How to Contact The MathWorks

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

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

508-647-7000 (Phone)
508-647-7001 (Fax)

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

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

MATLAB Function Reference


The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

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

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

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

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

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desktop Tools and Development Environment (p. 1-3)</td>
<td>Startup, Command Window, help, editing and debugging, tuning, other general functions</td>
</tr>
<tr>
<td>Mathematics (p. 1-13)</td>
<td>Arrays and matrices, linear algebra, other areas of mathematics</td>
</tr>
<tr>
<td>Data Analysis (p. 1-43)</td>
<td>Basic data operations, descriptive statistics, covariance and correlation, filtering and convolution, numerical derivatives and integrals, Fourier transforms, time series analysis</td>
</tr>
<tr>
<td>Programming and Data Types (p. 1-51)</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>Object-Oriented Programming (p. 1-77)</td>
<td>Functions for working with classes and objects</td>
</tr>
<tr>
<td>File I/O (p. 1-81)</td>
<td>General and low-level file I/O, plus specific file formats, like audio, spreadsheet, HDF, images</td>
</tr>
<tr>
<td>3-D Visualization (p. 1-104)</td>
<td>Surface and mesh plots, view control, lighting and transparency, volume visualization</td>
</tr>
</tbody>
</table>
GUI Development (p. 1-110)  
GUIDE, programming graphical user interfaces

External Interfaces (p. 1-115)  
Interfaces to shared libraries, Java, .NET, COM and ActiveX, Web services, and serial port devices, and C and Fortran routines
Desktop Tools and Development Environment

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

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

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

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

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

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

**Startup and Shutdown**

exit  
Terminate MATLAB® program (same as quit)

finish  
Termination M-file for MATLAB program

matlab (UNIX)  
Start MATLAB program (UNIX® platforms)

matlab (Windows)  
Start MATLAB program (Windows® platforms)

matlabrc  
Startup M-file for MATLAB program

prefdir  
Directory containing preferences, history, and layout files

preferences  
Open Preferences dialog box

quit  
Terminate MATLAB program
startup

Startup M-file for user-defined options

userpath

View or change user portion of search path

**Command Window and History**

clc

Clear Command Window

commandhistory

Open Command History window, or select it if already open

commandwindow

Open Command Window, or select it if already open

diary

Save session to file

dos

Execute DOS command and return result

format

Set display format for output

home

Move cursor to upper-left corner of Command Window

matlabcolon (matlab:)

Run specified function via hyperlink

more

Control paged output for Command Window

perl

Call Perl script using appropriate operating system executable

system

Execute operating system command and return result

unix

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

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

Workspace (p. 1-6) Manage variables
Search Path (p. 1-6) View and change MATLAB search path
File Operations (p. 1-7) View and change files and directories

Workspace

assignin Assign value to variable in specified workspace
clear Remove items from workspace, freeing up system memory
evalin Execute MATLAB expression in specified workspace
exist Check existence of variable, function, directory, or Java™ programming language class
openvar Open workspace variable in Variable Editor or other graphical editing tool
pack Consolidate workspace memory
uiimport Open Import Wizard to import data
which Locate functions and files
who, whos List variables in workspace
workspace Open Workspace browser to manage workspace

Search Path

addpath Add directories to search path
genpath Generate path string
partialpath
path
path2rc
pathsep
pathtool
restoredefaultpath
rmpath
savepath
userpath

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

cd
copyfile
delete
dir
exist
fileattrib
filebrowser
isdir
lookfor

<table>
<thead>
<tr>
<th>Partial pathname description</th>
</tr>
</thead>
<tbody>
<tr>
<td>View or change search path</td>
</tr>
<tr>
<td>Save current search path to pathdef.m file</td>
</tr>
<tr>
<td>Path separator for current platform</td>
</tr>
<tr>
<td>Open Set Path dialog box to view and change search path</td>
</tr>
<tr>
<td>Restore default search path</td>
</tr>
<tr>
<td>Remove directories from search path</td>
</tr>
<tr>
<td>Save current search path</td>
</tr>
<tr>
<td>View or change user portion of search path</td>
</tr>
<tr>
<td>Change working directory</td>
</tr>
<tr>
<td>Copy file or directory</td>
</tr>
<tr>
<td>Remove files or graphics objects</td>
</tr>
<tr>
<td>Directory listing</td>
</tr>
<tr>
<td>Check existence of variable, function, directory, or Java programming language class</td>
</tr>
<tr>
<td>Set or get attributes of file or directory</td>
</tr>
<tr>
<td>Open Current Directory browser, or select it if already open</td>
</tr>
<tr>
<td>Determine whether input is directory</td>
</tr>
<tr>
<td>Search for keyword in all help entries</td>
</tr>
<tr>
<td>Function</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>ls</td>
</tr>
<tr>
<td>matlabroot</td>
</tr>
<tr>
<td>mkdir</td>
</tr>
<tr>
<td>movefile</td>
</tr>
<tr>
<td>pwd</td>
</tr>
<tr>
<td>recycle</td>
</tr>
<tr>
<td>rehash</td>
</tr>
<tr>
<td>rmdir</td>
</tr>
<tr>
<td>toolboxdir</td>
</tr>
<tr>
<td>type</td>
</tr>
<tr>
<td>visdiff</td>
</tr>
<tr>
<td>what</td>
</tr>
<tr>
<td>which</td>
</tr>
</tbody>
</table>

**Programming Tools**

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

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

**M-File Performance**

- **bench**
  - MATLAB benchmark
- **mlint**
  - Check M-files for possible problems
- **mlintrpt**
  - Run `mlint` for file or directory, reporting results in browser
- **pack**
  - Consolidate workspace memory
profile
profsave
rehash
sparse
zeros

Profile execution time for function
Save profile report in HTML format
Refresh function and file system path caches
Create sparse matrix
Create array of all zeros

Source Control
checkin
checkout
cmopts
customverctrl
undocheckout
verctrl

Check files into source control system (UNIX platforms)
Check files out of source control system (UNIX platforms)
Name of source control system
Allow custom source control system (UNIX platforms)
Undo previous checkout from source control system (UNIX platforms)
Source control actions (Windows platforms)

Publishing
grabcode
notebook
publish
snapnow

MATLAB code from M-files published to HTML
Open M-book in Microsoft® Word software (on Microsoft Windows platforms)
Publish M-file containing cells, saving output to file of specified type
Force snapshot of image for inclusion in published document
System

Operating System Interface (p. 1-11)  Exchange operating system information and commands with MATLAB

MATLAB Version and License (p. 1-12)  Information about MATLAB version and license

Operating System Interface

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

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

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

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

Operators (p. 1-15) Arithmetic operators

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

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

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

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

Basic Information

disp Display text or array
display Display text or array (overloaded method)
isempty Determine whether array is empty
isequal Test arrays for equality
isequalwithnan Test arrays for equality, treating NaNs as equal
isfinite Array elements that are finite
isfloat Determine whether input is floating-point array
isinfi Array elements that are infinite
isinteger Determine whether input is integer array
islogical
Determine whether input is logical array

isnan
Array elements that are NaN

isnumeric
Determine whether input is numeric array

isscalar
Determine whether input is scalar

issparse
Determine whether input is sparse

isvector
Determine whether input is vector

length
Length of vector

max
Largest elements in array

min
Smallest elements in array

ndims
Number of array dimensions

numel
Number of elements in array or subscripted array expression

size
Array dimensions

**Operators**

+ Addition
+ Unary plus
- Subtraction
- Unary minus
* Matrix multiplication
^ Matrix power
\ Backslash or left matrix divide
/ Slash or right matrix divide
\' Transpose
\' Nonconjugated transpose
.* Array multiplication (element-wise)
\textbf{Elementary Matrices and Arrays}

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

**Array Operations**

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumarray</td>
<td>Construct array with accumulation</td>
</tr>
<tr>
<td>arrayfun</td>
<td>Apply function to each element of array</td>
</tr>
<tr>
<td>bsexfun</td>
<td>Apply element-by-element binary operation to two arrays with singleton expansion enabled</td>
</tr>
<tr>
<td>cast</td>
<td>Cast variable to different data type</td>
</tr>
<tr>
<td>cross</td>
<td>Vector cross product</td>
</tr>
<tr>
<td>cumprod</td>
<td>Cumulative product</td>
</tr>
<tr>
<td>cumsum</td>
<td>Cumulative sum</td>
</tr>
</tbody>
</table>
dot Vector dot product
idivide Integer division with rounding option
kron Kronecker tensor product
prod Product of array elements
sum Sum of array elements
tril Lower triangular part of matrix
triu Upper triangular part of matrix

Array Manipulation

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

**Specialized Matrices**

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

Rotate matrix 90 degrees
Shift dimensions
Sort array elements in ascending or
descending order
Sort rows in ascending order
Remove singleton dimensions
Vectorize expression
Concatenate arrays vertically
Companion matrix
Test matrices
Hadamard matrix
Hankel matrix
Hilbert matrix
Inverse of Hilbert matrix
Magic square
Pascal matrix
Classic symmetric eigenvalue test
problem
Toeplitz matrix
Vandermonde matrix
Wilkinson’s eigenvalue test matrix
### Linear Algebra

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

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

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

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

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

### Matrix Analysis

<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>Matrix determinant</td>
</tr>
<tr>
<td>norm</td>
<td>Vector and matrix norms</td>
</tr>
<tr>
<td>normest</td>
<td>2-norm estimate</td>
</tr>
<tr>
<td>null</td>
<td>Null space</td>
</tr>
<tr>
<td>orth</td>
<td>Range space of matrix</td>
</tr>
<tr>
<td>rank</td>
<td>Rank of matrix</td>
</tr>
<tr>
<td>rcond</td>
<td>Matrix reciprocal condition number estimate</td>
</tr>
<tr>
<td>rref</td>
<td>Reduced row echelon form</td>
</tr>
</tbody>
</table>
subspace  
Angle between two subspaces

trace  
Sum of diagonal elements

**Linear Equations**

chol  
Cholesky factorization

cholinc  
Sparse incomplete Cholesky and Cholesky-Infinity factorizations

cond  
Condition number with respect to inversion

condest  
1-norm condition number estimate

funm  
Evaluate general matrix function

ilu  
Sparse incomplete LU factorization

inv  
Matrix inverse

linsolve  
Solve linear system of equations

lscov  
Least-squares solution in presence of known covariance

lsqnonneg  
Solve nonnegative least-squares constraints problem

lu  
LU matrix factorization

luinc  
Sparse incomplete LU factorization

pinv  
Moore-Penrose pseudoinverse of matrix

qr  
Orthogonal-triangular decomposition

rcond  
Matrix reciprocal condition number estimate
## Eigenvalues and Singular Values

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

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

## Factorization

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

qz
QZ factorization for generalized eigenvalues

rssf2csf
Convert real Schur form to complex Schur form

svd
Singular value decomposition

Elementary Math

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

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

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

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

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

Trigonometric

acos
Inverse cosine; result in radians

acosd
Inverse cosine; result in degrees

acosh
Inverse hyperbolic cosine

acot
Inverse cotangent; result in radians

acotd
Inverse cotangent; result in degrees

acoth
Inverse hyperbolic cotangent

acsc
Inverse cosecant; result in radians

acscd
Inverse cosecant; result in degrees

acsch
Inverse hyperbolic cosecant
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asec</td>
<td>Inverse secant; result in radians</td>
</tr>
<tr>
<td>asecd</td>
<td>Inverse secant; result in degrees</td>
</tr>
<tr>
<td>asech</td>
<td>Inverse hyperbolic secant</td>
</tr>
<tr>
<td>asin</td>
<td>Inverse sine; result in radians</td>
</tr>
<tr>
<td>asind</td>
<td>Inverse sine; result in degrees</td>
</tr>
<tr>
<td>asinh</td>
<td>Inverse hyperbolic sine</td>
</tr>
<tr>
<td>atan</td>
<td>Inverse tangent; result in radians</td>
</tr>
<tr>
<td>atan2</td>
<td>Four-quadrant inverse tangent</td>
</tr>
<tr>
<td>atand</td>
<td>Inverse tangent; result in degrees</td>
</tr>
<tr>
<td>atanh</td>
<td>Inverse hyperbolic tangent</td>
</tr>
<tr>
<td>cos</td>
<td>Cosine of argument in radians</td>
</tr>
<tr>
<td>cosp</td>
<td>Cosine of argument in degrees</td>
</tr>
<tr>
<td>cosh</td>
<td>Hyperbolic cosine</td>
</tr>
<tr>
<td>cot</td>
<td>Cotangent of argument in radians</td>
</tr>
<tr>
<td>cotd</td>
<td>Cotangent of argument in degrees</td>
</tr>
<tr>
<td>coth</td>
<td>Hyperbolic cotangent</td>
</tr>
<tr>
<td>csc</td>
<td>Cosecant of argument in radians</td>
</tr>
<tr>
<td>cscd</td>
<td>Cosecant of argument in degrees</td>
</tr>
<tr>
<td>csch</td>
<td>Hyperbolic cosecant</td>
</tr>
<tr>
<td>hypot</td>
<td>Square root of sum of squares</td>
</tr>
<tr>
<td>sec</td>
<td>Secant of argument in radians</td>
</tr>
<tr>
<td>secd</td>
<td>Secant of argument in degrees</td>
</tr>
<tr>
<td>sech</td>
<td>Hyperbolic secant</td>
</tr>
<tr>
<td>sin</td>
<td>Sine of argument in radians</td>
</tr>
<tr>
<td>sind</td>
<td>Sine of argument in degrees</td>
</tr>
<tr>
<td>sinh</td>
<td>Hyperbolic sine of argument in radians</td>
</tr>
<tr>
<td>tan</td>
<td>Tangent of argument in radians</td>
</tr>
</tbody>
</table>
tand  
Tangent of argument in degrees

\textbf{tanh}  
Hyperbolic tangent

\textbf{Exponential}

\textbf{exp}  
Exponential

\textbf{expm1}  
Compute $\exp(x) - 1$ accurately for small values of $x$

\textbf{log}  
Natural logarithm

\textbf{log10}  
Common (base 10) logarithm

\textbf{log1p}  
Compute $\log(1+x)$ accurately for small values of $x$

\textbf{log2}  
Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

\textbf{nextpow2}  
Next higher power of 2

\textbf{nthroot}  
Real $n$th root of real numbers

\textbf{pow2}  
Base 2 power and scale floating-point numbers

\textbf{reallog}  
Natural logarithm for nonnegative real arrays

\textbf{realpow}  
Array power for real-only output

\textbf{realsqrt}  
Square root for nonnegative real arrays

\textbf{sqrt}  
Square root

\textbf{Complex}

\textbf{abs}  
Absolute value and complex magnitude

\textbf{angle}  
Phase angle
complex Construct complex data from real and imaginary components
conj Complex conjugate
cplxpair Sort complex numbers into complex conjugate pairs
i Imaginary unit
imag Imaginary part of complex number
isreal Check if input is real array
j Imaginary unit
real Real part of complex number
sign Signum function
unwrap Correct phase angles to produce smoother phase plots

**Rounding and Remainder**

ceil Round toward positive infinity
fix Round toward zero
floor Round toward negative infinity
idivide Integer division with rounding option
mod Modulus after division
rem Remainder after division
round Round to nearest integer

**Discrete Math**

factor Prime factors
factorial Factorial function
gcd Greatest common divisor
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isprime</td>
<td>Array elements that are prime numbers</td>
</tr>
<tr>
<td>lcm</td>
<td>Least common multiple</td>
</tr>
<tr>
<td>nchoosek</td>
<td>Binomial coefficient or all combinations</td>
</tr>
<tr>
<td>perms</td>
<td>All possible permutations</td>
</tr>
<tr>
<td>primes</td>
<td>Generate list of prime numbers</td>
</tr>
<tr>
<td>rat, rats</td>
<td>Rational fraction approximation</td>
</tr>
</tbody>
</table>

### Polynomials

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

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

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

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

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

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

**Interpolation**

dsearch  
Search Delaunay triangulation for nearest point

dsearchn  
N-D nearest point search

griidata  
Data gridding

griidata3  
Data gridding and hypersurface fitting for 3-D data

griidatans  
Data gridding and hypersurface fitting (dimension \(\geq 2\))

interp1  
1-D data interpolation (table lookup)

interp1q  
Quick 1-D linear interpolation

interp2  
2-D data interpolation (table lookup)

interp3  
3-D data interpolation (table lookup)

interpft  
1-D interpolation using FFT method

interpn  
N-D data interpolation (table lookup)

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

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

### Delaunay Triangulation and Tessellation

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

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

Voronoi Diagrams

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

Domain Generation

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

Cartesian Coordinate System Conversion

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

Nonlinear Numerical Methods

Ordinary Differential Equations (p. 1-33)
Solve stiff and nonstiff differential equations, define the problem, set solver options, evaluate solution

Delay Differential Equations (p. 1-34)
Solve delay differential equations with constant and general delays, set solver options, evaluate solution

Boundary Value Problems (p. 1-35)
Solve boundary value problems for ordinary differential equations, set solver options, evaluate solution

Partial Differential Equations (p. 1-35)
Solve initial-boundary value problems for parabolic-elliptic PDEs, evaluate solution

Optimization (p. 1-35)
Find minimum of single and multivariable functions, solve nonnegative least-squares constraint problem

Numerical Integration (Quadrature) (p. 1-36)
Evaluate Simpson, Lobatto, and vectorized quadratures, evaluate double and triple integrals

Ordinary Differential Equations

decic Compute consistent initial conditions for ode15i
deval Evaluate solution of differential equation problem
ode15i
Solve fully implicit differential equations, variable order method

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

odefile
Define differential equation problem for ordinary differential equation solvers

odeget
Ordinary differential equation options parameters

odeset
Create or alter options structure for ordinary differential equation solvers

odextend
Extend solution of initial value problem for ordinary differential equation

**Delay Differential Equations**

dde23
Solve delay differential equations (DDEs) with constant delays

ddeget
Extract properties from delay differential equations options structure

ddesd
Solve delay differential equations (DDEs) with general delays

ddeset
Create or alter delay differential equations options structure

deval
Evaluate solution of differential equation problem
### Boundary Value Problems

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

### Partial Differential Equations

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

### Optimization

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

**Numerical Integration (Quadrature)**

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

**Specialized Math**

airy Airy functions
besselh Bessel function of third kind (Hankel function)
besseli Modified Bessel function of first kind
besselj Bessel function of first kind
besselk Modified Bessel function of second kind
bessely Bessel function of second kind
Mathematics

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

Sparse Matrices

**Elementary Sparse Matrices** (p. 1-38)
Create random and nonrandom sparse matrices

**Full to Sparse Conversion** (p. 1-38)
Convert full matrix to sparse, sparse matrix to full

**Sparse Matrix Manipulation** (p. 1-39)
Test matrix for sparseness, get information on sparse matrix, allocate sparse matrix, apply function to nonzero elements, visualize sparsity pattern

**Reordering Algorithms** (p. 1-39)
Random, column, minimum degree, Dulmage-Mendelsohn, and reverse Cuthill-McKee permutations
### Linear Algebra (p. 1-40)
Compute norms, eigenvalues, factorizations, least squares, structural rank

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

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

## Elementary Sparse Matrices

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

## Full to Sparse Conversion

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

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

**Reordering Algorithms**

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

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

**Linear Equations (Iterative Methods)**

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

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

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

Basic Operations (p. 1-43) Sums, products, sorting
Descriptive Statistics (p. 1-43) Statistical summaries of data
Filtering and Convolution (p. 1-44) Data preprocessing
Interpolation and Regression (p. 1-44) Data fitting
Fourier Transforms (p. 1-45) Frequency content of data
Derivatives and Integrals (p. 1-45) Data rates and accumulations
Time Series Objects (p. 1-46) Methods for timeseries objects
Time Series Collections (p. 1-49) Methods for tscollection objects

Basic Operations

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

descriptive Statistics

corrcoef Correlation coefficients
cov Covariance matrix
max Largest elements in array
mean Average or mean value of array
median Median value of array
min Smallest elements in array
mode Most frequent values in array
std Standard deviation
var Variance

**Filtering and Convolution**

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

**Interpolation and Regression**

interp1 1-D data interpolation (table lookup)
interp2 2-D data interpolation (table lookup)
interp3 3-D data interpolation (table lookup)
interpN N-D data interpolation (table lookup)
ldivide \, mrdivide /
left or right matrix division
polyfit Polynomial curve fitting
polyval Polynomial evaluation
### Fourier Transforms

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

### Derivatives and Integrals

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

**Time Series Objects**

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

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

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

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

**Utilities**

`get (timeseries)`
Query `timeseries` object property values

`getdatasamplesize`
Size of data sample in `timeseries` object

`getqualitydesc`
Data quality descriptions

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

`length (timeseries)`
Length of time vector

`plot (timeseries)`
Plot time series

`set (timeseries)`
Set properties of `timeseries` object

`size (timeseries)`
Size of `timeseries` object
timeseries
Create timeseries object

tsdata.event
Construct event object for timeseries object

tsprops
Help on timeseries object properties

tstool
Open Time Series Tools GUI

**Data Manipulation**

addsample
Add data sample to timeseries object

ctranspose (timeseries)
Transpose timeseries object

delsample
Remove sample from timeseries object

detrend (timeseries)
Subtract mean or best-fit line and all NaNs from time series

filter (timeseries)
Shape frequency content of time series

getabstime (timeseries)
Extract date-string time vector into cell array

getinterpmethod
Interpolation method for timeseries object

getsampleusingtime (timeseries)
Extract data samples into new timeseries object

idealfilter (timeseries)
Apply ideal (noncausal) filter to timeseries object

resample (timeseries)
Select or interpolate timeseries data using new time vector

setabstime (timeseries)
Set times of timeseries object as date strings

setinterpmethod
Set default interpolation method for timeseries object
synchronize  Synchronize and resample two timeseries objects using common time vector

transpose (timeseries)  Transpose timeseries object
vertcat (timeseries)  Vertical concatenation of timeseries objects

**Event Data**

addevent  Add event to timeseries object
delevent  Remove tsdata.event objects from timeseries object
gettsafterevent  New timeseries object with samples occurring at or after event
gettsbforevent  New timeseries object with samples occurring after event
gettsbforevent  New timeseries object with samples occurring at event
gettsbforevent  New timeseries object with samples occurring before or at event
gettsbforevent  New timeseries object with samples occurring before event
gettsbbetweenevents  New timeseries object with samples occurring between events

**Descriptive Statistics**
iqr (timeseries)  Interquartile range of timeseries data
max (timeseries)  Maximum value of timeseries data
mean (timeseries)  Mean value of timeseries data
median (timeseries)  Median value of timeseries data
min (timeseries)  Minimum value of timeseries data
std (timeseries)  Standard deviation of timeseries data
sum (timeseries)  Sum of timeseries data
var (timeseries)  Variance of timeseries data

**Time Series Collections**

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

**Utilities**

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

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


## Programming and Data Types

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

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

**Operators and Special Characters**
- Arithmetic, relational, and logical operators, and special characters

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

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

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

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

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

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

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

### Data Types

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

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

**Structures** (p. 1-54)
- Data of varying types and sizes stored in fields of a structure
Cell Arrays (p. 1-55)  
Data of varying types and sizes stored in cells of array

Function Handles (p. 1-56)  
Invoke a function indirectly via handle

Java Classes and Objects (p. 1-56)  
Access Java classes through MATLAB interface

Data Type Identification (p. 1-58)  
Determine data type of a variable

**Numeric Types**

arrayfun  
Apply function to each element of array

cast  
Cast variable to different data type

cat  
Concatenate arrays along specified dimension

class  
Create object or return class of object

find  
Find indices and values of nonzero elements

intmax  
Largest value of specified integer type

intmin  
Smallest value of specified integer type

intwarning  
Control state of integer warnings

ipermute  
Inverse permute dimensions of N-D array

isa  
Determine whether input is object of given class

isequal  
Test arrays for equality

isequalwithequalnans  
Test arrays for equality, treating NaNs as equal

isfinite  
Array elements that are finite
### Programming and Data Types

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

### Characters and Strings

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

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

- **sscanf**  
  Read formatted data from string

- **strcat**  
  Concatenate strings horizontally

- **strcmp, strcmpi**  
  Compare strings

- **strings**  
  String handling

- **strjust**  
  Justify character array

- **strmatch**  
  Find possible matches for string

- **strread**  
  Read formatted data from string

- **strrep**  
  Find and replace substring

- **strtrim**  
  Remove leading and trailing white space from string

- **strvcat**  
  Concatenate strings vertically

**Structures**

- **arrayfun**  
  Apply function to each element of array

- **cell2struct**  
  Convert cell array to structure array

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

- **deal**  
  Distribute inputs to outputs

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

- **getfield**  
  Field of structure array

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

- **isequal**  
  Test arrays for equality

- **isfield**  
  Determine whether input is structure array field

- **isscalar**  
  Determine whether input is scalar

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

Cell Arrays

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

**Function Handles**

class Create object or return class of object
feval Evaluate function
func2str Construct function name string from function handle
functions Information about function handle
function_handle (@) Handle used in calling functions indirectly
isa Determine whether input is object of given class
isequal Test arrays for equality
str2func Construct function handle from function name string

**Java Classes and Objects**

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

**Data Type Identification**

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

Data Type Conversion

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

Numeric

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

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

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

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

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

## Numeric to String

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

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

**Operators and Special Characters**

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

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

**Relational Operators**

<        Less than
<=       Less than or equal to

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

**Logical Operators**

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

**Special Characters**

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

Description of Strings in MATLAB (p. 1-64)
String Creation (p. 1-64)
String Identification (p. 1-65)
String Manipulation (p. 1-65)
String Parsing (p. 1-66)
String Evaluation (p. 1-66)
String Comparison (p. 1-66)

Basics of string handling in MATLAB
Create strings, cell arrays of strings, concatenate strings together
Identify characteristics of strings
Convert case, strip blanks, replace characters
Formatted read, regular expressions, locate substrings
Evaluate stated expression in string
Compare contents of strings

Description of Strings in MATLAB

strings
String handling

String Creation

blanks
Create string of blank characters

cellstr
Create cell array of strings from character array

char
Convert to character array (string)

sprintf
Write formatted data to string

strcat
 Concatenate strings horizontally

strvcat
 Concatenate strings vertically
### String Identification

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

### String Manipulation

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

findstr
regexp, regexpi
regexp
regexptranslate
sscanf
strfind
strread
strtok

Find string within another, longer string
Match regular expression
Replace string using regular expression
Translate string into regular expression
Read formatted data from string
Find one string within another
Read formatted data from string
Selected parts of string

String Evaluation

eval
evalc
evalin

Execute string containing MATLAB expression
Evaluate MATLAB expression with capture
Execute MATLAB expression in specified workspace

String Comparison

strcmp, strcmpi
strmatch
strncmp, strncmpi

Compare strings
Find possible matches for string
Compare first n characters of strings
**Bit-Wise Operations**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bitand</code></td>
<td>Bitwise AND</td>
</tr>
<tr>
<td><code>bitcmp</code></td>
<td>Bitwise complement</td>
</tr>
<tr>
<td><code>bitget</code></td>
<td>Bit at specified position</td>
</tr>
<tr>
<td><code>bitmax</code></td>
<td>Maximum double-precision floating-point integer</td>
</tr>
<tr>
<td><code>bitor</code></td>
<td>Bitwise OR</td>
</tr>
<tr>
<td><code>bitset</code></td>
<td>Set bit at specified position</td>
</tr>
<tr>
<td><code>bitshift</code></td>
<td>Shift bits specified number of places</td>
</tr>
<tr>
<td><code>bitxor</code></td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td><code>swapbytes</code></td>
<td>Swap byte ordering</td>
</tr>
</tbody>
</table>

**Logical Operations**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>all</code></td>
<td>Determine whether all array elements are nonzero</td>
</tr>
<tr>
<td><code>and</code></td>
<td>Find logical AND of array or scalar inputs</td>
</tr>
<tr>
<td><code>any</code></td>
<td>Determine whether any array elements are nonzero</td>
</tr>
<tr>
<td><code>false</code></td>
<td>Logical 0 (false)</td>
</tr>
<tr>
<td><code>find</code></td>
<td>Find indices and values of nonzero elements</td>
</tr>
<tr>
<td><code>isa</code></td>
<td>Determine whether input is object of given class</td>
</tr>
<tr>
<td><code>iskeyword</code></td>
<td>Determine whether input is MATLAB keyword</td>
</tr>
<tr>
<td><code>isvarname</code></td>
<td>Determine whether input is valid variable name</td>
</tr>
<tr>
<td><code>logical</code></td>
<td>Convert numeric values to logical</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>not</td>
<td>Find logical NOT of array or scalar input</td>
</tr>
<tr>
<td>or</td>
<td>Find logical OR of array or scalar inputs</td>
</tr>
<tr>
<td>true</td>
<td>Logical 1 (true)</td>
</tr>
<tr>
<td>xor</td>
<td>Logical exclusive-OR</td>
</tr>
</tbody>
</table>

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

**Relational Operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>Test for equality</td>
</tr>
<tr>
<td>ge</td>
<td>Test for greater than or equal to</td>
</tr>
<tr>
<td>gt</td>
<td>Test for greater than</td>
</tr>
<tr>
<td>le</td>
<td>Test for less than or equal to</td>
</tr>
<tr>
<td>lt</td>
<td>Test for less than</td>
</tr>
<tr>
<td>ne</td>
<td>Test for inequality</td>
</tr>
</tbody>
</table>

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

**Set Operations**

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

**Date and Time Operations**

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

**Programming in MATLAB**

M-Files and Scripts (p. 1-70)
Evaluation (p. 1-71)
Timer (p. 1-72)
| Variables and Functions in Memory (p. 1-73) | List files in memory, clear M-files in memory, assign to variable in nondefault workspace, refresh caches |
| Control Flow (p. 1-74) | if-then-else, for loops, switch-case, try-catch |
| Error Handling (p. 1-75) | Generate warnings and errors, test for and catch errors, retrieve most recent error message |
| MEX Programming (p. 1-76) | Compile MEX function from C or Fortran code, list MEX-files in memory, debug MEX-files |

**M-Files and Scripts**

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

**Evaluation**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ans</td>
<td>Most recent answer</td>
</tr>
<tr>
<td>arrayfun</td>
<td>Apply function to each element of array</td>
</tr>
<tr>
<td>assert</td>
<td>Generate error when condition is violated</td>
</tr>
<tr>
<td>builtin</td>
<td>Execute built-in function from overloaded method</td>
</tr>
<tr>
<td>cellfun</td>
<td>Apply function to each cell in cell array</td>
</tr>
<tr>
<td>echo</td>
<td>Echo M-files during execution</td>
</tr>
<tr>
<td>eval</td>
<td>Execute string containing MATLAB expression</td>
</tr>
<tr>
<td>evalc</td>
<td>Evaluate MATLAB expression with capture</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>evalin</td>
<td>Execute MATLAB expression in specified workspace</td>
</tr>
<tr>
<td>feval</td>
<td>Evaluate function</td>
</tr>
<tr>
<td>iskeyword</td>
<td>Determine whether input is MATLAB keyword</td>
</tr>
<tr>
<td>isvarname</td>
<td>Determine whether input is valid variable name</td>
</tr>
<tr>
<td>pause</td>
<td>Halt execution temporarily</td>
</tr>
<tr>
<td>run</td>
<td>Run script that is not on current path</td>
</tr>
<tr>
<td>script</td>
<td>Script M-file description</td>
</tr>
<tr>
<td>structfun</td>
<td>Apply function to each field of scalar structure</td>
</tr>
<tr>
<td>symvar</td>
<td>Determine symbolic variables in expression</td>
</tr>
<tr>
<td>tic, toc</td>
<td>Measure performance using stopwatch timer</td>
</tr>
</tbody>
</table>

**Timer**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete (timer)</td>
<td>Remove timer object from memory</td>
</tr>
<tr>
<td>disp (timer)</td>
<td>Information about timer object</td>
</tr>
<tr>
<td>get (timer)</td>
<td>Timer object properties</td>
</tr>
<tr>
<td>isvalid (timer)</td>
<td>Determine whether timer object is valid</td>
</tr>
<tr>
<td>set (timer)</td>
<td>Configure or display timer object properties</td>
</tr>
<tr>
<td>start</td>
<td>Start timer(s) running</td>
</tr>
<tr>
<td>startat</td>
<td>Start timer(s) running at specified time</td>
</tr>
<tr>
<td>stop</td>
<td>Stop timer(s)</td>
</tr>
</tbody>
</table>

1-72
### Programming and Data Types

- **timer**
  - Construct timer object
- **timerfind**
  - Find timer objects
- **timerfindall**
  - Find timer objects, including invisible objects
- **wait**
  - Wait until timer stops running

### Variables and Functions in Memory

- **ans**
  - Most recent answer
- **assignin**
  - Assign value to variable in specified workspace
- **datatipinfo**
  - Produce short description of input variable
- **genvarname**
  - Construct valid variable name from string
- **global**
  - Declare global variables
- **inmem**
  - Names of M-files, MEX-files, Sun Java classes in memory
- **isglobal**
  - Determine whether input is global variable
- **memory**
  - Display memory information
- **mislocked**
  - Determine whether M-file or MEX-file cannot be cleared from memory
- **mlock**
  - Prevent clearing M-file or MEX-file from memory
- **munlock**
  - Allow clearing M-file or MEX-file from memory
- **namelengthmax**
  - Maximum identifier length
- **pack**
  - Consolidate workspace memory
persistent
rehash

**Control Flow**

break
Terminate execution of *for* or *while* loop

case
Execute block of code if condition is true

catch
Specify how to respond to error in try statement

continue
Pass control to next iteration of *for* or *while* loop

extelse
Execute statements if condition is false

elseif
Execute statements if additional condition is true

end
Terminate block of code, or indicate last array index

error
Display message and abort function

for
Execute block of code specified number of times

if
Execute statements if condition is true

otherwise
Default part of switch statement

parfor
Parallel for-loop

return
Return to invoking function

switch
Switch among several cases, based on expression
try
attempt to execute block of code, and catch errors

while
repeatedly execute statements while condition is true

**Error Handling**

addCause (MException)
append MException objects

assert
generate error when condition is violated

catch
specify how to respond to error in try statement

disp (MException)
display MException object

eq (MException)
compare MException objects for equality

error
display message and abort function

ferror
query MATLAB software about errors in file input or output

getReport (MException)
get error message for exception

intwarning
control state of integer warnings

isequal (MException)
compare MException objects for equality

last (MException)
last uncaught exception

lasterr
last error message

lasterror
last error message and related information

lastwarn
last warning message

MException
construct MException object

ne (MException)
compare MException objects for inequality

rethrow
reissue error
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rethrow (MException)</td>
<td>Reissue existing exception</td>
</tr>
<tr>
<td>throw (MException)</td>
<td>Terminate function and issue exception</td>
</tr>
<tr>
<td>try</td>
<td>Attempt to execute block of code, and catch errors</td>
</tr>
<tr>
<td>warning</td>
<td>Warning message</td>
</tr>
</tbody>
</table>

**MEX Programming**

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

Classes and Objects (p. 1-77)

Get information about classes and objects

Handle Classes (p. 1-78)

Define and use handle classes

Events and Listeners (p. 1-79)

Define and use events and listeners

Meta-Classes (p. 1-79)

Access information about classes without requiring instances

Classes and Objects

class
Create object or return class of object
classdef
Class definition key words
fieldnames
Field names of structure, or public fields of object
import
Add package or class to current import list
inferiorto
Specify inferior class relationship
isa
Determine whether input is object of given class
isobject
Determine if input is MATLAB object
loadobj
Modify how load function loads objects
methods
Information on class methods
methodview
Information on class methods in separate window
properties
Display class property names
saveobj
Modify how save function saves objects
subsasgn
Subscripted assignment for objects
Function Reference

subsindex
Subscripted indexing using object as index

subsref
Subscripted reference for objects

substruct
Create structure argument for subsasgn or subsref

superiorto
Establish superior class relationship

Handle Classes

addlistener (handle)
Create event listener

addprop (dynamicprops)
Add dynamic property

delete (handle)
Handle object destructor function

dynamicprops
Abstract class used to derive handle class with dynamic properties

findobj (handle)
Find objects matching specified conditions

findprop (handle)
Find meta.property object associated with property name

get (hgsetget)
Query property values of handle objects derived from hgsetget class

getdisp (hgsetget)
Override to change command window display

handle
Abstract class for deriving handle classes

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

isvalid (handle)
Is object valid handle class object

notify (handle)
Notify listeners that event is occurring

relationaloperators (handle)
Equality and sorting of handle objects
set (hgsetget) 
Assign property values to handle objects derived from hgsetget class

setdisp (hgsetget) 
Override to change command window display

---

**Events and Listeners**

addlistener (handle) 
Create event listener

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

event.listener 
Class defining listener objects

event.PropertyEvent 
Listener for property events

event.proplistener 
Define listener object for property events

events 
Display class event names

notify (handle) 
Notify listeners that event is occurring

---

**Meta-Classes**

meta.class 
(meta.class class describes MATLAB classes

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

meta.DynamicProperty 
(meta.DynamicProperty class describes dynamic property of MATLAB object

meta.event 
(meta.event class describes MATLAB class events

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

File Name Construction (p. 1-81)  Get path, directory, filename information; construct filenames
File Opening, Loading, and Saving (p. 1-82)  Open files; transfer data between files and MATLAB workspace
Memory Mapping (p. 1-82)  Access file data via memory map using MATLAB array indexing
Low-Level File I/O (p. 1-82)  Low-level operations that use a file identifier
Text Files (p. 1-83)  Delimited or formatted I/O to text files
XML Documents (p. 1-84)  Documents written in Extensible Markup Language
Spreadsheets (p. 1-84)  Excel and Lotus 1-2-3 files
Scientific Data (p. 1-85)  CDF, FITS, HDF formats
Audio and Audio/Video (p. 1-88)  General audio functions; SparcStation, WAVE, AVI files
Images (p. 1-90)  Graphics files
Internet Exchange (p. 1-91)  URL, FTP, zip, tar, and e-mail

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

File Name Construction

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

Name of system’s temporary directory

tempname

Unique name for temporary file

**File Opening, Loading, and Saving**

daqread

Read Data Acquisition Toolbox™ (.daq) file

importdata

Load data from disk file

load

Load workspace variables from disk

open

Open files based on extension

save

Save workspace variables to disk

uiimport

Open Import Wizard to import data

winopen

Open file in appropriate application (Windows)

**Memory Mapping**

disp (memmapfile)

Information about memmapfile object

get (memmapfile)

Memmapfile object properties

memmapfile

Construct memmapfile object

**Low-Level File I/O**

close

Close one or more open files

feof

Test for end-of-file

ferror

Query MATLAB software about errors in file input or output
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgets</td>
<td>Read line from file, keeping newline characters</td>
</tr>
<tr>
<td>fopen</td>
<td>Open file, or obtain information about open files</td>
</tr>
<tr>
<td>fprintf</td>
<td>Write formatted data to file</td>
</tr>
<tr>
<td>fread</td>
<td>Read binary data from file</td>
</tr>
<tr>
<td>frewind</td>
<td>Move file position indicator to beginning of open file</td>
</tr>
<tr>
<td>fscanf</td>
<td>Read formatted data from a text file</td>
</tr>
<tr>
<td>fseek</td>
<td>Set file position indicator</td>
</tr>
<tr>
<td>ftell</td>
<td>File position indicator</td>
</tr>
<tr>
<td>fwrite</td>
<td>Write binary data to file</td>
</tr>
</tbody>
</table>

**Text Files**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csvread</td>
<td>Read comma-separated value file</td>
</tr>
<tr>
<td>csvwrite</td>
<td>Write comma-separated value file</td>
</tr>
<tr>
<td>dlmread</td>
<td>Read ASCII-delimited file of numeric data into matrix</td>
</tr>
<tr>
<td>dlmwrite</td>
<td>Write matrix to ASCII-delimited file</td>
</tr>
<tr>
<td>fileread</td>
<td>Return contents of file as string vector</td>
</tr>
<tr>
<td>textread</td>
<td>Read data from text file; write to multiple outputs</td>
</tr>
<tr>
<td>textscan</td>
<td>Read formatted data from text file or string</td>
</tr>
</tbody>
</table>
### XML Documents

- **xmlread**
  - Parse XML document and return Document Object Model node

- **xmlwrite**
  - Serialize XML Document Object Model node

- **xslt**
  - Transform XML document using XSLT engine

### Spreadsheets

- **Microsoft Excel** (p. 1-84)
  - **xlsinfo**
    - Determine whether file contains a Microsoft® Excel® spreadsheet
  - **xlsread**
    - Read Microsoft Excel spreadsheet file
  - **xlswrite**
    - Write Microsoft Excel spreadsheet file

- **Lotus 1-2-3** (p. 1-84)
  - **wk1info**
    - Determine whether file contains 1-2-3 WK1 worksheet
  - **wk1read**
    - Read Lotus 1-2-3 WK1 spreadsheet file into matrix
  - **wk1write**
    - Write matrix to Lotus 1-2-3 WK1 spreadsheet file
**Scientific Data**

- **Common Data Format (p. 1-85)**: Work with CDF files
- **Network Common Data Form (p. 1-85)**: Work with netCDF files
- **Flexible Image Transport System (p. 1-87)**: Work with FITS files
- **Hierarchical Data Format (p. 1-87)**: Work with HDF files
- **Band-Interleaved Data (p. 1-88)**: Work with band-interleaved files

**Common Data Format**

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

**Network Common Data Form**

**File Operations**

- **netcdf**: Summary of MATLAB Network Common Data Form (netCDF) capabilities
- **netcdf.abort**: Revert recent netCDF file definitions
- **netcdf.close**: Close netCDF file
- **netcdf.create**: Create new netCDF dataset
**netcdf.endDef**  
End netCDF file define mode

**netcdf.getConstant**  
Return numeric value of named constant

**netcdf.getConstantNames**  
Return list of constants known to netCDF library

**netcdf.inq**  
Return information about netCDF file

**netcdf.inqLibVers**  
Return netCDF library version information

**netcdf.open**  
Open netCDF file

**netcdf.reDef**  
Put open netCDF file into define mode

**netcdfsetDefaultFormat**  
Change default netCDF file format

**netcdf.setFill**  
Set netCDF fill mode

**netcdf.sync**  
Synchronize netCDF file to disk

### Dimensions

**netcdf.defDim**  
Create netCDF dimension

**netcdf.inqDim**  
Return netCDF dimension name and length

**netcdf.inqDimID**  
Return dimension ID

**netcdf.renameDim**  
Change name of netCDF dimension

### Variables

**netcdf.defVar**  
Create netCDF variable

**netcdf.getVar**  
Return data from netCDF variable

**netcdf.inqVar**  
Return information about variable

**netcdf.inqVarID**  
Return ID associated with variable name
File I/O

netcdf.putVar  Write data to netCDF variable
netcdf.renameVar  Change name of netCDF variable

Attributes

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

Flexible Image Transport System

fitsinfo  Information about FITS file
fitsread  Read data from FITS file

Hierarchical Data Format

hdf  Summary of MATLAB HDF4 capabilities
hdf5  Summary of MATLAB HDF5 capabilities
hdf5info  Information about HDF5 file
hdf5read  Read HDF5 file
hdf5write  Write data to file in HDF5 format
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdfinfo</td>
<td>Information about HDF4 or HDF-EOS file</td>
</tr>
<tr>
<td>hdfread</td>
<td>Read data from HDF4 or HDF-EOS file</td>
</tr>
<tr>
<td>hdftool</td>
<td>Browse and import data from HDF4 or HDF-EOS files</td>
</tr>
<tr>
<td><strong>Band-Interleaved Data</strong></td>
<td></td>
</tr>
<tr>
<td>multibandread</td>
<td>Read band-interleaved data from binary file</td>
</tr>
<tr>
<td>multibandwrite</td>
<td>Write band-interleaved data to file</td>
</tr>
<tr>
<td><strong>Audio and Audio/Video</strong></td>
<td></td>
</tr>
<tr>
<td>Utilities (p. 1-88)</td>
<td>Create audio player object, obtain information about multimedia files, convert to/from audio signal</td>
</tr>
<tr>
<td>SPARCstation-Specific Sound (p. 1-89)</td>
<td>Access NeXT/SUN (.au) sound files</td>
</tr>
<tr>
<td>Microsoft WAVE Sound (p. 1-89)</td>
<td>Access Microsoft WAVE (.wav) sound files</td>
</tr>
<tr>
<td>Audio/Video Interleaved (p. 1-90)</td>
<td>Access Audio/Video interleaved (.avi) sound files</td>
</tr>
<tr>
<td><strong>Utilities</strong></td>
<td></td>
</tr>
<tr>
<td>audiodevinfo</td>
<td>Information about audio device</td>
</tr>
<tr>
<td>audioplayer</td>
<td>Create audioplayer object</td>
</tr>
<tr>
<td>audiorecorder</td>
<td>Create audiorecorder object</td>
</tr>
<tr>
<td>beep</td>
<td>Produce beep sound</td>
</tr>
</tbody>
</table>
lin2mu  Convert linear audio signal to mu-law
mmfileinfo Information about multimedia file
mmreader Create multimedia reader object for reading video files
mmreader.isPlatformSupported Determine whether mmreader function is available on current platform
mu2lin Convert mu-law audio signal to linear
read Read video frame data from multimedia reader object
sound Convert vector into sound
soundsc Scale data and play as sound

**SPARCstation-Specific Sound**

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

**Microsoft WAVE Sound**

wavinfo Information about Microsoft WAVE (.wav) sound file
wavplay Play recorded sound on PC-based audio output device
wavread Read Microsoft WAVE (.wav) sound file
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wavrecord</td>
<td>Recordsound using PC-based audio input device</td>
</tr>
<tr>
<td>wavwrite</td>
<td>Write Microsoft WAVE (.wav) sound file</td>
</tr>
<tr>
<td><strong>Audio/Video Interleaved</strong></td>
<td></td>
</tr>
<tr>
<td>addframe</td>
<td>Add frame to Audio/Video Interleaved (AVI) file</td>
</tr>
<tr>
<td>avifile</td>
<td>Create new Audio/Video Interleaved (AVI) file</td>
</tr>
<tr>
<td>aviinfo</td>
<td>Information about Audio/Video Interleaved (AVI) file</td>
</tr>
<tr>
<td>aviread</td>
<td>Read Audio/Video Interleaved (AVI) file</td>
</tr>
<tr>
<td>close (avifile)</td>
<td>Close Audio/Video Interleaved (AVI) file</td>
</tr>
<tr>
<td>movie2avi</td>
<td>Create Audio/Video Interleaved (AVI) movie from MATLAB movie</td>
</tr>
<tr>
<td><strong>Images</strong></td>
<td></td>
</tr>
<tr>
<td>exifread</td>
<td>Read EXIF information from JPEG and TIFF image files</td>
</tr>
<tr>
<td>im2java</td>
<td>Convert image to Java image</td>
</tr>
<tr>
<td>imfinfo</td>
<td>Information about graphics file</td>
</tr>
<tr>
<td>imread</td>
<td>Read image from graphics file</td>
</tr>
<tr>
<td>imwrite</td>
<td>Write image to graphics file</td>
</tr>
</tbody>
</table>
Internet Exchange

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

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

URL, Zip, Tar, E-Mail

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

FTP

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

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

Basic Plots and Graphs

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

- `figurepalette` Show or hide figure palette
- `pan` Pan view of graph interactively
- `plotbrowser` Show or hide figure plot browser
- `plotedit` Interactively edit and annotate plots
- `plottools` Show or hide plot tools
- `propertyeditor` Show or hide property editor
- `rotate3d` Rotate 3-D view using mouse
- `showplottool` Show or hide figure plot tool
- `zoom` Turn zooming on or off or magnify by factor

**Annotating Plots**

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

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

Area, Bar, and Pie Plots

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

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

### Direction and Velocity Plots

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

### Discrete Data Plots

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

### Function Plots

- **ezcontour**
  - Easy-to-use contour plotter
- **ezcontourf**
  - Easy-to-use filled contour plotter
- **ezmesh**
  - Easy-to-use 3-D mesh plotter
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 function between specified limits</td>
</tr>
</tbody>
</table>

**Histograms**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hist</td>
<td>Histogram plot</td>
</tr>
<tr>
<td>histc</td>
<td>Histogram count</td>
</tr>
<tr>
<td>rose</td>
<td>Angle histogram plot</td>
</tr>
</tbody>
</table>

**Polygons and Surfaces**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convhull</td>
<td>Convex hull</td>
</tr>
<tr>
<td>cylinder</td>
<td>Generate cylinder</td>
</tr>
<tr>
<td>delaunay</td>
<td>Delaunay triangulation</td>
</tr>
<tr>
<td>delaunay3</td>
<td>3-D Delaunay tessellation</td>
</tr>
<tr>
<td>delaunayn</td>
<td>N-D Delaunay tessellation</td>
</tr>
<tr>
<td>dsearch</td>
<td>Search Delaunay triangulation for nearest point</td>
</tr>
<tr>
<td>dsearchn</td>
<td>N-D nearest point search</td>
</tr>
<tr>
<td>ellipsoid</td>
<td>Generate ellipsoid</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>fill</td>
<td>Filled 2-D polygons</td>
</tr>
<tr>
<td>fill3</td>
<td>Filled 3-D polygons</td>
</tr>
<tr>
<td>inpolygon</td>
<td>Points inside polygonal region</td>
</tr>
<tr>
<td>pcolor</td>
<td>Pseudocolor (checkerboard) plot</td>
</tr>
<tr>
<td>polyarea</td>
<td>Area of polygon</td>
</tr>
<tr>
<td>rectint</td>
<td>Rectangle intersection area</td>
</tr>
<tr>
<td>ribbon</td>
<td>Ribbon plot</td>
</tr>
<tr>
<td>slice</td>
<td>Volumetric slice plot</td>
</tr>
<tr>
<td>sphere</td>
<td>Generate sphere</td>
</tr>
<tr>
<td>tsearch</td>
<td>Search for enclosing Delaunay triangle</td>
</tr>
<tr>
<td>tsearchn</td>
<td>N-D closest simplex search</td>
</tr>
<tr>
<td>voronoi</td>
<td>Voronoi diagram</td>
</tr>
<tr>
<td>waterfall</td>
<td>Waterfall plot</td>
</tr>
</tbody>
</table>

**Scatter/Bubble Plots**

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

**Animation**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame2im</td>
<td>Return image data associated with movie frame</td>
</tr>
<tr>
<td>getframe</td>
<td>Capture movie frame</td>
</tr>
<tr>
<td>im2frame</td>
<td>Convert image to movie frame</td>
</tr>
</tbody>
</table>
movie
noanimate

**Bit-Mapped Images**

frame2im
im2frame
im2java
image
imagesc
imfinfo
imformats
imread
imwrite
ind2rgb

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

**Printing**

hgexport
orient
print, printopt
printdlg
printpreview
saveas

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

Graphics Object Identification (p. 1-100)  Find and manipulate graphics objects via their handles
Object Creation (p. 1-101)  Constructors for core graphics objects
Plot Objects (p. 1-101)  Property descriptions for plot objects
Figure Windows (p. 1-102)  Control and save figures
Axes Operations (p. 1-103)  Operate on axes objects
Object Property Operations (p. 1-103)  Query, set, and link object properties

Graphics Object Identification

allchild  Find all children of specified objects
ancestor  Ancestor of graphics object
copyobj  Copy graphics objects and their descendants
delete  Remove files or graphics objects
findall  Find all graphics objects
findfigs  Find visible offscreen figures
findobj  Locate graphics objects with specific properties
gca  Current axes handle
gcbf  Handle of figure containing object whose callback is executing
gcbo  Handle of object whose callback is executing
gco  Handle of current object
get  Query Handle Graphics® object properties
ishandle          Determine whether input is valid
                Handle Graphics handle
propedit          Open Property Editor
set               Set Handle Graphics object properties

**Object Creation**

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

**Plot Objects**

Annotation Arrow Properties     Define annotation arrow properties
Annotation Doublearrow Properties Define annotation doublearrow properties
Annotation Ellipse Properties    Define annotation ellipse properties
Annotation Line Properties      Define annotation line properties
Annotation Rectangle Properties | Define annotation rectangle properties
Annotation Textarrow Properties | Define annotation textarrow properties
Annotation Textbox Properties | Define annotation textbox properties
Areaseries Properties | Define areaseries properties
Barseries Properties | Define barseries properties
Contourgroup Properties | Define contourgroup properties
Errorbarseries Properties | Define errorbarseries properties
Image Properties | Define image properties
Lineseries Properties | Define lineseries properties
Quivergroup Properties | Define quivergroup properties
Scattergroup Properties | Define scattergroup properties
Stairseries Properties | Define stairseries properties
Stemseries Properties | Define stemseries properties
Surfaceplot Properties | Define surfaceplot properties

**Figure Windows**

clf | Clear current figure window
close | Remove specified figure
closereq | Default figure close request function
drawnow | Flush event queue and update figure window
gcf | Current figure handle
hgload | Load Handle Graphics object hierarchy from file
hgsave | Save Handle Graphics object hierarchy to file
### Graphics

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newplot</td>
<td>Determine where to draw graphics objects</td>
</tr>
<tr>
<td>opengl</td>
<td>Control OpenGL® rendering</td>
</tr>
<tr>
<td>refresh</td>
<td>Redraw current figure</td>
</tr>
<tr>
<td>saveas</td>
<td>Save figure or Simulink block diagram using specified format</td>
</tr>
</tbody>
</table>

### Axes Operations

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

### Object Property Operations

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

<table>
<thead>
<tr>
<th>Function Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface and Mesh Plots (p. 1-104)</td>
<td>Plot matrices, visualize functions of two variables, specify colormap</td>
</tr>
<tr>
<td>View Control (p. 1-106)</td>
<td>Control the camera viewpoint, zooming, rotation, aspect ratio, set axis limits</td>
</tr>
<tr>
<td>Lighting (p. 1-108)</td>
<td>Add and control scene lighting</td>
</tr>
<tr>
<td>Transparency (p. 1-108)</td>
<td>Specify and control object transparency</td>
</tr>
<tr>
<td>Volume Visualization (p. 1-108)</td>
<td>Visualize gridded volume data</td>
</tr>
</tbody>
</table>

### Surface and Mesh Plots

<table>
<thead>
<tr>
<th>Function Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface and Mesh Creation (p. 1-104)</td>
<td>Visualizing gridded and triangulated data as lines and surfaces</td>
</tr>
<tr>
<td>Domain Generation (p. 1-105)</td>
<td>Gridding data and creating arrays</td>
</tr>
<tr>
<td>Color Operations (p. 1-105)</td>
<td>Specifying, converting, and manipulating color spaces, colormaps, colorbars, and backgrounds</td>
</tr>
</tbody>
</table>

### Surface and Mesh Creation

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trimesh</td>
<td>Triangular mesh plot</td>
</tr>
<tr>
<td>triplot</td>
<td>2-D triangular plot</td>
</tr>
<tr>
<td>trisurf</td>
<td>Triangular surface plot</td>
</tr>
</tbody>
</table>

### Domain Generation

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

### Color Operations

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

**View Control**

**Camera Viewpoint**

- **surfnorm**
  - Compute and display 3-D surface normals

- **whitebg**
  - Change axes background color

**Camera Viewpoint (p. 1-106)**

- Orbiting, dollying, pointing, rotating camera positions and setting fields of view

**Aspect Ratio and Axis Limits**

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

**Object Manipulation**

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

**Region of Interest**

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

**Camera Viewpoint**

- **camdolly**
  - Move camera position and target

- **cameratoolbar**
  - Control camera toolbar programmatically

- **camlookat**
  - Position camera to view object or group of objects

- **camorbit**
  - Rotate camera position around camera target

- **campan**
  - Rotate camera target around camera position

- **campos**
  - Set or query camera position

- **camproj**
  - Set or query projection type

- **camroll**
  - Rotate camera about view axis

- **camtarget**
  - Set or query location of camera target
3-D Visualization

**camup**
Set or query camera up vector

**camva**
Set or query camera view angle

**camzoom**
Zoom in and out on scene

**makehgtform**
Create 4-by-4 transform matrix

**view**
Viewpoint specification

**viewmtx**
View transformation matrices

---

**Aspect Ratio and Axis Limits**

**daspect**
Set or query axes data aspect ratio

**pbaspect**
Set or query plot box aspect ratio

**xlim, ylim, zlim**
Set or query axis limits

---

**Object Manipulation**

**pan**
Pan view of graph interactively

**reset**
Reset graphics object properties to their defaults

**rotate**
Rotate object in specified direction

**rotate3d**
Rotate 3-D view using mouse

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

**zoom**
Turn zooming on or off or magnify by factor

---

**Region of Interest**

**dragrect**
Drag rectangles with mouse

**rbbox**
Create rubberband box for area selection
**Lighting**

- **camlight**
  - Create or move light object in camera coordinates

- **diffuse**
  - Calculate diffuse reflectance

- **light**
  - Create light object

- **lightangle**
  - Create or position light object in spherical coordinates

- **lighting**
  - Specify lighting algorithm

- **material**
  - Control reflectance properties of surfaces and patches

- **specular**
  - Calculate specular reflectance

**Transparency**

- **alim**
  - Set or query axes alpha limits

- **alpha**
  - Set transparency properties for objects in current axes

- **alphamap**
  - Specify figure alphamap (transparency)

**Volume Visualization**

- **coneplot**
  - Plot velocity vectors as cones in 3-D vector field

- **contourslice**
  - Draw contours in volume slice planes

- **curl**
  - Compute curl and angular velocity of vector field

- **divergence**
  - Compute divergence of vector field

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

Predefined Dialog Boxes (p. 1-110) Dialog boxes for error, user input, waiting, etc.
User Interface Deployment (p. 1-111) Open GUIs, create the handles structure
User Interface Development (p. 1-111) Start GUIDE, manage application data, get user input
User Interface Objects (p. 1-112) Create GUI components
Objects from Callbacks (p. 1-113) Find object handles from within callbacks functions
GUI Utilities (p. 1-113) Move objects, wrap text
Program Execution (p. 1-114) Wait and resume based on user input

Predefined Dialog Boxes

dialog Create and display empty dialog box
erroldlg Create and open error dialog box
export2wsdlg Export variables to workspace
helpdlg Create and open help dialog box
inputdlg Create and open input dialog box
listdlg Create and open list-selection dialog box

msgbox Create and open message box
printdlg Print dialog box
printpreview Preview figure to print
questdlg Create and open question dialog box
uigetdir Open standard dialog box for selecting directory
**GUI Development**

- `uigetfile` Open standard dialog box for retrieving files
- `uigetpref` Open dialog box for retrieving preferences
- `uiopen` Open file selection dialog box with appropriate file filters
- `uiputfile` Open standard dialog box for saving files
- `uisave` Open standard dialog box for saving workspace variables
- `uisetcolor` Open standard dialog box for setting object’s `ColorSpec`
- `uisetfont` Open standard dialog box for setting object’s font characteristics
- `waitbar` Open or update a wait bar dialog box
- `warndlg` Open warning dialog box

**User Interface Deployment**

- `guidata` Store or retrieve GUI data
- `guihandles` Create structure of handles
- `movegui` Move GUI figure to specified location on screen
- `openfig` Open new copy or raise existing copy of saved figure

**User Interface Development**

- `addpref` Add preference
- `getappdata` Value of application-defined data
- `getpref` Preference
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ginput</td>
<td>Graphical input from mouse or cursor</td>
</tr>
<tr>
<td>guidata</td>
<td>Store or retrieve GUI data</td>
</tr>
<tr>
<td>guide</td>
<td>Open GUI Layout Editor</td>
</tr>
<tr>
<td>inspect</td>
<td>Open Property Inspector</td>
</tr>
<tr>
<td>isappdata</td>
<td>True if application-defined data exists</td>
</tr>
<tr>
<td>ispref</td>
<td>Test for existence of preference</td>
</tr>
<tr>
<td>rmappdata</td>
<td>Remove application-defined data</td>
</tr>
<tr>
<td>rmpref</td>
<td>Remove preference</td>
</tr>
<tr>
<td>setappdata</td>
<td>Specify application-defined data</td>
</tr>
<tr>
<td>setpref</td>
<td>Set preference</td>
</tr>
<tr>
<td>uigetpref</td>
<td>Open dialog box for retrieving preferences</td>
</tr>
<tr>
<td>uisetpref</td>
<td>Manage preferences used in uigetpref</td>
</tr>
<tr>
<td>waitfor</td>
<td>Wait for condition before resuming execution</td>
</tr>
<tr>
<td>waitforbuttonpress</td>
<td>Wait for key press or mouse-button click</td>
</tr>
</tbody>
</table>

**User Interface Objects**

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menu</td>
<td>Generate menu of choices for user input</td>
</tr>
<tr>
<td>uibuttongroup</td>
<td>Create container object to exclusively manage radio buttons and toggle buttons</td>
</tr>
<tr>
<td>uicontextmenu</td>
<td>Create context menu</td>
</tr>
<tr>
<td>uicontrol</td>
<td>Create user interface control object</td>
</tr>
</tbody>
</table>
GUI Development

**Objects from Callbacks**

- `uimenu` Create menus on figure windows
- `uipanel` Create panel container object
- `uipushtool` Create push button on toolbar
- `uitable` Create 2-D graphic table GUI component
- `uitoggletool` Create toggle button on toolbar
- `uicontrol` Create toolbar on figure

- `findall` Find all graphics objects
- `findfigs` Find visible offscreen figures
- `findobj` Locate graphics objects with specific properties
- `gcbf` Handle of figure containing object whose callback is executing
- `gcbo` Handle of object whose callback is executing

**GUI Utilities**

- `getpixelposition` Get component position in pixels
- `listfonts` List available system fonts
- `selectmoveresize` Select, move, resize, or copy axes and uicontrol graphics objects
- `setpixelposition` Set component position in pixels
- `textwrap` Wrapped string matrix for given uicontrol
- `uistack` Reorder visual stacking order of objects
Program Execution

uiresume  Resume execution of blocked M-file
uiwait    Block execution and wait for resume
External Interfaces

Shared Libraries (p. 1-115)
Access functions stored in external shared library files

Java (p. 1-116)
Work with objects constructed from Java API and third-party class packages

.NET (p. 1-117)
Work with objects constructed from .NET assemblies

Component Object Model and ActiveX (p. 1-117)
Integrate COM components into your application

Web Services (p. 1-120)
Communicate between applications over a network using SOAP and WSDL

Serial Port Devices (p. 1-120)
Read and write to devices connected to your computer's serial port

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

Shared Libraries

calllib
Call function in shared library

libfunctions
Return information on functions in shared library

libfunctionsview
View functions in shared library

libisloaded
Determine if shared library is loaded

libpointer
Create pointer object for use with shared libraries

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

**Java**

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

<table>
<thead>
<tr>
<th>Method/Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>methodsview</td>
<td>Information on class methods in separate window</td>
</tr>
<tr>
<td>usejava</td>
<td>Determine whether Sun Java feature is supported in MATLAB software</td>
</tr>
</tbody>
</table>

### .NET

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

### Component Object Model and ActiveX

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

**Web Services**

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

**Serial Port Devices**

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

Arithmetic Operators + - * / \ ^ '
Relational Operators < > <= >= == ~=
Logical Operators: Elementwise & | ~
Logical Operators: Short-circuit && ||
Special Characters [ ] ( ) {} = ’ . ... , ; : % ! @
colon (:)
abs
accumarray
acos
acosd
acosh
acot
acotd
acoth
acsc
acscd
acsch
actxcontrol
actxcontrollist
actxcontrolselect
actxGetRunningServer
actxserver
addCause (MException)
addevent
addframe
addlistener (handle)
addOptional (inputParser)
addParamValue (inputParser)
addpath
addpref
addprop (dynamicprops)
addproperty
addRequired (inputParser)
addsample
addsampletocollection
addtodate
addtods
airy
align
align
alim
all
allchild
alpha
alphamap
amd
ancestor
and
angle
annotation
Annotation Arrow Properties
Annotation Doublearrow Properties
Annotation Ellipse Properties
Annotation Line Properties
Annotation Rectangle Properties
Annotation Textarrow Properties
Annotation Textbox Properties
ans
any
area
Areaseries Properties
arrayfun
ascii
asec
asecd
asech
asin
asind
asinh
assert
assignin
atan
atan2
atand
atanh
audiodevinfo
audioplayer
audiorecorder
aufinfo
auread
auwrite
avifile
aviinfo
aviread
axes
Axes Properties
axis
balance
bar, barh
bar3, bar3h
BarsSeries Properties
baryToCart
base2dec
beep
bench
besselh
besseli
besselj
besselk
bessely
beta
betainc
betaincinv
betaln
bicg
bicgstab
bicgstabl
bin2dec
binary
bitand
bitcmp
bitget
bitmax
bitor
bitset
bitshift
bitxor
blanks
blkdiag
box
break
brighten
brush
bsxfun
builddocsearchdb
builtin
bvp4c
bvp5c
bvpget
bvpinit
bvpset
bvpxtend
calendar
calllib
callSoapService
camdolly
cameratoolbar
camlight
camlookat
camorbit
campan
campos
camproj
camroll
camtarget
camup
camva
camzoom
cartToBary
cart2pol
cart2sph
case
cast
cat
catch
caxis
cd
convexHull
cd (ftp)
cdf2rdf
cdfepoch
cdfinfo
cdfread
cdfwrite
ceil
cell
cell2mat
cell2struct
celldisp
cellfun
cellplot
cellstr
cgs
char
checkin
checkout
chol
cholinc
cholupdate
circshift
circumcenters
cla
clabel
class
classdef
clc
clear
clearvars
serial.clear
clf
clipboard
clock
close
close (avifile)
close (ftp)
closereq
cmopts
cmpermute
cmunique
colamd
colorbar
colordef
colormap
colormapeditor
ColorSpec (Color Specification)
colperm
comet
comet3
commandhistory
commandwindow
compan
compass
complex
computer
cond
condeig
condest
coneplot
conj
continue
contour
contour3
contourc
contourf
Contourgroup Properties
contourslice
contrast
conv
conv2
convhull
convhulln
convn
copyfile
copyobj
corrcof
cos
cosd
cosh
cot
cotd
coth
cov
cplxpair
cputime
cputime
create (RandStream)
createClassFromWsdld
createCopy (inputParser)
createSoapMessage
cross
csc
cscd
csc 
csvread 
csvwrite 
ctranspose (timeseries) 
cumprod 
cumsum 
cumtrapz 
curl 
customverctrl 
cylinder 
daqread 
daspect 
datacursormode 
datatipinfo 
date 
datenum 
datestr 
datetick 
datevec 
dbclear 
dbcont 
dbdown 
dblquad 
dbmex 
dbquit 
dbstack 
dbstatus 
dbstep 
dbstop 
dbtype 
 dbup 
dde23 
ddeget 
ddesd 
ddeset 
deal 
debblank
debug
dec2base
dec2bin
dec2hex
decic
decconv
del2
DelaunayTri
DelaunayTri
delaunay
delaunay3
delaunayn
delete
delete (COM)
delete (ftp)
delete (handle)
serial.delete
delete (timer)
deleteproperty
delevent
delsample
delsamplefromcollection
demo
depdir
depfun
det
detrend
detrend (timeseries)
deval
diag
dialog
diary
diff
diffuse
dir
dir (ftp)
disp
disp (memmapfile)
disp (MException)
serial.disp
disp (timer)
display
dither
divergence
dlmtree
dlmsread
dlmswrite
dmperm
doc
doct
docsearch
dos
dot
double
dragread
drawnow
dsearch
dsearchn
dynamicprops
echo
echomenu
egeAttachments
edgesedit
eig
eigs
ellipj
ellipke
ellipsoid
else
elseif
enableservice
end
eomday
eps
eq
eq (MException)
erf, erfc, erfcx, erfinv, erfcinv
error
errorbar
Errorbarseries Properties
errordlg
etime
etree
etreeplot
eval
evalc
evalin
event.EventData
event.PropertyEvent
event.listener
event.proplistener
eventlisteners
events
events (COM)
Execute
exifread
exist
exit
exp
expint
expm
expm1
export2wsdlg
eye
ezcontour
ezcontourf
ezmesh
ezmeshc
ezplot
ezplot3
ezpolar
ezsurf
ezsurfc
faceNormals
factor
factorial
false
fclose
serial fclose
feather
featureEdges
feof
ferror
feval
feval (COM)
fft
fft2
fftn
fftshift
fftw
fgetl
serial.fgetl
fgets
serial.fgets
fieldnames
figure
Figure Properties
figurepalette
fileattrib
filebrowser
File Formats
filemarker
fileparts
fileread
filesep
fill
fill3
filter
filter (timeseries)
filter2
find
findall
findfigs
findobj
findobj (handle)
findprop (handle)
findstr
finish
fitsinfo
fitsread
fix
fliptim
fliplr
flipud
floor
flow
fminbnd
fminsearch
fopen
serial.fopen
for
format
fplot
fprintf
serialfprintf
frame2im
fread
serial.fread
freeBoundary
freqspace
frewind
fscanf
serial.fscanf
fseek
ftell
ftp
full
fullfile
func2str
function
function_handle (@)
functions
funm
fwrite
serial.fwrite
fzero
gallery
gamma, gammainc, gammainv
gammaincinv
gca
gcbf
gcbo
gcd
gcf
gco
ge
genpath
genvarname
get
get (COM)
get (hgsetget)
get (memmapfile)
get (RandStream)
serial.get
get (timer)
get (timeseries)
get (tscollection)
getabstime (timeseries)
getabstime (tscollection)
getappdata
GetCharArray
getdatasamplesize
getDefaultStream (RandStream)
getdisp (hgsetget)
getenv
getfield
getframe
GetFullMatrix
getinterpmethod
getpixelposition
getpref
getqualitydesc
getReport (MException)
getsampleusingtime (timeseries)
getsampleusingtime (tscollection)
gettimeseriesnames
gettsafteratevent
gettsafterevent
gettsatevent
gettsbeforeatevent
gettsbeforeevent
gettsbetweenevents
GetVariable
GetWorkspaceData
ginput
global
gmres
gplot
grabcode
gradient
graymon
grid
griddata
griddata3
griddatan
gsvd
gt
gtext
guida
guide
guihandles
gunzip
gzip
hadamard
handle
hankel
hdf
hdf5
hdf5info
hdf5read
hdf5write
hdfinfo
hdfread
hdftool
help
helpbrowser
helpdesk
helpdlg
helpwin
hess
hex2dec
hex2num
hgexport
hgroup
Hggroup Properties
hgload
hgsave
hgsetget
hgtransform
Hgtransform Properties
hidden
hilb
hist
histc
hold
home
horzcat
horzcat (tscollection)
hostid
hsv2rgb
hypot
i
idealfilter (timeseries)
idivide
if
ifft
ifft2
ifftn
ifftshift
ilu
im2frame
im2java
imag
image
Image Properties
imagesc
imapprox
imfinfo
imformats
import
importdata
imread
imwrite
incenters
inOutStatus
ind2rgb
ind2sub
Inf
inferior
info
inline
inmem
inpolygon
input
inputdlg
inputname
inputParser
inspect
instrcallback
instrfind
instrfindall
int2str
int8, int16, int32, int64
interfaces
interp1
interp1q
interp2
interp3
interpft
interpnt
interpstreamspeed
intersect
intmax
intmin
intwarning
inv
invhilb
invoke
ipermute
iqr (timeseries)
is*
is
isa
isappdata
iscell
iscellstr
ischar
iscom
isdir
isEdge
isempty
isempty (timeseries)
isempty (tscollection)
isequal
isequal (MException)
isequalwiththeequalnans
isevent
isfield
isfinite
isfloat
isglobal
ishandle
ishold
isinf
isinteger
isinterface
isjava
isKey (Map)
iskeyword
isletter
islogical
ismac
ismember
ismethod
isnan
isnumeric
isobject
isocaps
isocolors
isonormals
isosurface
ispc
ispref
isprime
isprop
isreal
isscalar
issorted
isspace
issparse
isstr
isstrprop
isstruct
isstudent
isunix
isvalid (handle)
serial.isvalid
isvalid (timer)
isvarname
isvector
j
javaaddpath
javaArray
javachk
javaclasspath
javaMethod
javaMethodEDT
javaObject
javaObjectEDT
javaRmpath
keyboard
keys (Map)
kron
last (MException)
lasterr
lasterror
lastwarn
lcm
ldl
ldivide, rdivide
le
legend
legendre
length
length (Map)
serial.length
length (timeseries)
length (tscollection)
libfunctions
libfunctionsview
libisloaded
libpointer
libstruct
license
light
Light Properties
lightangle
lighting
lin2mu
line
Line Properties
Lineseries Properties
LineSpec (Line Specification)
linkaxes
linkdata
linkprop
linsolve
linspace
list (RandStream)
listdlg
listfonts
load
load (COM)
serial.load
loadlibrary
loadobj
log
log10
log1p
log2
logical
loglog
logm
logspace
lookfor
lower
ls
ls cov
lsq nonneg
lsqr
lt
lu
lu inc
magic
make hgt form
containers.Map
mat2cell
mat2str
material
matlab colon (matlab:)
matlabrc
matlab root
matlab (UNIX)
matlab (Windows)
max
max (timeseries)
Maximize Command Window
max Num Comp Threads
mean
mean (timeseries)
median
median (timeseries)
mem map file
memory
menu
mesh, meshc, meshz
mesh grid
meta.class
meta.class. from Name
meta.DynamicProperty
meta.event
meta.method
meta.package
meta.package.fromName
meta.package.getAllPackages
meta.property
metaclass
methods
methodsview
mex
mex.getCompilerConfigurations
MException
mexext
mfilename
mget
min
min (timeseries)
MinimizeCommandWindow
minres
mislocked
mkdir
mkdir (ftp)
mkpp
mldivide \, mrdive /
mlint
mlintrpt
mlock
mmfileinfo
mmreader
mmreader.isPlatformSupported
mod
mode
more
move
movefile
movegui
movie
movie2avi
mput
msgbox
mtimes
mu2lin
multibandread
multibandwrite
munlock
namelengthmax
NaN
nargchk
nargin, nargout
nargoutchk
native2unicode
nchoosek
ndgrid
ndims
ne
nearestNeighbor
ne (MException)
neighbors
NET
NET.addAssembly
NET.convertArray
NET.createArray
NET.createGeneric
NET.GenericClass
NET.GenericClass
netcdf
netcdf.abort
netcdf.close
netcdf.copyAtt
netcdf.create
netcdf.defDim
netcdf.defVar
netcdf.delAtt
num2cell
num2hex
num2str
numel
nzmax
ode15i
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
odefile
odeget
odeset
odextend
onCleanup
ones
open
openfig
opengl
openvar
optimget
optimset
or
ordeig
orderfields
ordqz
ordschur
orient
orth
otherwise
pack
padecoef
pagesetupdlg
pan
pareto
parfor
parse (inputParser)
parseSoapResponse
partialpath
pascal
patch
Patch Properties
path
path2rc
pathsep
pathtool
pause
pbaspect
pcg
pchip
pcode
pcolor
pdepe
pdeval
peaks
perl
perms
permute
persistent
pi
pie
pie3
pinv
planerot
playshow
plot
plot (timeseries)
plot3
plotbrowser
plotedit
plotmatrix
plottools
plotyy
pointLocation
pol2cart
polar
poly
polyarea
polyder
polyeig
polyfit
polyint
polyval
polyvalm
pow2
power
ppval
prefdir
preferences
primes
print, printopt
printdlg
printpreview
prod
profile
profsave
propedit
propedit (COM)
properties
propertyeditor
psi
publish
PutCharArray
PutFullMatrix
PutWorkspaceData
pwd
qmr
qr
qrdelete
qrinsert
qrupdate
quad
quad2d
quadgk
reducevolume
refresh
refreshdata
regexp, regexpi
regexprep
regextranslate
registerevent
rehash
release
relationaloperators (handle)
rem
remove (Map)
removets
rename
repmat
resample (timeseries)
resample (tscollection)
reset
reset (RandStream)
reshape
residue
restoredefaultpath
rethrow
rethrow (MException)
return
rgb2hsv
rgb2ind
rgbplot
ribbon
rmappdata
rmdir
rmdir (ftp)
rmfield
rmpath
rmpref
root object
Root Properties
roots
rose
rosser
rot90
rotate
rotate3d
round
rref
rsf2csf
run
save
save (COM)
serial.save
saveas
saveobj
savepath
scatter
scatter3
Scattergroup Properties
schur
script
sec
secd
sech
selectmoveresize
semilogx, semilogy
sendmail
serial
serialbreak
set
set (COM)
set (hgsetget)
set (RandStream)
serial.set
set (timer)
set (timeseries)
set (tscollection)
setabstime (timeseries)
setabstime (tscollection)
setapppdata
setDefaultStream (RandStream)
setdiff
setdisp (hgsetget)
setenv
setfield
setinterpmethod
setpixelposition
setpref
setstr
settimeseriesnames
setxor
shading
shg
shiftdim
showplottool
shrinkfaces
sign
sin
sind
single
sinh
size
size (Map)
serial.size
size (timeseries)
size
size (tscollection)
slice
smooth3
snapnow
sort
sortrows
sound
soundsc
spalloc
sparse
spaugment
spconvert
spdiags
specular
speye
spfun
sph2cart
sphere
spinmap
spline
spones
spparms
sprand
sprandn
sprandsym
sprank
sprintf
spy
sqrt
sqrtm
squeeze
ss2tf
sscanf
stairs
Stairseries Properties
start
startat
startup
std
std (timeseries)
stem
stem3
Stemseries Properties
stop
stopasync
str2double
str2func
str2mat
str2num
strcat
strcmp, strcmpi
stream2
stream3
streamline
streamparticles
streamribbon
streamslice
streamtube
strfind
strings
strjust
strmatch
strncmp, strncmpi
strread
strrep
strtok
strtrim
struct
struct2cell
structfun
strvcat
sub2ind
subplot
subsasgn
subsindex
subspace
subsref
substrct
subvolume
sum
sum (timeseries)
superiorto
support
surf, surfc
surf2patch
surface
Surface Properties
Surfaceplot Properties
surfl
surf
surf
svd
svds
swapbytes
switch
symamd
symbfact
symmlq
symrcm
symvar
synchronize
syntax
system	an
tand
tanh
tar
tempdir
tempname
tetramesh
texlabel
text
Text Properties
textread
textscan
textran
tfqmr
throw (MException)
throwAsCaller (MException)
tic, toc
timer
timerfind
timerfindall
timeseries
title
todatenum
toeplitz
toolboxdir
trace
transpose (timeseries)
trapz
treelayout
treeplot
tril
trimesh
triplequad
triplot
TriRep
TriRep
TriScatteredInterp
TriScatteredInterp
trisurf
triu
true
try
tscollection
tsdale.data.event
tsearch
tsearchn
tsprops
tstool
type
typecast
uibuttongroup
Uibuttongroup Properties
uicontextmenu
Uicontextmenu Properties
uicontrol
Uicontrol Properties
uigetdir
uigetfile
uigetpref
uiimport
uimenu
Uimenu Properties
uint8, uint16, uint32, uint64
uiopen
uipanel
Uipanel Properties
uipushtool
Uipushtool Properties
uiputfile
uiresume
uisave
uisetcolor
uisetfont
uisetpref
uistack
uitable
Uitable Properties
uitoggletool
Uitoggletool Properties
uitoolbar
Uitoolbar Properties
uiwait
undocheckout
unicode2native
union
unique
unix
unloadlibrary
unmesh
unmkpp
unregisterallevents
unregisterevent
untar
unwrap
unzip
upper
urllread
urllwrite
usejava
userpath
validateattributes
valideatestring
values (Map)
vander
var
var (timeseries)
varargin
varargout
vectorize
ver
verctrl
verLessThan
version
vertcat
vertcat (timeseries)
vertcat (tscollection)
vertexAttachments
view
viewmtx
visdiff
volumebounds
voronoi
voronoiDiagram
voronoin
wait
waitbar
waitfor
waitforbuttonpress
warndlg  
warning  
waterfall  
wavinfo  
wavplay  
wavread  
wavrecord  
wavwrite  
weekday  
what  
whatsnew  
which  
while  
whitebg  
who, whos  
wilkinson  
winopen  
winqueryreg  
wk1info  
wk1read  
wk1write  
workspace  
xlabel, ylabel, zlabel  
xlim, ylim, zlim  
xlsinfo  
xlsread  
xlswrite  
xmlread  
xmlwrite  
xor  
xslt  
zeros  
zip  
zoom
**Purpose**
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 software 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.

Slash or matrix right division. \( B / A \) is roughly the same as \( B * \text{inv}(A) \). More precisely, \( B / A = (A' \backslash B')' \). See the reference page for \text{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, \( A \backslash B \) is roughly the same as \( \text{inv}(A) * B \), 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 \( X = A \backslash B \) is the solution to the equation \( AX = B \). A warning message is displayed if \( A \) is badly scaled or nearly singular. See the reference page for \text{mldivide} for more information.
If $A$ is an $m$-by-$n$ matrix with $m \neq n$ and $B$ is a column vector with $m$ components, or a matrix with several such columns, then $X = A \backslash B$ 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 2-2407 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 $\text{pinv}(A) \ast B$, which is the least squares solution with the smallest norm $\|X\|$.

\texttt{\backslash}

Array left division. $A \backslash B$ 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.

\texttt{^}

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 $[V,D] = \text{eig}(X)$, then $X^p = V \ast D \ast p \ast V$.

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.

\texttt{.^}

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.

\texttt{'}

Matrix transpose. $A'$ is the linear algebraic transpose of $A$. For complex matrices, this is the complex conjugate transpose.

\texttt{.'}

Array transpose. $A.'$ is the array transpose of $A$. For complex matrices, this does not involve conjugation.
Nondouble Data Type Support

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 software 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,

\[
x = \text{int8}(3) + \text{int8}(4);
class(x)
\]

\[
\text{ans} = \text{int8}
\]
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.

<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>
<tr>
<td>A.', A'</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**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,

```matlab
y = 5 + 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

The section “Numeric Classes”, under “Classes (Data Types)” in the MATLAB Programming Fundamentals documentation, provides more information about operations on nondouble data types.

**Remarks**

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary addition</td>
<td>A+B</td>
</tr>
<tr>
<td>Unary plus</td>
<td>+A</td>
</tr>
<tr>
<td>Binary subtraction</td>
<td>A-B</td>
</tr>
<tr>
<td>Unary minus</td>
<td>-A</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>A*B</td>
</tr>
<tr>
<td>Arraywise multiplication</td>
<td>A.*B</td>
</tr>
<tr>
<td>Matrix right division</td>
<td>A/B</td>
</tr>
<tr>
<td>Arraywise right division</td>
<td>A./B</td>
</tr>
<tr>
<td>Matrix left division</td>
<td>A\B</td>
</tr>
<tr>
<td>Arraywise left division</td>
<td>A\B</td>
</tr>
</tbody>
</table>

(*A*, *B* being arrays of type `int32`)
### Arithmetic Operators + - * / \ ^ '

<table>
<thead>
<tr>
<th>Matrix power</th>
<th>A^B</th>
<th>mpower(A,B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arraywise power</td>
<td>A.^B</td>
<td>power(A,B)</td>
</tr>
<tr>
<td>Complex transpose</td>
<td>A'</td>
<td>ctranspose(A)</td>
</tr>
<tr>
<td>Matrix transpose</td>
<td>A.'</td>
<td>transpose(A)</td>
</tr>
</tbody>
</table>

**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><code>x</code></td>
<td><code>y</code></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><code>x'</code></td>
<td><code>y'</code></td>
</tr>
<tr>
<td>1 2 3</td>
<td>4 5 6</td>
</tr>
<tr>
<td><code>x+y</code></td>
<td><code>x-y</code></td>
</tr>
<tr>
<td>5 7 9</td>
<td>-3 -3 -3</td>
</tr>
<tr>
<td><code>x + 2</code></td>
<td><code>x-2</code></td>
</tr>
<tr>
<td>3 4 5</td>
<td>-1 0 1</td>
</tr>
<tr>
<td>Matrix Operations</td>
<td>Array Operations</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>$x \times y$</td>
<td>Error</td>
</tr>
<tr>
<td>$x' \times y$</td>
<td>32</td>
</tr>
<tr>
<td>$x \times y'$</td>
<td>4 5 6</td>
</tr>
<tr>
<td>$x \times 2$</td>
<td>2 4 6</td>
</tr>
<tr>
<td>$x \div y$</td>
<td>16/7</td>
</tr>
<tr>
<td>$2 \div x$</td>
<td>1/2 1 3/2</td>
</tr>
<tr>
<td>$x \div y$</td>
<td>0 0 1/6 0 0 1/3</td>
</tr>
<tr>
<td>$x \div 2$</td>
<td>1/2 1 3/2</td>
</tr>
<tr>
<td>$x.\times y$</td>
<td>4 10 18</td>
</tr>
<tr>
<td>$x'.\times y$</td>
<td>Error</td>
</tr>
<tr>
<td>$x.\times y'$</td>
<td>Error</td>
</tr>
<tr>
<td>$x.\times 2$</td>
<td>2 4 6</td>
</tr>
<tr>
<td>$x.\div y$</td>
<td>4 5/2 2</td>
</tr>
<tr>
<td>$2.\div x$</td>
<td>2 1 2/3</td>
</tr>
<tr>
<td>$x.\div y$</td>
<td>1/4 2/5 1/2</td>
</tr>
<tr>
<td>$x.\div 2$</td>
<td>1/2 1 3/2</td>
</tr>
<tr>
<td>Matrix Operations</td>
<td>Array Operations</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>$x^y$</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>$x.^y$</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>729</td>
</tr>
<tr>
<td>$x^2$</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>$x.^2$</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td>$2^x$</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>$2.^x$</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>$(x+i*y)'$</td>
<td>$1 - 4i$</td>
</tr>
<tr>
<td></td>
<td>$2 - 5i$</td>
</tr>
<tr>
<td></td>
<td>$3 - 6i$</td>
</tr>
<tr>
<td>$(x+i*y).'$</td>
<td>$1 + 4i$</td>
</tr>
<tr>
<td></td>
<td>$2 + 5i$</td>
</tr>
<tr>
<td></td>
<td>$3 + 6i$</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.

- 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

References


### 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 logical 1 (true) where the relation is true, and elements set to logical 0 (false) 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

```plaintext
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:
Relational Operators < > <= >= == ~=

ans =

1 1 1
1 1 0
0 0 0

See Also all, any, find, strcmp

Logical Operators: Elementwise & | ~, Logical Operators: Short-circuit && ||
**Logical Operators: Elementwise & | ~**

**Purpose**
Elementwise logical operations on arrays

**Syntax**
expr1 & expr2
expr1 | expr2
~expr

**Description**
The symbols &, |, and ~ are the logical array operators AND, OR, and NOT. These operators are commonly used in conditional statements, such as if and while, to determine whether or not to execute a particular block of code. Logical operations return a logical array with elements set to 1 (true) or 0 (false), as appropriate.

expr1 & expr2 represents a logical AND operation between values, arrays, or expressions expr1 and expr2. In an AND operation, if expr1 is true and expr2 is true, then the AND of those inputs is true. If either expression is false, the result is false. Here is a pseudocode example of AND:

```
IF (expr1: all required inputs were passed) AND ...
    (expr2: all inputs are valid)
THEN (result: execute the function)
```

expr1 | expr2 represents a logical OR operation between values, arrays, or expressions expr1 and expr2. In an OR operation, if expr1 is true or expr2 is true, then the OR of those inputs is true. If both expressions are false, the result is false. Here is a pseudocode example of OR:

```
IF (expr1: S is a string) OR ...
    (expr2: S is a cell array of strings)
THEN (result: parse string S)
```

~expr represents a logical NOT operation applied to expression expr. In a NOT operation, if expr is false, then the result of the operation is true. If expr is true, the result is false. Here is a pseudocode example of NOT:

```
IF (expr: function returned a Success status) is NOT true
Logical Operators: Elementwise & | ~

THEN (result: throw an error)

The function `xor(A,B)` implements the exclusive OR operation.

Logical Operations on Arrays

The expression operands for AND, OR, and NOT are often arrays of nonsingleton dimensions. When this is the case, The MATLAB software performs the logical operation on each element of the arrays. The output is an array that is the same size as the input array or arrays.

If just one operand is an array and the other a scalar, then the scalar is matched against each element of the array. When the operands include two or more nonscalar arrays, the sizes of those arrays must be equal.

This table shows the output of AND, OR, and NOT statements that use scalar and/or array inputs. In the table, S is a scalar array, A is a nonscalar array, and R is the resulting array:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1 &amp; S2</td>
<td>R = S1 &amp; S2</td>
</tr>
<tr>
<td>S &amp; A</td>
<td>R(1) = S &amp; A(1); ...</td>
</tr>
<tr>
<td></td>
<td>R(2) = S &amp; A(2); ...</td>
</tr>
<tr>
<td>A1 &amp; A2</td>
<td>R(1) = A1(1) &amp; A2(1);</td>
</tr>
<tr>
<td></td>
<td>R(2) = A1(2) &amp; A2(2); ...</td>
</tr>
<tr>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>S</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>R(2) = S</td>
</tr>
<tr>
<td>A1</td>
<td>A2</td>
</tr>
<tr>
<td></td>
<td>R(2) = A1(2)</td>
</tr>
<tr>
<td>~S</td>
<td>R = ~S</td>
</tr>
<tr>
<td>~A</td>
<td>R(1) = ~A(1);</td>
</tr>
<tr>
<td></td>
<td>R(2) = ~A(2), ...</td>
</tr>
</tbody>
</table>
Logical Operators: Elementwise & | ~

Compound Logical Statements

The number of expressions that you can evaluate with AND or OR is not limited to two (e.g., A & B). Statements such as the following are also valid:

\[
eexpr1 \& eexpr2 \& eexpr3 \mid eexpr4 \& eexpr5
\]

Use parentheses to establish the order in which MATLAB evaluates a compound operation. Note the difference in the following two statements:

\[
\text{(expr1 \& expr2) \mid (expr3 \& expr4)} \quad \% \text{ 2-component OR}
\]

\[
eexpr1 \& \text{(expr2 \mid expr3)} \& eexpr4 \quad \% \text{ 3-component AND}
\]

Operator Precedence

The precedence for the logical operators with respect to each other is shown in the table below. 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 |.

<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>

Short-Circuiting in Elementwise Operators

The \&, and | operators do not short-circuit. See the documentation on the && and || operators if you need short-circuiting capability.

When used in the context of an if or while expression, and only in this context, the elementwise \& and | operators use short-circuiting in
evaluating their expressions. That is, $A \& B$ and $A \mid B$ ignore the second operand, $B$, if the first operand, $A$, is sufficient to determine the result.

So, although the statement $1 \mid []$ evaluates to false, the same statement evaluates to true when used in either an if or while expression:

```matlab
A = 1; B = []; if(A|B) disp 'The statement is true', end;
The statement is true
```

while the reverse logical expression, which does not short-circuit, evaluates to false:

```matlab
if(B|A) disp 'The statement is true', end;
```

Another example of short-circuiting with elementwise operators shows that a logical expression such as the one shown below, which under most circumstances is invalid due to a size mismatch between $A$ and $B$, works within the context of an if or while expression:

The $A \mid B$ statement generates an error:

```matlab
A = [1 1]; B = [2 0 1]; A\mid B
??? Error using ==> or
Matrix dimensions must agree.
```

But the same statement used to test an if condition does not error:

```matlab
if (A\mid B) disp 'The statement is true', end;
The statement is true
```

**Operator Truth Table**

The following is a truth table for the operators and functions in the previous example.
Logical Operators: Elementwise & | ~

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

Equivalent Functions

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

<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

Example 1 — Conditional Statement with OR

Using OR in a conditional statement, call function parseString on S, but only if S is a character array or a cell array of strings:

```matlab
if ischar(S) || iscellstr(S)
    parseString(S)
end
```

Example 2 — Array AND Array

Find those elements of array R that are both greater than 0.3 AND less than 0.9:

```matlab
rand('state',0);
R=rand(5,7);
R>0.3 & R<0.9
```
Logical Operators: Elementwise & | ~

Example 3 — Array AND Scalar

Find those elements of array R that are greater than or equal to 25 AND are less than or equal to 50:

```
rand('state',0);
R = rand(3,5) * 50;
R > 40
```

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

Example 4 — Check Status with NOT

Throw an error if the return status of a function does NOT indicate success:

```
[Z, status] = myfun(X, Y);
if ~(status == SUCCESS);
    error('Error in function myfun')
end
```

Example 5 — OR of Binary Arrays

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

```
u = [0 0 1 1 0 1];
v = [0 1 1 0 0 1];
u | v
```
Logical Operators: Elementwise & | ~

\[
\begin{array}{cccccc}
0 & 1 & 1 & 1 & 0 & 1
\end{array}
\]

See Also
all, any, find, logical, xor, true, false
Logical Operators: Short-circuit && ||
Relational Operators < > <= >= == ~=
**Purpose**
Logical operations, with short-circuiting capability

**Syntax**
expr1 && expr2
expr1 || expr2

**Description**
expr1 && expr2 represents a logical AND operation that employs short-circuiting behavior. With short-circuiting, the second operand expr2 is evaluated only when the result is not fully determined by the first operand expr1. For example, if A = 0, then the following statement evaluates to false, regardless of the value of B, so the MATLAB software does not evaluate B:

A && B

These two expressions must each be a valid MATLAB statement that evaluates to a scalar logical result.

expr1 || expr2 represents a logical OR operation that employs short-circuiting behavior.

**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 < > <= >= == !=
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Special characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>[ ]</td>
</tr>
<tr>
<td></td>
<td>{ }</td>
</tr>
<tr>
<td></td>
<td>( )</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td></td>
<td>’</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.( )</td>
</tr>
<tr>
<td></td>
<td>..</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...,</td>
</tr>
<tr>
<td></td>
<td>;</td>
</tr>
<tr>
<td></td>
<td>;</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>%</td>
</tr>
<tr>
<td></td>
<td>%{ %}</td>
</tr>
<tr>
<td></td>
<td>!</td>
</tr>
<tr>
<td></td>
<td>@</td>
</tr>
</tbody>
</table>
Brackets are used to form vectors and matrices. \[6.9 \ 9.64 \ \text{sqrt}(-1)\] 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.

\([A_1,A_2,A_3...]=\text{function}\) 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 \(A{2,2} = \text{('str')}.\) 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 subscripting 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. $S(m).f$ when S is a structure, accesses the contents of field f of that structure.
. ( Dynamic Field access. \( S.(df) \) when \( A \) is a structure, accesses
) the contents of dynamic field \( df \) of that structure. Dynamic
field names are defined at runtime.

.. Parent directory. See \texttt{cd}.

... Continuation. Three or more periods at the end of a line
continue the current function on the next line. Three or more
periods before the end of a line cause the MATLAB software to
generate the remaining text on the current line and continue the
function on the next line. This effectively makes a comment
out of anything on the current line that follows the three
periods. See “Entering Multiple-Line (Long) Statements —
Line Continuation” for more information.

, Comma. Used to separate matrix subscripts and function
arguments. Used to separate statements in multistatement
lines. For multistatement lines, the comma can be replaced by
a semicolon to suppress printing.

; Semicolon. Used inside brackets to end rows. Used after an
expression or statement to suppress printing or to separate
statements.

: Colon. Create vectors, array subscripting, and \texttt{for} loop
iterations. See \texttt{colon (:) } for details.

% Percent. The percent symbol denotes a comment; it indicates
a logical end of line. Any following text is ignored. MATLAB
displays the first contiguous comment lines in a M-file in
response to a \texttt{help} command.

%{ Percent-brace. The text enclosed within the %\{ and %\} symbols
%} is a comment block. Use these symbols to insert comments that
take up more than a single line in your M-file code. Any text
between these two symbols is ignored by MATLAB.

With the exception of whitespace characters, the %\{ and %\} operators must appear alone on the lines that immediately
precede and follow the block of help text. Do not include any
other text on these lines.
Exclamation point. Indicates that the rest of the input line is issued as a command to the operating system. See “Running External Programs” for more information.

Function handle. MATLAB data type that is a handle to a function. See function_handle (@) for details.

Remarks

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>MATLAB Equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal concatenation</td>
<td>horzcat(A,B,C...)</td>
<td>See help subsref.</td>
</tr>
<tr>
<td>Vertical concatenation</td>
<td>vertcat(A,B,C...)</td>
<td></td>
</tr>
<tr>
<td>Subscript reference</td>
<td>subsref(A,S)</td>
<td>See help subsref.</td>
</tr>
<tr>
<td>Subscript assignment</td>
<td>subsasgn(A,S,B)</td>
<td></td>
</tr>
</tbody>
</table>

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 < > <= => == ~=
Logical Operators: Elementwise & | –,
**Purpose**

Create vectors, array subscripting, and for-loop iterators

**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 \(i = 0\), 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 \(A\)

\[ A(:, :) \]

is the equivalent two-dimensional array. For matrices this is the same as \(A\).

\[ A(j:k) \]

is \(A(j), A(j+1), \ldots, A(k)\)

\[ A(:,j:k) \]

is \(A(:,j), A(:,j+1), \ldots, A(:,k)\)

\[ A(:, :, k) \]

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 \(A\).
For more information on how the colon operator works, see http://www.mathworks.com/support/solutions/data/1-4FLI96.html?solution=1-4FLI96.html

**Examples**

Using the colon with integers,

\[ D = 1:4 \]

results in

\[ D = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \]

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

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

results in

\[ E = \begin{bmatrix} 0 & 0.1000 & 0.2000 & 0.3000 & 0.4000 & 0.5000 \end{bmatrix} \]

The command

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

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

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

\[ A(:, :, 2) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 6 \end{bmatrix} \]

Using a colon with characters to iterate a for-loop,
Colon (:)

for x='a':'d',x,end

results in

x =
  a
x =
  b
x =
  c
x =
  d

See Also for, linspace, logspace, reshape
**Purpose**

Absolute value and complex magnitude

**Syntax**

`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(real(X).^2 + imag(X).^2)
```

**Examples**

```matlab
abs(-5)
ans =
    5

abs(3+4i)
ans =
    5
```

**See Also**

`angle`, `sign`, `unwrap`
Purpose
Construct array with accumulation

Syntax
A = accumarray(subs,val)
A = accumarray(subs,val,sz)
A = accumarray(subs,val,sz,fun)
A = accumarray(subs,val,sz,fun,fillval)
A = accumarray(subs,val,sz,fun,fillval,issparse)
A = accumarray({subs1, subs2, ...}, val, ...)

Description
accumarray groups elements from a data set and applies a function to each group. \( A = \text{accumarray}(\text{subs}, \text{val}) \) creates an array \( A \) by accumulating elements of the vector \( \text{val} \) using the elements of \( \text{subs} \) as indices. The position of an element in \( \text{subs} \) determines which value of \( \text{vals} \) it selects for the accumulated vector; the value of an element in \( \text{subs} \) determines the position of the accumulated vector in the output.

\( A = \text{accumarray}(\text{subs}, \text{val}, \text{sz}) \) creates an array \( A \) with size \( \text{sz} \), where \( \text{sz} \) is a vector of positive integers. If \( \text{subs} \) is nonempty with \( N>1 \) columns, then \( \text{sz} \) must have \( N \) elements, where \( \text{all(sz} \geq \text{max(subs,}([],1)) \). If \( \text{subs} \) is a nonempty column vector, then \( \text{sz} \) must be \([M 1] \), where \( M \geq \text{MAX(subs)} \). Specify \( \text{sz} \) as [] for the default behavior.

\( A = \text{accumarray}(\text{subs}, \text{val}, \text{sz}, \text{fun}) \) applies function \( \text{fun} \) to each subset of elements of \( \text{val} \). The default accumulating function is \( \text{sum} \). To specify another function \( \text{fun} \), use the @ symbol (e.g., @max). The function \( \text{fun} \) must accept a column vector and return a numeric, logical, or character scalar, or a scalar cell. Return value \( A \) has the same class as the values returned by \( \text{fun} \). Specify \( \text{fun} \) as [] for the default behavior.

\( A = \text{accumarray}(\text{subs}, \text{val}, \text{sz}, \text{fun}, \text{fillval}) \) puts the scalar value \( \text{fillval} \) in elements of \( A \) that are not referred to by any row of \( \text{subs} \). For example, if \( \text{subs} \) is empty, then \( A \) is \( \text{repmat(fillval,sz)} \). \( \text{fillval} \) and the values returned by \( \text{fun} \) must belong to the same class. The default value of \( \text{fillval} \) is 0.

\( A = \text{accumarray}(\text{subs}, \text{val}, \text{sz}, \text{fun}, \text{fillval}, \text{issparse}) \) creates an array \( A \) that is sparse if the scalar input \( \text{issparse} \) is equal to logical 1 (i.e., true), or full if \( \text{issparse} \) is equal to logical 0 (false). \( A \) is full by
default. If issparse is true, then fillval must be zero or [], and val and
the output of fun must be double.

A = accumarray({subs1, subs2, ...}, val, ...) passes multiple
subs vectors in a cell array. You can use any of the four optional inputs
(sz, fun, fillval, or issparse) with this syntax.

**Note** If the subscripts in subs are not sorted, fun should not depend on
the order of the values in its input data.

The function processes the input as follows:

1 Find out how many unique indices there are in subs. Each unique
index defines a bin in the output array. The maximum index value in
subs determines the size of the output array.

2 Find out how many times each index is repeated.

This determines how many elements of vals are going to be
accumulated at each bin in the output array.

3 Create an output array. The output array is of size max(subs) or
of size sz.

4 Accumulate the entries in vals into bins using the values of the
indices in subs and apply fun to the entries in each bin.

5 Fill the values in the output for positions not referred to by subs.
Default fill value is zero; use fillval to set a different value.
Note subs should contain positive integers. subs can also be a cell vector with one or more elements, each element a vector of positive integers. All the vectors must have the same length. In this case, subs is treated as if the vectors formed columns of an index matrix. val must be a numeric, logical, or character vector with the same length as the number of rows in subs. val can also be a scalar whose value is repeated for all the rows of subs.

Examples

Example 1
Create a 5-by-1 vector and sum values for repeated 1-D subscripts:

val = 101:105;
subs = [1; 2; 4; 2; 4]
subs =
  1
  2
  4
  2
  4

A = accumarray(subs, val)
A =
 101 % A(1) = val(1) = 101
206 % A(2) = val(2)+val(4) = 102+104 = 206
  0 % A(3) = 0
208 % A(4) = val(3)+val(5) = 103+105 = 208

Example 2
Create a 4-by-4 matrix and subtract values for repeated 2-D subscripts:

val = 101:106;
subs=[1 2; 1 2; 3 1; 4 1; 4 4; 4 1];
B = accumarray(subs,val,[],@(x)sum(diff(x)))
B =
The order of the subscripts matters:

```matlab
val = 101:106;
subs=[1 2; 3 1; 1 2; 4 4; 4 1; 4 1];
B1 = accumarray(subs,val,[],@(x)sum(diff(x)))
```

```
B1 =
    0    2    0    0
    0    0    0    0
    0    0    0    0
 -1    0    0    0
```

**Example 3**

Create a 2-by-3-by-2 array and sum values for repeated 3-D subscripts:

```matlab
val = 101:105;
subs = [1 1 1; 2 1 2; 2 3 2; 2 1 2; 2 3 2];
A = accumarray(subs,val)
A(:,:,1) =
    101    0    100
    0    0    0
A(:,:,2) =
    0    0    208
    206    0    208
```

**Example 4**

Create a 2-by-3-by-2 array, and sum values natively:

```matlab
val = 101:105;
subs = [1 1 1; 2 1 2; 2 3 2; 2 1 2; 2 3 2];
```
A = accumarray(subs, int8(val), [], @(x) sum(x,'native'))
A(:,:,1) =
101 0 0
 0 0 0
A(:,:,2) =
 0 0 0
127 0 127

class(A)
ans =
   int8

**Example 5**

Pass multiple subscript arguments in a cell array.

1 Create a 12-element vector \( V \):

\[
V = 101:112;
\]

2 Create three 12-element vectors, one for each dimension of the resulting array \( A \). Note how the indices of these vectors determine which elements of \( V \) are accumulated in \( A \):

\[
\begin{align*}
\text{rowsubs} &= [1 3 3 2 3 1 2 2 3 3 1 2]; \\
colsubs &= [3 4 2 1 4 3 4 2 2 4 3 4]; \\
pagsubs &= [1 1 2 2 1 1 2 2 1 1 2 2]; \\
\text{index 1} & \quad \text{index 6} \Rightarrow V(1)+V(6) \Rightarrow A(1,3,1) \\
\text{index 4} & \Rightarrow V(4) \Rightarrow A(2,1,2)
\end{align*}
\]

\[
\begin{align*}
A(1,3,1) &= V(1) + V(6) = 101 + 106 = 207 \\
A(2,1,2) &= V(4) = 104
\end{align*}
\]

3 Call `accumarray`, passing the subscript vectors in a cell array:

\[
A = \text{accumarray}([\text{rowsubs} \ \text{colsubs} \ \text{pagsubs}], V)
\]
A(:,:,1) =
0 0 207 0 % A(1,3,1) is 207
0 108 0 0
0 109 0 317
A(:,:,2) =
0 0 111 0
104 0 0 219 % A(2,1,2) is 104
0 103 0 0

Example 6
Create an array with the max function, and fill all empty elements of that array with NaN:

val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @max, NaN)
A =
101 NaN NaN NaN
104 NaN 105 NaN

Example 7
Create a sparse matrix using the prod function:

val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @prod, 0, true)
A =
(1,1) 101
(2,1) 10608
(2,3) 10815

Example 8
Count the number of entries accumulated in each bin:

val = 1;
accumarray

subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4])
A =
    1  0  0  0
    2  0  2  0

Example 9
Create a logical array that shows which bins will accumulate two or more values:

val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @(x) length(x) > 1)
A =
    0  0  0  0
    1  0  1  0

Example 10
Group values in a cell array:

val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @(x) {x})
A =
    [101]   []   []   []
    [2x1 double]   []   [2x1 double]   []
A{2}
an =
    104
    102

See Also
full, sparse, sum
**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 \([-1, 1]\), \( \text{acos}(X) \) is real and in the range \([0, \pi]\). For real elements of \( X \) outside the domain \([-1, 1]\), \( \text{acos}(X) \) is complex.

The \( \text{acos} \) 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 \(-1 \leq x \leq 1\).

\[
\begin{align*}
x &= \text{array}; \\
\text{plot}(x, \text{acos}(x)), \text{ grid on}
\end{align*}
\]
Definition

The inverse cosine can be defined as

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

Algorithm

acos uses FDLIBM, which was developed at SunSoft, a Sun Microsystems™ 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 |
Purpose  
Inverse hyperbolic cosine

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

Description  
$Y = \text{acosh}(X)$ 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 $1 \leq x \leq \pi$.

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

Definition  
The hyperbolic inverse cosine can be defined as
$$\cosh^{-1}(z) = \log\left[ z + (z^2 - 1)^{\frac{1}{2}} \right]$$

**Algorithm**

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

**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 $-2\pi \leq x < 0$ and $0 < x \leq 2\pi$.

```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
\[ \cot^{-1}(z) = \tan^{-1}\left(\frac{1}{z}\right) \]

**Algorithm**  acot uses FDLIBM, which was developed at SunSoft, a Sun Microsystems business, by Kwok C. Ng, and others. For information about FDLIBM, see [http://www.netlib.org](http://www.netlib.org).

**See Also**  cot, acotd, acoth
acotd

**Purpose**
Inverse cotangent; result in degrees

**Syntax**
Y = acosd(X)

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

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

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

**Description**  
\( Y = \text{acoth}(X) \) returns the inverse hyperbolic cotangent for each element of \( X \).

The \text{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 \(-30 \leq x < -1\) and \(1 < x \leq 30\).

\[
\begin{align*}
x1 &= -30:0.1:-1.1; \\
x2 &= 1.1:0.1:30; \\
\text{plot}(x1,\text{acoth}(x1),x2,\text{acoth}(x2)), \text{ grid on}
\end{align*}
\]

**Definition**  
The hyperbolic inverse cotangent can be defined as
acoth

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

Algorithm

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

See Also

acot, coth
Purpose  
Inverse cosecant; result in radians

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

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

The \text{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 \(-10 \leq x < -1\) and \(1 < x \leq 10\).

\[ x1 = -10:0.01:-1.01; \]
\[ x2 = 1.01:0.01:10; \]
\[ \text{plot}(x1, \text{acsc}(x1), x2, \text{acsc}(x2)), \text{grid on} \]
**Definition**  
The inverse cosecant can be defined as  
\[ \csc^{-1}(z) = \sin^{-1}\left(\frac{1}{z}\right) \]

**Algorithm**  
`acsc` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems business, by Kwok C. Ng, and others. For information about FDLIBM, see [http://www.netlib.org](http://www.netlib.org).

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

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

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

See Also  \text{cscd, acsc}
Purpose
Inverse hyperbolic cosecant

Syntax
Y = acsch(X)

Description
Y = acsch(X) returns the inverse hyperbolic cosecant for each element of X.
The `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 $-20 \leq x \leq -1$ and $1 \leq x \leq 20$.

```matlab
x1 = -20:0.01:-1; x2 = 1:0.01:20; plot(x1,acsch(x1),x2,acsch(x2)), grid on
```

Definition
The hyperbolic inverse cosecant can be defined as
\[ \text{csch}^{-1}(z) = \sinh^{-1}\left(\frac{1}{z}\right) \]

**Algorithm**

`acsc` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems business, by Kwok C. Ng, and others. For information about FDLIBM, see [http://www.netlib.org](http://www.netlib.org).

**See Also**

`acsc`, `csch`
**Purpose**
Create Microsoft ActiveX control in figure window

**Syntax**

```matlab
h = actxcontrol('progid')
h = actxcontrol('progid','param1',value1,...)
h = actxcontrol('progid', position)
h = actxcontrol('progid', position, fig_handle)
h = actxcontrol('progid',position,fig_handle,event_handler)
h = actxcontrol('progid',position,fig_handle,event_handler,'filename')
```

**Description**

$h = \text{actxcontrol}('progid')$ creates an ActiveX® control in a figure window. The programmatic identifier (progid) for the control determines the type of control created. (See the documentation provided by the control vendor to get this string.) The returned object, $h$, represents the default interface for the control.

Note that progid cannot be an ActiveX server because the MATLAB software cannot insert ActiveX servers in a figure. See `actxserver` for use with ActiveX servers.

$h = \text{actxcontrol}('progid','param1',value1,...)$ creates an ActiveX control using the optional parameter name/value pairs. Parameter names include:

- **position** — MATLAB position vector specifying the control’s position. The format is [left, bottom, width, height] using pixel units.
- **parent** — Handle to parent figure, model, or command window.
- **callback** — Name of event handler. Specify a single name to use the same handler for all events. Specify a cell array of event name/event handler pairs to handle specific events.
- **filename** — Sets the control’s initial conditions to those in the previously saved control.
- **licensekey** — License key to create licensed ActiveX controls that require design-time licenses. See “Deploying ActiveX Controls Requiring Run-Time Licenses” for information on how to use controls that require run-time licenses.
One possible format is:

```matlab
h = actxcontrol('myProgId','newPosition',[0 0 200 200],...
    'myFigHandle',gcf,...
    'myCallback',{ 'Click' 'myClickHandler';...
    'DblClick' 'myDblClickHandler';...
    'MouseDown' 'myMouseDownHandler'});
```

The following syntaxes are deprecated and will not become obsolete. They are included for reference, but the above syntaxes are preferred.

```matlab
h = actxcontrol('progid', position)
```
creates an ActiveX control having the location and size specified in the vector, position. The format of this vector is:

```matlab
[x y width height]
```

The first two elements of the vector determine where the control is placed in the figure window, with $x$ and $y$ being offsets, in pixels, from the bottom left corner of the figure window to the same corner of the control. The last two elements, width and height, determine the size of the control itself.

The default position vector is $[20 \ 20 \ 60 \ 60]$.

```matlab
h = actxcontrol('progid', position, fig_handle)
```
creates an ActiveX control at the specified position in an existing figure window. This window is identified by the Handle Graphics handle, `fig_handle`.

The current figure handle is returned by the `gcf` command.

**Note** If the figure window designated by `fig_handle` is invisible, the control is invisible. If you want the control you are creating to be invisible, use the handle of an invisible figure window.

```matlab
h = actxcontrol('progid',position,fig_handle,event_handler)
```
creates an ActiveX control that responds to events. Controls respond to events by invoking an M-file function whenever an event (such...
as clicking a mouse button) is fired. The `event_handler` argument identifies one or more M-file functions to be used in handling events (see “Specifying Event Handlers” on page 2-94 below).

```
actxcontrol('progid',position,fig_handle,event_handler,'filename')
```

creates an ActiveX control with the first four arguments, and sets its initial state to that of a previously saved control. MATLAB loads the initial state from the file specified in the string `filename`.

If you don’t want to specify an `event_handler`, you can use an empty string (``) as the fourth argument.

The `progid` argument must match the `progid` of the saved control.

### Specifying Event Handlers

There is more than one valid format for the `event_handler` argument. Use this argument to specify one of the following:

- A different event handler routine for each event supported by the control
- One common routine to handle selected events
- One common routine to handle all events

In the first case, use a cell array for the `event_handler` argument, with each row of the array specifying an event and handler pair:

```
{ 'event' 'eventhandler'; 'event2' 'eventhandler2'; ...}
```

event can be either a string containing the event name or a numeric event identifier (see Example 2 below), and `eventhandler` is a string identifying the M-file function you want the control to use in handling the event. Include only those events that you want enabled.

In the second case, use the same cell array syntax just described, but specify the same `eventhandler` for each event. Again, include only those events that you want enabled.
In the third case, make `event_handler` a string (instead of a cell array) that contains the name of the one M-file function that is to handle all events for the control.

There is no limit to the number of event and handler pairs you can specify in the `event_handler` cell array.

Event handler functions should accept a variable number of arguments.

Strings used in the `event_handler` argument are not case sensitive.

**Note** Although using a single handler for all events may be easier in some cases, specifying an individual handler for each event creates more efficient code that results in better performance.

**Remarks**

If the control implements any custom interfaces, use the `interfaces` function to list them, and the `invoke` function to get a handle to a selected interface.

When you no longer need the control, call `release` to release the interface and free memory and other resources used by the interface. Note that releasing the interface does not delete the control itself. Use the `delete` function to do this.

For more information on handling control events, see Writing Event Handlers in the External Interfaces documentation.

For an example event handler, see the file `sampev.m` in the `toolbox\matlab\winfun\comcli` directory.

COM functions are available on Microsoft Windows systems only.

**Note** If you encounter problems creating Microsoft Forms 2.0 controls in MATLAB software or other non-VBA container applications, see “Using Microsoft Forms 2.0 Controls” in the External Interfaces documentation.
**Examples**

**Example 1 — Basic Control Methods**

Start by creating a figure window to contain the control. Then create a control to run a Microsoft Calendar application in the window. Position the control at a [0 0] x-y offset from the bottom left of the figure window, and make it the same size (600 x 500 pixels) as the figure window.

```matlab
f = figure('position', [300 300 600 500]);
cal = actxcontrol('mscal.calendar', [0 0 600 500], f);
```

Call the get method on `cal` to list all properties of the calendar, including today’s date:

```matlab
cal.get
```

For example, MATLAB displays (in part):

```
BackgroundColor: 2.1475e+009
Day: 23
DayFont: [1x1 Interface.Standard_OLE_Types.Font]
Value: '8/20/2001'
```

Read today’s date:

```matlab
date = cal.Value
```

MATLAB displays a date similar to:

```matlab
date =
8/20/2001
```

Set the Day property to a new value:

```matlab
kal.Day = 5;
date = kal.Value
```
MATLAB displays a date similar to:

\[
\text{date} = \\
8/5/2001
\]

Call `invoke` to list all available methods:

\[
\text{meth} = \text{cal.invoke}
\]

MATLAB displays (in part):

\[
\text{meth} = \\
\begin{align*}
\text{NextDay:} & \quad \text{'HRESULT NextDay(handle)'} \\
\text{NextMonth:} & \quad \text{'HRESULT NextMonth(handle)'} \\
\text{NextWeek:} & \quad \text{'HRESULT NextWeek(handle)'} \\
\text{NextYear:} & \quad \text{'HRESULT NextYear(handle)'}
\end{align*}
\]

Invoke the `NextWeek` method to advance the current date by one week:

\[
\text{cal.NextWeek;}
\text{date} = \text{cal.Value}
\]

MATLAB displays a date similar to:

\[
\text{date} = \\
8/12/2001
\]

Call `events` to list all calendar events that can be triggered:

\[
\text{cal.events}
\]

MATLAB displays:

\[
\text{Click} = \text{void Click()}
\text{Db1Click} = \text{void Db1Click()}
\]
Example 2 — Event Handling

The `event_handler` argument specifies how you want the control to handle any events that occur. The control can handle all events with one common handler function, selected events with a common handler function, or each type of event can be handled by a separate function.

This command creates an `mwsamp` control that uses one event handler, `sampev`, to respond to all events:

```matlab
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
gcf, 'sampev');
```

The next command also uses a common event handler, but will only invoke the handler when selected events, `Click` and `DblClick` are fired:

```matlab
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
gcf, {'Click' 'sampev'; 'DblClick' 'sampev'});
```

This command assigns a different handler routine to each event. For example, `Click` is an event, and `myclick` is the routine that executes whenever a `Click` event is fired:

```matlab
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
gcf, {'Click', 'myclick'; 'DblClick' 'my2click'; ...
'MouseDown' 'mymoused'});
```

The next command does the same thing, but specifies the events using numeric event identifiers:

```matlab
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
gcf, {-600, 'myclick'; -601 'my2click'; -605 'mymoused'});
```
See the section, “Sample Event Handlers” in the External Interfaces documentation for examples of event handler functions and how to register them with MATLAB software.

See Also

actxserver, release, delete (COM), save (COM), load (COM), interfaces
**Purpose**
List all currently installed Microsoft ActiveX controls

**Syntax**
C = actxcontrollist

**Description**
C = actxcontrollist returns a list of each control, including its name, programmatic identifier (or ProgID), and filename, in output cell array C.

**Remarks**
COM functions are available on Microsoft Windows systems only.

**Examples**
Here is an example of the information that might be returned for several controls:

```matlab
list = actxcontrollist;
for k = 1:2
    sprintf(' Name = %s
ProgID = %s
File = %s', list{k,:})
end
```

MATLAB software displays information similar to:

```plaintext
ans =
Name = Calendar Control 11.0
ProgID = MSCAL.Calendar.7
File = C:\Program Files\MSOffice\OFFICE11\MSCAL.OCX

ans =
Name = CTreeView Control
ProgID = CTREEVIEW.CTreeViewCtrl.1
File = C:\WINNT\system32\dmocx.dll
```

**See Also**
actxcontrolselect, actxcontrol
**Purpose**
Open GUI to create Microsoft ActiveX control

**Syntax**

\[
h = \text{actxcontrolselect} \\
[h, \text{info}] = \text{actxcontrolselect}
\]

**Description**

\( h = \text{actxcontrolselect} \) displays a graphical interface that lists all ActiveX controls installed on the system and creates the one that you select from the list. The function returns a handle \( h \) for the object. Use the handle to identify this particular control object when calling other MATLAB COM functions.

\([h, \text{info}] = \text{actxcontrolselect}\) returns the handle \( h\) and also the 1-by-3 cell array \( \text{info} \) containing information about the control. The information returned in the cell array shows the name, programmatic identifier (or ProgID), and filename for the control.
The `actxcontrolselect` interface has a selection pane at the left of the window and a preview pane at the right. Click on one of the control names in the selection pane to see a preview of the control displayed. (If MATLAB cannot create the control, an error message is displayed in the preview pane.) Select an item from the list and click the **Create** button at the bottom.

**Remarks**

Click the **Properties** button on the `actxcontrolselect` window to enter nondefault values for properties when creating the control. You can select which figure window to put the control in (Parent field), where to position it in the window (X and Y fields), and what size to make the control (Width and Height).

You can also register any events you want the control to respond to and what event handling routines to use when any of these events fire. Do this by entering the name of the appropriate event handling routine to the right of the event, or clicking the **Browse** button to search for the event handler file.

COM functions are available on Microsoft Windows systems only.
**Note** If you encounter problems creating Microsoft Forms 2.0 controls in MATLAB software or other non-VBA container applications, see “Using Microsoft Forms 2.0 Controls” in the External Interfaces documentation.

### Examples

Open a window showing the ActiveX controls on your system:

```matlab
[h, info] = actxcontrolselect
```

Select the Calendar Control in the window and click Properties to open the window shown above. Enter new values for the size of the control, setting Width to 500 and Height to 350, then click OK. Click Create in the actxcontrolselect window to create the control.

The control appears in a MATLAB figure window. MATLAB displays information similar to (your version number may be different):

```matlab
h = COM.MSCAL_Calendar_7
info = [1x21 char] 'MSCAL.Calendar.7' [1x44 char]%}
```

Expand the info cell array to show the control name, ProgID, and filename:

```matlab
info{:}
```

MATLAB displays information similar to:

```matlab
ans = Calendar Control 11.0
ans = MSCAL.Calendar.7
ans =
actxcontrolselect

C:\Program Files\MSOffice\OFFICE11\MSCAL.OCX

See Also

actxcontrollist, actxcontrol
| **Purpose** | Get handle to running instance of Automation server |
| **Syntax** | `h = actxGetRunningServer('progid')` |
| **Description** | `h = actxGetRunningServer('progid')` gets a reference to a running instance of the OLE Automation server, where `progid` is the programmatic identifier of the Automation server object and `h` is the handle to the server object’s default interface. The function issues an error if the server specified by `progid` is not currently running or if the server object is not registered. When there are multiple instances of the Automation server already running, the behavior of this function is controlled by the operating system. |
| **Remarks** | COM functions are available on Microsoft Windows systems only. |
| **Example** | `h = actxGetRunningServer('matlab.application')` |
| **See Also** | `actxcontrol, actxserver` |
**Purpose**
Create COM server

**Syntax**

```matlab
h = actxserver('progid')
h = actxserver('progid', 'machine', 'machineName')
h = actxserver('progid', 'interface', 'interfaceName')
h = actxserver('progid', 'machine', 'machineName', 'interface', 'interfaceName')
h = actxserver('progid', machine)
```

**Description**

$h = \text{actxserver('progid')}$ creates a local OLE Automation server, where $progid$ is the programmatic identifier of the COM server, and $h$ is the handle of the server’s default interface.

Get $progid$ from the control or server vendor’s documentation. To see the $progid$ values for MATLAB software, refer to “Programmatic Identifiers” in the MATLAB External Interfaces documentation.

$h = \text{actxserver('progid', 'machine', 'machineName')}$ creates an OLE Automation server on a remote machine, where $machineName$ is a string specifying the name of the machine on which to start the server.

$h = \text{actxserver('progid', 'interface', 'interfaceName')}$ creates a Custom interface server, where $interfaceName$ is a string specifying the interface name of the COM object. Values for $interfaceName$ are

- IUnknown — Use the IUnknown interface.
- The Custom interface name

You must know the name of the interface and have the server vendor’s documentation in order to use the $interfaceName$ value. See “COM Server Types” in the MATLAB External Interfaces documentation for information about Custom COM servers and interfaces.

**Note**
The MATLAB COM Interface does not support invoking functions with optional parameters.
h = actxserver('progid', 'machine', 'machineName', 'interface', 'interfaceName') creates a Custom interface server on a remote machine.

The following syntaxes are deprecated and will not become obsolete. They are included for reference, but the syntaxes described earlier are preferred:

h = actxserver('progid', machine) creates a COM server running on the remote system named by the machine argument. This can be an IP address or a DNS name. Use this syntax only in environments that support Distributed Component Object Model (DCOM).

Remarks

For components implemented in a dynamic link library (DLL), actxserver creates an in-process server. For components implemented as an executable (EXE), actxserver creates an out-of-process server. Out-of-process servers can be created either on the client system or on any other system on a network that supports DCOM.

If the control implements any Custom interfaces, use the interfaces function to list them, and the invoke function to get a handle to a selected interface.

You can register events for COM servers.

COM functions are available on Microsoft Windows systems only.

Microsoft Excel Workbook Example

This example creates an OLE Automation server, Excel® version 9.0, and manipulates a workbook in the application:

```matlab
% Create a COM server running Microsoft Excel
e = actxserver ('Excel.Application')
%

% Make the Excel frame window visible
e.Visible = 1;
```
% Use the get method on the Excel object "e"
% to list all properties of the application:
e.get
%
   Application: [1x1
   Interface.Microsoft_Excel_9.0_Object_Library._Application]
   Creator: 'xlCreatorCode'
   .
   .
   .
   Workbooks: [1x1
   Interface.Microsoft_Excel_9.0_Object_Library.Workbooks]
   .
   .
   .
   Caption: 'Microsoft Excel - Book1'
   CellDragAndDrop: 0
   ClipboardFormats: {3x1 cell}
   .
   .
   .
   Cursor: 'xlNorthwestArrow'
   .
   .
   .
%
% Create an interface "eWorkBooks"
eWorkbooks = e.Workbooks
%
   eWorkbooks =
   Interface.Microsoft_Excel_9.0_Object_Library.Workbooks
%
% List all methods for that interface
eWorkbooks.invoke
%
   Add: 'handle Add(handle, [Optional]Variant)'
   Close: 'void Close(handle)'

2-108
Item: 'handle Item(handle, Variant)'
Open: 'handle Open(handle, string, [Optional]Variant)'
OpenText: 'void OpenText(handle, string, [Optional]Variant)'

%
%
%
%
%
%} Add a new workbook "w",
% also creating a new interface
w = eWorkbooks.Add
%
w =
  Interface.Microsoft_Excel_9.0_Object_Library._Workbook
%
% Close Excel and delete the object
e.Quit;
e.delete;

See Also

actxcontrol, actxGetRunningServer, release, delete (COM), save (COM), load (COM), interfaces
addCause (MException)

**Purpose**
Append MException objects

**Syntax**

```
new_ME = addCause(base_ME, cause_ME)
base_ME = addCause(base_ME, cause_ME)
```

**Description**

`new_ME = addCause(base_ME, cause_ME)` creates a new MException object `new_ME` from two existing MException objects, `base_ME` and `cause_ME`. `addCause` constructs `new_ME` by making a copy of the `base_ME` object and appending `cause_ME` to the `cause` property of that object.

If other errors have contributed to the exception currently being thrown, you can add the MException objects that represent these errors to the `cause` field of the current MException to provide further information for diagnosing the error at hand. All objects of the MException class have a property called `cause` which is defined as a vector of additional MException objects that can be added onto a base object of that class.

`base_ME = addCause(base_ME, cause_ME)` modifies existing MException object `base_ME` by appending `cause_ME` to the `cause` property of that object.

**Examples**

**Example 1**

This example attempts to assign data from array `D`. If `D` does not exist, the code attempts to recreate `D` by loading it from a MAT-file. The code constructs a new MException object `new_ME` to store the causes of the first two errors, `cause1_ME` and `cause2_ME`:

```
try
    x = D(1:25);
catch cause1_ME
    try
        filename = 'test204';
        testdata = load(filename);
        x = testdata.D(1:25)
    catch cause2_ME
        base_ME = MException('MATLAB:LoadErr', ...
            'Unable to load from file %s', filename);
    end
end
```
When you run the code, the MATLAB software displays the following message:

```plaintext
??? Unable to load from file test204
```

There are two exceptions in the cause field of `new_ME`:

```plaintext
new_ME.cause
ans =
[1x1 MException]
[1x1 MException]
```

Examine the cause field of `new_ME` to see the related errors:

```plaintext
new_ME.cause{:}
ans =

MException object with properties:

    identifier: 'MATLAB:UndefinedFunction'
    message: 'Undefined function or method 'D' for input arguments of type 'double'.'
    stack: [0x1 struct]
    cause: {}

ans =

MException object with properties:

    identifier: 'MATLAB:load:couldNotReadFile'
    message: 'Unable to read file test204: No such file or directory.'
    stack: [0x1 struct]
```
Example 2

This example attempts to open a file in a directory that is not on the MATLAB path. It uses a nested try-catch block to give the user the opportunity to extend the path. If the file still cannot be found, the program issues an exception with the first error appended to the second using addCause:

```matlab
function data = read_it(filename);
    try
        fid = fopen(filename, 'r');
        data = fread(fid);
    catch ME1
        if strcmp(ME1.identifier, 'MATLAB:FileIO:InvalidFid')
            msg = sprintf('
%s%s%s', 'Cannot open file ', ...
                filename, '. Try another location? ');
            reply = input(msg, 's')
            if reply(1) == 'y'
                newdir = input('Enter directory name: ', 's');
            else
                throw(ME1);
            end
            addpath(newdir);
        end
        try
            fid = fopen(filename, 'r');
            data = fread(fid);
        catch ME2
            ME3 = addCause(ME2, ME1)
            throw(ME3);
        end
        rmpath(newdir);
    end
    fclose(fid);
```
If you run this function in a try-catch block at the command line, you can look at the MException object by assigning it to a variable (e) with the catch command.

```matlab
try
d = read_it('anytextfile.txt');
catch e
end

e
```

```
e =
MException object with properties:
  identifier: 'MATLAB:FileIO:InvalidFid'
  message: 'Invalid file identifier. Use fopen to generate a valid file identifier.'
  stack: [1x1 struct]
  cause: {[1x1 MException]}

Cannot open file anytextfile.txt. Try another location?
y Enter directory name: xxxxxxx
Warning: Name is nonexistent or not a directory: xxxxxxx.
> In path at 110
  In addpath at 89
```

**See Also**

try, catch, error, assert, throw(MException),
rethrow(MException), throwAsCaller(MException),
getReport(MException), disp(MException), isequal(MException),
eq(MException), ne(MException), last(MException)
Purpose
Add event to timeseries object

Syntax
\[
\begin{align*}
\text{ts} &= \text{addevent}(	ext{ts}, \text{e}) \\
\text{ts} &= \text{addevent}(	ext{ts}, \text{Name, Time})
\end{align*}
\]

Description
\( \text{ts} = \text{addevent}(	ext{ts}, \text{e}) \) adds one or more \text{tsdata.event} objects, \text{e}, to the timeseries object \text{ts}. \text{e} is either a single \text{tsdata.event} object or an array of \text{tsdata.event} objects.

\( \text{ts} = \text{addevent}(	ext{ts}, \text{Name, Time}) \) constructs one or more \text{tsdata.event} objects and adds them to the \text{Events} property of \text{ts}. \text{Name} is a cell array of event name strings. \text{Time} is a cell array of event times.

Examples
Create a time-series object and add an event to this object.

\begin{verbatim}
%% Import the sample data
load count.dat

%% Create time-series object
count1=timeseries(count(:,1),1:24,'name', 'data');

%% Modify the time units to be 'hours' ('seconds' is default)
count1.TimeInfo.Units = 'hours';

%% Construct and add the first event at 8 AM
e1 = tsdata.event('AMCommute',8);

e1.Units = 'hours';

View the properties (EventData, Name, Time, Units, and StartDate) of the event object.

get(e1)
\end{verbatim}

MATLAB software responds with

\begin{verbatim}
EventData: []
\end{verbatim}
Name: 'AMCommute'
Time: 8
Units: 'hours'
StartDate: ''

```matlab
count1 = addevent(count1,e1);
```

An alternative syntax for adding two events to the time series `count1` is as follows:

```matlab
count1 = addevent(count1,{'AMCommute' 'PMCommute'},[8 18])
```

See Also

`timeseries`, `tsdata.event`, `tsprops`
Purpose

Add frame to Audio/Video Interleaved (AVI) file

Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aviobj = addframe(aviobj,frame)</td>
<td>append 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.</td>
</tr>
<tr>
<td>aviobj = addframe(aviobj,frame1,frame2,frame3,...)</td>
<td>adds multiple frames to an AVI file.</td>
</tr>
<tr>
<td>aviobj = addframe(aviobj,mov)</td>
<td>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.</td>
</tr>
<tr>
<td>aviobj = addframe(aviobj,h)</td>
<td>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.</td>
</tr>
</tbody>
</table>

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`.

```matlab
t = linspace(0,2.5*pi,40);
fact = 10*sin(t);
fig = figure;
aviobj = avifile('example.avi')
[x,y,z] = peaks;
for k=1:length(fact)
    h = surf(x,y,fact(k)*z);
    axis([-3 3 -3 3 -80 80])
    axis off
    caxis([-90 90])
    F = getframe(fig);
    aviobj = addframe(aviobj,F);
end
close(fig)
 aviobj = close(aviobj);
```

See Also

`avifile`, `close`, `movie2avi`
Purpose

Create event listener

Syntax

\[ lh = \text{addlistener}(Hsource,'EventName',\text{callback}) \]
\[ lh = \text{addlistener}(Hsource,\text{property},'EventName',\text{callback}) \]

Description

\[ lh = \text{addlistener}(Hsource,'EventName',\text{callback}) \] creates a listener for the specified event.

\[ lh = \text{addlistener}(Hsource,\text{property},'EventName',\text{callback}) \] creates a listener for one of the predefined property events. There are four property events:

- \textbf{PreSet} — triggered just before the property value is set, before calling its set access method.
- \textbf{PostSet} — triggered just after the property value is set.
- \textbf{PreGet} — triggered just before a property value query is serviced, before calling its get access method.
- \textbf{PostGet} — triggered just after returning the property value to the query.

See “Defining Events and Listeners — Syntax and Techniques” for more information.

Arguments

- \textbf{Hsource}
  Handle of the object that is the source of the event, or an array of source handles.

- \textbf{EventName}
  Name of the event, which is triggered by the source objects.

- \textbf{callback}
  Function handle referencing a function to execute when the event is triggered.

- \textbf{property}
  Character string that can be:
• the name of the property
• a cell array of strings where each string is the name of a property that exists in object array Hsource
• a meta.property object or an array of meta.property objects
• a cell array of meta.property objects

If Hsource is a scalar, then any of the properties can be dynamic properties. If Hsource is non-scalar, then the properties must belong to the class of Hsource and can not include dynamic properties (which are not part of the class definition).

For more information, see the following sections:
• The GetObservable and SetObservable property attributes in the “Property Attributes” table.
• “Creating Property Listeners”
• “Dynamic Properties — Adding Properties to an Instance”

Handle of the event.listener object returned by addlistener.

Removing a Listener

To remove a listener, delete the listener object returned by addlistener. For example,

delete(lh)

calls the handle class delete method to delete the object from the workspace and remove the listener.

See Also
delete (handle), handle, notify (handle)
**addOptional** *(inputParser)*

**Purpose**
Add optional argument to **inputParser** schema

**Syntax**
p.addOptional(argname, default, validator)
addOptional(p, argname, default, validator)

**Description**
p.addOptional(argname, default, validator) updates the schema for **inputParser** object p by adding an optional argument, argname. Specify the argument name in a string enclosed within single quotation marks. The default input specifies the value to use when the optional argument argname is not present in the actual inputs to the function. The optional validator input is a handle to a function that the MATLAB software uses during parsing to validate the input arguments. If the validator function returns false or errors, the parsing fails and MATLAB throws an error.

MATLAB parses parameter-value arguments after required arguments and optional arguments.

addOptional(p, argname, default, validator) is functionally the same as the syntax above.

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

**Examples**
Write an M-file function called publish_ip, based on the MATLAB publish function, to illustrate the use of the **inputParser** class.

There are three calling syntaxes for this function:

```matlab
publish_ip('script')
publish_ip('script', 'format')
publish_ip('script', options)
```

From these three syntaxes, you can see that there is one required argument (script), one optional argument (format), and some number of optional arguments that are specified as parameter-value pairs (options).
Begin writing the example publish_ip M-file by entering the following two statements. The second statement calls the class constructor for inputParser to create an instance of the class. This class instance, or object, gives you access to all of the methods and properties of the class:

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

Following the constructor, add this block of code to the M-file. This code uses the addRequired(inputParser), addOptional, and addParamValue(inputParser) methods to define the input arguments to the function:

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

Also add the next two lines to the M-file. The Parameters property of inputParser lists all of the arguments that belong to the object p:

```
disp 'The input parameters for this program are'
disp(p.Parameters)
```

Save the M-file using the Save option on the MATLAB File menu, and then run it to see the following list displayed:

```
The input parameters for this program are
'format'
'maxHeight'
'maxWidth'
'outputDir'
'script'
```
addOptional (inputParser)

See Also

inputParser, addRequired(inputParser),
addParamValue(inputParser), parse(inputParser),
createCopy(inputParser)
Purpose
Add parameter-value argument to inputParser schema

Syntax
p.addParamValue(argname, default, validator)
addParamValue(p, argname, default, validator)

Description
p.addParamValue(argname, default, validator) updates the schema for inputParser object p by adding a parameter-value argument, argname. Specify the argument name in a string enclosed within single quotation marks. The default input specifies the value to use when the optional argument name is not present in the actual inputs to the function. The optional validator is a handle to a function that the MATLAB software uses during parsing to validate the input arguments. If the validator function returns false or errors, the parsing fails and MATLAB throws an error.

MATLAB parses parameter-value arguments after required arguments and optional arguments.

addParamValue(p, argname, default, validator) is functionally the same as the syntax above.

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

Examples
Write an M-file function called publish_ip, based on the MATLAB publish function, to illustrate the use of the inputParser class. There are three calling syntaxes for this function:

```matlab
publish_ip('script')
publish_ip('script', 'format')
publish_ip('script', options)
```

From these calling syntaxes, you can see that there is one required argument (script), one optional argument (format), and a number of optional arguments that are specified as parameter-value pairs (options).
Begin writing the example publish_ip M-file by entering the following two statements. Call the class constructor for inputParser to create an instance of the class. This class instance, or object, gives you access to all of the methods and properties of the class:

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

After calling the constructor, add the following lines to the M-file. This code uses the addRequired(inputParser), addOptional(inputParser), and addParamValue methods to define the input arguments to the function:

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

Also add the next two lines to the M-file. The Parameters property of inputParser lists all of the arguments that belong to the object p:

```matlab
  disp 'The input parameters for this program are
  disp(p.Parameters)
```

Save the M-file using the Save option on the MATLAB File menu, and then run it to see the following list displayed:

```
The input parameters for this program are
  'format'
  'maxHeight'
  'maxWidth'
  'outputDir'
  'script'
```
addParamValue (inputParser)

See Also

inputParser, addRequired(inputParser),
addOptional(inputParser), parse(inputParser),
createCopy(inputParser)
**Purpose**
Add directories to search path

**GUI Alternatives**
As an alternative to the addpath function, use the Set Path dialog box.

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

**Description**
- `addpath('directory')` adds the specified directory to the top (also called front) of the current MATLAB search path. Use the full pathname for `directory`.
- `addpath('dir','dir2','dir3' ...)` adds all the specified directories to the top of the path. Use the full pathname for each `dir`.
- `addpath('dir','dir2','dir3' ...'-flag')` adds the specified directories to either the top or bottom of 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 <code>begin</code></td>
<td>Add specified directories to the top of the path</td>
</tr>
<tr>
<td>1 or <code>end</code></td>
<td>Add specified directories to the bottom (also called end) of the path</td>
</tr>
</tbody>
</table>

`addpath 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 “Automatically Modifying the Search Path.”
Path at Startup” in the MATLAB Desktop Tools and Development Environment documentation.

**Examples**

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

```
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 running

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

Verify that the files were added to the path by running

```
path
```

and MATLAB returns

```
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. For example, to add `/control` and its subdirectories to the path, use

```
addpath(genpath(fullfile(matlabroot,'toolbox/control')))
```

**See Also**

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

“Search Path” in the MATLAB Desktop Tools and Development Environment documentation
### Purpose
Add preference

### Syntax
addpref('group','pref',val)
addpref('group',{'pref1','pref2',...'prefn'},[val1,val2,...,valn])

### Description
`addpref('group','pref',val)` creates the preference specified by `group` and `pref` and sets its value to `val`. It is an error to add a preference that already exists.

group labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g. 'ApplicationOnePrefs'. The input argument `pref` identifies an individual preference in that group, and must be a legal variable name.

`addpref('group',{'pref1','pref2',...'prefn'},[val1,val2,...,valn])` creates the preferences specified by the cell array of names 'pref1', 'pref2',..., 'prefn', setting each to the corresponding value.

---

**Note** Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

### Examples
This example adds a preference called version to the mytoolbox group of preferences and sets its value to the string 1.0.

```matlab
addpref('mytoolbox','version','1.0')
```

### See Also
getpref, ispref, rmpref, setpref, uigetpref, uisetpref
**Purpose**
Add dynamic property

**Syntax**
P = addprop(Hobj,'PropName')

**Description**
P = addprop(Hobj,'PropName') adds a property named PropName to each object in array Hobj. The class definition is not affected by the addition of dynamic properties. Note that you can add dynamic properties only to objects derived from the dynamicprops class. You can set and retrieve the data in dynamic properties as you would any property.

The output argument P is an array the same size as Hobj of meta.DynamicProperty objects, which you can use to assign SetMethod and GetMethod functions to the property. These functions operate just like property set and get access methods.

See “Dynamic Properties — Adding Properties to an Instance” for more information and examples.

**See Also**
handle, dynamicprops
**Purpose**
Add custom property to COM object

**Syntax**

```
    h.addproperty('propertyname')
    addproperty(h, 'propertyname')
```

**Description**

`h.addproperty('propertyname')` adds the custom property specified in the string, `propertyname`, to the object or interface, `h`. Use `set` to assign a value to the property.

`addproperty(h, 'propertyname')` is an alternate syntax for the same operation.

**Remarks**

COM functions are available on Microsoft Windows systems only.

**Examples**

Create an `mwsamp` control and display its properties:

```matlab
    f = figure('position', [100 200 200 200]);
    h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
    h.get
```

MATLAB software displays:

```
    Label: 'Label'
    Radius: 20
```

Add a new property named `Position` to the control. Assign an array value to the property:

```matlab
    h.addproperty('Position');
    h.Position = [200 120];
    h.get
```

MATLAB displays (in part):

```
    Label: 'Label'
    Radius: 20
    Position: [200 120]```
Delete the custom Position property:

```matlab
    h.deleteproperty('Position');
    h.get
```

MATLAB displays:

```
Label: 'Label'
Radius: 20
```

Create an `mwsamp` control and add a new property named `Position` to it. Assign an array value to the property:

```matlab
    f = figure('position', [100 200 200 200]);
    h = actxcontrol(['mwsamp.mwsampctrl1.2', [0 0 200 200], f);
    h.get
        Label: 'Label'
        Radius: 20
    h.addproperty('Position');
    h.Position = [200 120];
    h.get
        Label: 'Label'
        Radius: 20
        Position: [200 120]

    h.get('Position')
    ans =
        200   120
```

Delete the custom Position property:

```matlab
    h.deleteproperty('Position');
    h.get
        Label: 'Label'
        Radius: 20
```

**See Also**

`deleteproperty`, `get (COM)`, `set (COM)`, `inspect`
addRequired (inputParser)

**Purpose**
Add required argument to inputParser schema

**Syntax**

```
p.addRequired(argname, validator)
addRequired(p, argname, validator)
```

**Description**

p.addRequired(argname, validator) updates the schema for inputParser object p by adding a required argument, argname. Specify the argument name in a string enclosed within single quotation marks. The optional validator is a handle to a function that the MATLAB software uses during parsing to validate the input arguments. If the validator function returns false or errors, the parsing fails and MATLAB throws an error.

MATLAB parses required arguments before optional or parameter-value arguments.

addRequired(p, argname, validator) is functionally the same as the syntax above.

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

**Examples**

Write an M-file function called publish_ip, based on the MATLAB publish function, to illustrate the use of the inputParser class. There are three calling syntaxes for this function:

```
publish_ip('script')
publish_ip('script', 'format')
publish_ip('script', options)
```

From these calling syntaxes, you can see that there is one required argument (script), one optional argument (format), and a number of optional arguments that are specified as parameter-value pairs (options).
Begin writing the example publish_ip M-file by entering the following two statements. Call the class constructor for `inputParser` to create an instance of the class. This class instance, or object, gives you access to all of the methods and properties of the class:

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

After calling the constructor, add the following lines to the M-file. This code uses the `addRequired`, `addOptional`, and `addParamValue` methods to define the input arguments to the function:

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

Also add the next two lines to the M-file. The `Parameters` property of `inputParser` lists all of the arguments that belong to the object `p`:

```matlab
disp 'The input parameters for this program are'
disp(p.Parameters)
```

Save the M-file using the **Save** option on the MATLAB **File** menu, and then run it to see the following list displayed:

```
The input parameters for this program are
'format'
'maxHeight'
'maxWidth'
'outputDir'
'script'
```
addRequired (inputParser)

See Also
inputParser, addOptional(inputParser), addParamValue(inputParser), parse(inputParser), createCopy(inputParser)
Purpose
Add data sample to timeseries object

Syntax
\[ ts = \text{addsample}(ts, 'Field1', \text{Value1}, 'Field2', \text{Value2}, ...) \]
\[ ts = \text{addsample}(ts, s) \]

Description
\( ts = \text{addsample}(ts, 'Field1', \text{Value1}, 'Field2', \text{Value2}, ...) \) adds one or more data samples to the timeseries object \( ts \), where one field must specify Time and another must specify Data. You can also specify the following optional property-value pairs:

- 'Quality' — Array of data quality codes
- 'OverwriteFlag' — Logical value that controls whether to overwrite a data sample at the same time with the new sample you are adding to your timeseries object. When set to true, the new sample overwrites the old sample at the same time.

\( ts = \text{addsample}(ts, s) \) adds one or more new samples stored in a structure \( s \) to the timeseries object \( ts \). You must define the fields of the structure \( s \) before passing it as an argument to addsample by assigning values to the following optional \( s \) fields:

- \( s.data \)
- \( s.time \)
- \( s.quality \)
- \( s.overwriteflag \)

Remarks
A time-series data sample consists of one or more values recorded at a specific time. The number of data samples in a time series is the same as the length of the time vector.

The Time value must be a valid time vector.

Suppose that \( N \) is the number of samples. The sample size of each time series is given by \( \text{SampleSize} = \text{getsamplesize}(ts) \). When
ts.IsTimeFirst is true, the size of the data is N-by-SampleSize. When ts.IsTimeFirst is false, the size of the data is SampleSize-by-N.

**Examples**

Add a data value of 420 at time 3.

```matlab
ts = ts.addsample('Time',3,'Data',420);
```

Add a data value of 420 at time 3 and specify quality code 1 for this data value. Set the flag to overwrite an existing value at time 3.

```matlab
ts = ts.addsample('Data',3.2,'Quality',1,'OverwriteFlag',true,'Time',3);
```

**See Also**
delsample, getdatasamplesize, tsprops
Purpose
Add sample to tscollection object

Syntax
tsc = addsampletocollection(tsc,'time',Time,TS1Name,TS1Data, TSnName,TSnData)

Description
tsc = addsampletocollection(tsc,'time',Time,TS1Name,TS1Data, TSnName,TSnData) adds data samples TSnData to the collection member TSnName in the tscollection object tsc at one or more Time values. Here, TSnName is the string that represents the name of a time series in tsc, and TSnData is an array containing data samples.

Remarks
If you do not specify data samples for a time-series member in tsc, that time-series member will contain missing data at the times given by Time (for numerical time-series data), NaN values, or (for logical time-series data) false values.

When a time-series member requires Quality values, you can specify data quality codes together with the data samples by using the following syntax:

\[
\text{tsc} = \text{addsampletocollection}(\text{tsc},'\text{time}',\text{time},\text{TS1Name},... \\
\text{ts1cellarray},\text{TS2Name},\text{ts2cellarray},...)
\]

Specify data in the first cell array element and Quality in the second cell array element.

Note If a time-series member already has Quality values but you only provide data samples, 0s are added to the existing Quality array at the times given by Time.

Examples
The following example shows how to create a tscollection that consists of two timeseries objects, where one timeseries does not have quality codes and the other does. The final step of the example adds a sample to the tscollection.
1. Create two timeseries objects, ts1 and ts2.
   
   ```matlab
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0],1:5,...
                     'name','acceleration');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1],1:5,...
                     'name','speed');
   ```

2. Define a dictionary of quality codes and descriptions for ts2.
   
   ```matlab
ts2.QualityInfo.Code = [0 1];
ts2.QualityInfo.Description = {'bad','good'};
   ```

3. Assign a quality of code of 1, which is equivalent to 'good', to each data value in ts2.
   
   ```matlab
ts2.Quality = ones(5,1);
   ```

4. Create a time-series collection tsc, which includes time series ts1 and ts2.
   
   ```matlab
tsc = tscollection({ts1,ts2});
   ```

5. Add a data sample to the collection tsc at 3.5 seconds.
   
   ```matlab
tsc = addsampletocollection(tsc,'time',3.5,'acceleration',10,
                             'speed',{5 1});
   ```

   The cell array for the timeseries object 'speed' specifies both the data value 5 and the quality code 1.

**Note** If you do not specify a quality code when adding a data sample to a time series that has quality codes, then the lowest quality code is assigned to the new sample by default.

**See Also**
delsamplefromcollection, tscollection, tsprops
Purpose
Modify date number by field

Syntax
R = addtodate(D, Q, F)

Description
R = addtodate(D, Q, F) adds quantity Q to the indicated date field F of a scalar 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', 'day', 'hour', 'minute', 'second', or 'millisecond'.

If the addition to the date field causes the field to roll over, the MATLAB software 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
Modify the hours, days, and minutes of a given date:

```matlab
t = datenum('07-Apr-2008 23:00:00');
datestr(t)
ans =
    07-Apr-2008 23:00:00

t = addtodate(t, 2, 'hour');
datestr(t)
ans =
    08-Apr-2008 01:00:00

t = addtodate(t, -7, 'day');
datestr(t)
ans =
    01-Apr-2008 01:00:00

t = addtodate(t, 59, 'minute');
datestr(t)
ans =
```

2-139
01-Apr-2008 01:59:00

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

\[
R = \text{addtodate(}
\text{datenum('12/24/2007 12:45')}\), \text{20, 'day')};
\]

datestr(R)
ans =
13-Jan-1985 12:45:00

**See Also**

date, datenum, datestr, datevec
**Purpose**  
Add timeseries object to tscollection object

**Syntax**  
```
tsc = addts(tsc,ts)
tsc = addts(tsc,ts)
tsc = addts(tsc,ts,Name)
tsc = addts(tsc,Data,Name)
```

**Description**  
```
tsc = addts(tsc,ts) adds the timeseries object ts to tscollection object tsc.

tsc = addts(tsc,ts) adds a cell array of timeseries objects ts to the tscollection tsc.

tsc = addts(tsc,ts,Name) adds a cell array of timeseries objects ts to tscollection tsc. Name is a cell array of strings that gives the names of the timeseries objects in ts.

tsc = addts(tsc,Data,Name) creates a new timeseries object from Data with the name Name and adds it to the tscollection object tsc. Data is a numerical array and Name is a string.
```

**Remarks**  
The timeseries objects you add to the collection must have the same time vector as the collection. That is, the time vectors must have the same time values and units.

Suppose that the time vector of a timeseries object is associated with calendar dates. When you add this timeseries to a collection with a time vector without calendar dates, the time vectors are compared based on the units and the values relative to the StartDate property. For more information about properties, see the timeseries reference page.

**Examples**  
The following example shows how to add a time series to a time-series collection:

1. Create two timeseries objects, ts1 and ts2.

   ```
   ts1 = timeseries([1.1 2.9 3.7 4.0 3.0],1:5,...
                   'name','acceleration');
   ```
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1],1:5,...
  'name','speed');

2 Create a time-series collection tsc, which includes ts1.

    tsc = tscollection(ts1);

3 Add ts2 to the tsc collection.

    tsc = addts(tsc, ts2);

4 To view the members of tsc, type

    tsc

at the MATLAB prompt. the response is

    Time Series Collection Object: unnamed

    Time vector characteristics
    Start time 1 seconds
    End time 5 seconds

    Member Time Series Objects:
    acceleration
    speed

The members of tsc are listed by name at the bottom: acceleration and speed. These are the Name properties of the timeseries objects ts1 and ts2, respectively.

See Also
    removets, tscollection
**Purpose**
Airy functions

**Syntax**

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

**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

\[
Ai(Z) = \left[\frac{1}{\pi \sqrt[3]{Z/3}}\right] K_{1/3}(\zeta)
\]

\[
Bi(Z) = \sqrt[3]{Z/3} \left[I_{-1/3}(\zeta) + I_{1/3}(\zeta)\right]
\]

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><strong>Returns</strong></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
Align user interface controls (uicontrols) and axes

### Syntax
```matlab
align(HandleList,'HorizontalAlignment','VerticalAlignment')
Positions = align(HandleList,'HorizontalAlignment',
    'VerticalAlignment')
Positions = align(CurPositions,'HorizontalAlignment',
    'VerticalAlignment')
```

### Description
`align(HandleList,'HorizontalAlignment','VerticalAlignment')` aligns the `uicontrol` and axes objects in `HandleList`, a vector of handles, according to the options `HorizontalAlignment` and `VerticalAlignment`. The following table shows the possible values for `HorizontalAlignment` and `VerticalAlignment`.

<table>
<thead>
<tr>
<th><code>HorizontalAlignment</code></th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No horizontal alignment is made</td>
</tr>
<tr>
<td>Left</td>
<td>Shifts the objects’ left edges to that of the first object selected</td>
</tr>
<tr>
<td>Center</td>
<td>Shifts objects to center their positions to the average of the extreme x-values of the group</td>
</tr>
<tr>
<td>Right</td>
<td>Shifts the objects’ right edges to that of the first object selected</td>
</tr>
<tr>
<td>Distribute</td>
<td>Equalizes x-distances between all objects within the span of the extreme x-values</td>
</tr>
<tr>
<td>Fixed</td>
<td>Spaces objects to have a specified number of points between them in the y-direction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>VerticalAlignment</code></th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No vertical alignment is made</td>
</tr>
<tr>
<td><strong>VerticalAlignment</strong></td>
<td><strong>Definition</strong></td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Top</td>
<td>Shifts the objects’ top edges to that of the first object selected</td>
</tr>
<tr>
<td>Middle</td>
<td>Shifts objects to center their positions to the average of the extreme y-values of the group</td>
</tr>
<tr>
<td>Bottom</td>
<td>Shifts the objects’ bottom edges to that of the first object selected</td>
</tr>
<tr>
<td>Distribute</td>
<td>Equalizes y-distances between all objects within the span of the extreme y-values</td>
</tr>
<tr>
<td>Fixed</td>
<td>Spaces objects to have a specified number of points between them in the x-direction</td>
</tr>
</tbody>
</table>

Aligning objects does not change their absolute sizes. All alignment options align the objects within the bounding box that encloses the objects. Distribute and Fixed align objects to the bottom left of the bounding box. Distribute evenly distributes the objects while Fixed distributes the objects with a fixed distance (in points) between them. When you specify both horizontal and vertical distance together, the keywords 'HorizontalAlignment' and 'VerticalAlignment' are not necessary.

If you use Fixed for Horizontal Alignment or Vertical Alignment, then you must specify the distance, in points, as an extra argument. These are some examples:

```matlab
align(HandleList, 'Fixed', Distance, 'VerticalAlignment')
```

distributes the specified components Distance points horizontally and aligns them vertically as specified.

```matlab
align(HandleList, 'HorizontalAlignment', 'Fixed', Distance)
```

aligns the specified components horizontally as specified and distributes them Distance points vertically.
align(HandleList,'Fixed',HorizontalDistance,...
     'Fixed',VerticalDistance)

distributes the specified components HorizontalDistance points horizontally and distributes them VerticalDistance points vertically.

**Note** 72 points equals 1 inch.

Positions = align(HandleList,'HorizontalAlignment',
     'VerticalAlignment') returns updated positions for the specified objects as a vector of Position vectors. The position of the objects on the figure does not change.

Positions = align(CurPositions,'HorizontalAlignment',
     'VerticalAlignment') returns updated positions for the objects whose positions are contained in CurPositions, where CurPositions is a vector of Position vectors. The position of the objects on the figure does not change.

**Examples**

Create a GUI with three buttons and use align to line up the buttons.

Create a figure window and one button object.

f=figure;
u1 = uicontrol('Style','push', 'parent', f,'pos',...
     [20 100 100 100], 'string', 'button1');
Create two more button objects, not aligned with each other or any part of the figure window.

```matlab
u2 = uicontrol('Style','push', 'parent', f,'pos',[150 250 100 100], 'string', 'button2');
u3 = uicontrol('Style','push', 'parent', f,'pos',[250 100 100 100], 'string', 'button3');
```
Align the button objects with the bottom of the first button object, equalizing the distance between the objects within the span of the extreme x-values.

```
align([u1 u2 u3], 'distribute', 'bottom');
```
align

See Also  uicontrol, uistack
Purpose
Set or query axes alpha limits

Syntax
alpha_limits = alim
alim([amin amax])
alim_mode = alim('mode')
alim('alim_mode')
alim(axes_handle,...)

Description
alpha_limits = alim returns the alpha limits (the axes ALim property) of the current axes.

alim([amin amax]) 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_mode = alim('mode') returns the alpha limits mode (the axes ALimMode property) of the current axes.

alim('alim_mode') sets the alpha limits mode on the current axes.
alim_mode can be

• auto — The MATLAB software automatically sets the alpha limits based on the alpha data of the objects in the axes.

• manual — MATLAB does not change the alpha limits.

alim(axes_handle,...) operates on the specified axes.

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**
Determine whether all array 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 1 (true).

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

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

**Examples**
Given

A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69]

then B = (A < 0.5) returns logical 1 (true) 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:

```matlab
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. If handle_list is a vector of handles, the output is a cell array.

Examples
Compare the results returned by these two statements.

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

See Also
findall, findobj
Purpose
Set transparency properties for objects in current axes

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

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 Transparency” 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 — Alpha Data” for more information.

**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” on page 1-108 for related functions

“Transparency” in 3-D Visualization for examples
Purpose
Specify figure alphamap (transparency)

Syntax
alphamap
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, FaceVertexAlphaData, AlphaDataMapping

Surface: FaceAlpha, AlphaData, AlphaDataMapping

Transparency for related functions

“Transparency” in 3-D Visualization for examples
Purpose
Approximate minimum degree permutation

Syntax

\[
P = \text{amd}(A) \\
P = \text{amd}(A, \text{opts})
\]

Description

\[
P = \text{amd}(A)
\]
returns the approximate minimum degree permutation vector for the sparse matrix \( C = A + A' \). The Cholesky factorization of \( C(P,P) \) or \( A(P,P) \) tends to be sparser than that of \( C \) or \( A \). The \text{amd} function tends to be faster than \text{symamd}, and also tends to return better orderings than \text{symamd}. Matrix \( A \) must be square. If \( A \) is a full matrix, then \text{amd}(A) \) is equivalent to \text{amd}(\text{sparse}(A))

\[
P = \text{amd}(A, \text{opts})
\]
allows additional options for the reordering. The \text{opts} input is a structure with the two fields shown below. You only need to set the fields of interest:

- **dense** — A nonnegative scalar value that indicates what is considered to be dense. If \( A \) is \( n \)-by-\( n \), then rows and columns with more than \( \max(16, (\text{dense} \times \sqrt{n})) \) entries in \( A + A' \) are considered to be "dense" and are ignored during the ordering. MATLAB software places these rows and columns last in the output permutation. The default value for this field is 10.0 if this option is not present.

- **aggressive** — A scalar value controlling aggressive absorption. If this field is set to a nonzero value, then aggressive absorption is performed. This is the default if this option is not present.

MATLAB software performs an assembly tree post-ordering, which is typically the same as an elimination tree post-ordering. It is not always identical because of the approximate degree update used, and because “dense” rows and columns do not take part in the post-order. It well-suited for a subsequent \text{chol} operation, however, If you require a precise elimination tree post-ordering, you can use the following code:

\[
P = \text{amd}(S); \\
C = \text{spones}(S) + \text{spones}(S'); % Skip this line if S is already symmetric \[\text{[ignore, Q]} = \text{etree}(C(P,P));
\]
This example constructs a sparse matrix and computes two Cholesky factors: one of the original matrix and one of the original matrix preordered by `amd`. Note how much sparser the Cholesky factor of the preordered matrix is compared to the factor of the matrix in its natural ordering:

```matlab
A = gallery('wathen',50,50);
p = amd(A);
L = chol(A,'lower');
Lp = chol(A(p,p),'lower');
```

```matlab
figure;
subplot(2,2,1); spy(A);
title('Sparsity structure of A');
subplot(2,2,2); spy(A(p,p));
title('Sparsity structure of AMD ordered A');
subplot(2,2,3); spy(L);
title('Sparsity structure of Cholesky factor of A');
subplot(2,2,4); spy(Lp);
title('Sparsity structure of Cholesky factor of AMD ordered A');
```

```
set(gcf,'Position',[100 100 800 700]);
```

### See Also
`colamd`, `colperm`, `symamd`, `symrcm`, `/`

### References
AMD Version 1.2 is written and copyrighted by Timothy A. Davis, Patrick R. Amestoy, and Iain S. Duff. It is available at http://www.cise.ufl.edu/research/sparse/amd.

The authors of the code for `symamd` are Stefan I. Larimore and Timothy A. Davis (davis@cise.ufl.edu), University of Florida. The algorithm was developed in collaboration with John Gilbert,
Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. Sparse Matrix Algorithms Research at the University of Florida: http://www.cise.ufl.edu/research/sparse/
**Purpose**

Ancestor of graphics object

**Syntax**

```matlab
p = ancestor(h,type)
p = ancestor(h,type,'toplevel')
```

**Description**

`p = ancestor(h,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 the MATLAB software 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 = ancestor(h,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'});
g p(p,'Type')
an s =

hggroup
```

Now get the top-level ancestor
p=ancestor(hgg,{'figure','axes','hgroup'},'toplevel');
get(p,'type')
ans =

figure

See Also

findobj
**Purpose**
Find logical AND of array or scalar inputs

**Syntax**
A & B & ...  
and(A, B)

**Description**
A & B & ... performs a logical AND of all input arrays A, B, etc., and returns an array containing elements set to either logical 1 (true) or logical 0 (false). An element of the output array is set to 1 if all input arrays contain a nonzero element at that same array location. Otherwise, that element is set to 0.

Each input of the expression can be an array or can be a scalar value. All nonscalar input arrays must have equal dimensions. If one or more inputs are an array, then the output is an array of the same dimensions. If all inputs are scalar, then the output is scalar.

If the expression contains both scalar and nonscalar inputs, then each scalar input is treated as if it were an array having the same dimensions as the other input arrays. In other words, if input A is a 3-by-5 matrix and input B is the number 1, then B is treated as if it were a 3-by-5 matrix of ones.

and(A, B) is called for the syntax A & B when either A or B is an object.

**Note**
The symbols & and && perform different operations in the MATLAB software. The element-wise AND operator described here is &. The short-circuit AND operator is &&.

**Examples**
If matrix A is

```
 0.4235  0.5798   0   0.7942   0  
 0.5155   0   0.7833  0.0592  0.8744  
 0.3340   0   0   0  0.0150  
 0.4329  0.6405  0.6808  0.0503  0  
```

and matrix B is

```
2 166
```
\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 \\
\end{array} \]

then

A & B
ans =
\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{array} \]

See Also bitand, or, xor, not, any, all, logical operators, logical types, bitwise functions
Purpose
Phase angle

Syntax
P = angle(Z)

Description
P = angle(Z) returns the phase angles, in radians, for each element of complex array Z. The angles lie between ±π.

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

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

and the statement

\[ Z = R \cdot \exp(i \cdot \theta) \]

converts back to the original complex Z.

Examples
Z = [
  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
]

P = angle(Z)

P =

\[
\begin{bmatrix}
  -0.7854 & 0.4636 & -0.3218 & 0.2450 \\
  1.1071 & -0.7854 & 0.5880 & -0.4636 \\
 -1.2490 & 0.9828 & -0.7854 & 0.6435 \\
  1.3258 & -1.1071 & 0.9273 & -0.7854 \\
\end{bmatrix}
\]

Algorithm
The angle function can be expressed as \( \text{angle}(z) = \text{imag}(\log(z)) = \text{atan2}(\text{imag}(z), \text{real}(z)) \).

See Also
abs, atan2, unwrap
**Purpose**
Create annotation objects

**GUI Alternatives**
Create several types of annotations with the Figure Palette and modify annotations with the Property Editor, components of the plotting tools. Directly manipulate annotations in plot edit mode. For details, see “How to Annotate Graphs” and “Working in Plot Edit Mode” in the MATLAB Graphics documentation.

**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 text box),
- `'textbox'`
- `'ellipse'`
- `'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 text box.

annotation('textbox',[x y w h]) creates an editable text box 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 in the text box, 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 nor 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.

**Deleting Annotations**

Existing annotations persist on a plot when you replace its data. This might not be what you want to do. If it is not, or if you want to remove annotation objects for any reason, you can do so manually, or sometimes programmatically, in several ways:

- To manually delete, click the `Edit Plot` tool or invoke `plottools`, select the annotation(s) you want to remove, and do one of the following:
  - Press the `Delete` key.
  - Press the `Backspace` key.
  - Select `Clear` from the `Edit` menu.
  - Select `Delete` from the context menu (one annotation at a time).
- If you obtained a handle for the annotation when you created it, use the `delete` function:

  `delete(anno_obj_handle)`

There is no reliable way to obtain handles for annotations from a figure’s property set; you must keep track of them yourself.
To delete all annotations at once (as well as all plot contents), type

clf

**Normalized Coordinates**

By default, annotation objects use normalized 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 and proportions. Set the *Units* property of annotation objects to change their coordinates from normalized to inches, centimeters, points, pixels, or characters.

When their *Units* property is other than normalized, annotation objects have absolute positions with respect to the figure’s origin, and fixed sizes. Therefore, they will shift position with respect to axes when you resize figures. When units are normalized, annotations shrink and grow when you resize figures; this can cause lines of text in textbox annotations to wrap. However, if you set the *FontUnits* property of an annotation textbox object to normalized, the text changes size rather than wraps if the textbox size changes.

You can use either the **set** command or the **Inspector** to change a selected annotation object’s *Units* property:

```matlab
set(gco,'Units','inches') % or
inspect(gco)
```

For more information see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

**See Also**

Properties for the annotation objects **Annotation Arrow Properties**, **Annotation Doublearrow Properties**, **Annotation Ellipse Properties**, **Annotation Line Properties**, **Annotation Rectangle Properties**, **Annotation Textarrow Properties**, **Annotation Textbox Properties**

See “Annotating Graphs” and “Annotation Objects” for more information.
Annotation Arrow Properties

Purpose
Define annotation arrow 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. For an example of its use, see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

Annotation Arrow Property Descriptions

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

Color
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object’s color.

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

HeadLength
scalar value in points

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

HeadStyle
select string from list

Style of the arrowhead. Specify this property as one of the strings from the following 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>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>

**HeadWidth**

Scalar value in points

*Width of the arrowhead.* 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 object. Available line styles are shown in the following table.

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

**LineWidth**

scalar

*The width of linear objects and edges of filled areas.* Specify this value in points (1 point = $\frac{1}{72}$ inch). The default LineWidth is 0.5 points.

**Position**

four-element vector $[x, y, \text{width, height}]$

*Size and location of the object.* Specify the lower left corner of the object with the first two elements of the vector defining the point $x$, $y$ in units normalized to the figure (when Units property is normalized). The third and fourth elements specify the object’s $dx$ and $dy$, respectively, in units normalized to the figure.

**Units**

{normalized} | inches | centimeters | points | pixels

*position units.* MATLAB uses this property to determine the units used by the Position property. All positions are measured from the lower left corner of the figure window. Normalized units interpret Position as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the
size of the object accordingly. pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).

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

*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.

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

*Y-coordinates of the beginning and ending points for line.* 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.
**Purpose**
Define 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. For an example of its use, see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

**Annotation Doublearrow Property Descriptions**

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

**Color**
ColorSpec

*Color of the object.* A three-element RGB vector or one of the MATLAB predefined names, specifying the object’s color.

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

**Head1Length**
scalar value in points

*Length of the first arrowhead.* Specify this property in points (1 point = 1/72 inch). See also `Head1Width`.

The first arrowhead 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 arrowhead.* Specify this property in points (1 point = 1/72 inch). See also `Head1Width`. 
The first arrowhead is located at the end defined by the point \( x(\text{end}) \), \( y(\text{end}) \). See also the \( X \) and \( Y \) properties.

**Head1Style**
select string from list

*Style of the first arrowhead.* Specify this property as one of the strings from the following table.

**Head2Style**
select string from list

*Style of the second arrowhead.* 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>
</tbody>
</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>cback2</td>
<td>→</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cback3</td>
<td>→</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Head1Width**
scalar value in points

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

**Head2Width**
scalar value in points

*Width of the second arrowhead.* Specify this property in points (1 point = 1/72 inch). See also Head2Length.

**LineStyle**
{-} | − | : | . | none

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

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

LineWidth
scalar

*The width of linear objects and edges of filled areas.* Specify this value in points (1 point = \(1/72\) inch). The default LineWidth is 0.5 points.

Position
four-element vector \([x, y, \text{width}, \text{height}]\)

*Size and location of the object.* Specify the lower left corner of the object with the first two elements of the vector defining the point \(x, y\) in units normalized to the figure (when Units property is normalized). The third and fourth elements specify the object’s \(dx\) and \(dy\), respectively, in units normalized to the figure.

Units
{normalized} | inches | centimeters | points | pixels

*position units.* MATLAB uses this property to determine the units used by the Position property. All positions are measured from the lower left corner of the figure window. Normalized units interpret Position as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. pixels, inches, centimeters, and points are absolute units (1 point = \(1/72\) inch).

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

*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.

\(Y\)
vector \([Y_{\text{begin}} \ Y_{\text{end}}]\)
**Annotation Doublearrow Properties**

*Y-coordinates of the beginning and ending points for line.* 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.
Annotation Ellipse Properties

Purpose
Define annotation ellipse 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. For an example of its use, see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

Annotation Ellipse Property Descriptions

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

EdgeColor
ColorSpec {{0 0 0}} | none |

Color of the object’s edges. 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
{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.
See the ColorSpec reference page for more information on specifying color.

**LineStyle**

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

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

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

**LineWidth**

*scalar*

*The width of linear objects and edges of filled areas*. Specify this value in points (1 point = \(\frac{1}{72}\) inch). The default LineWidth is 0.5 points.

**Position**

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

*Size and location of the object*. Specify the lower left corner of the object with the first two elements of the vector defining the point \(x, y\) in units normalized to the figure (when Units property is normalized). The third and fourth elements specify the object’s \(dx\) and \(dy\), respectively, in units normalized to the figure.

**Units**

```
{normalized} | inches | centimeters | points | pixels
```
position units. MATLAB uses this property to determine the units used by the Position property. All positions are measured from the lower left corner of the figure window. Normalized units interpret Position as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).
Annotation Line Properties

Purpose
Define 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. For an example of its use, see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

Annotation Line Property Descriptions

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

Color
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object’s color.

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

LineStyle
{-} | – | : | -. | none

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

<table>
<thead>
<tr>
<th>Specifier String</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Solid line (default)</td>
</tr>
<tr>
<td>--</td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
</tbody>
</table>
Annotation Line Properties

<table>
<thead>
<tr>
<th>Specifier String</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>-.</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>

**LineWidth**
scalar

*The width of linear objects and edges of filled areas.* Specify this value in points (1 point = \(\frac{1}{72}\) inch). The default LineWidth is 0.5 points.

**Position**
four-element vector \([x, y, \text{width}, \text{height}]\)

*Size and location of the object.* Specify the lower left corner of the object with the first two elements of the vector defining the point \(x, y\) in units normalized to the figure (when Units property is normalized). The third and fourth elements specify the object’s \(dx\) and \(dy\), respectively, in units normalized to the figure.

**Units**

\{'normalized\} | inches | centimeters | points | pixels

*position units.* MATLAB uses this property to determine the units used by the Position property. All positions are measured from the lower left corner of the figure window. Normalized units interpret Position as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. pixels, inches, centimeters, and points are absolute units (1 point = \(\frac{1}{72}\) inch).

**X**

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

*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.

\[ Y \]

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

*Y-coordinates of the beginning and ending points for line.* 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.
Annotation Rectangle Properties

**Purpose**
Define annotation rectangle 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. For an example of its use, see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

**Annotation Rectangle Property Descriptions**

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

- **EdgeColor**
  
  ColorSpec `{{0 0 0}}` | none |
  
  *Color of the object’s edges.* 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.

- **FaceAlpha**
  
  Scalar alpha value in range `[0 1]`
  
  *Transparency of object background.* This property defines the degree to which the object’s background color is transparent. A value of 1 (the default) makes the background completely transparent (i.e., invisible). The default FaceAlpha is 1.

- **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.

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

**LineStyle**

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

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

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

**LineWidth**

`scalar`

*The width of linear objects and edges of filled areas.* Specify this value in points (1 point = 1/72 inch). The default LineWidth is 0.5 points.
Position
four-element vector [x, y, width, height]

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point x, y in units normalized to the figure (when Units property is normalized). The third and fourth elements specify the object’s dx and dy, respectively, in units normalized to the figure.

Units
{normalized} | inches | centimeters | points | pixels

position units. MATLAB uses this property to determine the units used by the Position property. All positions are measured from the lower left corner of the figure window. Normalized units interpret Position as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).
Annotation Textarrow Properties

Purpose

Define annotation textarrow 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. For an example of its use, see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

Annotation Textarrow Property Descriptions

Properties You Can Modify

This section lists the properties you can modify on an annotation textarrow 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 to display the text box, set the `TextEdgeColor` to `none`.

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

**FontAngle**

{normal} | italic | oblique

*Character slant.* MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

**FontName**

A name, such as *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).

**FontUnits**

{points} | normalized | inches | centimeters | pixels

*Font size units.* MATLAB uses this property to determine the units used by the `FontSize` property. Normalized units interpret `FontSize` as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

**FontWeight**

light | {normal} | demi | bold
**Annotation Textarrow Properties**

*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 arrowhead.* Specify this property in points (1 point = 1/72 inch). See also *HeadWidth*.

**HeadStyle**
select string from list

*Style of the arrowhead.* 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><img src="image" alt="arrow head" /></td>
<td>rectangle</td>
<td><img src="image" alt="rectangle head" /></td>
</tr>
<tr>
<td>ellipse</td>
<td><img src="image" alt="ellipse head" /></td>
<td>diamond</td>
<td><img src="image" alt="diamond head" /></td>
</tr>
<tr>
<td>vback1</td>
<td><img src="image" alt="vback1 head" /></td>
<td>rose</td>
<td><img src="image" alt="rose head" /></td>
</tr>
<tr>
<td>vback2 (Default)</td>
<td><img src="image" alt="vback2 head" /></td>
<td>hypocycloid</td>
<td><img src="image" alt="hypocycloid head" /></td>
</tr>
<tr>
<td>vback3</td>
<td><img src="image" alt="vback3 head" /></td>
<td>astroid</td>
<td><img src="image" alt="astroid head" /></td>
</tr>
<tr>
<td>cback1</td>
<td><img src="image" alt="cback1 head" /></td>
<td>deltoid</td>
<td><img src="image" alt="deltoid head" /></td>
</tr>
</tbody>
</table>
Annotation Textarrow Properties

<table>
<thead>
<tr>
<th>Head Style String</th>
<th>Head Style String</th>
<th>Head Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>cback2</td>
<td></td>
<td>scalar value in points</td>
</tr>
<tr>
<td>cback3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

HeadWidth
scalar value in points

Width of the arrowhead. 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 justification of the text string. It determines where MATLAB places the string with regard to the point specified by the Position property. The following picture illustrates the alignment options.

HorizontalAlignment viewed with the VerticalAlignment set to middle (the default).

See the Extent property for related information.

Interpreter
latex | {tex} | none
Interpret $T_\text{E}X$ instructions. This property controls whether MATLAB interprets certain characters in the String property as $T_\text{E}X$ instructions (default) or displays all characters literally. The options are:

- \texttt{latex} — Supports the full $L_\text{A}T_\text{E}X$ markup language.
- \texttt{tex} — Supports a subset of plain $T_\text{E}X$ markup language. See the String property for a list of supported $T_\text{E}X$ instructions.
- \texttt{none} — Displays literal characters.

\texttt{LineStyle}

\{\} | \_ | : \_ | . \_ | none

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

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

\texttt{LineWidth}

scalar

The width of linear objects and edges of filled areas. Specify this value in points \(1\text{ point} = \frac{1}{72}\text{ inch}\). The default LineWidth is 0.5 points.

\texttt{Position}

four-element vector \([x, y, \text{width}, \text{height}]\)
**Annotation Textarrow Properties**

*Size and location of the object.* Specify the lower left corner of the object with the first two elements of the vector defining the point \( x, y \) in units normalized to the figure (when `Units` property is normalized). The third and fourth elements specify the object’s \( dx \) and \( dy \), respectively, in units normalized to the figure.

*String*

string

*The text string.* Specify this property as a quoted string for single-line strings, or as a cell array of strings, or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as line breaks in text strings, and are drawn as part of the text string. See Mathematical Symbols, Greek Letters, and TeX Characters for an example.

When the text `Interpreter` property is set to `Tex` (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequences used to define them.

<table>
<thead>
<tr>
<th>Character Sequence</th>
<th>Symbol</th>
<th>Character Sequence</th>
<th>Symbol</th>
<th>Character Sequence</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\alpha</td>
<td>α</td>
<td>\upsilon</td>
<td>υ</td>
<td>\sim</td>
<td>~</td>
</tr>
<tr>
<td>\beta</td>
<td>β</td>
<td>\phi</td>
<td>Φ</td>
<td>\leq</td>
<td>≤</td>
</tr>
<tr>
<td>\gamma</td>
<td>γ</td>
<td>\chi</td>
<td>χ</td>
<td>\infty</td>
<td>∞</td>
</tr>
<tr>
<td>\delta</td>
<td>δ</td>
<td>\psi</td>
<td>ψ</td>
<td>\clubsuit</td>
<td>♣</td>
</tr>
<tr>
<td>\epsilon</td>
<td>ε</td>
<td>\omega</td>
<td>ω</td>
<td>\diamondsuit</td>
<td>♦</td>
</tr>
<tr>
<td>\zeta</td>
<td>ζ</td>
<td>\Gamma</td>
<td>Γ</td>
<td>\heartsuit</td>
<td>♥</td>
</tr>
<tr>
<td>\eta</td>
<td>η</td>
<td>\Delta</td>
<td>Δ</td>
<td>\spadesuit</td>
<td>♠</td>
</tr>
</tbody>
</table>
### Annotation Textarrow Properties

<table>
<thead>
<tr>
<th>Character Sequence</th>
<th>Symbol</th>
<th>Character Sequence</th>
<th>Symbol</th>
<th>Character Sequence</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\theta</td>
<td>θ</td>
<td>\Theta</td>
<td>Θ</td>
<td>\leftarrow</td>
<td>←</td>
</tr>
<tr>
<td>\vartheta</td>
<td>\Lambda</td>
<td>Λ</td>
<td>\leftarrow</td>
<td></td>
<td>←</td>
</tr>
<tr>
<td>\iota</td>
<td>ι</td>
<td>\Xi</td>
<td>Ξ</td>
<td>\uparrow</td>
<td>↑</td>
</tr>
<tr>
<td>\kappa</td>
<td>κ</td>
<td>\Pi</td>
<td>Π</td>
<td>\rightarrow</td>
<td>→</td>
</tr>
<tr>
<td>\lambda</td>
<td>λ</td>
<td>\Sigma</td>
<td>Σ</td>
<td>\downarrow</td>
<td>↓</td>
</tr>
<tr>
<td>\mu</td>
<td>µ</td>
<td>\Upsilon</td>
<td>Υ</td>
<td>\circ</td>
<td>◦</td>
</tr>
<tr>
<td>\nu</td>
<td>ν</td>
<td>\Phi</td>
<td>Φ</td>
<td>\pm</td>
<td>±</td>
</tr>
<tr>
<td>\xi</td>
<td>ξ</td>
<td>\Psi</td>
<td>Ψ</td>
<td>\geq</td>
<td>≥</td>
</tr>
<tr>
<td>\pi</td>
<td>π</td>
<td>\Omega</td>
<td>Ω</td>
<td>\propto</td>
<td>∝</td>
</tr>
<tr>
<td>\rho</td>
<td>ρ</td>
<td>\forall</td>
<td>∀</td>
<td>\partial</td>
<td>\partial</td>
</tr>
<tr>
<td>\sigma</td>
<td>σ</td>
<td>\exists</td>
<td>∃</td>
<td>\bullet</td>
<td>•</td>
</tr>
<tr>
<td>\varsigma</td>
<td>σ</td>
<td>\ni</td>
<td>_AES</td>
<td>\div</td>
<td>÷</td>
</tr>
<tr>
<td>\tau</td>
<td>τ</td>
<td>\cong</td>
<td>≈</td>
<td>\neq</td>
<td>≠</td>
</tr>
<tr>
<td>\equiv</td>
<td>≡</td>
<td>\approx</td>
<td>≃</td>
<td>\aleph</td>
<td>ℵ</td>
</tr>
<tr>
<td>\Im</td>
<td>ℤ</td>
<td>\Re</td>
<td>ℜ</td>
<td>\wp</td>
<td>∅</td>
</tr>
<tr>
<td>\otimes</td>
<td>⊗</td>
<td>\oplus</td>
<td>⊕</td>
<td>\oslash</td>
<td>∅</td>
</tr>
<tr>
<td>\cap</td>
<td>∩</td>
<td>\cup</td>
<td>∪</td>
<td>\supseteq</td>
<td>⊇</td>
</tr>
<tr>
<td>\supset</td>
<td>⊃</td>
<td>\subseteq</td>
<td>⊆</td>
<td>\subset</td>
<td>⊂</td>
</tr>
<tr>
<td>\int</td>
<td>∫</td>
<td>\in</td>
<td>∊</td>
<td>\o</td>
<td>o</td>
</tr>
<tr>
<td>\rfloor</td>
<td>\lfloor</td>
<td>\lceil</td>
<td>\nabla</td>
<td>\nabla</td>
<td>∇</td>
</tr>
<tr>
<td>\lfloor</td>
<td>\lfloor</td>
<td>\cdots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>...</td>
</tr>
<tr>
<td>\perp</td>
<td>⊥</td>
<td>\neg</td>
<td>¬</td>
<td>\prime</td>
<td>′</td>
</tr>
</tbody>
</table>
You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However, you can use \texttt{\fontname} in combination with one of the other modifiers:

**TextBackgroundColor**

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

**TextColor**

\texttt{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 \texttt{ColorSpec} reference page for more information on specifying color. Setting the \texttt{Color} property also sets this property.

**TextEdgeColor**

\texttt{ColorSpec or none Default: none}
Annotation Textarrow Properties

*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 TextLineWidth 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 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 are absolute and not relative to previous rotations; a rotation of 0 degrees is always horizontal.

**Units**

{normalized} | inches | centimeters | points | pixels

*position units.* MATLAB uses this property to determine the units used by the Position property. All positions are measured from the lower left corner of the figure window. Normalized units interpret Position as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. pixels, inches, centimeters, and points are absolute units (1 point = \(1/72\) inch).
VerticalAlignment
top | cap | {middle} | baseline | bottom

*Vertical alignment of text.* This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the Position property. The possible values mean

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

The following picture illustrates the alignment options.

*Text VerticalAlignment property viewed with the HorizontalAlignment property set to left (the default).*

```
Middle     Top     Cap
Baseline   Bottom
```
X
vector \([X_{\text{begin}} X_{\text{end}}]\)

*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.

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

*Y-coordinates of the beginning and ending points for line.* 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.
### Annotation Textbox Properties

**Purpose**

Define annotation textbox 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. For an example of its use, see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

**AnnotationTextbox Property Descriptions**

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

**Properties You Can Modify**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BackgroundColor</code></td>
<td>Color of text background rectangle. A three-element RGB vector or one of the MATLAB predefined names, specifying the rectangle background color. The default value is 'none'. See the <code>ColorSpec</code> reference page for more information on specifying color.</td>
</tr>
<tr>
<td><code>Color</code></td>
<td>Text color. A three-element RGB vector or one of the predefined names, specifying the text color. The default value is black. See <code>ColorSpec</code> for more information on specifying color.</td>
</tr>
<tr>
<td><code>EdgeColor</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
Annotation Textbox Properties

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

FaceAlpha
Scalar alpha value in range [0 1]

*Transparency of object background.* This property defines the degree to which the object’s background color is transparent. A value of 1 (the default) makes to color opaque, a value of 0 makes the background completely transparent (i.e., invisible). The default FaceAlpha is 1.

FitBoxToText
on | off

*Automatically adjust text box width and height to fit text.* When this property is on (the default), MATLAB automatically resizes textboxes to fit the x-extents and y-extents of the text strings they contain. When it is off, text strings are wrapped to fit the width of their textboxes, which can cause them to extend below the bottom of the box.

If you resize a textbox in plot edit mode or change the width or height of its position property directly, MATLAB sets the object’s FitBoxToText property to ‘off’. You can toggle this property with set, with the Property Inspector, or in plot edit mode via the object’s context menu.

FitHeightToText
on | off

*Automatically adjust text box width and height to fit text.* MATLAB automatically wraps text strings to fit the width of the text box. However, if the text string is long enough, it can extend beyond the bottom of the text box.
**Note** The `FitHeightToText` property is obsolete. To control line wrapping behavior in textboxes, use `fitBoxToText` instead.

When you set this mode to on, MATLAB automatically adjusts the height of the text box to accommodate the string, doing so as you create or edit the string.

The fit-size-to-text behavior turns off if you resize the text box programmatically or manually in plot edit mode.
However, if you resize the text box from any other handles, the position you set is honored without regard to how the text fits the box.

**FontAngle**

\{normal\} | italic | oblique

*Character slant.* MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to *italic* or *oblique* selects a slanted font.

**FontName**

A name, such as *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.*
Annotation Textbox Properties

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).

FontUnits

{points} | normalized | inches | centimeters | pixels

Font size units. MATLAB uses this property to determine the units used by the FontSize property. Normalized units interpret FontSize as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen FontSize accordingly. pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).

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.

HorizontalAlignment

{left} | center | right

Horizontal alignment of text. This property specifies the horizontal justification of the text string. It determines where MATLAB places the string with regard to the point specified by the Position property. The following picture illustrates the alignment options.

HorizontalAlignment viewed with the VerticalAlignment set to middle (the default).

Left   Center   Right
See the Extent property for related information.

**Interpreter**

latex | {tex} | none

*Interpret T\(_E\)X instructions.* This property controls whether MATLAB interprets certain characters in the String property as T\(_E\)X instructions (default) or displays all characters literally. The options are:

- latex — Supports the full L\(_A\)T\(_E\)X markup language.
- tex — Supports a subset of plain T\(_E\)X markup language. See the String property for a list of supported T\(_E\)X instructions.
- none — Displays literal characters.

**LineStyle**

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

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

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

**LineWidth**

scalar

*The width of linear objects and edges of filled areas.* Specify this value in points (1 point = \(\frac{1}{72}\) inch). The default LineWidth is 0.5 points.
Annotation Textbox Properties

Margin
dimension in pixels default: 5

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

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

*Size and location of the object.* Specify the lower left corner of the object with the first two elements of the vector defining the point x, y in units normalized to the figure (when Units property is normalized). The third and fourth elements specify the object’s dx and dy, respectively, in units normalized to the figure.

String
string

*The text string.* Specify this property as a quoted string for single-line strings, or as a cell array of strings, or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as line breaks in text strings, and are drawn as part of the text string. See Mathematical Symbols, Greek Letters, and TeX Characters for an example.

When the text Interpreter property is set to Tex (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequences used to define them.

<table>
<thead>
<tr>
<th>Character Sequence</th>
<th>Symbol</th>
<th>Character Sequence</th>
<th>Symbol</th>
<th>Character Sequence</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\alpha</td>
<td>α</td>
<td>\upsilon</td>
<td>υ</td>
<td>\sim</td>
<td>~</td>
</tr>
<tr>
<td>\beta</td>
<td>β</td>
<td>\phi</td>
<td>Φ</td>
<td>\leq</td>
<td>≤</td>
</tr>
<tr>
<td>Character Sequence</td>
<td>Symbol</td>
<td>Character Sequence</td>
<td>Symbol</td>
<td>Character Sequence</td>
<td>Symbol</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------</td>
<td>--------------------</td>
<td>--------</td>
<td>--------------------</td>
<td>--------</td>
</tr>
<tr>
<td>\gamma</td>
<td>γ</td>
<td>\chi</td>
<td>χ</td>
<td>\infty</td>
<td>∞</td>
</tr>
<tr>
<td>\delta</td>
<td>δ</td>
<td>\psi</td>
<td>ψ</td>
<td>\clubsuit</td>
<td>♠</td>
</tr>
<tr>
<td>\epsilon</td>
<td>ε</td>
<td>\omega</td>
<td>ω</td>
<td>\diamondsuit</td>
<td>♦</td>
</tr>
<tr>
<td>\zeta</td>
<td>ζ</td>
<td>\Gamma</td>
<td>Γ</td>
<td>\heartsuit</td>
<td>♥</td>
</tr>
<tr>
<td>\eta</td>
<td>η</td>
<td>\Delta</td>
<td>Δ</td>
<td>\spadesuit</td>
<td>♠</td>
</tr>
<tr>
<td>\theta</td>
<td>Θ</td>
<td>\Theta</td>
<td>Θ</td>
<td>\leftrightarrow</td>
<td>↔</td>
</tr>
<tr>
<td>\vartheta</td>
<td>θ</td>
<td>\Lambda</td>
<td>Λ</td>
<td>\leftarrow</td>
<td>←</td>
</tr>
<tr>
<td>\iota</td>
<td>ι</td>
<td>\Xi</td>
<td>Ξ</td>
<td>\uparrow</td>
<td>↑</td>
</tr>
<tr>
<td>\kappa</td>
<td>κ</td>
<td>\Pi</td>
<td>Π</td>
<td>\rightarrow</td>
<td>→</td>
</tr>
<tr>
<td>\lambda</td>
<td>λ</td>
<td>\Sigma</td>
<td>Σ</td>
<td>\downarrow</td>
<td>↓</td>
</tr>
<tr>
<td>\mu</td>
<td>µ</td>
<td>\Upsilon</td>
<td>Υ</td>
<td>\circ</td>
<td>◦</td>
</tr>
<tr>
<td>\nu</td>
<td>ν</td>
<td>\Phi</td>
<td>Φ</td>
<td>\pm</td>
<td>±</td>
</tr>
<tr>
<td>\xi</td>
<td>ξ</td>
<td>\Psi</td>
<td>Ψ</td>
<td>\geq</td>
<td>≥</td>
</tr>
<tr>
<td>\pi</td>
<td>π</td>
<td>\Omega</td>
<td>Ω</td>
<td>\propto</td>
<td>∝</td>
</tr>
<tr>
<td>\rho</td>
<td>ρ</td>
<td>\forall</td>
<td>∀</td>
<td>\partial</td>
<td>∂</td>
</tr>
<tr>
<td>\sigma</td>
<td>σ</td>
<td>\exists</td>
<td>∃</td>
<td>\bullet</td>
<td>•</td>
</tr>
<tr>
<td>\varsigma</td>
<td>ς</td>
<td>\ni</td>
<td>ℓ</td>
<td>\div</td>
<td>÷</td>
</tr>
<tr>
<td>\tau</td>
<td>τ</td>
<td>\cong</td>
<td>≅</td>
<td>\neq</td>
<td>≠</td>
</tr>
<tr>
<td>\equiv</td>
<td>≡</td>
<td>\approx</td>
<td>≈</td>
<td>\aleph</td>
<td>ℵ</td>
</tr>
<tr>
<td>\Im</td>
<td>ℑ</td>
<td>\Re</td>
<td>ℜ</td>
<td>\wp</td>
<td>⊥</td>
</tr>
<tr>
<td>\otimes</td>
<td>⊗</td>
<td>\oplus</td>
<td>⊕</td>
<td>\oslash</td>
<td>∅</td>
</tr>
<tr>
<td>\cap</td>
<td>∩</td>
<td>\cup</td>
<td>∪</td>
<td>\supseteq</td>
<td>⊃</td>
</tr>
</tbody>
</table>
### Annotation Textbox Properties

<table>
<thead>
<tr>
<th>Character Sequence</th>
<th>Symbol</th>
<th>Character Sequence</th>
<th>Symbol</th>
<th>Character Sequence</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\supset</td>
<td>⊃</td>
<td>\subseteq</td>
<td>⊆</td>
<td>\subset</td>
<td>⊂</td>
</tr>
<tr>
<td>\int</td>
<td>∫</td>
<td>\in</td>
<td>∈</td>
<td>\o</td>
<td>o</td>
</tr>
<tr>
<td>\rfloor</td>
<td>\lfloor</td>
<td>\lceil</td>
<td>\rceil</td>
<td>\nabla</td>
<td>∇</td>
</tr>
<tr>
<td>\lceil</td>
<td>\lceil</td>
<td>\cdot</td>
<td>·</td>
<td>\ldots</td>
<td>…</td>
</tr>
<tr>
<td>\perp</td>
<td>⊥</td>
<td>\neg</td>
<td>¬</td>
<td>\prime</td>
<td>’</td>
</tr>
<tr>
<td>\wedge</td>
<td>^</td>
<td>\times</td>
<td>x</td>
<td>\0</td>
<td>∅</td>
</tr>
<tr>
<td>\rceil</td>
<td>\rceil</td>
<td>\surd</td>
<td>√</td>
<td>\mid</td>
<td></td>
</tr>
<tr>
<td>\vee</td>
<td>∨</td>
<td>\varpi</td>
<td>ϖ</td>
<td>\copyright</td>
<td>©</td>
</tr>
<tr>
<td>\langle</td>
<td>∠</td>
<td>\rangle</td>
<td>∠</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However, you can use \fontname in combination with one of the other modifiers:

**Units**

{normalized} | inches | centimeters | points | pixels

*position units*. MATLAB uses this property to determine the units used by the Position property. All positions are measured from the lower left corner of the figure window. Normalized units interpret Position as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).

**VerticalAlignment**

top | cap | {middle} | baseline | bottom
**Vertical alignment of text.** This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the `Position` property. The possible values mean

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

The following picture illustrates the alignment options.

*Text `HorizontalAlignment` property viewed with the `HorizontalAlignment` property set to left (the default).*

```
Middle  Top  Cap
```

```
Baseline  Bottom
```
**Purpose**  
Most recent answer

**Syntax**  
ans

**Description**  
The MATLAB software creates the ans variable automatically when you specify no output argument.

**Examples**  
The statement

\[
2+2
\]

is the same as

\[
\text{ans} = 2+2
\]

**See Also**  
display
**Purpose**
Determine whether any array elements are nonzero

**Syntax**

```matlab
B = any(A)
B = any(A,dim)
```

**Description**

B = any(A) tests whether any of the elements along various dimensions of an array is a nonzero number or is logical 1 (true). any ignores entries that are NaN (Not a Number).

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

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

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

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

**Examples**

**Example 1 – Reducing a Logical Vector to a Scalar Condition**

Given

```matlab
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69]
```

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

```
0 0 1 1 1 1 0
```
The `any` function reduces such a vector of logical conditions to a single condition. In this case, `any(B)` yields logical 1.

This makes `any` particularly useful in `if` statements:

```matlab
if any(A < 0.5) do something
end
```

where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

**Example 2– Reducing a Logical Matrix to a Scalar Condition**

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

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

**Example 3 – Testing Arrays of Any Dimension**

You can use the following type of statement on an array of any dimensions. This example tests a 3-D array to see if any of its elements are greater than 3:

```matlab
x = rand(3,7,5) * 5;
any(x(:) > 3)
ans =
    1
```

or less than zero:

```matlab
any(x(:) < 0)
ans =
    0
```

**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.
### Purpose
Filled area 2-D plot

![Area Plot](image)

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

### Syntax

```plaintext
area(Y)
area(X,Y)
area(...,basevalue)
area(...,'PropertyName',PropertyValue,...)
area(axes_handle,...)
h = area(...)
hpatches = 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)`. If X is a matrix, `size(X)` must equal `size(Y)`.
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_handle,...) plots into the axes with the handle axes_handle instead of into 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.

**Note** The v6 option enables users of MATLAB Version 7.x of to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

**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.

Some areaseries object properties that you set on an individual areaseries object set the values 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. 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.
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 = [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'
```
$Y(3,1) = -1; \text{ % Was 1}$

```matlab
h = area(Y);
set(gca,'Layer','top')
grid on
colormap summer
```

The area graph now looks like this:

![Area Graph with BaseValue](image)

Adjusting the `BaseValue` property improves the appearance of the graph:

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

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” on page 1-95 for related functions

“Area Graphs” for more examples

Areaseries Properties for property descriptions
**Areaseries Properties**

**Purpose**
Define 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.

**Annotation**

`hg.Annotation` object Read Only

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

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

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

<table>
<thead>
<tr>
<th><strong>IconDisplayStyle Value</strong></th>
<th><strong>Purpose</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Include the areaseries object in a legend as one entry, but not its children objects</td>
</tr>
<tr>
<td>off</td>
<td>Do not include the areaseries or its children in a legend (default)</td>
</tr>
<tr>
<td>children</td>
<td>Include only the children of the areaseries as separate entries in the legend</td>
</tr>
</tbody>
</table>
Setting the IconDisplayStyle Property

These commands set the IconDisplayStyle of a graphics object with handle hobj to children, which causes each child object to have an entry in the legend:

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

Using the IconDisplayStyle Property

See “Controlling Legends” for more information and examples.

BaseValue
double: y-axis value

*Value where filled area base is drawn.* Specify the value along the y-axis at which the MATLAB software draws the baseline of the bottommost 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}
**Areaseries Properties**

*Callback routine interruption.* The **BusyAction** property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the **Interruptible** property of the object whose callback is executing is set to **on** (the default), then interruption occurs at the next point where the event queue is processed. If the **Interruptible** property is **off**, the **BusyAction** property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

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

**ButtonDownFcn**

*string or function handle*

*Button press callback function.* A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the **HitTestArea** property for information about selecting objects of this type.

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

This property can be:

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

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.
See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

**Children**
array of graphics object handles

*Children of this object.* The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object’s `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object’s `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

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

**Clipping**
{on} | off

*Clipping mode.* MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

**CreateFcn**
string or function handle

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

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

where `@CallbackFcn` is a function handle that references the callback function.
MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

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

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

**DeleteFcn**
string or function handle

*Callback executed during object deletion.* A callback that executes when this object is deleted (e.g., this might happen when you issue a delete command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

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

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

See the BeingDeleted property for related information.

**DisplayName**
string (default is empty string)

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

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

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

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

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

See “Controlling Legends” for more examples.

**EdgeColor**

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

*Color of line that separates filled areas.* You can set the color of the edges of 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 objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.
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:

- **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.

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

HandleVisibility
{on} | callback | off

Control access to object’s handle by command-line users and GUIs. This property determines when an object’s handle is visible in its parent’s list of children. HandleVisibility is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- **on** — Handles are always visible when HandleVisibility is on.
- **callback** — Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions.
invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.

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

### Functions Affected by Handle Visibility

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

### Properties Affected by Handle Visibility

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

### Overriding Handle Visibility

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

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

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

**HitTest**

{on} | off

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

**HitTestArea**

on | {off}

*Select 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 object’s callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the ButtonDownFcn property are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback only when it encounters a drawnow, figure, getframe, or pause command in the routine. See the BusyAction property for related information.

Setting Interruptible to on allows any graphics object’s callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the gca or gcf command) when an interruption occurs.

**LineStyle**

```
{-} | – | : | -. | none
```

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

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

**LineWidth**

Scalar
The width of linear objects and edges of filled areas. Specify this value in points (1 point = 1/72 inch). The default LineWidth is 0.5 points.

Parent

handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object’s parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

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

Selected

on | {off}

Is object selected? When you set this property to on, MATLAB displays selection "handles" at the corners and midpoints if the SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

SelectionHighlight

{on} | off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

Tag

string

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

For example, you might create an areaseries object and set the Tag property.

\[ t = \text{area}(Y,'Tag','area1') \]

When you want to access objects of a given type, you can use findobj to find the object’s handle. The following statement changes the FaceColor property of the object whose Tag is area1.

\[ \text{set(findobj('Tag','area1'),'FaceColor','red')} \]

**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.

\[ t = \text{findobj(gca,'Type','hggroup');} \]

**UIContextMenu**

handle of a uicontextmenu object

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

**UserData**

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

Visible
{on} | off

Visibility of this object and its children. By default, a new object’s visibility is on. This means all children of the object are visible unless the child object’s Visible property is set to off. Setting an object’s Visible property to off prevents the object from being displayed. However, the object still exists and you can set and query its properties.

XData
vector or matrix

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

You can use XData to define meaningful coordinates for an underlying surface whose topography is being mapped. See for more information.

XDataMode
{auto} | manual

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

If you set XDataMode to auto after having specified XData, MATLAB resets the x-axis ticks to 1:size(YData,1) or to the
column indices of the ZData, overwriting any previous values for XData.

XDataSource
string (MATLAB variable)

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

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

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

See the refreshdata reference page for more information.

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

YData
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 \( YData \).

\textbf{YDataSource}

string (MATLAB variable)

\textit{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.

\textbf{Note} If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
Purpose
Apply function to each element of array

Syntax
A = arrayfun(fun, S)
A = arrayfun(fun, S, T, ...)
[A, B, ...] = arrayfun(fun, S, ...)
[A, ...] = arrayfun(fun, S, ..., 'param1', value1, ...)

Description
A = arrayfun(fun, S) applies the function specified by fun to each element of array S, and returns the results in array A. The value A returned by arrayfun is the same size as S, and the (I,J,...)th element of A is equal to fun(S(I,J,...)). The first input argument fun is a function handle to a function that takes one input argument and returns a scalar value. fun must return values of the same class each time it is called.

If fun is bound to more than one built-in or M-file (that is, if it represents a set of overloaded functions), then the class of the values that arrayfun actually provides as input arguments to fun determines which functions are executed.

The order in which arrayfun computes elements of A is not specified and should not be relied upon.

A = arrayfun(fun, S, T, ...) evaluates fun using elements of the arrays S, T, ... as input arguments. The (I,J,...)th element of A is equal to fun(S(I,J,...), T(I,J,...), ...). All input arguments must be of the same size.

[A, B, ...] = arrayfun(fun, S, ...) evaluates fun, which is a function handle to a function that returns multiple outputs, and returns arrays A, B, ..., each corresponding to one of the output arguments of fun. arrayfun calls fun each time with as many outputs as there are in the call to arrayfun. fun can return output arguments having different classes, but the class of each output must be the same each time fun is called.

[A, ...] = arrayfun(fun, S, ..., 'param1', value1, ...) enables you to specify optional parameter name and value pairs.
Parameters recognized by `arrayfun` are shown below. Enclose each parameter name with single quotes.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UniformOutput</td>
<td>A logical 1 (<code>true</code>) or 0 (<code>false</code>), indicating whether or not the outputs of <code>fun</code> can be returned without encapsulation in a cell array. If <code>true</code> (the default), <code>fun</code> must return scalar values that can be concatenated into an array. These values can also be a cell array. If <code>false</code>, <code>arrayfun</code> returns a cell array (or multiple cell arrays), where the <code>(I,J,...)</code>th cell contains the value <code>fun(S(I,J,...), ...)</code>.</td>
</tr>
<tr>
<td>ErrorHandler</td>
<td>A function handle, specifying the function that <code>arrayfun</code> is to call if the call to <code>fun</code> fails. If an error handler is not specified, <code>arrayfun</code> rethrows the error from the call to <code>fun</code>.</td>
</tr>
</tbody>
</table>

**Remarks**

The MATLAB software provides two functions that are similar to `arrayfun`; these are `structfun` and `cellfun`. With `structfun`, you can apply a given function to all fields of one or more structures. With `cellfun`, you apply the function to all cells of one or more cell arrays.

**Examples**

**Example 1 — Operating on a Single Input.**

Create a 1-by-15 structure array with fields `f1` and `f2`, each field containing an array of a different size. Make each `f1` field be unequal to the `f2` field at that same array index:

```matlab
for k=1:15
    s(k).f1 = rand(k+3,k+7) * 10;
    s(k).f2 = rand(k+3,k+7) * 10;
```
Set three \( f_1 \) fields to be equal to the \( f_2 \) field at that array index:

\[
\begin{align*}
  s(3).f2 &= s(3).f1; \\
  s(9).f2 &= s(9).f1; \\
  s(12).f2 &= s(12).f1;
\end{align*}
\]

Use `arrayfun` to compare the fields at each array index. This compares the array of \( s(1).f1 \) with that of \( s(1).f2 \), the array of \( s(2).f1 \) with that of \( s(2).f2 \), and so on through the entire structure array.

The first argument in the call to `arrayfun` is an anonymous function. Anonymous functions return a function handle, which is the required first input to `arrayfun`:

\[
\begin{align*}
  z &= \text{arrayfun}(@(x)\text{isequal}(x.f1, x.f2), s) \\
  z &= 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0
\end{align*}
\]

**Example 2 — Operating on Multiple Inputs.**

This example performs the same array comparison as in the previous example, except that it compares the same field of more than one structure array rather than different fields of the same structure array. This shows how you can use more than one array input with `arrayfun`.

Make copies of array \( s \), created in the last example, to arrays \( t \) and \( u \):

\[
\begin{align*}
  t &= s; \\
  u &= s;
\end{align*}
\]

Make one element of structure array \( t \) unequal to the same element of \( s \). Do the same with structure array \( u \):

\[
\begin{align*}
  t(4).f1(12) &= 0; \\
  u(14).f1(6) &= 0;
\end{align*}
\]

Compare field \( f_1 \) of the three arrays \( s, t, \) and \( u \):

\[
\begin{align*}
  z &= \text{arrayfun}(@(a,b,c)\text{isequal}(a.f1, b.f1, c.f1), s, t, u) \\
  z &= \end{align*}
\]
Example 3 — Generating Nonuniform Output.

Generate a 1-by-3 structure array s having random matrices in field f1:

```
rand('state', 0);
s(1).f1 = rand(7,4) * 10;
s(2).f1 = rand(3,7) * 10;
s(3).f1 = rand(5,5) * 10;
```

Find the maximum for each f1 vector. Because the output is nonscalar, specify the UniformOutput option as false:

```
sMax = arrayfun(@(x) max(x.f1), s, 'UniformOutput', false)
sMax =
   [1x4 double]       [1x7 double]       [1x5 double]
sMax{::}
ans =
   9.5013  9.2181  9.3547  8.1317
ans =
ans =
   6.8222  8.6001  8.9977  8.1797  8.385
```

Find the mean for each f1 vector:

```
sMean = arrayfun(@(x) mean(x.f1), s, ...    
                  'UniformOutput', false)
sMean =
   [1x4 double]       [1x7 double]       [1x5 double]
sMean{::}
ans =
   6.2628  6.2171  5.4231  3.3144
ans =
   1.6209  7.079  5.7696  4.6665  5.1301  5.7136  4.8099
ans =
```
Example 4 — Assigning to More Than One Output Variable.

The next example uses the `lu` function on the same structure array, returning three outputs from `arrayfun`:

```matlab
[l u p] = arrayfun(@(x)lu(x.f1), s, 'UniformOutput', false)
```

```
L =
[7x4 double]  [3x3 double]  [5x5 double]
u =
[4x4 double]  [3x7 double]  [5x5 double]
p =
[7x7 double]  [3x3 double]  [5x5 double]
```

```
l{3}
an =
1    0    0    0    0    0
0.44379    1    0    0    0
0.79398    0.79936    1    0    0
0.27799    0.066014  -0.77517    1    0
0.28353    0.85338    0.29223    0.67036    1
```

```
u{3}
an =
6.8222    3.7837    8.9977    3.4197    3.0929
0    6.9209    4.2232    1.3796    7.0124
0    0   -4.0708   -0.40607   -2.3804
0    0    0    6.8232    2.1729
0    0    0    0   -0.35098
```

```
p{3}
an =
0    0    1    0    0
0    0    0    1    0
0    0    0    0    1
1    0    0    0    0
0    1    0    0    0
```

3.8195  5.8816  6.9128  4.9022  5.9541
See Also

structfun, cellfun, spfun, function_handle, cell2mat
**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
Purpose
Inverse secant; result in radians

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

Description
\[ Y = \text{asec}(X) \] returns the inverse secant (arcsecant) for each element of \( X \).

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

Examples
Graph the inverse secant over the domains \( 1 \leq x \leq 5 \) and \(-5 \leq x \leq -1\).

\[
\begin{align*}
  x1 &= -5:0.01:-1; \\
  x2 &= 1:0.01:5; \\
  \text{plot}(x1, \text{asec}(x1), x2, \text{asec}(x2)), \text{ grid on}
\end{align*}
\]
**Definition**

The inverse secant can be defined as

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

**Algorithm**

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

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

**Description**
\( Y = \text{asech}(X) \) returns the inverse hyperbolic secant for each element of \( X \).

The \text{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 \(0.01 \leq x \leq 1\).

\[
\begin{align*}
x &= 0.01:0.001:1; \\
\text{plot}(x, \text{asech}(x)), \text{ grid on}
\end{align*}
\]

**Definition**
The hyperbolic inverse secant can be defined as
\[ \text{sech}^{-1}(z) = \cosh^{-1}\left(\frac{1}{z}\right) \]

**Algorithm**  
aasech uses FDLIBM, which was developed at SunSoft, a Sun Microsystems 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 \(-1 \leq x \leq 1\).

```matlab
x = -1:.01:1;
plot(x,asin(x)), grid on
```
**Definition**  The inverse sine can be defined as

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

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

**See Also**  asind, asinh, sin, sind, sinh
**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**
asin, asinh, sin, sind, sinh
Purpose  
Inverse hyperbolic sine

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

Description  
$Y = \text{asinh}(X)$ returns the inverse hyperbolic sine for each element of $X$. The \text{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 $-5 \leq x \leq 5$.

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

Definition  
The hyperbolic inverse sine can be defined as
asinh

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

**Algorithm**
asinh uses FDLIBM, which was developed at SunSoft, a Sun Microsystems business, by Kwok C. Ng, and others. For information about FDLIBM, see [http://www.netlib.org](http://www.netlib.org).

**See Also**
asin, asind, sin, sinh, sind
**Purpose**  
Generate error when condition is violated

**Syntax**

```
assert(expression)
assert(expression, 'errmsg')
assert(expression, 'errmsg', value1, value2, ...)
assert(expression, 'msg_id', 'errmsg', value1, value2, ...)
```

**Description**

`assert(expression)` evaluates `expression` and, if it is false, displays the error message: **Assertion Failed**.

`assert(expression, 'errmsg')` evaluates `expression` and, if it is false, displays the string contained in `errmsg`. This string must be enclosed in single quotation marks. When `errmsg` is the last input to `assert`, the MATLAB software displays it literally, without performing any substitutions on the characters in `errmsg`.

`assert(expression, 'errmsg', value1, value2, ...)` evaluates `expression` and, if it is false, displays the formatted string contained in `errmsg`. The `errmsg` string can include escape sequences such as `\t` or `\n`, as well as any of the C language conversion operators supported by the `sprintf` function (e.g., `%s` or `%d`). Additional arguments `value1`, `value2`, etc. provide values that correspond to and replace the conversion operators.

See “Formatting Strings” in the MATLAB Programming Fundamentals documentation for more detailed information on using string formatting commands.

MATLAB makes substitutions for escape sequences and conversion operators in `errmsg` in the same way that it does for the `sprintf` function.

`assert(expression, 'msg_id', 'errmsg', value1, value2, ...)` evaluates `expression` and, if it is false, displays the formatted string `errmsg`, also tagging the error with the message identifier `msg_id`. See “Message Identifiers” in the MATLAB Programming Fundamentals documentation for information.
**Examples**

This function tests input arguments using `assert`:

```matlab
function write2file(varargin)

min_inputs = 3;
assert(nargin >= min_inputs, ...
    'You must call function %s with at least %d inputs', ...
    mfilename, min_inputs)

infile = varargin{1};
assert(ischar(infile), ...
    'First argument must be a filename.');
assert(exist(infile)==0, 'File %s not found.', infile)

fid = fopen(infile, 'w');
assert(fid > 0, 'Cannot open file %s for writing', infile)

fwrite(fid, varargin{2}, varargin{3});
```

**See Also**

`error`, `eval`, `sprintf`
**Purpose**
Assign value to variable in specified workspace

**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});
```
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 `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 $-20 \leq x \leq 20$.

```matlab
x = -20:0.01:20;
plot(x,atan(x)), grid on
```

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

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

**See Also**  
atan2, tan, atand, atanh
Purpose

Four-quadrant inverse tangent

Syntax

\[ P = \text{atan2}(Y,X) \]

Description

\[ P = \text{atan2}(Y,X) \]

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 of the inputs are ignored.

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

\[ \text{atan2}(Y,X) \]

contrasts with \( \text{atan}(Y/X) \), 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,

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

\[ r = 5 \]
\[ \theta = 0.6435 \]

This is a common operation, so MATLAB software provides a function, \( \text{angle}(z) \), that computes \( \theta = \text{atan2}(|z|, \text{sign}(z)) \).

To convert back to the original complex number

\[ z = r \exp(i \theta) \]
\[ z = 4.0000 + 3.0000i \]

**Algorithm**  
atan2 uses FDLIBM, which was developed at SunSoft, a Sun Microsystems 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
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Inverse tangent; result in degrees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>( Y = \text{atand}(X) )</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>( Y = \text{atand}(X) ) is the inverse tangent, expressed in degrees, of the elements of ( X ).</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td>tand, atan</td>
</tr>
</tbody>
</table>
**Purpose**  
Inverse hyperbolic tangent

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

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

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

**Examples**  
Graph the inverse hyperbolic tangent function over the domain 
\[-1 < x < 1\].

\[
x = -0.99:0.01:0.99;
\text{plot}(x, \text{atanh}(x)), \text{grid on}
\]

**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 business, by Kwok C. Ng, and others. For information about FDLIBM, see http://www.netlib.org.

**See Also**

atan2, atan, tanh
**Purpose**

Information about audio device

**Syntax**

```matlab
devinfo = audiodevinfo
devs = audiodevinfo(IO)
name = audiodevinfo(IO, ID)
ID = audiodevinfo(IO, name)
DriverVersion = audiodevinfo(IO, ID, 'DriverVersion')
ID = audiodevinfo(IO, rate, bits, chans)
doesSupport = audiodevinfo(IO, ID, rate, bits, chans)
```

**Description**

**Note** You can use audiodevinfo only on Microsoft Windows operating systems.

`devinfo = audiodevinfo` returns a structure, `devinfo`, containing two fields, input and output. Each field is an array of structures, with each structure containing information about one of the audio input or output devices on the system. The individual device structure fields are:

- **Name** — A string indicating the name of the device.
- **DriverVersion** — A string indicating the version of the installed device driver.
- **ID** — The ID of the device.

`devs = audiodevinfo(IO)` returns the number of input or output audio devices on the system. Use an IO value of 1 to indicate input, and an IO value of 0 to indicate output.

`name = audiodevinfo(IO, ID)` returns the name of the input or output audio device identified by device ID.

`ID = audiodevinfo(IO, name)` returns the device ID of the input or output audio device identified by the given name (partial matching, case sensitive). If no audio device is found with the given name, -1 is returned.
DriverVersion = audiodevinfo(IO, ID, 'DriverVersion') returns a string indicating the driver version of the specified audio input or output device.

ID = audiodevinfo(IO, rate, bits, chans) returns the device ID of the first input or output device that supports the sample rate, number of bits, and number of channels specified by the values of rate, bits, and chans, respectively. If no supporting device is found, -1 is returned.

doesSupport = audiodevinfo(IO, ID, rate, bits, chans) returns 1 or 0 for whether or not the input or output audio device specified by ID can support the given sample rate, number of bits, and number of channels.

See Also

audioplayer, audiorecorder
**Purpose**  
Create audioplayer object

**Syntax**

```matlab
player = audioplayer(Y, Fs)
player = audioplayer(Y, Fs, nBits)
player = audioplayer(Y, Fs, nBits, ID)
player = audioplayer(R)
player = audioplayer(R, ID)
```

**Description**

**Requirements** To use all of the features of the audioplayer object, ensure that your system has 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.

`player = audioplayer(Y, Fs)` creates an audioplayer object for signal Y, using sample rate Fs. The function returns `player`, a handle to the audioplayer object. The audioplayer object supports methods and properties that you can use to control how the audio data is played.

The input signal Y can be a vector or two-dimensional array containing `single`, `double`, `int8`, `uint8`, or `int16` MATLAB data types. 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.

`player = audioplayer(Y, Fs, nBits)` creates an audioplayer object and uses nBits bits per sample for floating-point signal Y. Valid values for nBits are 8, 16, and 24 on Windows operating systems, and 8 and 16 on UNIX operating systems. The default number of bits per sample for floating-point signals is 16.

`player = audioplayer(Y, Fs, nBits, ID)` creates an audioplayer object using audio device identifier ID for output. (You can obtain the ID of a device with the `audiodevinfo` function.) If ID equals -1, the default output device is used. This option is available only on Windows operating systems.
`player = audioplayer(R)` creates an `audioplayer` object using audio recorder object `R`.

`player = audioplayer(R, ID)` creates an `audioplayer` object from audio recorder object `R` using audio device identifier `ID` for output. (You can obtain the `ID` of a device with the `audiodevinfo` function.) This option is available only on Windows operating systems.

### Remarks

#### Value Range and Data Type

The value range of the input sample depends on the MATLAB data type. The following table lists these ranges.

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

#### Object Scope

When using an `audioplayer` object inside a function, the scope of the object is limited to the duration of the function. When the function is completed, the object is cleared, so that you cannot access any of its data or callbacks after that point. For example, in this function, `play` does not block execution, so the function ends immediately after the playing starts, deleting all callbacks and data.

```plaintext
function makePlayer(waveData, Fs, timerCallback, timerPeriod)

% Create an audioplayer object
playObject = audioplayer(waveData, Fs);

% Set callback to be called every timerPeriod seconds
set(playObject, ...
```
% Start playing
play(playObject);

end

To work around this problem, you can try any of the following:

- Create the audioplayer object outside your function, before calling that function.
- Use playblocking instead of play. This synchronously blocks execution until the waveform is completed. Note, however, that the object exists only during execution of your function.
- Pass the audioplayer object back out of the function to the MATLAB workspace:

  function playObject = makeRecorder(waveData, Fs, ...
                              timerCallback, timerPeriod)

Examples

Load a sample audio file of Handel's Hallelujah Chorus, create an audioplayer object, and play back only the first 3 seconds. y contains the audio samples, and Fs is the sampling rate.

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

To stop the playback, use this command:

    stop(player); % Equivalent to player.stop

Methods

After you create an audioplayer object, you can use the following methods on that object. player represents a handle to the audioplayer object.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>play(player)</td>
<td>Starts playback from the beginning and plays to the end of audioplayer object player.</td>
</tr>
<tr>
<td>play(player, start)</td>
<td>Play audio from the sample indicated by start to the end, or from the sample indicated by start up to the sample indicated by stop. You can also specify the values of start and stop in a two-element vector range.</td>
</tr>
<tr>
<td>play(player, [start stop])</td>
<td></td>
</tr>
<tr>
<td>play(player, range)</td>
<td></td>
</tr>
<tr>
<td>playblocking(player)</td>
<td>Same as play, but does not return control until playback completes.</td>
</tr>
<tr>
<td>playblocking(player, start)</td>
<td></td>
</tr>
<tr>
<td>playblocking(player, [start stop])</td>
<td></td>
</tr>
<tr>
<td>playblocking(player, range)</td>
<td></td>
</tr>
<tr>
<td>stop(player)</td>
<td>Stops playback.</td>
</tr>
<tr>
<td>pause(player)</td>
<td>Pauses playback.</td>
</tr>
<tr>
<td>resume(player)</td>
<td>Restarts playback from where playback was paused.</td>
</tr>
<tr>
<td>isplaying(player)</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>display(player)</td>
<td>Displays all property information about audioplayer object player.</td>
</tr>
<tr>
<td>disp(player)</td>
<td></td>
</tr>
<tr>
<td>get(player)</td>
<td></td>
</tr>
</tbody>
</table>
Properties

audioplayer objects have the following properties. To specify a user-settable property, use this syntax:

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

To view a read-only property, use this syntax:

```
get(player,'property')    % Displays 'property' setting.
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitsPerSample</td>
<td>Number of bits per sample.</td>
<td>Read-only</td>
</tr>
<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>NumberOfChannels</td>
<td>Number of channels.</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>
<tr>
<td>SampleRate</td>
<td>Sampling frequency in Hz.</td>
<td>User-settable</td>
</tr>
<tr>
<td>TotalSamples</td>
<td>Total length, in samples, of the audio data.</td>
<td>Read-only</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object label string.</td>
<td>User-settable</td>
</tr>
<tr>
<td>Type</td>
<td>Name of the object’s class.</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 and Executing Callback Functions” in the MATLAB documentation. Note that for audioplayer object callbacks, eventStruct(event) is currently empty ([]).
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartFcn</td>
<td>Handle to a user-specified callback function that is executed once when playback starts.</td>
<td>User-settable</td>
</tr>
<tr>
<td>StopFcn</td>
<td>Handle to a user-specified callback function that is executed once when playback stops.</td>
<td>User-settable</td>
</tr>
<tr>
<td>TimerFcn</td>
<td>Handle to a user-specified callback function that is executed repeatedly (at TimerPeriod intervals) 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>
</tbody>
</table>

**See Also**

audiodevinfo, audiorecorder, get, methods, set, sound, wavplay, wavread, wavwrite
**Purpose**
Create audiorecorder object

**Syntax**

\[
\begin{align*}
y &= \text{audiorecorder} \\
y &= \text{audiorecorder}(\text{Fs}, \text{nbits}, \text{nchans}) \\
y &= \text{audiorecorder}(\text{Fs}, \text{nbits}, \text{channels}, \text{ID})
\end{align*}
\]

**Description**

**Requirements** To use all of the features of the audiorecorder object, ensure that your system has a properly installed and configured sound card with 8- and 16-bit I/O and support for sampling rates of up to 48 kHz.

\[
y = \text{audiorecorder}
\]
creates an 8000 Hz, 8-bit, 1–channel audiorecorder object. \(y\) is a handle to the object. The audiorecorder object supports methods and properties that you can use to record audio data.

\[
y = \text{audiorecorder}(\text{Fs}, \text{nbits}, \text{nchans})
\]
creates an audiorecorder object using the sampling rate \(\text{Fs}\) (in Hz), the sample size \(\text{nbits}\), and the number of channels \(\text{nchans}\). \(\text{Fs}\) can be any sampling rate supported by the audio hardware. Common sampling rates are 8000, 11025, 22050, and 44100 (only 44100 on Macintosh® operating systems). The value of \(\text{nbits}\) must be 8, 16, or 24 on Microsoft Windows operating systems, and 8 or 16 on UNIX operating systems. The number of channels, \(\text{nchans}\) must be 1 (mono) or 2 (stereo).

\[
y = \text{audiorecorder}(\text{Fs}, \text{nbits}, \text{channels}, \text{ID})
\]
creates an audiorecorder object using the audio device specified by its ID for input. (You can obtain the ID of a device with the audiodevinfo function.) If ID equals -1, the default input device is used. This option is available only on Windows operating systems.

**Remarks**

**Performance with Large Recordings**
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.

**Object Scope**

When using an audiorecorder object inside a function, the scope of the object is limited to the duration of the function. When the function is completed, the object is cleared, so that you cannot access any of its data or callbacks after that point. For example, in this function, record does not block execution, so the function ends immediately after the recording starts, deleting all callbacks and data.

```matlab
function makeRecorder(timerCallback, timerPeriod)

% Create an audio recorder object
recObject = audiorecorder();

% Set callback to be called every timerPeriod seconds
set(recObject, ...
    'TimerFcn', timerCallback, ...
    'TimerPeriod', timerPeriod);

% Start recording
record(recObject);

end
```

To work around this problem, you can try any of the following:

- Create the audiorecorder object outside your function, before calling that function.
- Use recordblocking instead of record. This synchronously blocks execution until recording completes. Note, however, that the object exists only during execution of your function, so you might call getaudiodata inside the function.
- Pass the audiorecorder object back out of the function to the MATLAB workspace:
function recObject = makeRecorder(timerCallback, timerPeriod)

Examples
Using a microphone, record your voice with a sample rate of 44100 Hz, 16 bits per sample, and one channel. Speak into the microphone, then pause the recording. Play back what you have recorded so far. Record some more, then stop the recording. Finally, return the recorded data to the MATLAB workspace as an int16 array.

```matlab
r = audiorecorder(44100, 16, 1);
record(r);       % speak into microphone...
pause(r);
p = play(r);     % listen
resume(r);       % speak again
stop(r);
p = play(r);     % listen to complete recording
mySpeech = getaudiodata(r, 'int16'); % get data as int16 array
```

Methods
After you create an audiorecorder object, you can use the following methods on that object. y represents the name of the returned audiorecorder object.

<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>
<tr>
<td>isrecording(y)</td>
<td>Indicates the status of recording. If 0, recording is not in progress. If 1, recording is in progress.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>play(y)</td>
<td>Creates an audioplayer, plays the recorded audio data, and returns a handle to the created audioplayer.</td>
</tr>
<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 you omit type the defaults is 'double'. For double and single, the array contains values from -1 to 1. For int8, values are from -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>getaudiodata(y,'type')</td>
<td></td>
</tr>
<tr>
<td>display(y)</td>
<td>Displays all property information about audiorecorder object y.</td>
</tr>
<tr>
<td>disp(y)</td>
<td></td>
</tr>
<tr>
<td>get(y)</td>
<td></td>
</tr>
</tbody>
</table>

**Properties**

audiorecorder objects have the following properties. To specify a user-settable property, use this syntax:

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

To view a read-only property, use this syntax:

```matlab
get(y, 'property') % Displays 'property' setting.
```
<table>
<thead>
<tr>
<th><strong>Property</strong></th>
<th><strong>Description</strong></th>
<th><strong>Type</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BitsPerSample</td>
<td>Number of bits per recorded sample.</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, CurrentSample is the next sample to be recorded with record or resume).</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>Running</td>
<td>Status of the audio recorder (‘on’ or ‘off’).</td>
<td>Read-only</td>
</tr>
<tr>
<td>SampleRate</td>
<td>Sampling frequency in Hz.</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>Type</td>
<td>Name of the object’s class.</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 properties, see “Creating and Executing Callback Functions” in the MATLAB documentation. Note that for audio object callbacks, `eventStruct(event)` is currently empty ([]).

<table>
<thead>
<tr>
<th><strong>Property</strong></th>
<th><strong>Description</strong></th>
<th><strong>Type</strong></th>
</tr>
</thead>
<tbody>
<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><strong>Property</strong></td>
<td><strong>Description</strong></td>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td><strong>NumberOfBuffers</strong></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><strong>StartFcn</strong></td>
<td>Handle to a user-specified callback function that is executed once when recording starts.</td>
<td>User-settable</td>
</tr>
<tr>
<td><strong>StopFcn</strong></td>
<td>Handle to a user-specified callback function that is executed once when recording stops.</td>
<td>User-settable</td>
</tr>
<tr>
<td><strong>Tag</strong></td>
<td>User-specified object label string.</td>
<td>User-settable</td>
</tr>
<tr>
<td><strong>TimerFcn</strong></td>
<td>Handle to a user-specified callback function that is executed repeatedly (at <strong>TimerPeriod</strong> intervals) during recording.</td>
<td>User-settable</td>
</tr>
<tr>
<td><strong>TimerPeriod</strong></td>
<td>Time, in seconds, between <strong>TimerFcn</strong> callbacks.</td>
<td>User-settable</td>
</tr>
</tbody>
</table>

**See Also**

audiodevinfo, audioplayer, get, methods, set, wavread, wavrecord, wavwrite
**Purpose**
Information about 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
**Purpose**
Read NeXT/SUN (.au) sound file

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

**Syntax**
```matlab
y = auread('aufile')
[y,Fs,bits] = auread('aufile')
[...] = auread('aufile',N)
[...] = auread('aufile',[N1 N2])
siz = auread('aufile','size')
```

**Description**
`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**

```matlab
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 new Audio/Video Interleaved (AVI) file

**Syntax**

```matlab
aviobj = avifile(filename)
aviobj = avifile(filename, 'Param1', Val1, 'Param2', Val2, ...)
```

**Description**

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

`avifile` returns a handle to an AVI file object `aviobj`. 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, 'Param1', Val1, 'Param2', Val2, ...)` creates an `avifile` object with the property values specified by parameter/value pairs. 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. This parameter can be specified only when the 'compression' parameter is set to 'MSVC', 'RLE', or 'None'</td>
<td>There is no default colormap.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Default</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
</tbody>
</table>
| 'compression' | A text string specifying the compression codec to use. On Microsoft Windows operating systems:  

- 'Indeo3'  
- 'Indeo5'  
- 'Cinepak'  
- 'MSVC'  
- 'RLE'  
- 'None'  

- To use a custom compression codec on Windows systems, 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.  

On UNIX operating systems:  

- 'None'  
|                                                       | 'Indeo5' on Windows systems.  

- 'None' on UNIX systems. |
<p>| 'fps'         | A scalar value specifying the speed of the AVI movie in frames per second (fps). | 15 fps           |
| 'keyframe'    | For compressors that support temporal compression, this is the number of key frames per second. | 2.1429 key frames per second. |</p>
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<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. You must set this parameter before calling addframe. This parameter has no effect on uncompressed movies.</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 and must be set before using addframe.</td>
<td>The default is the filename.</td>
</tr>
</tbody>
</table>

You can also use structure syntax (also called dot notation) to set `avifile` object properties. The property name must be typed in full, however, it is not case sensitive. For example, to set the quality property to 100, use the following syntax:

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

All the field names of an `avifile` object are the same as the parameter names listed in the table, except for the `keyframe` parameter. To set this property using dot notation, specify the `KeyFramePerSec` property. For example, to change the value of `keyframe` to 2.5, type

```matlab
aviobj.KeyFramePerSec = 2.5;
```

**Example**

This example uses the `avifile` function to create the AVI file `example.avi`.

```matlab
t = linspace(0, 2.5*pi, 40);
fact = 10*sin(t);
fig = figure;
aviobj = avifile('example.avi')
[x, y, z] = peaks;
for k = 1:length(fact)
    h = surf(x, y, fact(k)*z);
```
```
axis([-3 3 -3 3 -80 80])
axis off
caxis([-90 90])
F = getframe(fig);
    aviobj = addframe(aviobj,F);
end
close(fig)
    aviobj = close(aviobj);
```

**See Also**
addframe, close, movie2avi
**Purpose**
Information about 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>Field Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</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>
<tr>
<td>Quality</td>
<td>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.</td>
</tr>
<tr>
<td>VideoCompression</td>
<td>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.</td>
</tr>
<tr>
<td>Width</td>
<td>Integer indicating the width of the AVI movie in pixels</td>
</tr>
</tbody>
</table>

See also

avifile, aviread
Purpose
Read Audio/Video Interleaved (AVI) file

Syntax
mov = aviread(filename)
mov = aviread(filename, index)

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

mov has two fields, cdata and 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 of uint8 values</td>
<td>Empty</td>
</tr>
<tr>
<td>Indexed</td>
<td>Height-by-width array of uint8 values</td>
<td>m-by-3 array of double values</td>
</tr>
</tbody>
</table>

aviread supports 8-bit frames, for indexed and grayscale images, 16-bit grayscale images, or 24-bit truecolor images. Note, however, that movie only accepts 8-bit image frames; it does not accept 16-bit grayscale image frames.

mov = aviread(filename, index) reads only the frames specified by index. 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.

Note If you are using MATLAB on a Windows platform, consider using the new mmreader function, which adds support for more video formats and codecs.

See also
avifile, aviinfo, mmreader, movie
**Purpose**
Create axes graphics object

**GUI Alternatives**
To create a figure select **New > Figure** from the MATLAB Desktop or a figure’s **File** menu. To add an axes to a figure, click one of the **New Subplots** icons in the Figure Palette, and slide right to select an arrangement of new axes. For details, see “Plotting Tools — Interactive Plotting” in the MATLAB Graphics documentation.

**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 and brings the figure containing it into focus. 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, rectangle, surface, and text graphics objects.

If you want to make an axes the current axes without changing the state of the parent figure, set the `CurrentAxes` property of the figure containing the axes:

```
set(figure_handle,'CurrentAxes',axes_handle)
```
This is useful if you want a figure to remain minimized or stacked below other figures, but want to specify the current axes.

\[ h = \text{axes(...)} \] 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,

\[ h = \text{axes('Position',position_{rectangle})} \]

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,

\[ \text{position}_{rectangle} = [\text{left}, \text{bottom}, \text{width}, \text{height}] \]

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:

\[
\begin{align*}
\text{axes('position',}[.1 & .1 & .8 & .6]) \\
\text{mesh(peaks(20))}; \\
\text{axes('position',}[.1 & .7 & .8 & .2]) \\
\text{pcolor([1:10;1:10])};
\end{align*}
\]

In this example, the first plot occupies the bottom two-thirds of the figure, and the second occupies the top third.
You can set default axes properties on the figure and root object 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.

**See Also**

axis, cla, clf, figure, gca, grid, subplot, title, xlabel, ylabel, zlabel, view

“Axes Operations” on page 1-103 for related functions

“Axes Properties” for more examples
See “Types of Graphics Objects” for information on core, group, plot, and annotation objects.
Axes Properties

**Purpose**

Modify 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 let you set and query the values of properties.

To change the default values of properties, see “Setting Default Property Values” in the Handle Graphics Objects documentation.

**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 you resize the figure (interactively or during a printing or exporting operation).

See `OuterPosition` and `Position` for related properties.

See Automatic Axes Resize for a discussion of how to use axes positioning properties.

**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. MATLAB linearly interpolates data values in between
across the alphamap and clamps data values outside to either the first or last alphamap value, whichever is closest.

If the axes contains multiple graphics objects, MATLAB sets ALim to span the range of all objects’ AlphaData (or FaceVertexAlphaData for patch objects).

See the alpha function reference page for additional information.

**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 ALim when the AlphaData limits of axes children change. Setting the ALim property sets ALimMode to manual.

**AmbientLightColor**

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 AmbientLightColor. If there are light objects in the axes, the AmbientLightColor is added to the other light sources.

**AspectRatio**

(Obsolete)

This property produces a warning message when queried or changed. The DataAspectRatio[Mode] and PlotBoxAspectRatio[Mode] properties have superseded it.

**BeingDeleted**

on | {off}
This object is being deleted. The BeingDeleted property provides a mechanism 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.

See the close and delete function reference pages for related information.

Box

on | {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.

BusyAction

cancel | {queue}

Callback routine interruption. The BusyAction property lets you control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback 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 as follows:
• **cancel** — Discard the event that attempted to execute a second callback routine.

• **queue** — Queue the event that attempted to execute a second callback routine until the current callback finishes.

### ButtonDownFcn

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

*Button press callback function.* A callback that executes whenever you press a mouse button while the pointer is within the axes, but not over another graphics object parented to the axes. For 3-D views, the active area is a rectangle that encloses the axes.

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

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of axes associated with the button down event and an event structure, which is empty for this property).

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

### Some Plotting Functions Reset the ButtonDownFcn

Most MATLAB plotting functions clear the axes and reset a number of axes properties, including the *ButtonDownFcn* before plotting data. To create an interface that lets users plot data interactively, consider using a control device such as a push button (uicontrol), which plotting functions do not affect. See “Example — Using Function Handles in GUIs” for an example.

If you must use the axes *ButtonDownFcn* to plot data, then you should use low-level functions such as *line*, *patch*, and *surface*
and manage the process with the figure and axes `NextPlot` properties.

See “High-Level Versus Low-Level Functions” for information on how plotting functions behave.

See “Preparing Figures and Axes for Graphics” for more information.

**Camera Properties**

See View Control with the Camera Toolbar for information related to the Camera properties

**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.
Axes Properties

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.

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.
Axes Properties

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°).

The following table summarizes MATLAB camera behavior using various combinations of CameraViewAngleMode, CameraTargetMode, and CameraPositionMode:

<table>
<thead>
<tr>
<th>CameraViewAngleMode</th>
<th>CameraTargetMode</th>
<th>CameraPositionMode</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>
</tbody>
</table>
### Axes Properties

<table>
<thead>
<tr>
<th>CameraViewAngleMode</th>
<th>CameraTargetMode</th>
<th>CameraPositionMode</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<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>User specifies all camera properties.</td>
</tr>
</tbody>
</table>

**Children**

*vector of graphics object handles*

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 image, light, line, patch, rectangle, surface, and text. Change the order of the handles to change the stacking of the objects on the display.

The text objects used to label the x-, y-, and z-axes and the title are also children of axes, but their HandleVisibility properties are
Axes Properties

set to off. 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, its parent’s Children property does not list it. See HandleVisibility for more information.

CLim

[cmin, cmax]

Color axis limits. A two-element vector that determines how MATLAB maps the CData values of surface and patch objects to the figure’s colormap. cmin is the value of the data mapped to the first color in the colormap, and cmax is the value of the data mapped to the last color in the colormap. MATLAB linearly interpolates data values in between across the colormap and clamps data values outside to either the first or last alphamap colormap color, whichever is closest.

When CLimMode is auto (the default), MATLAB assigns cmin the minimum data value and cmax the maximum data value in the graphics object’s CData. This maps 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 CLim to span the range of all objects’ CData.

See the caxis function reference page for related information.

CLimMode

{auto} | manual

Color axis limits mode. In auto mode, MATLAB sets the CLim property to span the CData limits of the graphics objects displayed in the axes. If CLimMode is manual, MATLAB does not change
Axes Properties

the value of CLim when the CData limits of axes children change. Setting the CLim property sets this property to manual.

Clipping
{on} | off

This property has no effect on axes.

Color
{none} | ColorSpec

*Color of the axes back planes.* Setting this property to none means the axes is transparent and the figure color shows through. A ColorSpec is a three-element RGB vector or one of the MATLAB predefined names. Note that while the default value is none, the matlabrc.m file may set the axes color to a specific color.

ColorOrder
m-by-3 matrix of RGB values

*Colors to use for multiline plots.* ColorOrder is an m-by-3 matrix of RGB values that define the colors used by the plot and plot3 functions to color each line plotted. If you do not specify a line color with plot and plot3, these functions cycle through the ColorOrder to obtain the color for each line plotted. To obtain the current ColorOrder, which may be set during startup, get the property value:

```matlab
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.
Axes Properties

CreateFcn

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

Callback function executed during object creation. A callback function that executes when MATLAB creates an axes object. You must define this property as a default value for axes. For example, the statement

    set(0,'DefaultAxesCreateFcn',@ax_create)

defines a default value on the Root level that sets axes properties whenever you (or MATLAB) create an axes.

    function ax_create(src,evnt)
      set(src,'Color','b',...
         'XLim',[1 10],...
         'YLim',[0 100])
    end

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

MATLAB passes the handle of the object whose CreateFcn is being executed as the first argument to the callback function and is also accessible 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 at the last mouse click. MATLAB returns the coordinates with respect to the requested axes.

**Clicking Within the Axes — Orthogonal Projection**

The two points lie on the line that is perpendicular to the plane of the screen and passes through the pointer. This is true for both 2-D and 3-D views.

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:

$$
\begin{bmatrix}
  x_{\text{front}} & y_{\text{front}} & z_{\text{front}} \\
  x_{\text{back}} & y_{\text{back}} & z_{\text{back}}
\end{bmatrix}
$$

where $\text{front}$ defines the point nearest to the camera position. Therefore, if the `CurrentPoint` property returns the `cp` matrix, then the first row,

$$
cp(1,:)$$

specifies the point nearest the viewer and the second row,

$$
cp(2,:)$$

specifies the point furthest from the viewer.

**Clicking Outside the Axes — Orthogonal Projection**

When you click outside the axes volume, but within the figure, the returned values are:

- Back point — a point in the plane of the camera target (which is perpendicular to the viewing axis).
Axes Properties

- Front point — a point in the camera position plane (which is perpendicular to the viewing axis).

These points lie on a line that passes through the pointer and is perpendicular to the camera target and camera position planes.

**Clicking Within the Axes — Perspective Projection**

The values of the current point when using perspective projection can be different from the same point in orthographic projection because the shape of the axes volume can be different.

**Clicking Outside the Axes — Perspective Projection**

Clicking outside of the axes volume returns the front point as the current camera position at all times. Only the back point updates with the coordinates of a point that lies on a line extending from the camera position through the pointer and intersecting the camera target at the point.

**Related Information**

See Defining Scenes with Camera Graphics for information on the camera properties.

See View Projection Types for information on orthogonal and perspective projections.

See the figure CurrentPoint property for more information.

**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 you disable stretch-to-fill behavior.

<table>
<thead>
<tr>
<th>X-, Y-, Z-LimitModes</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. MATLAB modifies DataAspectRatio to achieve the requested PlotBoxAspectRatio within the limits the software selected.</td>
</tr>
</tbody>
</table>
### Axes Properties

<table>
<thead>
<tr>
<th>X-, Y-, Z-LimitModes</th>
<th>DataAspectRatio</th>
<th>PlotBoxAspectRatio</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>manual</td>
<td>auto</td>
<td>Limits chosen to span data range in all dimensions. MATLAB modifies PlotBoxAspectRatio to achieve the requested DataAspectRatio within the limits the software selected.</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>
<tr>
<td>manual</td>
<td>auto</td>
<td>auto</td>
<td>MATLAB honors limits and modifies the DataAspectRatio and PlotBoxAspectRatio as necessary.</td>
</tr>
<tr>
<td>X-, Y-, Z-Limit Modes</td>
<td>DataAspectRatio</td>
<td>PlotBoxAspectRatio</td>
<td>Behavior</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------</td>
<td>--------------------</td>
<td>----------</td>
</tr>
<tr>
<td>manual</td>
<td>auto</td>
<td>manual</td>
<td>MATLAB honors limits and PlotBoxAspectRatio and modifies DataAspectRatio as necessary.</td>
</tr>
<tr>
<td>manual</td>
<td>manual</td>
<td>auto</td>
<td>MATLAB honors limits and DataAspectRatio and modifies the PlotBoxAspectRatio as necessary.</td>
</tr>
<tr>
<td>1 manual</td>
<td>manual</td>
<td>manual</td>
<td>MATLAB selects the 2 automatic limits to honor the specified aspect ratios and limit. See &quot;Examples.&quot;</td>
</tr>
<tr>
<td>2 auto</td>
<td>manual</td>
<td>manual</td>
<td></td>
</tr>
<tr>
<td>2 or 3 manual</td>
<td>manual</td>
<td>manual</td>
<td>MATLAB honors limits and DataAspectRatio while ignoring PlotBoxAspectRatio.</td>
</tr>
</tbody>
</table>

See “Understanding Axes Aspect Ratio” for more information.

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
Axes Properties

property to manual. Changing DataAspectRatioMode to manual disables the stretch-to-fill behavior if DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all auto.

DeleteFcn

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

*Delete axes callback function.* A callback function that executes when you delete the axes object (e.g., when you issue a *delete* or *clf* command). MATLAB executes the routine before destroying the object’s properties so the callback can query these values.

MATLAB passes the handle of the object whose DeleteFcn is executing as the first argument to the callback function. The handle is also accessible 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.

DrawMode

{normal} | fast

*Rendering mode.* This property controls the way MATLAB renders 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 can produce undesirable results because it
bypasses the hidden surface elimination and object intersection handling provided by normal `DrawMode`.

When the figure `Renderer` is `zbuffer`, it ignores `DrawMode` and always provides hidden surface elimination and object intersection handling.

**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 MATLAB does not display the x-, y-, and z-axis labels 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, set `FontName` to the string `FixedWidth`:

```
set(axes_handle,'FontName','FixedWidth')
```

This eliminates the need to hardcode the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding (such as in Japan, where character sets can be multibyte). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` (note that this string is case sensitive) and rely
Axes Properties

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 and the maximum allowable font size depends on your OS. MATLAB does not display x-, y-, and z-axis text labels 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.

Note that if you set both the FontSize and the FontUnits in one function call, you must set the FontUnits property first so that MATLAB can correctly interpret the specified FontSize.

FontWeight
{normal} | bold | light | demi
Axes Properties

Select bold or normal font. The character weight for axes text. MATLAB does not display the x-, y-, and z-axis text labels 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 children. HandleVisibility is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when HandleVisibility is on.

Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

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

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

When you restrict a handle’s visibility by 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 `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).

**Interruptible**

{on} | off

*Callback routine interruption mode.* The `Interruptible` property controls whether an axes callback routine can be
interrupted by subsequently invoked callback routines. The Interruptible property only affects callback routines defined for the ButtonDownFcn. 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 whether to draw axis lines and tick marks 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 {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', '-*|:|o')
```

setsLineStyleOrder 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.

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 and LineStyleOrder**

Note that, if the axes NextPlot property is set to replace (the default), high-level functions like plot reset the LineStyleOrder property before determining the line style to use. If you want MATLAB to use a LineStyleOrder that is different from the default, set NextPlot to replacechildren.

**Specifying a Default LineStyleOrder**

You can also specify your own default LineStyleOrder. For example:

```
set(0,'DefaultAxesLineStyleOrder',{"-*","-o","-.","o"})
```

creates a default value for the axes LineStyleOrder that high-level plotting functions will not reset.

**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 = \(\frac{1}{72}\) inch).
Axes Properties

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).

The newplot function simplifies the use of the NextPlot property and is useful for 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 as follows:

    [left bottom width height]
Axes Properties

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.

When `ActivePositionProperty` is set to `OuterPosition` (the default), resizing the figure will not clip any of the text. The default value of `[0 0 1 1]` (normalized units) includes the interior of the figure.

The units property specifies all measurement units.

See the property for related information.
Axes Properties

See “Automatic Axes Resize” for a discussion of how to use axes positioning properties.

Parent

figure or uipanel handle

Axes parent. The handle of the axes’ parent object. The parent of an axes object is the figure which displays it 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 MATLAB displays graphics objects. Setting the PlotBoxAspectRatio 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
Axes Properties

manual disables stretch-to-fill behavior if DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all auto.

Position

four-element vector

*Position of axes.* A four-element vector specifying a rectangle that locates the axes within its parent container (figure or uipanel). 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 container to the lower-left corner of the rectangle. *width* and *height* are the dimensions of the rectangle. The *Units* property specifies the units for all measurements.

When you enable axes stretch-to-fill behavior (when DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all auto), MATLAB stretches the axes to fill the Position rectangle. When you disable stretch-to-fill, MATLAB makes the axes as large as possible, while obeying all other properties, without extending outside the Position rectangle.

See the OuterPosition property for related information.

See “Automatic Axes Resize” for a discussion of how to use axes positioning properties.

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 and draws parallel lines in the data 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; it displays a distant line segment 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**

*Highlights objects 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 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.

**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 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'.
Axes Properties

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

Axes position units. The units used to interpret the Position property. MATLAB measures all units from the lower left corner of the figure window.

Note The Units property controls the positioning of the axes within the figure. This property does not affect the data units used for graphing. See the axes XLim, YLim, and ZLim properties to set the limits of each axis data units.

- 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 uses characters from the default system font to define units; 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.

When specifying the units as property/value pairs during object creation, you must set the Units property before specifying the properties that you want to use these units.
Axes Properties

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 axes camera properties now controls the functionality provided by the `View` property — `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. This property applies to 2-D views only.

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. This property applies
Axes Properties

to 2–D views only. 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 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.

```matlab
set(gca,'ZDir','reverse')
```

XGrid
YGrid
ZGrid

```matlab
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.

```matlab
set(gca,'XGrid','on')
```

XLLabel
YLLabel
ZLLabel

```matlab
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:

```matlab
set(get(gca,'XLabel'),'String','axis label')
```

MATLAB places the string 'axis label' appropriately for an x-axis label and moves any text object whose handle you specify as an XLabel, YLabel, or ZLabel property to the appropriate location for the respective label.
Alternatively, you can use the xlabel, ylabel, and zlabel functions, which generally provide a simpler means to label axis lines.

Note that using a bitmapped font (e.g., Courier is usually a bitmapped font) might cause the labels to rotate improperly. As a workaround, use a TrueType font (e.g., Courier New) for axis labels. See your system documentation to determine the types of fonts installed on your system.

\begin{verbatim}
XLim
YLim
ZLim
[minimum maximum]
\end{verbatim}

Axis limits. A two-element vector specifying the minimum and maximum values of the respective axis. The data you plot determines these values.

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].

See the axis, datetick, xlim, ylim, and zlim commands to set these properties.

\begin{verbatim}
XLimMode
YLimMode
ZLimMode
{auto} | manual
\end{verbatim}

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 XLim, YLim, or ZLim property, in which case, the respective limits mode is set to manual.
Axes Properties

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

    set(gca,'XTickLabel',{'One';'Two';'Three';'Four'})

labels the first four tick marks on the x-axis and then reuses the labels for the remaining ticks.

Labels can be cell arrays of strings, padded string matrices, string vectors separated by vertical slash characters, or numeric vectors (where MATLAB implicitly converts each number to the equivalent string using num2str). All of the following are equivalent:

    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

{auto} | manual
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
{auto} | 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
axis([xmin xmax ymin ymax])
axis([xmin xmax ymin ymax zmin zmax cmin cmax])
v = axis
axis auto
axis manual
axis tight
axis fill
axis ij
axis xy
axis equal
axis image
axis square
axis vis3d
axis normal
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 default behavior to compute 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). This option 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

```matlab
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.

Keywords to axis can be combined, separated by a space (e.g., `axis tight equal`). These are evaluated from left to right, so subsequent keywords can overwrite properties set by prior ones.

**Remarks**

You can create an axes (and a figure for it) if none exists with the axis command. However, if you specify non-default limits or formatting for the axes when doing this, such as `[4 8 2 9]`, `square`, `equal`, or `image`, the property is ignored because there are no axis limits to adjust in the absence of plotted data. To use axis in this manner, you can set hold on to keep preset axes limits from being overridden.
Examples

The statements

\[
x = 0:.025:pi/2;
\]
\[
plot(x,\tan(x),'-ro')
\]

use the automatic scaling of the \textit{y-axis} based on \(y_{\text{max}} = \tan(1.57)\), which is well over 1000:

The right figure shows a more satisfactory plot after typing

\[
\text{axis}([0 \ pi/2 \ 0 \ 5])
\]
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`. 

---

**Algorithm**

- `axis x` sets the current axes `XLimMode`, `YLimMode`, and `ZLimMode` properties to 'x'.
- `axis y` sets the current axes `XLimMode`, `YLimMode`, and `ZLimMode` properties to 'y'.
- `axis z` sets the current axes `XLimMode`, `YLimMode`, and `ZLimMode` properties to 'z'.
- `axis none` sets the current axes `XLimMode`, `YLimMode`, and `ZLimMode` properties to 'none'.

The following table shows the values of the axes properties set by `axis x`, `axis y`, `axis z`, and `axis none`. 

---

**Example**

```matlab
axis([0 10 0 10])
```
<table>
<thead>
<tr>
<th>Axes Property or Behavior</th>
<th>axis equal</th>
<th>axis normal</th>
<th>axis square</th>
<th>axis image</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataAspectRatio property</td>
<td>[1 1 1]</td>
<td>not set</td>
<td>not set</td>
<td>[1 1 1]</td>
</tr>
<tr>
<td>DataAspectRatioMode property</td>
<td>manual</td>
<td>auto</td>
<td>auto</td>
<td>manual</td>
</tr>
<tr>
<td>PlotBoxAspectRatio property</td>
<td>[3 4 4]</td>
<td>not set</td>
<td>[1 1 1]</td>
<td>auto</td>
</tr>
<tr>
<td>PlotBoxAspectRatioMode property</td>
<td>manual</td>
<td>auto</td>
<td>manual</td>
<td>auto</td>
</tr>
<tr>
<td>Stretch-to-fill behavior;</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” on page 1-103 for related functions

For aspect ratio behavior, see *Related Information* in the axes properties reference page.
**Purpose**
Diagonal scaling to improve eigenvalue accuracy

**Syntax**

```matlab
[T,B] = balance(A)
[S,P,B] = balance(A)
B = balance(A)
B = balance(A,'noperm')
```

**Description**

[T,B] = balance(A) returns a similarity transformation T such that B = T\(A^{}T\), 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 roundoff error. If A is symmetric, then B == A and T is the identity matrix.

[S,P,B] = 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) = diag(S) and B(P,P) = diag(1./S)*A*diag(S).

B = balance(A) returns just the balanced matrix B.

B = balance(A,'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,

$$
\text{cond}(V) = \text{norm}(V)\text{norm}(\text{inv}(V))
$$

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, `eig(A)`, automatically balances $A$ before computing its eigenvalues. Turn off the balancing with `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.

$$A = \begin{bmatrix} 1 & 100 & 10000; & .01 & 1 & 100; & .0001 & .01 & 1 \end{bmatrix}$$

$$A =
\begin{bmatrix}
  10000 & 0 & 0 \\
  0.01 & 10000 & 0 \\
  0.0001 & 0.0001 & 0.01 \\
  0.0000 & 0.0000 & 0.0001
\end{bmatrix}$$

Balancing produces a diagonal matrix $T$ with elements that are powers of two and a balanced matrix $B$ that is closer to symmetric than $A$.

$$[T,B] = balance(A)$$

$$T =
\begin{bmatrix}
  2.0480 & 0 & 0 \\
  0 & 0.0320 & 0 \\
  0 & 0 & 0.0003
\end{bmatrix}$$

$$B =
\begin{bmatrix}
  1.0000 & 1.5625 & 1.2207 \\
  0.6400 & 1.0000 & 0.7813 \\
  0.8192 & 1.2800 & 1.0000
\end{bmatrix}$$
To see the effect on eigenvectors, first compute the eigenvectors of A, shown here as the columns of V.

\[ [V, E] = \text{eig}(A); V \]
\[ V = 
\begin{bmatrix}
-1.0000 & 0.9999 & 0.9937 \\
0.0050 & 0.0100 & -0.1120 \\
0.0000 & 0.0001 & 0.0010 \\
\end{bmatrix} \]

Note that all three vectors have the first component the largest. This indicates V is badly conditioned; in fact \( \text{cond}(V) \) is \( 8.7766\times10^3 \). 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 \( \text{cond}(V) \) is \( 1.4421 \). The ill conditioning is concentrated in the scaling matrix; \( \text{cond}(T) \) is \( 8192 \).

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 CGBA K (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 might scale them up to make them as significant as the other elements of the original matrix.

See Also

eig

References

Purpose

Plot bar graph (vertical and horizontal)

GUI

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

Syntax

```matlab
bar(Y)
bar(x,Y)
bar(...,width)
bar(...,'style')
bar(...,'bar_color')
bar(...,'PropertyName',PropertyValue,...)
bar(axes_handle,...)
barh(axes_handle,...)
```

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 up 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. The default is to scale the x-axis to the highest x-tick on the plot, (a multiple of 10, 100, etc.). If
you want the x-axis scale to end exactly at the last bar, you can use the 
default, and then, for example, type

```matlab
set(gca,'xlim',[1 length(Y)])
```

at the MATLAB prompt.

`bar(x,Y)` draws a bar for each element in `Y` at locations specified in `x`, where `x` is a vector defining the x-axis intervals for the vertical bars. The x-values can be nonmonotonic, but cannot contain duplicate values. 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. The value of `width` must be a scalar.

`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.
- `'histc'` displays the graph in histogram format, in which bars touch one another.
- `'hist'` also displays the graph in histogram format, but centers each bar over the x-ticks, rather than making bars span x-ticks as the `histc` option does.
bar, barh

**Note** When you use either the hist or histc option, you cannot also use parameter/value syntax. These two options create graphic objects that are patches rather than barseries. See “Backward-Compatible Versions” on page 2-346 for details.

`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(...,'PropertyName',PropertyValue,...)` set the named property or properties to the specified values. Properties cannot be specified when the hist or histc options are used. See the barseries property descriptions for information on what properties you can set.

`bar(axes_handle,...)` and `barh(axes_handle,...)` plot into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = bar(...)` returns a vector of handles to barseries graphics objects, one for each created. When `Y` is a matrix, `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 vector defining the y-axis intervals for horizontal bars. The `x`-values can be nonmonotonic, but cannot contain duplicate values.

**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. Patch objects are also created when the hist and histc options are used, even if the V6 option is not. See patch object properties for a discussion of the properties you can set to control the appearance of these bar graphs.
Note  The v6 option enables users of MATLAB Version 7.x of to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See
Plot Objects and Backward Compatibility for more information.

**Barseries Objects**

Creating a bar graph of an \( m \)-by-\( n \) matrix creates \( m \) groups of \( n \) barseries objects. Each barseries object contains the data for corresponding \( x \) values of each bar group (as indicated by the coloring of the bars).

Note that some barseries object properties set on an individual barseries object set the values for all barseries objects in the graph. See the barseries 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.

\[
\begin{align*}
  x &= \text{-2.9:0.2:2.9}; \\
  \text{bar}(x, \exp(-x.*x), 'r')
\end{align*}
\]
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'
```
Setting Properties with Multiobject Graphs

This example creates a graph that displays three groups of bars and contains five bar series objects. Since all bar series objects in a graph share the same baseline, you can set values using any bar series object’s Baseline property. This example uses the first handle returned in h.

```matlab
Y = randn(3,5);
h = bar(Y);
set(get(h(1),'BaseLine'),'LineWidth',2,'LineStyle',':')
colormap summer % Change the color scheme
```
See Also

bar3, ColorSpec, patch, stairs, hist

“Area, Bar, and Pie Plots” on page 1-95 for related functions

Barseries Properties

“Bar and Area Graphs” for more examples
**Purpose**

Plot 3-D bar chart

**GUI Alternatives**

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

**Syntax**

```matlab
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 vector defining the y-axis intervals for vertical bars. The x-values can be nonmonotonic, but cannot contain duplicate values. 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.

`bar3(...,LineSpec)` displays all bars using the color specified by LineSpec.

`bar3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (gca).

`h = bar3(...)` returns a vector of handles to patch graphics objects, one for each created. `bar3` creates one patch object per column in Y. When Y is a matrix, `bar3` creates one patch graphics object per column in Y.

`bar3h(...)` and `h = bar3h(...)` create horizontal bars. Y determines the bar length. The vector x is a vector defining the y-axis intervals for horizontal bars.
Examples

This example creates six subplots showing the effects of different arguments for `bar3`. The data `Y` is a 7-by-3 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')
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])
```
bar3, bar3h
See Also  
bar, LineSpec, patch

“Area, Bar, and Pie Plots” on page 1-95 for related functions
“Bar and Area Graphs” for more examples
Barseries Properties

**Purpose**

Define 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.

**Annotation**

hg.Annotation object Read Only

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

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

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

<table>
<thead>
<tr>
<th>IconDisplayStyle</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td></td>
</tr>
<tr>
<td>off</td>
<td></td>
</tr>
<tr>
<td>children</td>
<td>Include only the children of the barseries as separate entries in the legend</td>
</tr>
</tbody>
</table>
Setting the IconDisplayStyle Property

These commands set the IconDisplayStyle of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

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

Using the IconDisplayStyle Property

See “Controlling Legends” for more information and examples.

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
**Barseries Properties**

*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 the MATLAB software draws the baseline.

**BeingDeleted**

on | {off} Read Only

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

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

**BusyAction**

cancel | {queue}

2-358
Callback routine interruption. The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the Interruptible property is off, the BusyAction property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

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

**ButtonDownFcn**

string or function handle

*Button press callback function.* A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the HitTestArea property for information about selecting objects of this type.

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

This property can be:
- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.
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 this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object’s HandleVisibility property is set to callback or off, its handle does not show up in this object’s Children property unless you set the root ShowHiddenHandles property to on:

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

Clipping
{on} | off

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set Clipping to off, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set hold to on, freeze axis scaling (axis manual), and then create a larger plot object.

CreateFcn
string or function handle

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

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

where @CallbackFcn is a function handle that references the callback function.
MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

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

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

**DeleteFcn**

string or function handle

*Callback executed during object deletion.* A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

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

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

See the `BeingDeleted` property for related information.

**DisplayName**

string (default is empty string)

*String used by legend for this barseries object.* The `legend` function uses the string defined by the `DisplayName` property to label this barseries object in the legend.
Barseries Properties

- If you specify string arguments with the `legend` function, `DisplayName` is set to this barseries object’s corresponding string and that string is used for the legend.

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

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

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

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

See “Controlling Legends” for more examples.

**EdgeColor**

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

*Color of line that separates filled areas.* You can set the color of the edges of 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 objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.
Barseries 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:

- **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.

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

**HandleVisibility**

{on} | callback | off

*Control access to object’s handle by command-line users and GUIs.* This property determines when an object’s handle is visible in its parent’s list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- **on** — Handles are always visible when `HandleVisibility` is on.
- **callback** — Setting `HandleVisibility` to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions.
invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.

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

**Functions Affected by Handle Visibility**

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

**Properties Affected by Handle Visibility**

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

**Overriding Handle Visibility**

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

**Handle Validity**
Barseries 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.

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

**HitTest**

{on} | off

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

**HitTestArea**

on | {off}

*Select 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 an object’s callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the **ButtonDownFcn** property are affected by the **Interruptible** property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the **BusyAction** property for related information.

Setting **Interruptible** to **on** allows any graphics object’s callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

**LineStyle**

{\texttt{-} | \texttt{--} | \texttt{:} | \texttt{-.} | \texttt{none}}

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

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

**LineWidth**

scalar
**The width of linear objects and edges of filled areas.** Specify this value in points (1 point = 1/72 inch). The defaultLineWidth is 0.5 points.

**Parent**

handle of parent axes, hggroup, or hgtransform

*Parent of this object.* This property contains the handle of the object’s parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

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

**Selected**

on | {off}

*Is object selected?* When you set this property to on, MATLAB displays selection "handles" at the corners and midpoints if the SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

**SelectionHighlight**

{on} | off

*Objects are highlighted when selected.* When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

**ShowBaseLine**

{on} | off
Barseries Properties

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
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 barseries object and set the Tag property:

\[
t = \text{bar}(Y, 'Tag', 'bar1')
\]

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.

\[
\text{set}(\text{findobj}(\text{'Tag', 'bar1'}), \text{'FaceColor', 'red'})
\]

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.

\[
t = \text{findobj}(\text{gca}, \text{'Type', 'hggroup'});
\]

UIContextMenu
handle of a uicontextmenu object
Barseries Properties

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

UserData
array

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

Visible
{on} | off

Visibility of this object and its children. By default, a new object’s visibility is on. This means all children of the object are visible unless the child object’s Visible property is set to off. Setting an object’s Visible property to off prevents the object from being displayed. However, the object still exists and you can set and query its properties.

XData
array

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 and uses the specified values to label the x-axis.

If you set XDataMode to auto after having specified XData, MATLAB resets the x-axis ticks to 1:size(YData,1) or to the column indices of the ZData, overwriting any previous values for XData.

XDataSource
string (MATLAB variable)

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

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

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

See the refreshdata reference page for more information.

---

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

YData
scalar, vector, or matrix
Bar Series Properties

*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 XYData is a matrix, the bar function creates a "group" or a "stack" of bars for each column in the matrix. See “Bar Graph Options” in the bar, barh reference page 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.
Purpose  Converts point coordinates from barycentric to Cartesian

Syntax  \[ XC = \text{baryToCart}(TR, SI, B) \]

Description  \( XC = \text{baryToCart}(TR, SI, B) \) returns the Cartesian coordinates \( XC \) of each point in \( B \) that represents the barycentric coordinates with respect to its associated simplex \( SI \).

Inputs  
\( TR \)  Triangulation representation.
\( SI \)  Column vector of simplex indices that index into the triangulation matrix \( TR.Triangulation \)
\( B \)  \( B \) is a matrix that represents the barycentric coordinates of the points to convert with respect to the simplices \( SI \). \( B \) is of size \( m \)-by-\( k \), where \( m = \text{length}(SI) \), the number of points to convert, and \( k \) is the number of vertices per simplex.

Outputs  
\( XC \)  Matrix of cartesian coordinates of the converted points. \( XC \) is of size \( m \)-by-\( n \), where \( n \) is the dimension of the space where the triangulation resides. That is, the Cartesian coordinates of the point \( B(j) \) with respect to simplex \( SI(j) \) is \( XC(j) \).

Definitions  
A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

Example  Compute the Delaunay triangulation of a set of points.
\[
   x = [0 4 8 12 0 4 8 12]' ;
   y = [0 0 0 8 8 8 8]' ;
   dt = \text{DelaunayTri}(x,y)
\]

Compute the barycentric coordinates of the incenters.
cc = incenters(dt);
tri = dt(:,:);
subplot(1,2,1);
triplot(dt); hold on;
plot(cc(:,1), cc(:,2), '*r'); hold off;
axis equal;
title(sprintf('Original triangulation and ... reference points.\n'));

Stretch the triangulation and compute the mapped locations of the incenters on the deformed triangulation.

b = cartToBary(dt,[1:length(tri)]',cc);
y = [0 0 0 0 16 16 16 16]';
tr = TriRep(tri,x,y)
xc = baryToCart(tr, [1:length(tri)]', b);
subplot(1,2,2);
triplot(tr); hold on;
plot(xc(:,1), xc(:,2), '*r'); hold off;
axis equal;
title(sprintf('Deformed triangulation and mapped\n ... locations of the reference points.\n'));
See Also

TriRep.cartToBary
DelaunayTri.pointLocation
DelaunayTri class
Purpose
Convert base N number string to decimal number

Syntax
d = base2dec('strn', base)

Description
d = base2dec('strn', base) converts the string number strn of the
specified base into its decimal (base 10) equivalent. base must be an
integer between 2 and 36. If 'strn' is a character array, each row is
interpreted as a string in the specified base.

Examples
The expression base2dec('212', 3) converts 212₃ to decimal, returning
23.

See Also
dec2base
**Purpose**

Produce 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**  
MATLAB benchmark

**Syntax**
bench
bench(N)
bench(0)
t = bench(N)

**Description**  
bench times six different MATLAB tasks and compares the execution speed with the speed of several other computers. The six tasks are:

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>Performance Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU</td>
<td>Perform LU of a full matrix</td>
<td>Floating-point, regular memory access</td>
</tr>
<tr>
<td>FFT</td>
<td>Perform FFT of a full vector</td>
<td>Floating-point, irregular memory access</td>
</tr>
<tr>
<td>ODE</td>
<td>Solve van der Pol equation with ODE45</td>
<td>Data structures and M-files</td>
</tr>
<tr>
<td>Sparse</td>
<td>Solve a symmetric sparse linear system</td>
<td>Mixed integer and floating-point</td>
</tr>
<tr>
<td>2-D</td>
<td>Plot Bernstein polynomial graph</td>
<td>2-D line drawing graphics</td>
</tr>
<tr>
<td>3-D</td>
<td>Display animated L-shape membrane logo</td>
<td>3-D animated OpenGL graphics</td>
</tr>
</tbody>
</table>

A final bar chart shows speed, which is inversely proportional to time. The longer bars represent faster machines, and the shorter bars represent the slower ones.

bench(N) runs each of the six tasks N times.

bench(0) just displays the results from other machines.

t = bench(N) returns an N-by-6 array with the execution times.

**Remarks**  
The comparison data for other computers is stored in the following text file. Updated versions of this file are available from MATLAB Central:

http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=1836&objectType=file#
This benchmark is intended to compare performance of one particular version of MATLAB on different machines. It does not offer direct comparisons between different versions of MATLAB. The tasks and problem sizes change from version to version.

The LU and FFT tasks involve large matrices and long vectors. Machines with less than 64 megabytes of physical memory or without optimized Basic Linear Algebra Subprograms may show poor performance.

The 2-D and 3-D tasks measure graphics performance, including software or hardware support for OpenGL. The command

```
OpenGL info
```

describes the OpenGL support available on a particular machine.

Fluctuations of five or ten percent in the measured times of repeated runs on a single machine are not uncommon. Your own mileage may vary.

**See Also**

profile, profsave, mlint, mlintrpt, memory, pack, tic, cputime, rehash
**Purpose**

Bessel function of third kind (Hankel function)

**Syntax**

- `H = besselh(nu,K,Z)`
- `H = besselh(nu,Z)`
- `H = besselh(nu,K,Z,1)`
- `[H,ierr] = besselh(...)`

**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 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 besselj, and \( Y_\nu(z) \) is 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, `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 \( K = 2 \).

\[ [H,ierr] = \text{besselh}(\ldots) \] 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>\text{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 \( H^{(1)}_{0}(z) \) shown on page 359 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

It first generates the modulus contour plot

\[
[X,Y] = \text{meshgrid}(-4:0.025:2,-1.5:0.025:1.5);
H = \text{besselh}(0,1,X+i*Y);
\text{contour}(X,Y,\text{abs}(H),0:0.2:3.2), \text{hold on}
\]
then adds the contour plot of the phase of the same function.

\texttt{contour(X,Y,(180/pi)*angle(H),-180:10:180); hold off}
See Also
besselj, bessely, besseli, besselk

References
**besseli**

**Purpose**
Modified Bessel function of first kind

**Syntax**

\[
I = \text{besseli}(n, Z)
\]
\[
I = \text{besseli}(n, Z, 1)
\]
\[
[I, ierr] = \text{besseli}(\ldots)
\]

**Definitions**
The differential equation

\[
z^2 \frac{d}{dz} \left( \frac{2}{z} \frac{dy}{dz} \right) + z \frac{dy}{dz} - (z^2 + v^2)y = 0
\]

where \( v \) is a real constant, is called the *modified Bessel’s equation*, and its solutions are known as *modified Bessel functions*.

\( I_v(z) \) and \( I_{-v}(z) \) form a fundamental set of solutions of the modified Bessel’s equation for noninteger \( v \). \( I_v(z) \) is defined by

\[
I_v(z) = \left( \frac{z}{2} \right)^v \sum_{k=0}^{\infty} \frac{\left( \frac{z^2}{4} \right)^k}{k! \Gamma(v + k + 1)}
\]

where \( \Gamma(a) \) is the gamma function.

\( K_v(z) \) is a second solution, independent of \( I_v(z) \). It can be computed using \text{besselk}.

**Description**

\( I = \text{besseli}(n, Z) \) computes the modified Bessel function of the first kind, \( I_v(z) \), for each element of the array \( Z \). The order \( n \) need not be an integer, but must be real. The argument \( Z \) can be complex. The result is real where \( Z \) is positive.

If \( n \) 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 = besseli(nu,Z,1) computes
besseli(nu,Z).*exp(-abs(real(Z))).

[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 use a Fortran MEX-file to call a library developed by D.E. Amos [3] [4].

See Also

`airy`, `besselh`, `besselj`, `besselk`, `bessely`

References


Purpose  
Bessel function of first kind

Syntax  
J = besselj(nu,Z)  
J = besselj(nu,Z,1)  
[J,ierr] = besselj(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.

\( J_\nu(z) \) and \( J_{-\nu}(z) \) form a fundamental set of solutions of Bessel’s equation for noninteger \( \nu \). \( J_\nu(z) \) is defined by

\[
J_\nu(z) = \left( \frac{z}{2} \right)^\nu \sum_{k=0}^{\infty} \frac{\left( \frac{z}{2} \right)^k}{k! \Gamma(\nu + k + 1)}
\]

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 bessely.

Description  
\( J = besselj(nu,Z) \) 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 \[ \text{besselj}(\nu, Z) \times \exp(-\text{abs}\text{.imag}(Z)). \]

\[[J, \text{ierr}] = \text{besselj}(\nu, Z)\] also returns completion flags in an array the same size as \(J\).

<table>
<thead>
<tr>
<th>ierr</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\text{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 (\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 \text{besselh}, \(J_{\nu}(z)\) is \text{besselj}, and \(Y_{\nu}(z)\) is \text{bessely}. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see \text{besselh}).

**Examples**

**Example 1**

```plaintext
format long
z = (0:0.2:1)';

besselj(1,z)
```
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].

References


See Also

`besseln`, `besseli`, `besselk`, `bessely`
besselk

**Purpose**
Modified Bessel function of second kind

**Syntax**

```plaintext
K = besselk(nu, Z)
K = besselk(nu, Z, 1)
[K, ierr] = besselk(...)```

**Definitions**
The differential equation

\[ z^2 \frac{d}{dz} \left( \frac{2}{z} \frac{dy}{dz} \right) + z \frac{dy}{dz} - (z^2 + \nu^2) y = 0 \]

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 \( \Gamma(\alpha) \) is the gamma function. \( K_{\nu}(z) \) is independent of \( I_{\nu}(z) \).
\( I_{\nu}(z) \) can be computed using `besseli`.

**Description**

\( K = besselk(nu, Z) \) 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.

\[ K = \text{besselk}(\nu,Z,1) \]

computes \( \text{besselk}(\nu,Z) \cdot \exp(Z) \).

\([K,ierr] = \text{besselk}(\ldots)\) 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} 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**

```
format long
z = (0:0.2:1)';

besselk(1,z)
```

```
ans =
         Inf
    4.77597254322047
    2.18435442473269
    1.30283493976350
    0.86178163447218
    0.60190723019723
```
Example 2

besselk(3:9,(0:.2:10)',1) generates part of the table on page 424 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

Algorithm

The besselk function uses a Fortran MEX-file to call a library developed by D.E. Amos [3][4].

References


See Also

airy, besselh, besseli, besselj, bessely
**Purpose**  
Bessel function of second kind

**Syntax**  
Y = bessely(nu,Z)  
Y = bessely(nu,Z,1)  
[Y,ierr] = bessely(nu,Z)

**Definition**  
The differential equation

\[
\frac{z^2 d^2 y}{dz^2} + \frac{d y}{dz} + (z^2 - \nu^2)y = 0
\]

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 \( \nu \)

\[
J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{z^2}{4}\right)^k}{k! \Gamma(\nu + k + 1)}
\]

and \( \Gamma(\alpha) \) is the gamma function. \( Y_\nu(z) \) is linearly independent of \( J_\nu(z) \).

\( J_\nu(z) \) can be computed using 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.

\[
Y = \text{bessely}(\nu,Z,1) \text{ computes } \text{bessely}(\nu,Z) \cdot \exp(-\text{abs}(\text{imag}(Z))).
\]

\([Y,ierr] = \text{bessely}(\nu,Z) \) 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>\text{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 ( \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>

**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 \( \text{besselh} \), \( J_{\nu}(z) \) is \( \text{besselj} \), and \( Y_{\nu}(z) \) is \( \text{bessely} \). The Hankel functions also form a fundamental set of solutions to Bessel’s equation (see \( \text{besselh} \)).
Examples

Example 1

```matlab
format long
z = (0:0.2:1)';
bessely(1,z)
```

```
ans =
   -Inf
   -3.32382498811185
   -1.78087204427005
   -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].

References


bessely

See Also besselh, besseli, besselj, besselk
**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,

\[
\begin{align*}
\text{beta}(n,3) & = (n-1)!*2!/(n+2)!
\text{\quad} = 2/(n*(n+1)*(n+2))
\end{align*}
\]

is the ratio of fairly small integers, and the rational format is able to recover the exact result.

\[
\text{format rat}
\text{beta((0:10)',3)}
\]

\[
\text{ans} =
\begin{align*}
1/0 & \\
1/3 & \\
1/12 & \\
1/30 & \\
1/60 & \\
1/105 & \\
1/168 & \\
1/252 & \\
\end{align*}
\]
Algorithm

\[
\beta(z, w) = \exp(\text{gammaln}(z) + \text{gammaln}(w) - \text{gammaln}(z + w))
\]

See Also

betainc, betaln, gammaln
**Purpose**
Incomplete beta function

**Syntax**

\[
I = \text{betainc}(X,Z,W) \\
I = \text{betainc}(X,Z,\text{tail})
\]

**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.

\(I = \text{betainc}(X,Z,\text{tail})\) specifies the tail of the incomplete beta function. Choices are:

<table>
<thead>
<tr>
<th>'lower' (the default)</th>
<th>Computes the integral from 0 to (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>'upper'</td>
<td>Computes the integral from (x) to 1</td>
</tr>
</tbody>
</table>

These functions are related as follows:

\[
1 - \text{betainc}(X,Z,W) = \text{betainc}(X,Z,W,\text{'upper'})
\]

Note that especially when the upper tail value is close to 0, it is more accurate to use the 'upper' option than to subtract the 'lower' value from 1.
Examples  

```matlab
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, betainc
**Purpose**
Beta inverse cumulative distribution function

**Syntax**

```matlab
x = betaincinv(y,z,w)
```

```matlab
x = betaincinv(y,a,tail)
```

**Description**

`x = betaincinv(y,z,w)` computes the inverse incomplete beta function for corresponding elements of `x`, `z`, and `w`, such that `y = betainc(x,z,w)`. The elements of `y` must be in the closed interval `[0,1]`, and those of `z` and `w` must be nonnegative. `y`, `z`, and `w` must all be real and the same size (or any of them can be scalar).

`x = betaincinv(y,a,tail)` specifies the tail of the incomplete beta function. Choices are `lower` (the default) to use the integral from 0 to `x`, or `'upper'` to use the integral from `x` to 1. These two choices are related as follows: `betaincinv(y,z,w,'upper') = betaincinv(1-y,z,w,'lower')`. When `y` is close to 0, the upper option provides a way to compute `X` more accurately than by subtracting from `y` from 1.

**Definition**
The incomplete beta function is defined as

\[
I_x(z,w) = \frac{1}{\beta(z,w)} \int_0^x t^{(z-1)}(1-t)^{(w-1)} dt
\]

`betaincinv` computes the inverse of the incomplete beta function with respect to the integration limit `x` using Newton’s method.

**See Also**
`betainc`, `beta`, `betaln`
**Purpose**
Logarithm of beta function

**Syntax**
\[ L = \text{betaln}(Z, W) \]

**Description**
\[ L = \text{betaln}(Z, W) \] 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**
\[
\begin{align*}
\text{x} &= 5 \quad 1 \quad 0 \\
\text{betaln}(\text{x}, \text{x}) \\
\text{ans} &= \\
&= -708.8616
\end{align*}
\]

-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**
\[ \text{beta}, \text{betainc}, \text{gammaln} \]
Purpose

Biconjugate gradients method

Syntax

\[
\begin{align*}
x &= \text{bicg}(A,b)
\end{align*}
\]

bicg(A,b,tol)

bicg(A,b,tol,maxit)

bicg(A,b,tol,maxit,M)

bicg(A,b,tol,maxit,M1,M2)

bicg(A,b,tol,maxit,M1,M2,x0)

[x,flag] = bicg(A,b,...)

[x,flag,relres] = bicg(A,b,...)

[x,flag,relres,iter] = bicg(A,b,...)

[x,flag,relres,iter,resvec] = bicg(A,b,...)

Description

\[
\begin{align*}
x &= \text{bicg}(A,b) \text{ attempts to solve the system of linear equations } A* x = b \text{ for } x. \text{ The } n\text{-by-}n \text{ coefficient matrix } A \text{ must be square and should be large and sparse. The column vector } b \text{ must have length } n. \text{ A can be a function handle afun such that afun(x,'notransp') returns } A*x \text{ and afun(x,'transp') returns } A'*x. \text{ See “Function Handles” in the MATLAB Programming documentation for more information.}
\end{align*}
\]

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

If bicg converges, it displays a message to that effect. If 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 norm(b-A*x)/norm(b) and the iteration number at which the method stopped or failed.

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

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

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

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

[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;
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

Example 2

This example replaces the matrix A in Example 1 with a handle to a matrix-vector product function afun. The example is contained in an M-file run_bicg that

- Calls bicg with the function handle @afun as its first argument.
- Contains afun as a nested function, so that all variables in run_bicg are available to afun.

The following shows the code for run_bicg:

```matlab
function x1 = run_bicg
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = bicg(@afun,b,tol,maxit,M1,M2);

function y = afun(x,transp_flag)
    if strcmp(transp_flag,'transp') % y = A'*x
```
When you enter

```
x1 = run_bicg;
```

MATLAB software displays the message

```
bicg converged at iteration 9 to a solution with ...
relative residual
5.3e-009
```

**Example 3**

This example 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.

```
load west0479;
A = west0479;
b = sum(A,2);
```

You can accurately solve \( A*x = b \) using backslash since \( A \) is not so large.

```
x = A \ b;
norm(b-A*x) / norm(b)
```

```
ans =
8.3154e-017
```
Now try to solve $A\times x = b$ with `bicg`.

```matlab
[x,flag,relres,iter,resvec] = bicg(A,b)
```

```matlab
flag = 1
relres = 1
iter = 0
```

The value of `flag` indicates that `bicg` iterated the default 20 times without converging. The value of `iter` shows that the method behaved so badly that the initial all-zero guess was better than all the subsequent iterates. The value of `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.

```matlab
semilogy(0:20,resvec/norm(b),'-o')
exlabel('Iteration Number')
ylabel('Relative Residual')
```
Now, try an incomplete LU factorization with a drop tolerance of $1e^{-5}$ 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)

ans =

1887

ans =

5562

ans =

4320
The zero on the main diagonal of the upper triangular \( U_1 \) indicates that \( U_1 \) 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 \( U_1 \) using backslash. \texttt{bicg} is forced to return the initial estimate since no other iterates were produced.

Try again with a slightly less sparse preconditioner.

\[
[L2, U2] = \text{luinc}(A, 1e-6);
\]

\[
\text{nnz}(L2), \phantom{} \text{nnz}(U2)
\]

\[
\begin{align*}
\text{ans} &= 6231 \\
\text{ans} &= 4559
\end{align*}
\]

This time \( U_2 \) is nonsingular and may be an appropriate preconditioner.

\[
[x, \text{flag}, \text{relres}, \text{iter}, \text{resvec}] = \text{bicg}(A, b, 1e-15, 10, L2, U2)
\]

\[
\begin{align*}
\text{flag} &= 0 \\
\text{relres} &= 2.8664e-016 \\
\text{iter} &= \phantom{} \phantom{}
\end{align*}
\]
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.

References

See Also

bicgstab, cgs, gmres, ilu, lsqr, lhuinc, minres, pcg, qmr, symmlq, function_handle (@), mldivide (\)
bicgstab

Purpose
Biconjugate gradients stabilized method

Syntax
x = bicgstab(A,b)
bicgstab(A,b,tol)
bicgstab(A,b,tol,maxit)
bicgstab(A,b,tol,maxit,M)
bicgstab(A,b,tol,maxit,M1,M2)
bicgstab(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicgstab(A,b,...)
[x,flag,relres] = bicgstab(A,b,...)
[x,flag,relres,iter] = bicgstab(A,b,...)
[x,flag,relres,iter,resvec] = bicgstab(A,b,...)

Description
x = 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 handle afun such that afun(x) returns A\*x. See “Function 
Handles” in the MATLAB Programming documentation for more 
information.

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

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

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

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

bicgstab(A,b,tol,maxit,M) and bicgstab(A,b,tol,maxit,M1,M2)
use preconditioner M or M = M1*M2 and effectively solve the system
bicgstab(A, b, tol, maxit, M1, M2, x0) specifies the initial guess. If x0 is [], then bicgstab uses the default, an all zero vector.

[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 halfway 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 1**

This example first solves $Ax = b$ by providing $A$ and the preconditioner $M_1$ directly as arguments.

A = gallery('wilk',21);  
b = sum(A,2);  
tol = 1e-12;  
maxit = 15;  
M1 = diag([10:-1:1 1 1:10]);

x = bicgstab(A,b,tol,maxit,M1);

displays the message

bicgstab converged at iteration 12.5 to a solution with relative 
residual 6.7e-014

**Example 2**

This example replaces the matrix $A$ in Example 1 with a handle to a 
matrix-vector product function `afun`, and the preconditioner $M_1$ with a 
handle to a backsolve function `mfun`. The example is contained in an 
M-file `run_bicgstab` that

- Calls `bicgstab` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in 
  `run_bicgstab` are available to `afun` and `mfun`.

The following shows the code for `run_bicgstab`:

```matlab
function x1 = run_bicgstab
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);
x1 = bicgstab(@afun,b,tol,maxit,@mfun);
```
function y = afun(x)
    y = [0; x(1:n-1)] + ...
        [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
        [x(2:n); 0];
end

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

When you enter

    x1 = run_bicgstab;

MATLAB software displays the message

    bicgstab converged at iteration 12.5 to a solution with relative residual 6.7e-014

**Example 3**

This example 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.

    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 \( 1 \cdot 10^{-6} \) within the default 20 iterations.

    [L1,U1] = luinc(A,1e-5);
    [x1,flag1] = bicgstab(A,b,1e-6,20,L1,U1)
flag1 is 2 because the upper triangular $U1$ has a zero on its diagonal. This causes bicgstab to fail in the first iteration when it tries to solve a system such as $U1*y = r$ using backslash.

```matlab
[L2,U2] = luinc(A,1e-6);
[x2,flag2,relres2,iter2,resvec2] = bicgstab(A,b,1e-15,10,L2,U2)
```

flag2 is 0 because 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 bicgstab by plotting the relative residuals at the halfway point and end of each iteration starting from the initial estimate (iterate number 0).

```matlab
semilogy(0:0.5:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
```
References


See Also

bicg, cgs, gmres, lsqr, luinc, minres, pcg, qmr, symmlq, function_handle (@), mldivide (\)
**Purpose**
Biconjugate gradients stabilized (l) method

**Syntax**
```matlab
x = bicgstabl(A,b)
x = bicgstabl(afun,b)
x = bicgstabl(A,b,tol)
x = bicgstabl(A,b,tol,maxit)
x = bicgstabl(A,b,tol,maxit,M)
x = bicgstabl(A,b,tol,maxit,M1,M2)
x = bicgstabl(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicgstabl(A,b,...)
[x,flag,relres] = bicgstabl(A,b,...)
[x,flag,relres,iter] = bicgstabl(A,b,...)
[x,flag,relres,iter,resvec] = bicgstabl(A,b,...)
```

**Description**
`x = bicgstabl(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 the right-hand side column vector $b$ must have length $n$.

`x = bicgstabl(afun,b)` accepts a function handle `afun` instead of the matrix $A$. `afun(x)` accepts a vector input $x$ and returns the matrix-vector product $A*x$. In all of the following syntaxes, you can replace $A$ by `afun`.

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

`x = bicgstabl(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is [] then `bicgstabl` uses the default, min(N,20).

`x = bicgstabl(A,b,tol,maxit,M)` and `x = bicgstabl(A,b,tol,maxit,M1,M2)` use preconditioner $M$ or $M=M1*M2$ and effectively solve the system $A*inv(M)*x = b$ for $x$. If $M$ is [] then a preconditioner is not applied. $M$ may be a function handle returning $M\backslash x$.

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

`[x,flag] = bicgstabl(A,b,...)` also returns a convergence `flag`:
### Table: Flag Convergence

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

[x, flag, relres] = `bicgstabl`(A, b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x, flag, relres, iter] = `bicgstabl`(A, b,...) also returns the iteration number at which x was computed, where 0 <= iter <= maxit. iter can be k/4 where k is some integer, indicating convergence at a given quarter iteration.

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

#### Example

```matlab
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M = diag([10:-1:1 1 1:10]);
x = bicgstabl(A,b,tol,maxit,M);
```

You can also use this matrix-vector product function:

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

(1:(n-1)/2)'.*x+[x(2:n); 0];

and this preconditioner backsolve function:

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

as inputs to bicgstabl:

x1 = bicgstabl(@(x)afun(x,n),b,tol,maxit,@(x)mfun(x,n));

See Also
bicgstab, bicg, cgs, gmres, lsqr, luinc, minres, pcg, qmr, symmlq, function_handle (@), mldivide (\)
**Purpose**  
Convert binary number string to decimal number

**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')
   ans =
       23
```

Because space characters are ignored, this string yields the same result:

```matlab
   bin2dec(' 010 111 ')
   ans =
       23
```

**See Also**  
dec2bin
**Purpose**
Set FTP transfer type to binary

**Syntax**
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
    dir: /
    mode: binary

Note that the FTP object’s mode is again set to binary.

See Also
ftp, ascii
Purpose
Bitwise AND

Syntax
C = bitand(A, B)

Description
C = bitand(A, B) returns the bitwise AND of arguments A and B, where A and B are unsigned integers or arrays of unsigned integers.

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:

\[
C = \text{bitand(uint8}(13), \text{uint8}(27))
\]
\[
C = 9
\]

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

2-424
Purpose: Bitwise complement

Syntax:

\[
C = \text{bitcmp}(A) \\
C = \text{bitcmp}(A, n)
\]

Description:

\(C = \text{bitcmp}(A)\) returns the bitwise complement of \(A\), where \(A\) is an unsigned integer or an array of unsigned integers.

\(C = \text{bitcmp}(A, n)\) returns the bitwise complement of \(A\) as an \(n\)-bit unsigned integer \(C\). Input \(A\) may not have any bits set higher than \(n\) (that is, \(A\) may not have a value greater than \(2^n - 1\)). The value of \(n\) can be no greater than the number of bits in the unsigned integer class of \(A\). For example, if the class of \(A\) is \(\text{uint32}\), then \(n\) must be a positive integer less than 32.

Examples

Example 1

With eight-bit arithmetic, the one’s complement of 01100011 (decimal 99) is 10011100 (decimal 156):

\[
C = \text{bitcmp}(\text{uint8}(99)) \\
C = 156
\]

Example 2

The complement of hexadecimal A5 (decimal 165) is 5A:

\[
x = \text{hex2dec('A5')} \\
x = 165
\]

\[
\text{dec2hex(\text{bitcmp}(x, 8))} \\
\text{ans} = 5A
\]

Next, find the complement of hexadecimal 000000A5:

\[
\text{dec2hex(\text{bitcmp}(x, 32))}
\]
bitsmp

ans =
FFFFFF5A

See Also: bitand, bitget, bitmax, bitor, bitset, bitshift, bitxor
Purpose

Bit at specified position

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, a double, or an array containing unsigned integers, doubles or both. The bit input 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).

Examples

**Example 1 — Binary Conversion**

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 — Binary Compare**

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.
```

**Example 3 — Vector and Array Operations**

Get the value of the second most significant bit of the number sequence 5 through 75, counting by tens:
bitget

\[
\text{bitget}(5:10:65, [2 3 4 5 5 5 6 7])
\]
\[
\text{ans} = \\
0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1
\]

Do the same, but using 2-by-4 matrices:

\[
\text{bitget}([5 15 25 35; 45 55 65 75], ... \\
[2 3 4 5; 5 5 6 7])
\]
\[
\text{ans} = \\
0 \ 1 \ 1 \ 0 \\
0 \ 1 \ 0 \ 1
\]

**See Also**

bitand, bitcmp, bitmax, bitor, bitset, bitshift, bitxor
**Purpose**
Maximum double-precision 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 =
  433ffffffffffffff

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 x 2^52 with 52 trailing 1’s, or 2^53 - 1.

**See Also**

bitand, bitcmp, bitget, bitor, bitset, bitshift, bitxor
Purpose
Bitwise OR

Syntax
C = bitor(A, B)

Description
C = bitor(A, B) returns the bitwise OR of arguments A and B, where A and B are unsigned integers or arrays of unsigned integers.

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.

C = bitor(uint8(13), uint8(27))
C =
31

Example 2
Create a truth table for a logical OR operation:

A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);

TT = bitor(A, B)
TT =
0  1
1  1

See Also
bitand, bitcmp, bitget, bitmax, bitset, bitshift, bitxor
Purpose
Set bit at specified position

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 or an array of unsigned integers, 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:

```matlab
C = bitset(uint8(9), 5)
C =
25
```

Example 2
Repeatedly subtract powers of 2 from the largest uint32 value:

```matlab
a = intmax('uint32')
for k = 1:32
    a = bitset(a, 32-k+1, 0)
end
```

See Also
bitand, bitcmp, bitget, bitmax, bitor, bitshift, bitxor
Purpose  
Shift bits specified number of places

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 \( A \) must be an unsigned integer or an array of unsigned integers. 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.

\[ C = \text{bitshift}(A, k, n) \] causes any bits that overflow \( n \) bits to be dropped. The value of \( n \) must be less than or equal to the length in bits of the unsigned integer class of \( A \) (e.g., \( n \leq 32 \) for uint32).

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:

\[
a = \text{intmax('uint16')}; \\
disp(sprintf( ... \\
   'Initial uint16 value %5d is %16s in binary', ... \\
a, dec2bin(a)))
\]
for k = 1:16
    a = bitshift(a, 1);
    disp(sprintf('Shifted uint16 value %5d is %16s in binary', a, dec2bin(a)));
end

See Also

bitand, bitcmp, bitget, bitmax, bitor, bitset, bitxor, fix
Purpose  
Bitwise XOR

Syntax  
C = bitxor(A, B)

Description  
C = bitxor(A, B) returns the bitwise XOR of arguments A and B, where A and B are unsigned integers or arrays of 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**  
Create string of blank characters

**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 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

\[
\begin{bmatrix}
a & 0 & 0 & 0 & 0 \\
0 & b & 0 & 0 & 0 \\
0 & 0 & c & 0 & 0 \\
0 & 0 & 0 & d & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

The input matrices do not have to be square, nor do they have to be of equal size.

See Also
diag, horzcat, vertcat
**Purpose**

Axes border

**Syntax**

```matlab
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” on page 1-103 for related functions
### Purpose
Terminate execution of `for` 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) || ~ischar(line), break, end
    s = sprintf('%s%s\n', s, line);
end
disp(s);
fclose(fid);
```

### 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:

```matlab
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” on page 1-105 for related functions

“Altering Colormaps” for more information
**Purpose**

Interactively mark, delete, modify, and save observations in graphs.

**GUI Alternatives**

To turn data brushing on or off, use the Data Brushing tool in the figure toolbar, the right side of which drops down as a color palette for changing the current brushing color. For details, see “Marking Up Graphs with Data Brushing” in the MATLAB Data Analysis documentation.

**Syntax**

- brush on
- brush off
- brush
- brush color
- brush(figure_handle,...)
- brushobj = brush(figure_handle)

**Description**

Data brushing is a mode for interacting with graphs in figure windows in which you can click data points or drag a selection rectangle around data points to highlight observations in a color of your choice. Highlighting takes different forms for different types of graphs, and brushing marks persist—even in other interactive modes—until removed by deselecting them.

- `brush on` turns on interactive data brushing mode.
- `brush off` turns brushing mode off, leaving any brushed observations still highlighted.
- `brush` by itself toggles the state of the data brushing tool.
- `brush color` sets the current color used for brushing graphics to the specified `ColorSpec`. Changing brush color affects subsequent brushing, but does not change the color of observations already brushed or the brush tool’s state.
- `brush(figure_handle,...)` applies the function to the specified figure handle.
- `brushobj = brush(figure_handle)` returns a *brush mode object* for that figure, useful for controlling and customizing the figure’s brushing.
state. The following properties of such objects can be modified using `get` and `set`:

- **Enable 'on' | {'off'}**
  Specifies whether this figure mode is currently enabled on the figure.

- **FigureHandle**
  The associated figure handle. This property supports `get` only.

- **Color**
  Specifies the color to be used for brushing.

`brush` cannot return a brush mode object at the same time you are calling it to set a brushing option.

**Remarks**

- “Types of Plots You Can Brush” on page 2-443
- “Plot Types You Cannot Brush” on page 2-445
- “Mode Exclusivity and Persistence” on page 2-446
- “How Data Linking Affects Data Brushing” on page 2-447
- “Mouse Gestures for Data Brushing” on page 2-448

**Types of Plots You Can Brush**

Data brushing places lines and patches on plots to create highlighting, marking different types of graphs as follows (brushing marks are shown in red):

<table>
<thead>
<tr>
<th>Graph Type</th>
<th>Brushing Annotation</th>
<th>Overlays?</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>lineseries</td>
<td>Colored lines slightly wider than those in the lineseries with a marker distinct from those on the lineseries (filled circles if none) to identify brushed vertices. Only those line segments that connect brushed vertices are highlighted</td>
<td>Y</td>
<td>![Example Image]</td>
</tr>
</tbody>
</table>

2-443
<table>
<thead>
<tr>
<th>Graph Type</th>
<th>Brushing Annotation</th>
<th>Overlays?</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>scattergroup</td>
<td>Line withLineStyle 'none' and a marker with a color distinct from and slightly larger than the base scattergroup marker.</td>
<td>Y</td>
<td><img src="image" alt="Example Image" /></td>
</tr>
<tr>
<td>stemseries</td>
<td>The brushed stems and stem heads are shaded in the brushing color.</td>
<td>Y</td>
<td><img src="image" alt="Example Image" /></td>
</tr>
<tr>
<td>barseries</td>
<td>The interior of selected bars is filled in the brushing color.</td>
<td>N</td>
<td><img src="image" alt="Example Image" /></td>
</tr>
<tr>
<td>histogram</td>
<td>The bars to which brushed observations contribute are proportionately filled from the bottom up with the brushing color.</td>
<td>N</td>
<td><img src="image" alt="Example Image" /></td>
</tr>
<tr>
<td>Graph Type</td>
<td>Brushing Annotation</td>
<td>Overlays?</td>
<td>Example</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>areaseries</td>
<td>Patches filling the region between selected points and the x-axis in the brushing color.</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>surfaceplot</td>
<td>Patches with edges slightly wider than the surfaceplot line width and with a marker distinct from that of the surfaceplot (X if none) to identify brushed vertices. Patches are plotted only when all four vertices that define them are brushed. The brushed observations are the set of marked vertices, not the patches.</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

When using the linked plots feature, a graph can become brushed when you brush another graph that displays some of the same data, potentially brushing the same observations more than once. The overlaid brushing marks (whether lines or markers) are slightly wider than the brushing marks that they overlay; this makes multiply brushed observations visually distinct. The wider brushing marks are placed under the narrower ones, so that if they happen to have different colors, you can see all the colors. See the subsection “How Data Linking Affects Data Brushing” on page 2-447 for more information about brushing linked figures.

As the above table indicates, only lineseries, scatterseries, and stemseries brushing marks can be overlaid in this manner. Although you can brush them, you cannot overlay brushing marks on areaseries, barseries, histograms, or surfaceplots.

**Plot Types You Cannot Brush**

Currently, not all plot types enable data brushing. Graph functions that *do not* support brushing are:

<table>
<thead>
<tr>
<th>Graph Type</th>
<th>Brushing Annotation</th>
<th>Overlays?</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>areaseries</td>
<td>Patches filling the region between selected points and the x-axis in the brushing color.</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>surfaceplot</td>
<td>Patches with edges slightly wider than the surfaceplot line width and with a marker distinct from that of the surfaceplot (X if none) to identify brushed vertices. Patches are plotted only when all four vertices that define them are brushed. The brushed observations are the set of marked vertices, not the patches.</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

When using the linked plots feature, a graph can become brushed when you brush another graph that displays some of the same data, potentially brushing the same observations more than once. The overlaid brushing marks (whether lines or markers) are slightly wider than the brushing marks that they overlay; this makes multiply brushed observations visually distinct. The wider brushing marks are placed under the narrower ones, so that if they happen to have different colors, you can see all the colors. See the subsection “How Data Linking Affects Data Brushing” on page 2-447 for more information about brushing linked figures.

As the above table indicates, only lineseries, scatterseries, and stemseries brushing marks can be overlaid in this manner. Although you can brush them, you cannot overlay brushing marks on areaseries, barseries, histograms, or surfaceplots.

**Plot Types You Cannot Brush**

Currently, not all plot types enable data brushing. Graph functions that *do not* support brushing are:
• Line plots created with `line`
• Scatter plots created with `spy`
• Contour plots created with `contour`, `contourf`, or `contour3`
• Pie charts created with `pie` or `pie3`
• Radial graphs created with `polar`, `compass`, or `rose`
• Direction graphs created with `feather`, `quiver`, or `comet`
• Area and image plots created with `fill`, `image`, `imagesc`, or `pcolor`
• Bar graphs created with `pareto` or `errorbar`
• Functional plots created with `ezcontour` or `ezcontourf`
• 3-D plot types `other than plot3`, `stem3`, `scatter3`, `mesh`, `meshc`, `surf`, `surfl`, and `surfz`

You can use some of these functions to display base data that do not need to be brushable. For example, use `line` to plot mean y-values as horizontal lines that you do not need or want to brush.

**Mode Exclusivity and Persistence**

Data brushing mode is exclusive, like zoom, pan, data cursor, or plot edit mode. However, brush marks created in data brushing mode persist through all changes in mode. Brush marks that appear in other graphs while they are linked via `linkdata` also persist even when data linking is subsequently turned off. That is, severing connections to a graph's data sources does not remove brushing marks from it. The only ways to remove brushing marks are (in brushing mode):

• Brush an empty area in a brushed graph.
• Right-click and select **Clear all brushing** from the context menu.

Changing the brushing color for a figure does not recolor brushing marks on it until you brush it again. If you hold down the Shift key, all existing brush marks change to the new color. All brush marks that appear on linked plots in the same or different figure also change to the new color.
if the brushing action affects them. The behavior is the same whether you select a brushing color from the Brush Tool dropdown palette, set it by calling `brush(colorspec)`, or by setting the Color property of a brush mode object (e.g., `set(brushobj,'Color',colorspec)`).

**How Data Linking Affects Data Brushing**

When you use the Data Linking tool or call the `linkdata` function, brushing marks that you make on one plot appear on other plots that depict the same variable you are brushing—if they are also linked. This happens even if the affected plot is not in Brushing mode. That is, brushing marks appear on a linked plot in any mode when you brush another plot linked to it via a common variable or brush that variable in the Variable Editor. Two limiting conditions apply, however:

- The graph type must support data brushing (see “Types of Plots You Can Brush” on page 2-443 and “Plot Types You Cannot Brush” on page 2-445)

- The graphed variable should not be complex; if you can plot a complex variable you can brush it, but such graphs do not respond when you brush the complex variable in another linked plot.

For more information about linking complex variables, see Example 3 in the `linkdata` reference page.

Brush marks on a an unlinked graph can change color when data linking is turned on for that figure. They can, in fact, vanish and be replaced by marks in the same or different color when the plot enters a linked state. This happens because in the linked state, the variables (data sources) are brushed, not just the graphics. If different observations for the same variable on a linked figure are brushed, those brushed variables override the brushed graphics on the newly linked plot. In other words, the newly linked graph loses all its previous brush marks when it “joins the club” of common data sources.
**Mouse Gestures for Data Brushing**

You can brush graphs in several ways. The basic operation is to drag the mouse to highlight all observations within the rectangle you define. The following table lists data brushing gestures and their effects.

<table>
<thead>
<tr>
<th>Action</th>
<th>Gesture</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select data using a region of interest</td>
<td>ROI mouse drag</td>
<td>Region of interest (ROI) rectangle (or rectangular prism for 3-D axes) appears during the gesture and all brushable observations within the rectangle are highlighted. All other brushing marks in the axes are removed. The ROI rectangle disappears when the mouse button is released.</td>
</tr>
<tr>
<td>Select a single point</td>
<td>Single left-click on a graphic object that supports data brushing</td>
<td>Produces an equivalent result to ROI rectangle, brushing where the rectangle encloses only the single vertex on the graphical object closest to the mouse. All other brushing annotations in the figure are removed.</td>
</tr>
<tr>
<td>Add a point to the selection or remove a highlighted one</td>
<td>Single left-click on a graphic object that supports data brushing, with the <strong>Shift</strong> key down</td>
<td>Equivalent brushing by dragging an ROI rectangle that encloses only the single vertex on the graphic object closest to the mouse. All other brushed regions in the figure remain brushed.</td>
</tr>
<tr>
<td>Select all data associated with a graphic object</td>
<td>Double left-click on a graphic object that supports data brushing</td>
<td>All vertices for the graphic object are brushed.</td>
</tr>
<tr>
<td><strong>Action</strong></td>
<td><strong>Gesture</strong></td>
<td><strong>Result</strong></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>Add to or subtract from region of interest</td>
<td>Click or ROI drag with the <strong>Shift</strong> or <strong>Ctrl</strong> keys down</td>
<td>Region of interest grows; all unbrushed vertices within the rectangle become brushed and all brushed observations in it become unbrushed. All brushed vertices outside the ROI remain brushed.</td>
</tr>
<tr>
<td>Copy brushed data to Editor, Command Window, Variable Editor, or Workspace Browser</td>
<td>Drag brushed data to another window or to a program/icon on the system desktop</td>
<td>Equivalent to copying brushed data and pasting into other window or an existing/new variable.</td>
</tr>
</tbody>
</table>

**Examples**

**Example 1**

On a scatterplot, drag out a rectangle to brush the graph:

```plaintext
x = rand(20,1);
y = rand(20,1);
scatter(x,y,80,'s')
brush on
```
Example 2

Brush observations from -.2 to .2 on a lineseries plot in dark red:

```matlab
x = [-2*pi:.1:2*pi];
y = sin(x);
plot(x,y);
h = brush;
set(h,'Color',[.6 .2 .1],'Enable','on');
```
See Also

linkaxes, linkdata, pan, rotate3d, zoom
bsxfun

**Purpose**
Apply element-by-element binary operation to two arrays with singleton expansion enabled

**Syntax**
C = bsxfun(fun,A,B)

**Description**
C = bsxfun(fun,A,B) applies an element-by-element binary operation to arrays A and B, with singleton expansion enabled. fun is a function handle, and can either be an M-file function or one of the following built-in functions:

- @plus  
  Plus
- @minus  
  Minus
- @times  
  Array multiply
- @rdivide  
  Right array divide
- @ldivide  
  Left array divide
- @power  
  Array power
- @max  
  Binary maximum
- @min  
  Binary minimum
- @rem  
  Remainder after division
- @mod  
  Modulus after division
- @atan2  
  Four quadrant inverse tangent
- @hypot  
  Square root of sum of squares
- @eq  
  Equal
- @ne  
  Not equal
- @lt  
  Less than
- @le  
  Less than or equal to
- @gt  
  Greater than
- @ge  
  Greater than or equal to
@and Element-wise logical AND
@or Element-wise logical OR
@xor Logical exclusive OR

If an M-file function is specified, it must be able to accept either two column vectors of the same size, or one column vector and one scalar, and return as output a column vector of the size as the input values.

Each dimension of A and B must either be equal to each other, or equal to 1. Whenever a dimension of A or B is singleton (equal to 1), the array is virtually replicated along the dimension to match the other array. The array may be diminished if the corresponding dimension of the other array is 0.

The size of the output array C is equal to:
max(size(A),size(B)).*(size(A)>0 & size(B)>0).

**Examples**

In this example, bsxfun is used to subtract the column means from the corresponding columns of matrix A.

```matlab
A = magic(5);
A = bsxfun(@minus, A, mean(A))
```

A =

```
  4  11 -12  -5   2
 10  -8  -6   1   3
 -9  -7   0   7   9
-13  -1   6   8  -10
-2   5  12  -11  -4
```

**See Also**

repmat, arrayfun
builddocsearchdb

**Purpose**  
Build searchable documentation database

**Syntax**  
`builddocsearchdb help_location`

**Description**  
`builddocsearchdb help_location` builds a searchable database of user-added HTML and related help files in the specified help location. The `help_location` argument is the full path to the directory containing the help files. The database enables the Help browser to search for content within the help files.

`builddocsearchdb` creates a directory named `helpsearch` under `help_location`. The `helpsearch` directory contains the search database files. Add the location of the `helpsearch` directory to your `info.xml` file.

The `helpsearch` directory works only with the version of MATLAB software used to create it.

For a full discussion of this process, refer to “Adding HTML Help Files for Your Own Toolbox”.

**Examples**  
Build a search database for the documentation files found at `D:\work\mytoolbox\help`.

```plaintext
builddocsearchdb D:\work\mytoolbox\help
```

**See Also**  
doc, help
Purpose
Execute built-in function from overloaded method

Syntax
\[
\texttt{builtin(} \texttt{function, x1, ..., xn)} \\
\texttt{[y1, ..., yn] = builtin(} \texttt{function, x1, ..., xn)}
\]

Description
\texttt{builtin} is used in methods that overload built-in functions to execute the original built-in function. If \texttt{function} is a string containing the name of a built-in function, then \texttt{builtin(function, x1, ..., xn)} evaluates the specified function at the given arguments \texttt{x1} through \texttt{xn}. The \texttt{function} argument must be a string containing a valid function name. \texttt{function} cannot be a function handle.

\texttt{[y1, ..., yn] = builtin(function, x1, ..., xn)} returns multiple output arguments.

Remarks
\texttt{builtin(...)} is the same as \texttt{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 \texttt{builtin}.)

See Also
\texttt{feval}
**Purpose**
Solve boundary value problems for ordinary differential equations

**Syntax**

```matlab
sol = bvp4c(odefun,bcfun,solinit)
sol = bvp4c(odefun,bcfun,solinit,options)
solinit = bvpinit(x, yinit, params)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| `odefun` | A function handle that evaluates the differential equations $f(x,y)$. It can have the form  
  
  \begin{align*}
  dydx &= odefun(x,y) \\
  dydx &= odefun(x,y,parameters)
  \end{align*}
  
  where $x$ is a scalar corresponding to $X$, and $y$ is a column vector corresponding to $Y$. `parameters` is a vector of unknown parameters. The output `dydx` is a column vector. |
| `bcfun`  | A function handle 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  
  
  \begin{align*}
  res &= bcfun(ya,yb) \\
  res &= bcfun(ya,yb,parameters)
  \end{align*}
  
  where $ya$ and $yb$ are column vectors corresponding to $y(a)$ and $y(b)$. `parameters` is a vector of unknown parameters. The output `res` is a column vector.  
  
  See “Multipoint Boundary Value Problems” on page 2-459 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. |
### Description

\[
\text{sol} = \text{bvp4c}(\text{odefun}, \text{bcfun}, \text{solinit}) \text{ integrates a system of ordinary differential equations of the form}
\]

\[y' = f(x, y)\]

on the interval \([a,b]\) subject to two-point boundary value conditions

\[bc(y(a), y(b)) = 0\]

\textit{odefun} and \textit{bcfun} are function handles. See “Function Handles” in the MATLAB Programming documentation for more information.

“Parametrizing Functions” in the MATLAB mathematics documentation, explains how to provide additional parameters to the function \textit{odefun}, as well as the boundary condition function \textit{bcfun}, if necessary.

\textit{bvp4c} can also solve multipoint boundary value problems. See “Multipoint Boundary Value Problems” on page 2-459. You can use the function \textit{bvpinit} to specify the boundary points, which are stored in the input argument \textit{solinit}. See the reference page for \textit{bvpinit} for more information.

<table>
<thead>
<tr>
<th>Ordered nodes of the initial mesh. Boundary conditions are imposed at (a = \text{solinit.x}(1)) and (b = \text{solinit.x}(\text{end})).</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y) Initial guess for the solution such that (\text{solinit.y}(:,i)) is a guess for the solution at the node (\text{solinit.x}(i)).</td>
</tr>
<tr>
<td>parameters Optional. A vector that provides an initial guess for unknown parameters.</td>
</tr>
<tr>
<td>The structure can have any name, but the fields must be named (x, y), and (parameters). You can form (\text{solinit}) with the helper function \textit{bvpinit}. See \textit{bvpinit} for details.</td>
</tr>
<tr>
<td>options Optional integration argument. A structure you create using the \textit{bvpset} function. See \textit{bvpset} for details.</td>
</tr>
</tbody>
</table>
The bvp4c solver can also find unknown parameters $\mathbf{p}$ for problems of the form

$$y' = f(x, y, \mathbf{p})$$

$$0 = bc(y(a), y(b), \mathbf{p})$$

where $\mathbf{p}$ 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 $[a,b]$.

$$\text{sxint} = \text{deval(sol,xint)}$$

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.

$$\text{sol} = \text{bvp4c(odefun,bcfun,solinit,options)}$$

solves as above with default integration properties replaced by the values in options, a structure created with the bvpset function. See bvpset for details.
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

```matlab
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,


\[
y_{\text{left}}(:, 1) = y(a)
\]

and

\[
y_{\text{right}}(:, \text{end}) = y(b)
\]

When you create an initial guess with

\[
solinit = bvpinit(xinit, yinit),
\]

use double entries in \(xinit\) for each interface point. See the reference page for \(bvpinit\) for more information.

If \(yinit\) is a function, \(bvpinit\) calls \(y = yinit(x, k)\) to get an initial guess for the solution at \(x\) in region \(k\). In the solution structure \(sol\) returned by \(bvp4c\), \(sol.x\) has double entries for each interface point. The corresponding columns of \(sol.y\) contain the left and right solution at the interface, respectively.

For an example of solving a three-point boundary value problem, type \(threebbvp\) at the MATLAB command prompt to run a demonstration.

**Note** The \(bvp5c\) function is used exactly like \(bvp4c\), with the exception of the meaning of error tolerances between the two solvers. If \(S(x)\) approximates the solution \(y(x)\), \(bvp4c\) controls the residual \(|S'(x) - f(x, S(x))|\). This controls indirectly the true error \(|y(x) - S(x)|\). \(bvp5c\) controls the true error directly. \(bvp5c\) is more efficient than \(bvp4c\) for small error tolerances.

**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 \texttt{bvp4c}, you must write the differential equation as a system of two first-order ODEs

\[ y_1' = y_2 \]
\[ y_2' = -|y_1| \]

Here \( y_1 = y \) and \( y_2 = y' \). This system has the required form

\[ y' = f(x, y) \]
\[ bc(y(a), y(b)) = 0 \]

The function \( f \) and the boundary conditions \( bc \) are coded in MATLAB software as functions \texttt{twoode} and \texttt{twobc}.

```matlab
function dydx = twoode(x,y)
    dydx = [ y(2) -abs(y(1)) ];
end

function res = twobc(ya,yb)
    res = [ ya(1) yb(1) + 2 ];
end
```

Form a guess structure consisting of an initial mesh of five equally spaced points in \([0,4]\) and a guess of constant values \( y_1(x) \equiv 1 \) and \( y_2(x) \equiv 0 \) with the command

\[ \text{solinit} = \text{bvpinit(linspace(0,4,5),[1 0])}; \]

Now solve the problem with

\[ \text{sol} = \text{bvp4c @} \text{twoode, twobc, solinit}; \]
Evaluate the numerical solution at 100 equally spaced points and plot $y(x)$ with

```matlab
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

```matlab
solinit = bvpinit(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 \textit{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 \texttt{bvp4c} in a single M-file.

    function mat4bvp
lambda = 15;
solinit = bvpinit(linspace(0,pi,10),@mat4init,lambda);
sol = bvp4c(@mat4ode,@mat4bc,solinit);

fprintf('The fourth eigenvalue is approximately %7.3f.\n',...  
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 \texttt{bvpinit}, the third argument (\texttt{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.

**References**


**See Also**

function_handle (@), bvp5c, bvpget, bvpinit, bvpset, bvpxtend, deval
**Purpose**  
Solve boundary value problems for ordinary differential equations

**Syntax**

```matlab
sol = bvp5c(odefun,bcfun,solinit)
sol = bvp5c(odefun,bcfun,solinit,options)
solinit = bvpinit(x, yinit, params)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| **odefun** | A function handle that evaluates the differential equations \( f(x,y) \). It can have the form

\[
\begin{align*}
\text{dydx} &= \text{odefun}(x,y) \\
\text{dydx} &= \text{odefun}(x,y,\text{parameters})
\end{align*}
\]
where \( x \) is a scalar corresponding to \( x \), and \( y \) is a column vector corresponding to \( y \). \( \text{parameters} \) is a vector of unknown parameters. The output \( \text{dydx} \) is a column vector. |
| **bcfun** | A function handle 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

\[
\begin{align*}
\text{res} &= \text{bcfun}(ya,yb) \\
\text{res} &= \text{bcfun}(ya,yb,\text{parameters})
\end{align*}
\]
where \( ya \) and \( yb \) are column vectors corresponding to \( y(a) \) and \( y(b) \). \( \text{parameters} \) is a vector of unknown parameters. The output \( \text{res} \) is a column vector. |
| **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 \( a = \text{solinit}.x(1) \) and \( b = \text{solinit}.x(\text{end}) \). |
**Description**

\[ y' = f(x, y) \]

on the interval \([a,b]\) subject to two-point boundary value conditions

\[ bc(y(a), y(b)) = 0 \]

odefun and bcfun are function handles. See “Function Handles” in the MATLAB Programming documentation for more information.

“Parametrizing Functions” in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function odefun, as well as the boundary condition function bcfun, if necessary. 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 bvp5c 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 \( \mathcal{P} \) corresponds to parameters. You provide \( \text{bvp5c} \) an initial guess for any unknown parameters in \( \text{solinit}.\text{parameters} \). The \( \text{bvp5c} \) solver returns the final values of these unknown parameters in \( \text{sol}.\text{parameters} \).

\( \text{bvp5c} \) produces a solution that is continuous on \([a,b]\) and has a continuous first derivative there. Use the function \( \text{deval} \) and the output \( \text{sol} \) of \( \text{bvp5c} \) to evaluate the solution at specific points \( x_{\text{int}} \) in the interval \([a,b]\).

\[
s_{\text{int}} = \text{deval}(\text{sol}, x_{\text{int}})
\]

The structure \( \text{sol} \) returned by \( \text{bvp5c} \) has the following fields:

- \( \text{sol}.x \) Mesh selected by \( \text{bvp5c} \)
- \( \text{sol}.y \) Approximation to \( y(x) \) at the mesh points of \( \text{sol}.x \)
- \( \text{sol}.\text{parameters} \) Values returned by \( \text{bvp5c} \) for the unknown parameters, if any
- \( \text{sol}.\text{solver} \) 'bvp5c'

The structure \( \text{sol} \) can have any name, and \( \text{bvp5c} \) creates the fields \( x, y, \text{parameters}, \) and \( \text{solver} \).  

\( \text{sol} = \text{bvp5c}(\text{odefun}, \text{bcfun}, \text{solinit}, \text{options}) \) solves as above with default integration properties replaced by the values in \( \text{options} \), a structure created with the \( \text{bvpset} \) function. See \( \text{bvpset} \) for details.

\( \text{solinit} = \text{bvpinit}(x, y_{\text{init}}, \text{params}) \) forms the initial guess \( \text{solinit} \) with the vector \( \text{params} \) of guesses for the unknown parameters.

**Singular Boundary Value Problems**

\( \text{bvp5c} \) solves a class of singular boundary value problems, including problems with unknown parameters \( p \), of the form
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**

`bvp5c` 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. `bvp5c` enumerates the regions from left to right (from \(a\) to \(b\)), with indices starting from 1. In region \(k\), \([a_{k - 1}, a_k]\), `bvp5c` evaluates the derivative as

\[yp = \text{odefun}(x, y, k)\]

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

\[yright(:, \text{end}) = y(b)\]

When you create an initial guess with
solinit = bvpinit(xinit, yinit),

use double entries in xinit for each interface point. See the reference page for bvpinit for more information.

If yinit is a function, bvpinit calls y = yinit(x, k) to get an initial guess for the solution at x in region k. In the solution structure sol returned by bvp5c, sol.x has double entries for each interface point. The corresponding columns of sol.y contain the left and right solution at the interface, respectively.

For an example of solving a three-point boundary value problem, type threebvp at the MATLAB command prompt to run a demonstration.

**Algorithms**

bvp5c is a finite difference code that implements the four-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fifth-order accurate uniformly in [a,b]. The formula is implemented as an implicit Runge-Kutta formula. bvp5c solves the algebraic equations directly; bvp4c uses analytical condensation. bvp4c handles unknown parameters directly; while bvp5c augments the system with trivial differential equations for unknown parameters.

**References**

[1] Shampine, L.F., M.W. Reichelt, and J. Kierzenka “Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c” http://www.mathworks.com/bvp_tutorial. Note that this tutorial uses the bvp4c function, however in most cases the solvers can be used interchangeably.

**See Also**

function_handle (@), bvp4c, bvpget, bvpinit, bvpset, bvpxtend, deval
bvpget

**Purpose**
Extract properties from options structure created with bvpset

**Syntax**

```matlab
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,

```matlab
val = bvpget(opts,'RelTol',1e-4);
```

returns val = 1e-4 if the RelTol is not specified in opts.

**See Also**

bvp4c, bvp5c, bvpinit, bvpset, deval
**Purpose**
Form initial guess for bvp4c

**Syntax**

\[
\begin{align*}
\text{solinit} &= \text{bvpinit}(x,yinit) \\
\text{solinit} &= \text{bvpinit}(x,yinit,\text{parameters}) \\
\text{solinit} &= \text{bvpinit}(\text{sol},[\text{anew bnew}]) \\
\text{solinit} &= \text{bvpinit}(\text{sol},[\text{anew bnew}],\text{parameters})
\end{align*}
\]

**Description**

\[
\text{solinit} = \text{bvpinit}(x,yinit)
\]
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 \(xb=\text{nlin}\text{space}(a,b,10)\) 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:
bvpinit

- 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, \( y_{\text{init}}(i) \) is a constant guess for the \( i \)th component \( y_{\text{init}}(i,:) \) 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, 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 \( y \) 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 \( k \).

\[
solinit = \text{bvpinit}(x,y_{\text{init}},\text{parameters})
\]

indicates that the boundary value problem involves unknown parameters. Use the vector \( \text{parameters} \) to provide a guess for all unknown parameters.

\( \text{solinit} \) is a structure with the following fields. The structure can have any name, but the fields must be named \( x, y, \) and \( \text{parameters} \).
x  Ordered nodes of the initial mesh.

y  Initial guess for the solution with \( \text{solinit}.y(:,i) \)
a guess for the solution at the node \( \text{solinit}.x(i) \).

parameters  Optional. A vector that provides an initial guess for unknown parameters.

\[
\text{solinit} = \text{bvpinit}(\text{sol},[\text{anew bnew}])
\]
forms an initial guess on the interval \([\text{anew bnew}]\) from a solution \(\text{sol}\) on an interval \([a, b]\).
The new interval must be larger than the previous one, so either
\[
\text{anew} \leq a < b \leq \text{bnew} \quad \text{or} \quad \text{anew} \geq a > b \geq \text{bnew}.
\]
The solution \(\text{sol}\) is extrapolated to the new interval. If \(\text{sol}\) contains parameters, they are copied to \(\text{solinit}\).

\[
\text{solinit} = \text{bvpinit}(\text{sol},[\text{anew bnew}],\text{parameters})
\]
forms \(\text{solinit}\) as described above, but uses \(\text{parameters}\) as a guess for unknown parameters in \(\text{solinit}\).

**See Also**

@ (function_handle), bvp4c,bvp5c, bvpget, bvpset, bvpxtend, deval
Purpose

Create or alter options structure of boundary value problem

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 that you can supply to the boundary value problem solver bvp4c, in which the named properties have the specified values. Any unspecified properties retain their default values. For all properties, it is sufficient to type only the leading characters that uniquely identify the property. bvpset ignores case for property names.

options = bvpset(oldopts,'name1',value1,...) alters an existing options structure oldopts. This overwrites any values in oldopts that are specified using name/value pairs and returns the modified structure as the output argument.

options = bvpset(oldopts,newopts) combines an existing options structure oldopts with a new options structure newopts. Any values set in newopts overwrite the corresponding values in oldopts.

bvpset with no input arguments displays all property names and their possible values, indicating defaults with braces {}.

You can use the function bvpget to query the options structure for the value of a specific property.

BVP Properties

bvpset enables you to specify properties for the boundary value problem solver bvp4c. There are several categories of properties that you can set:

- “Error Tolerance Properties” on page 2-477
- “Vectorization” on page 2-478
- “Analytical Partial Derivatives” on page 2-479
- “Singular BVPs” on page 2-482
Error Tolerance Properties

Because `bvp4c` uses a collocation formula, the numerical solution is based on a mesh of points at which the collocation equations are satisfied. Mesh selection and error control are based on the residual of this solution, such that the computed solution $S(x)$ is the exact solution of a perturbed problem $S'(x) = f(x, S(x)) + res(x)$. On each subinterval of the mesh, a norm of the residual in the $i$th component of the solution, $res(i)$, is estimated and is required to be less than or equal to a tolerance. This tolerance is a function of the relative and absolute tolerances, $RelTol$ and $AbsTol$, defined by the user.

$$\left\| (res(i)/\max(|abs(f(i)|,AbsTol(i)/RelTol)) \right\| \leq RelTol$$

The following table describes the error tolerance properties.
### BVP Error Tolerance Properties

<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 error tolerance that applies to all components of the residual vector. It is a measure of the residual relative to the size of $\vec{f}(x, y)$. The default, 1e-3, corresponds to 0.1% accuracy. The computed solution $S(x)$ is the exact solution of $S'(x) = F(x, S(x)) + \text{res}(x)$. On each subinterval of the mesh, the residual $\text{res}(x)$ satisfies $</td>
</tr>
<tr>
<td>AbsTol</td>
<td>Positive scalar or vector {1e-6}</td>
<td>Absolute error tolerances that apply to the corresponding components of the residual vector. $\text{AbsTol}(i)$ is a threshold below which the values of the corresponding components are unimportant. If a scalar value is specified, it applies to all components.</td>
</tr>
</tbody>
</table>

### Vectorization

The following table describes the BVP vectorization property. Vectorization of the ODE function used by bvp4c differs from the vectorization used by the ODE solvers:

- For bvp4c, the ODE function must be vectorized with respect to the first argument as well as the second one, so that $F([x1 \ x2 \ldots], [y1 \ y2 \ldots])$ returns $[F(x1,y1) \ F(x2,y2) \ldots]$.  
- bvp4c benefits from vectorization even when analytical Jacobians are provided. For stiff ODE solvers, vectorization is ignored when analytical Jacobians are used.
## Vectorization Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorized</td>
<td>on</td>
<td>Set on to inform bvp4c that you have coded the ODE function F so that F([x1 x2 ...],[y1 y2 ...]) returns [F(x1,y1) F(x2,y2) ...]. That is, your ODE function can pass to the solver a whole array of column vectors at once. This enables the solver to reduce the number of function evaluations and may significantly reduce solution time. With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. In the shockbvp example shown previously, the shockODE function has been vectorized using colon notation into the subscripts and by using the array multiplication (.* operator).</td>
</tr>
<tr>
<td></td>
<td>{off}</td>
<td></td>
</tr>
</tbody>
</table>

```matlab
function dydx = shockODE(x,y,e)
pix = pi*x;
dydx = [ y(2,:)
         -x/e.*y(2,:) -pi^2*cos(pix)
         -pix/e.*sin(pix)];
```

## Analytical Partial Derivatives

By default, the bvp4c solver approximates all partial derivatives with finite differences. bvp4c can be more efficient if you provide analytical partial derivatives $\frac{\partial f}{\partial y}$ of the differential equations, and analytical
partial derivatives, $\partial bc/\partial y_a$ and $\partial bc/\partial y_b$, of the boundary conditions. If the problem involves unknown parameters, you must also provide partial derivatives, $\partial f/\partial p$ and $\partial bc/\partial p$, with respect to the parameters.

The following table describes the analytical partial derivatives properties.
## BVP Analytical Partial Derivative Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FJacobian</td>
<td>Function handle</td>
<td>Handle to a function that computes the analytical partial derivatives of $f(x, y)$. When solving $y' = f(x, y)$, set this property to @fjac if $dfdy = fjac(x, y)$ evaluates the Jacobian $\frac{\partial f}{\partial y}$. If the problem involves unknown parameters $P$, $[dfdy, dfdp] = fjac(x, y, p)$ must also return the partial derivative $\frac{\partial f}{\partial p}$. For problems with constant partial derivatives, set this property to the value of $dfdy$ or to a cell array ${dfdy, dfdp}$. See “Function Handles” in the MATLAB Programming documentation for more information.</td>
</tr>
<tr>
<td>BCJacobian</td>
<td>Function handle</td>
<td>Handle to a function that computes the analytical partial derivatives of $bc(ya, yb)$. For boundary conditions $bc(ya, yb)$, set this property to @bcjac if $[dbcdnya, dbcdnyb] = bcjac(ya, yb)$ evaluates the partial derivatives $\frac{\partial bc}{\partial ya}$ and $\frac{\partial bc}{\partial yb}$. If the problem involves unknown parameters $P$, $[dbcdnya, dbcdnyb, dbcdnp] = bcjac(ya, yb, p)$ must also return the partial derivative $\frac{\partial bc}{\partial p}$. For problems with constant partial derivatives, set this property to a cell array ${dbcdnya, dbcdnyb}$ or ${dbcdnya, dbcdnyb, dbcdnp}$.</td>
</tr>
</tbody>
</table>
**Singular BVPs**

`bvp4c` can solve singular problems of the form

\[ y' = S \frac{y}{x} + f(x, y, p) \]

posed on the interval \([0, b]\) where \(b > 0\). For such problems, specify the constant matrix \(S\) as the value of `SingularTerm`. For equations of this form, `odefun` evaluates only the \(f(x, y, p)\) term, where \(p\) represents unknown parameters, if any.

**Singular BVP Property**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SingularTerm</code></td>
<td>Constant matrix</td>
<td>Singular term of singular BVPs. Set to the constant matrix (S) for equations of the form [ y' = S \frac{y}{x} + f(x, y, p) ] posed on the interval ([0, b]) where (b &gt; 0).</td>
</tr>
</tbody>
</table>

**Mesh Size Property**

`bvp4c` solves a system of algebraic equations to determine the numerical solution to a BVP at each of the mesh points. The size of the algebraic system depends on the number of differential equations (\(n\)) and the number of mesh points in the current mesh (\(N\)). When the allowed number of mesh points is exhausted, the computation stops, `bvp4c` displays a warning message and returns the solution it found so far. This solution does not satisfy the error tolerance, but it may provide an
excellent initial guess for computations restarted with relaxed error tolerances or an increased value of NMax.

The following table describes the mesh size property.

### BVP Mesh Size Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMax</td>
<td>positive integer ({\text{floor}(1000/n)})</td>
<td>Maximum number of mesh points allowed when solving the BVP, where (n) is the number of differential equations in the problem. The default value of NMax limits the size of the algebraic system to about 1000 equations. For systems of a few differential equations, the default value of NMax should be sufficient to obtain an accurate solution.</td>
</tr>
</tbody>
</table>

### Solution Statistic Property

The Stats property lets you view solution statistics.

The following table describes the solution statistics property.
BVP Solution Statistic Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stats</td>
<td>on</td>
<td>Specifies whether statistics about the computations are displayed. If the stats property is on, after solving the problem, bvp4c displays:</td>
</tr>
<tr>
<td></td>
<td>{off}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The number of points in the mesh</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The maximum residual of the solution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The number of times it called the differential equation function odefun to evaluate $f(x, y)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The number of times it called the boundary condition function bcfun to evaluate $bc(y(a), y(b))$</td>
</tr>
</tbody>
</table>

Example

To create an options structure that changes the relative error tolerance of bvp4c from the default value of $1e^{-3}$ to $1e^{-4}$, enter

```matlab
options = bvpset('RelTol', 1e-4);
```

To recover the value of 'RelTol' from options, enter

```matlab
bvpget(options, 'RelTol')
```

```
ans =
1.0000e-004
```

See Also
@ (function_handle), bvp4c, bvp5c, bvpget, bvpinit, deval
Purpose
Form guess structure for extending boundary value solutions

Syntax
solinit = bvpxtend(sol,xnew,ynew)
solinit = bvpxtend(sol,xnew,extrap)
solinit = bvpxtend(sol,xnew)
solinit = bvpxtend(sol,xnew,ynew,pnew)
solinit = bvpxtend(sol,xnew,extrap,pnew)

Description
solinit = bvpxtend(sol,xnew,ynew) uses solution sol computed on [a,b] to form a solution guess for the interval extended to xnew. The extension point xnew must be outside the interval [a,b], but on either side. The vector ynew provides an initial guess for the solution at xnew.

solinit = bvpxtend(sol,xnew,extrap) forms the guess at xnew by extrapolating the solution sol. extrap is a string that determines the extrapolation method. extrap has three possible values:

- 'constant' — ynew is a value nearer to end point of solution in sol.
- 'linear' — ynew is a value at xnew of linear interpolant to the value and slope at the nearer end point of solution in sol.
- 'solution' — ynew is the value of (cubic) solution in sol at xnew.

The value of extrap is case-insensitive and only the leading, unique portion needs to be specified.

solinit = bvpxtend(sol,xnew) uses the extrapolating solution where extrap is 'constant'. If there are unknown parameters, values present in sol are used as the initial guess for parameters in solinit.

solinit = bvpxtend(sol,xnew,ynew,pnew) specifies a different guess pnew. pnew can be used with extrapolation, using the syntax solinit = bvpxtend(sol,xnew,extrap,pnew). To modify parameters without changing the interval, use [] as place holder for xnew and ynew.

See Also
bvp4c, bvp5c, bvpinit
Purpose

Calendar for specified month

Syntax

c = calendar

c = calendar(d)

c = calendar(y, m)

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.

Examples

The command

calendar(1957,10)

reveals that the Space Age began on a Friday (on October 4, 1957, when Sputnik 1 was launched).

```
Oct 1957
S M Tu W Th F S
0 0 1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31 0 0
0 0 0 0 0 0 0
```

See Also

datenum
Purpose

Call function in shared library

Syntax

\[ x_1, \ldots, x_N = \text{calllib('libname', 'funcname', arg1, \ldots, argN)} \]

Description

\[ x_1, \ldots, x_N = \text{calllib('libname', 'funcname', arg1, \ldots, argN)} \]
callsthefunction
funcname
inlibrary
libname,
passinginput
arguments
arg1 through argN.
calllib returns output values obtained
from function funcname in x1 through xN.

All scalar values returned by MATLAB are of type double.

If you used an alias when initially loading the library, then you must
use that alias for the libname argument.

Ways to Call calllib

The following examples show ways calls to calllib. By using
libfunctionsview, you determined that the addStructByRef function
in the shared library shrlibsamplesample requires a pointer to a c_struct
data type as its argument.

Load the library:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsamplesample shrlibsamplesample.h
```

Create a MATLAB structure:

```
struct.p1 = 4; struct.p2 = 7.3; struct.p3 = -290;
```

Use libstruct to create a C structure of the proper type (c_struct):

```
[res, st] = calllib('shrlibsamplesample','addStructByRef', ...
libstruct('c_struct', struct));
```

Let MATLAB convert struct to the proper type of C structure:

```
[res, st] = calllib('shrlibsamplesample','addStructByRef', struct);
```
Pass an empty array to `libstruct` and assign the values from your C function:

```matlab
[res,st] = calllib('shrlibsample','addStructByRef',... libstruct('c_struct',[]));
```

Let MATLAB create the proper type of structure and assign values from your C function:

```matlab
[res,st] = calllib('shrlibsample','addStructByRef',[]);
```

Remove the library from memory:

```matlab
unloadlibrary shrlibsample
```

**Examples**

To call functions in the MATLAB libmx library, see “Invoking Library Functions”.

**See Also**

`loadlibrary`, `libfunctions`, `libfunctionsview`, `unloadlibrary`

See Passing Arguments for information on defining the correct data types for library function arguments.
### Purpose
Send SOAP message to endpoint

### Syntax
```matlab
response = callSoapService(endpoint, soapAction, message)
```

### Description
`response = callSoapService(endpoint, soapAction, message)` sends `message`, a Sun Java document object model (DOM), to the `soapAction` service at `endpoint`. Create `message` using `createSoapMessage`, and extract results from `response` using `parseSoapResponse`.

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

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

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

author = parseSoapResponse(response)

MATLAB returns:

author = Kate Alvin

where author is a char class (type).

See Also

createClassFromWsdl, createSoapMessage, parseSoapResponse, urlread, xmlread

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

Move 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 the camera moves:

- **movetarget** (default) — Move both the camera and the target.
- **fixtarget** — Move only the camera.

camdolly(dx,dy,dz,'targetmode','coordsys') The **coordsys** argument can take on three values that determine how the MATLAB software 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.

```matlab
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

“Camera Viewpoint” on page 1-106 for related functions

See “Defining Scenes with Camera Graphics” for more information on camera properties.
### Purpose
Control camera toolbar programatically

### 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')
cameratoolbar(fig,...)
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'. For descriptions of the various modes, see “Camera Toolbar” in the MATLAB 3-D Visualization User’s Guide. You can also set these modes using the toolbar, by clicking on the respective buttons.

`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).
cameratoolbar(fig,...) specifies the figure to operate on by passing the figure handle as the first argument.
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

“Camera Toolbar”
Purpose
Create or move 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.
camlight

Examples
This example creates a light positioned to the left of the camera and then repositions the light each time the camera is moved:

    surf(peaks)
    axis vis3d
    h = camlight('left');
    for i = 1:20;
        camorbit(10,0)
        camlight(h,'left')
        drawnow;
    end

See Also
light, lightangle

“Lighting” on page 1-108 for related functions

“Lighting as a Visualization Tool” for more information on using lights
Purpose

Position camera to view 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

“Camera Viewpoint” on page 1-106 for related functions

“Defining Scenes with Camera Graphics” for more information
**Purpose**

Rotate camera position around camera target

**Syntax**

```
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:

```
surf(peaks)
```
camorbit

```matlab
axis vis3d
for i=1:36
    camorbit(10,0,'data',[0 1 0])
    drawnow
end
```

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
```

Remarks
The behavior of cameraorbit differs from the rotate3d function in that while the rotate3d tool modifies the View property of the axes, the cameraorbit function fixes the aspect ratio and modifies the CameraTarget, CameraPosition and CameraUpVector properties of the axes. See Axes Properties for more information.

You can also enable 3-D rotation from the figure Tools menu or the figure toolbar.

See Also
axes, axis('vis3d'), camdolly, campan, camzoom, camroll
“Camera Viewpoint” on page 1-106 for related functions
“Defining Scenes with Camera Graphics” for more information
Axes Properties for related properties
Purpose

Rotate camera target around camera position

Syntax

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

“Camera Viewpoint” on page 1-106 for related functions

“Defining Scenes with Camera Graphics” for more information
Purpose

Set or query 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:

```matlab
surf(peaks)
axis vis3d off
for x = -200:5:200
   campos([x,5,10])
drawnow
```
See Also

axis, camproj, camtarget, camup, camva

The axes properties CameraPosition, CameraTarget, CameraUpVector, CameraViewAngle, Projection

“Camera Viewpoint” on page 1-106 for related functions

“Defining Scenes with Camera Graphics” for more information
camproj

**Purpose**
Set or query projection type

**Syntax**
camproj
```
camproj('projection_type')
camproj(axes_handle, ...)
```

**Description**
The projection type determines whether MATLAB 3-D views use a perspective or orthographic projection.

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

“Camera Viewpoint” on page 1-106 for related functions

“Defining Scenes with Camera Graphics” for more information
**Purpose**  
Rotate camera about 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  
“Camera Viewpoint” on page 1-106 for related functions  
“Defining Scenes with Camera Graphics” for more information
Purpose
Set or query location of 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, the camera target is 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

“Camera Viewpoint” on page 1-106 for related functions

“Defining Scenes with Camera Graphics” for more information
camup

**Purpose**
Set or query 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, `[0 1 0]` is the up vector of for 2-D views. This means the z-axis points up.

`camup('manual')` sets the camera up vector mode to `manual`. In `manual` mode, the value of the camera up vector does not change unless you set it.

`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

“Camera Viewpoint” on page 1-106 for related functions

“Defining Scenes with Camera Graphics” for more information
Purpose  
Set or query camera view angle

Syntax

```matlab
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`, the camera view angle adjusts so that the scene fills the available space in the window. If you move the camera to a different position, the camera view angle changes 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 axes reference page for more information.

**Examples**

This example creates two pushbuttons, one that zooms in and another that zooms out.

```matlab
text{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:

```
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.

**See Also**

axis, camproj, campos, camup, camtarget

The axes properties CameraPosition, CameraTarget, CameraUpVector, CameraViewAngle, Projection

“Camera Viewpoint” on page 1-106 for related functions

“Defining Scenes with Camera Graphics” for more information
Purpose

Zoom in and out on 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

“Camera Viewpoint” on page 1-106 for related functions

“Defining Scenes with Camera Graphics” for more information
Purpose

Convert point coordinates from cartesian to barycentric

Syntax

B = cartToBary(TR, SI, XC)

Description

B = cartToBary(TR, SI, XC) returns the barycentric coordinates of each point in XC with respect to its associated simplex SI.

Inputs

TR
Triangulation representation.

SI
Column vector of simplex indices that index into the triangulation matrix TR.Triangulation.

XC
Matrix that represents the Cartesian coordinates of the points to be converted. XC is of size m-by-n, where m is of length(SI), the number of points to convert, and n is the dimension of the space where the triangulation resides.

Outputs

B
Matrix of dimension m-by-k where k is the number of vertices per simplex.

Definitions

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

Examples

Compute the Delaunay triangulation of a set of points.

```matlab
x = [0 4 8 12 0 4 8 12]';
y = [0 0 0 0 8 8 8 8]';
dt = DelaunayTri(x,y)
```

Compute the barycentric coordinates of the incenters.

```matlab
cc = incenters(dt);
tri = dt(:,:);
subplot(1,2,1);
subplot(1,2,1);
triplot(dt); hold on;
```
plot(cc(:,1), cc(:,2), '*r');
hold off;
axis equal;
title(sprintf('Original triangulation and reference ... points.'));

Stretch the triangulation and compute the mapped locations of the incenters on the deformed triangulation.

b = cartToBary(dt,[1:length(tri)'],cc);
y = [0 0 0 0 16 16 16 16]';
tr = TriRep(tri,x,y)
xc = baryToCart(tr, [1:length(tri)'], b);
subplot(1,2,2);
triplot(tr);
hold on;
plot(xc(:,1), xc(:,2), '*r');
hold off;
axis equal;
title(sprintf('Deformed triangulation and mapped
locations of the reference points.'));
See Also

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

Three-Dimensional Mapping
\[
\text{theta} = \text{atan2}(y, x) \\
\text{rho} = \text{sqrt}(x.^2 + y.^2) \\
\text{z} = z
\]
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 (or any of them can be scalar).

Algorithm
The mapping from three-dimensional Cartesian coordinates to spherical coordinates is

\[
\theta = \text{atan2}(y, x) \\
\phi = \text{atan2}(z, \sqrt{x^2 + y^2}) \\
R = \sqrt{x^2 + y^2 + z^2}
\]

The notation for spherical coordinates is not standard. For the cart2sph function, the angle PHI is measured from the x-y plane. Notice that if PHI = 0 then the point is in the x-y plane and if PHI = pi/2 then the point is on the positive z-axis.

See Also
cart2pol, pol2cart, sph2cart
Purpose

Execute block of code if condition is true

Syntax

switch switch_expr
    case case_expr
        statement, ..., statement
    case {case_expr1, case_expr2, case_expr3, ...}
        statement, ..., statement
    otherwise
        statement, ..., statement
end

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.

case case_expr compares the value of the expression switch_expr declared in the preceding switch statement with one or more values in case_expr, and executes the block of code that follows if any of the comparisons yield a true result.

You typically use multiple case statements in the evaluation of a single switch statement. The block of code associated with a particular case statement is executed only if its associated case expression (case_expr) is the first to match the switch expression (switch_expr).

To enter more than one case expression in a switch statement, put the expressions in a cell array, as shown above.

Examples

To execute a certain block of code based on what the string, method, is set to,

    method = 'Bilinear';

    switch lower(method)
        case {'linear','bilinear'}
            disp('Method is linear')
        case 'cubic'

case

disp('Method is cubic')
case 'nearest'
disp('Method is nearest')
otherwise
disp('Unknown method.')
end

Method is linear

See Also switch, otherwise, end, if, else, elseif, while
Purpose

Cast variable to different data type

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.

Examples

a = int8(5);
b = cast(a,'uint8');
class(b)

ans =

uint8

See Also

class
cat

Purpose

Concatenate arrays along specified dimension

Syntax

C = cat(dim, A, B)
C = cat(dim, A1, A2, A3, A4, ...)

Description

C = cat(dim, A, B) concatenates the arrays A and B along dim.

C = cat(dim, A1, A2, A3, A4, ...) concatenates all the input arrays (A1, A2, A3, A4, and so on) along dim.

cat(2, A, B) is the same as [A, B], and cat(1, A, B) is the same as [A; B].

Remarks

When used with comma-separated list syntax, cat(dim, C{:}) or cat(dim, C.field) is a convenient way to concatenate a cell or structure array containing numeric matrices into a single matrix.

Examples

Given

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

concatenating along different dimensions produces

The commands

\[
C = \text{cat}(1,A,B) \quad C = \text{cat}(2,A,B) \quad C = \text{cat}(3,A,B)
\]
A = magic(3); B = pascal(3);
C = cat(4, A, B);

produce a 3-by-3-by-1-by-2 array.

See Also vertcat, horzcat, strcat, strvcat, num2cell, special character []
**Purpose**
Specify how to respond to error in `try` statement

**Syntax**
catch ME

**Description**
catch ME marks the start of a `catch block` in a `try-catch` statement. It returns object ME, which is an instance of the MATLAB class `MException`. This object contains information about an error caught in the preceding `try block` and can be useful in helping your program respond to the error appropriately.

A `try-catch` statement is a programming device that enables you to define how certain errors are to be handled in your program. This bypasses the default MATLAB error-handling mechanism when these errors are detected. The `try-catch` statement consists of two blocks of MATLAB code, a `try block` and a `catch block`, delimited by the keywords `try`, `catch`, and `end`:

```matlab
try
    MATLAB commands % Try block
catch ME
    MATLAB commands % Catch block
end
```

Each of these blocks consists of one or more MATLAB commands. The `try` block is just another piece of your program code; the commands in this block execute just like any other part of your program. Any errors MATLAB encounters in the `try` block are dealt with by the respective `catch` block. This is where you write your error-handling code. If the `try` block executes without error, MATLAB skips the `catch` block entirely. If an error occurs while executing the `catch` block, the program terminates unless this error is caught by another `try-catch` block.

catch marks the start of a catch block but does not return an `MException` object. You can obtain the error string that was generated by calling the `lasterror` function.
Specifying the `try`, `catch`, and `end` commands, as well as the commands that make up the `try` and `catch` blocks, on separate lines is recommended. If you combine any of these components on the same line, separate them with commas:

```
try, surf, catch ME, ME.stack, end
```

```
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

the term `matlabroot` represents the string returned by the `matlabroot` function.

**Examples**

The `catch` block in this example checks to see if the specified file could not be found. If this is the case, the program allows for the possibility that a common variation of the filename extension (e.g., `jpeg` instead of `jpg`) was used by retrying the operation with a modified extension. This is done using a `try-catch` statement that is nested within the original `try-catch`.

```
function d_in = read_image(filename)
    [path name ext] = fileparts(filename);
    try
        fid = fopen(filename, 'r');
        d_in = fread(fid);
    catch ME1
        % Get last segment of the error message identifier.
        idSegLast = regexp(ME1.identifier, '(?<=:)\w+$', ... 
                          'match');
        % Did the read fail because the file could not be found?
        if strcmp(idSegLast, 'InvalidFid') && ... 
            ~exist(filename, 'file')
            % Yes. Try modifying the filename extension.
            switch ext
```
catch

case '.jpg' % Change jpg to jpeg
    filename = strrep(filename, '.jpg', '.jpeg')
case '.jpeg' % Change jpeg to jpg
    filename = strrep(filename, '.jpeg', '.jpg')
case '.tif' % Change tif to tiff
    filename = strrep(filename, '.tif', '.tiff')
case '.tiff' % Change tiff to tif
    filename = strrep(filename, '.tiff', '.tif')
otherwise
    fprintf('File %s not found
', filename);
    rethrow(ME1);
end

% Try again, with modified filenames.
try
    fid = fopen(filename, 'r');
    d_in = fread(fid);
    catch ME2
        fprintf('Unable to access file %s
', filename);
        ME2 = addCause(ME2, ME1);
        rethrow(ME2)
    end
end
end

See Also  try, rethrow, end, lasterror, eval, evalin
**Purpose**  
Color axis scaling

**Syntax**  
- `caxis([cmin cmax])`
- `caxis auto`
- `caxis manual`
- `caxis(caxis) freeze`
- `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` computes 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. The following linear transformation is performed 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).

\[
\text{index} = \text{fix}((C-\text{cmin})/(\text{cmax}-\text{cmin})*(m-1))+1
\]

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

\[
\text{caxis([-1 0])}
\]

To use only the bottom half of the color table, enter

\[
\text{caxis([-1 3])}
\]

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

\[
\text{caxis auto}
\]
resets axis scaling back to autoranging and you see all the colors in the surface. In this case, entering

```matlab
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)
```

This adjusts 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” on page 1-105 for related functions
“Axes Color Limits — the CLim Property” for more examples
Purpose

Change working directory

GUI Alternatives

As an alternative to the cd function, you can change the current directory using the current directory field on the desktop toolbar or using the Current Directory browser.

Syntax

cd
w = cd
cd('directory')
cd('..')
cd directory

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 path 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

UNIX Platforms

On UNIX platforms, to change the current working directory to ctrldemos for the Control System Toolbox software, run

    cd('/usr/local/matlab/toolbox/control/ctrldemos')

Windows Platforms

On Microsoft Windows platforms, to change the current working directory to ctrldemos for the Control System Toolbox software, run

1. UNIX is a registered trademark of The Open Group in the United States and other countries.
cd('c:/matlab/toolbox/control/ctrl1demos')

Then change the current working directory to control by running

cd ..

Then change the current working directory to toolbox by running

cd ..

**Change to matlabroot Directory**

On any platform, use cd with the matlabroot function to change to a directory relative to the directory in which the MATLAB executable is installed. For example,

```matlab
cd([matlabroot '/toolbox/control/ctrl1demos'])
```

changes the current working directory to ctrl1demos for the Control System Toolbox software.

See Also

dir, fileparts, mfilename, path, pwd, what

“Managing Files and Working with the Current Directory”
**Purpose**
Convex hull

**Syntax**

```
K = convexHull(DT)
[K AV] = convexHull(DT)
```

**Description**

- `K = convexHull(DT)` returns the indices into the array of points `DT.X` that correspond to the vertices of the convex hull.
- `[K AV] = convexHull(DT)` returns the convex hull and the area or volume bounded by the convex hull.

**Inputs**

- `DT` Delaunay triangulation.

**Outputs**

- `K` If the points lie in 2-D space, `K` is a column vector of length `numf`. Otherwise `K` is a matrix of size `numf`-by-`ndim`, `numf` being the number of facets in the convex hull, and `ndim` the dimension of the space where the points reside.
- `AV` The area or volume of the convex hull.

**Definitions**
The convex hull of a set of points `X` is the smallest convex polygon (or polyhedron in higher dimensions) containing all of the points of `X`.

**Examples**

**Example 1**
Compute the convex hull of a set of random points located within a unit square in 2-D space.

```matlab
x = rand(10,1);
y = rand(10,1);
dt = DelaunayTri(x,y);
k = convexHull(dt);
plot(x,y, '.', 'markersize',10);
hold on;
```
Example 2

Compute the convex hull of a set of random points located within a unit cube in 3-D space and the volume bounded by the convex hull.

```matlab
X = rand(25,3);
dt = DelaunayTri(X);
[ch v] = convexHull(dt);
trisurf(ch, X(:,1),X(:,2),X(:,3), 'FaceColor', 'cyan')
```
**DelaunayTri.convexHull**

**See Also**

- DelaunayTri.voronoiDiagram
- TriRep
- convhull
- convhulln
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)
```

Change the current directory to pub.

```
  cd(tmw,'pub');
```

View the contents of pub.

```
dir(tmw)
```

See Also

dir (ftp), ftp
**Purpose**
Convert complex diagonal form to real block diagonal form

**Syntax**

```
[V,D] = cdf2rdf(V,D)
```

**Description**
If the eigensystem \([V,D] = \text{eig}(X)\) has complex eigenvalues appearing in complex-conjugate pairs, `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.

\[
[V,D] = \text{eig}(X)
\]

\[
V = 
\begin{bmatrix}
1.0000 & -0.0191 - 0.4002i & -0.0191 + 0.4002i \\
0 & 0 - 0.6479i & 0 + 0.6479i \\
0 & 0.6479 & 0.6479 \\
\end{bmatrix}
\]

\[
D = 
\begin{bmatrix}
1.0000 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]
Converting this to real block diagonal form produces

\[
\begin{bmatrix}
V & D
\end{bmatrix}
\]

\[
V =
\begin{bmatrix}
1.0000 & -0.0191 & -0.4002 \\
0 & 0 & -0.6479 \\
0 & 0.6479 & 0
\end{bmatrix}
\]

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

**Algorithm**
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 cdfepoch object for Common Data Format (CDF) export

### Syntax
```matlab
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.

Use the `todatenum` function to convert a `cdfepoch` object into a MATLAB serial date number.

---

**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
Information about Common Data Format (CDF) file

Syntax
info = cdfinfo(filename)

Description
info = cdfinfo(filename) returns information about the Common Data Format (CDF) file specified in the string filename.

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>Field</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Subfiles</td>
<td>Filenames containing the CDF file’s data, if it is a multfile CDF</td>
</tr>
<tr>
<td>VariableAttributes</td>
<td>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.</td>
</tr>
<tr>
<td><strong>Field</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>Variables</td>
<td>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:</td>
</tr>
<tr>
<td>Column 1</td>
<td>Text string specifying name of variable</td>
</tr>
<tr>
<td>Column 2</td>
<td>Double array specifying the dimensions of the variable, as returned by the <code>size</code> function</td>
</tr>
<tr>
<td>Column 3</td>
<td>Double scalar specifying the number of records assigned for the variable</td>
</tr>
<tr>
<td>Column 4</td>
<td>Text string specifying the data type of the variable, as stored in the CDF file</td>
</tr>
</tbody>
</table>
| Column 5  | 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.  

- T/ (scalar variable)  
- F/T (one-dimensional variable)  
- T/TFF (three-dimensional variable) |
| Column 6  | Text string specifying the sparsity of the variable’s records, with these possible values:  

- 'Full'  
- 'Sparse (padded)'  
- 'Sparse (nearest)' |
**cdfinfo**

**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: '09-Mar-2001 15:45:22'
    FileSize: 1240
    Format: 'CDF'
    FormatVersion: '2.7.0'
    FileSettings: [1x1 struct]
    Subfiles: {}
    Variables: [5x6 cell]
    GlobalAttributes: [1x1 struct]
    VariableAttributes: [1x1 struct]

info.Variables
ans =
    'Time' [1x2 double] [24] 'epoch' 'T/' 'Full'
    'Longitude' [1x2 double] [ 1] 'int8' 'F/FT' 'Full'
    'Latitude' [1x2 double] [ 1] 'int8' 'F/TF' 'Full'
    'Data' [1x3 double] [ 1] 'double' 'T/TTT' 'Full'
    'multidim' [1x4 double] [ 1] 'uint8' 'T/TTTT' 'Full'
```

**See Also**

cdfread
Purpose
Read data from Common Data Format (CDF) file

Syntax

data = cdfread(filename)

data = cdfread(filename, param1, val1, param2, val2, ...)

[data, info] = cdfread(filename, ...)

Description
data = cdfread(filename) reads all the data from the Common Data Format (CDF) file specified in the string filename. CDF data sets typically contain a set of variables, of a specific data type, each with an associated set of records. The variable might represent time values with each record representing a specific time that an observation was recorded. cdfread returns all the data in a cell array where each column represents a variable and each row represents a record associated with a variable. If the variables have varying numbers of associated records, cdfread pads the rows to create a rectangular cell array, using pad values defined in the CDF file.

Note
Because cdfread creates temporary files, the current working directory must be writeable.

data = cdfread(filename, param1, val1, param2, val2, ...) reads data from the file, where param1, param2, and so on, can be any of the following parameters.

Note
Note: When working with large data files, use of the 'ConvertEpochToDatenum' and 'CombineRecords' options can significantly improve performance.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Records'</td>
<td>A vector specifying which records to read. Record numbers are zero-based. <code>cdfread</code> returns a cell array with the same number of rows as the number of records read and as many columns as there are variables.</td>
</tr>
<tr>
<td>'Variables'</td>
<td>A 1-by-( n ) or ( n )-by-1 cell array specifying the names of the variables to read from the file. ( n ) must be less than or equal to the total number of variables in the file. <code>cdfread</code> returns a cell array with the same number of columns as the number of variables read, and a row for each record read.</td>
</tr>
<tr>
<td>'Slices'</td>
<td>An ( m )-by-3 array, where each row specifies where to start reading along a particular dimension of a variable, the skip interval to use on that dimension (every item, every other item, etc.), and the total number of values to read on that dimension. ( m ) must be less than or equal to the number of dimensions of the variable. If ( m ) is less than the total number of dimensions, <code>cdfread</code> reads every value from the unspecified dimensions ([0 1 n]), where ( n ) is the total number of elements in the dimension. Note: Because the 'Slices' parameter describes how to process a single variable, it must be used in conjunction with the 'Variables' parameter.</td>
</tr>
</tbody>
</table>
Parameter | Value
--- | ---
'ConvertEpochToDatenum' | A Boolean value that determines whether `cdfread` automatically converts CDF epoch data types to MATLAB serial date numbers. If set to `false` (the default), `cdfread` wraps epoch values in MATLAB `cdfepoch` objects. Note: For better performance when reading large data sets, set this parameter to `true`.

'CombineRecords' | A Boolean value that determines how `cdfread` returns the CDF data sets read from the file. If set to `false` (the default), `cdfread` stores the data in an `m`-by-`n` cell array, where `m` is the number of records and `n` is the number of variables requested. If set to `true`, `cdfread` combines all records for a particular variable into one cell in the output cell array. In this cell, `cdfread` stores scalar data as a column array. `cdfread` extends the dimensionality of nonscalar and string data. For example, instead of creating 1000 elements containing 20-by-30 arrays for each record, `cdfread` stores all the records in one cell as a 1000-by-20-by-30 array. Note: If you use the 'Records' parameter to specify which records to read, you cannot use the 'CombineRecords' parameter. Note: When using the 'Variable' parameter to read one variable, if the 'CombineRecords' parameter is `true`, `cdfread` returns the data as an `M`-by-`N` numeric or character array; it does not put the data into a cell array.

```matlab
[data, info] = cdfread(filename, ...) returns details about the CDF file in the info structure.
```

**Examples**

Read all the data from a CDF file.

```matlab
data = cdfread('example.cdf');
```

Read the data from the 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 matrix indexing, as in the following.

data{1}(1, 2, [1 3], :)

Collapse the records from a data set and convert CDF epoch data types to MATLAB serial date numbers.

data = cdfread('example.cdf', ...
    'CombineRecords', true, ...
    'ConvertEpochToDatenum', true);

**See Also**
cdfepoch, cdfinfo, cdfwrite

For more information about using this function, see “Common Data Format (CDF) Files”.
Purpose

Write data to Common Data Format (CDF) file

Syntax

cdfwrite(filename,variablelist)
cdfwrite(...,'PadValues',padvals)
cdfwrite(...,'GlobalAttributes',gattrib)
cdfwrite(...,'VariableAttributes',vattrib)
cdfwrite(...,'WriteMode',mode)
cdfwrite(...,'Format',format)

Description

cdfwrite(filename,variablelist) writes out a Common Data Format (CDF) file, specified in filename. The filename input is a string enclosed in single quotes. 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 invalid in your MATLAB application, 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].

```matlab
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.

```matlab
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.

```matlab
varAttribStruct.validmin = {'longitude' [10]};
cdfwrite('example', {'Longitude' 0:360}, 'VarAttribStruct', ...
          varAttribStruct);
```

**See Also**
cdfread, cdfinfo, cdfepoch
**Purpose**
Round toward positive infinity

**Syntax**
\[ B = \text{ceil}(A) \]

**Description**
\( B = \text{ceil}(A) \) 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**
\[
a = [-1.9, -0.2, 3.4, 5.6, 7, 2.4+3.6i]
\]

\[
a =
\begin{array}{cccc}
-1.9000 & -0.2000 & 3.4000 & 5.6000 \\
\end{array}
\]

\[
\begin{array}{cc}
7.0000 & 2.4000 + 3.6000i \\
\end{array}
\]

\[
\text{ceil}(a)
\]

\[
\begin{array}{cccc}
-1.0000 & 0 & 4.0000 & 6.0000 \\
\end{array}
\]

\[
\begin{array}{cc}
7.0000 & 3.0000 + 4.0000i \\
\end{array}
\]

**See Also**
fix, floor, round
Purpose

Construct cell array

Syntax

\[
\begin{align*}
c &= \text{cell}(n) \\
c &= \text{cell}(m, n) \\
c &= \text{cell}([m, n]) \\
c &= \text{cell}(m, n, p, ...) \\
c &= \text{cell}([m n p ...]) \\
c &= \text{cell}(\text{size}(A)) \\
c &= \text{cell}(\text{javaobj})
\end{align*}
\]

Description

\(c = \text{cell}(n)\) creates an \(n\)-by-\(n\) cell array of empty matrices. An error message appears if \(n\) is not a scalar.

\(c = \text{cell}(m, n)\) or \(c = \text{cell}([m, n])\) creates an \(m\)-by-\(n\) cell array of empty matrices. Arguments \(m\) and \(n\) must be scalars.

\(c = \text{cell}(m, n, p, ...)\) or \(c = \text{cell}([m n p ...])\) creates an \(m\)-by-\(n\)-by-\(p\)-... cell array of empty matrices. Arguments \(m, n, p, ...\) must be scalars.

\(c = \text{cell}(\text{size}(A))\) creates a cell array the same size as \(A\) containing all empty matrices.

\(c = \text{cell}(\text{javaobj})\) converts a Java array or Java object \(\text{javaobj}\) 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.

Remarks

This type of cell is not related to “cell mode”, a MATLAB feature used in debugging and publishing.

Examples

This example creates a cell array that is the same size as another array, \(A\).

\[
A = \text{ones}(2,2)
\]

\[
A =
\begin{bmatrix}
1 & 1 \\
1 & 1
\end{bmatrix}
\]
The next example converts an array of java.lang.String objects into a MATLAB cell array.

```matlab
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 to 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:

\[ \text{cell2mat}(c) \]

<table>
<thead>
<tr>
<th>10x25</th>
<th>10x25</th>
</tr>
</thead>
<tbody>
<tr>
<td>20x25</td>
<td>20x25</td>
</tr>
<tr>
<td>30x25</td>
<td>30x25</td>
</tr>
</tbody>
</table>

**Remarks**

The dimensionality (or number of dimensions) of \( m \) will match the highest dimensionality contained in the cell array.

\texttt{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, \( M \):

\[ C = \{[1] \quad [2 \ 3 \ 4]; \quad [5; \ 9] \quad [6 \ 7 \ 8; \ 10 \ 11 \ 12]\} \]
cell2mat

C =

[ 1] [1x3 double]
[2x1 double] [2x3 double]

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.

\[
c = \{\text{'birch'}, \text{'betula'}, 65; \text{'maple'}, \text{'acer'}, 50\}
\]

\[
c = \begin{bmatrix}
\text{'birch'} & \text{'betula'} & [65] \\
\text{'maple'} & \text{'acer'} & [50]
\end{bmatrix}
\]

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.

\[
\text{fields} = \{\text{'name'}, \text{'genus'}, \text{'height'}\}; \\
\text{s} = \text{cell2struct(c, fields, 2)};
\]

This yields the following 2-by-1 structure array.

\[
\text{s(1)} \quad \text{s(2)} \\
\text{ans =} \quad \text{ans =}
\begin{bmatrix}
\text{name: 'birch'} \\
\text{name: 'maple'}
\end{bmatrix}
\]
genus: 'betula'  genus: 'acer'
height: 65      height: 50

See Also
struct2cell, cell, iscell, struct, isstruct, fieldnames, dynamic field names
Purpose  
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).

Examples  
Use celldisp to display the contents of a 2-by-3 cell array:

```matlab
C = {{[1 2] 'Tony' 3+4i; [1 2;3 4] -5 'abc'}};
celldisp(C)
```

```matlab
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
Purpose

Apply function to each cell in cell array

Syntax

A = cellfun(fun, C)
A = cellfun(fun, C, D, ...)
[A, B, ...] = cellfun(fun, C, ...)
[A, ...] = cellfun(fun, C, ..., 'param1', value1, ...)
A = cellfun('fname', C)
A = cellfun('size', C, k)
A = cellfun('iscell', C, 'classname')

Description

A = cellfun(fun, C) applies the function specified by fun to the contents of each cell of cell array C, and returns the results in array A. The value A returned by cellfun is the same size as C, and the (I,J,...)th element of A is equal to fun(C{I,J,...}). The first input argument fun is a function handle to a function that takes one input argument and returns a scalar value. fun must return values of the same class each time it is called. The order in which cellfun computes elements of A is not specified and should not be relied upon.

If fun is bound to more than one built-in or M-file (that is, if it represents a set of overloaded functions), then the class of the values that cellfun actually provides as input arguments to fun determines which functions are executed.

A = cellfun(fun, C, D, ...) evaluates fun using the contents of the cells of cell arrays C, D, ... as input arguments. The (I,J,...)th element of A is equal to fun(C{I,J,...}, D{I,J,...}, ...). All input arguments must be of the same size and shape.

[A, B, ...] = cellfun(fun, C, ...) evaluates fun, which is a function handle to a function that returns multiple outputs, and returns arrays A, B, ..., each corresponding to one of the output arguments of fun. cellfun calls fun each time with as many outputs as there are in the call to cellfun. fun can return output arguments having different classes, but the class of each output must be the same each time fun is called.

[A, ...] = cellfun(fun, C, ..., 'param1', value1, ...) enables you to specify optional parameter name and value pairs.
Parameters recognized by `cellfun` are shown below. Enclose each parameter name with single quotes.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UniformOutput</td>
<td>Logical 1 (true) or 0 (false), indicating whether or not the outputs of <code>fun</code> can be returned without encapsulation in a cell array. See “UniformOutput Parameter” on page 2-561 below.</td>
</tr>
<tr>
<td>ErrorHandler</td>
<td>Function handle, specifying the function that <code>cellfun</code> is to call if the call to <code>fun</code> fails. See “ErrorHandler Parameter” on page 2-561 below.</td>
</tr>
</tbody>
</table>

**UniformOutput Parameter**

If you set the `UniformOutput` parameter to `true` (the default), `fun` must return scalar values that can be concatenated into an array. These values can also be a cell array.

If `UniformOutput` is `false`, `cellfun` returns a cell array (or multiple cell arrays), where the (I,J,...)th cell contains the value

```
fun(C{I,J,...}, ...)
```

**ErrorHandler Parameter**

The MATLAB software calls the function represented by the `ErrorHandler` parameter with two input arguments:

- A structure having three fields, named `identifier`, `message`, and `index`, respectively containing the identifier of the error that occurred, the text of the error message, and a linear index into the input array or arrays for which the error occurred
- The set of input arguments for which the call to the function failed

The error handling function must either rethrow the error that was caught, or it must return the output values from the call to `fun`.
handling functions that do not rethrow the error must have the same
number of outputs as fun. MATLAB places these output values in the
output variables used in the call to arrayfun.

Shown here is an example of a simple error handling function, errorfun:

```matlab
function [A, B] = errorfun(S, varargin)
    warning(S.identifier, S.message);
    A = NaN; B = NaN;
```

If 'UniformOutput' is set to logical 1 (true), the outputs of the error
handler must be scalars and of the same data type as the outputs of
function fun.

If you do not specify an error handler, cellfun rethrows the error.

**Backward Compatibility**

The following syntaxes are also accepted for backward compatibility:

A = cellfun('fname', C) applies the function fname to the elements
of cell array C and returns the results in the double array A. Each
element of A contains the value returned by fname for the corresponding
element in C. The output array A 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>prodoftime</td>
<td>Number of dimensions of the cell element</td>
</tr>
</tbody>
</table>

A = cellfun('size', C, k) returns the size along the kth dimension
of each element of C.
A = cellfun('isclass', C, 'classname') returns logical 1 (true) for each element of C that matches classname. This function syntax returns logical 0 (false) for objects that are a subclass of classname.

**Note** For the previous three syntaxes, if C contains objects, cellfun does not call any overloaded versions of MATLAB functions corresponding to the above strings.

**Examples**

Compute the mean of several data sets:

\[ C = \{1:10, [2; 4; 6], []\}; \]

\[ Cmeans = \text{cellfun}(@\text{mean}, C) \]
\[ Cmeans = \]
\[ 5.5000 \quad 4.0000 \quad \text{NaN} \]

Compute the size of these data sets:

\[ [\text{Cnrows}, \text{Cncols}] = \text{cellfun}(@\text{size}, C) \]
\[ \text{Cnrows} = \]
\[ 1 \quad 3 \quad 0 \]
\[ \text{Cncols} = \]
\[ 10 \quad 1 \quad 0 \]

Again compute the size, but with UniformOutput set to false:

\[ Csize = \text{cellfun}(@\text{size}, C, 'UniformOutput', \text{false}) \]
\[ Csize = \]
\[ [1x2 \text{ double}] \quad [1x2 \text{ double}] \quad [1x2 \text{ double}] \]

\[ Csize{:} \]
\[ \text{ans} = \]
\[ 1 \quad 10 \]
\[ \text{ans} = \]
\[ 3 \quad 1 \]
\[ \text{ans} = \]
Find the positive values in several data sets.

```matlab
C = {randn(10,1), randn(20,1), randn(30,1)};

Cpositives = cellfun(@(x) x(x>0), C, 'UniformOutput',false)
Cpositives =
    [6x1 double]    [11x1 double]    [15x1 double]
```

```matlab
Cpositives{:}
an =
   0.1253
   0.2877
   1.1909
   etc.
an =
   0.7258
   2.1832
   0.1139
   etc.
an =
   0.6900
   0.8156
   0.7119
   etc.
```

Compute the covariance between several pairs of data sets:

```matlab
C = {randn(10,1), randn(20,1), randn(30,1)};
D = {randn(10,1), randn(20,1), randn(30,1)};

CDcovs = cellfun(@cov, C, D, 'UniformOutput',false)
CDcovs =
    [2x2 double]    [2x2 double]    [2x2 double]
```

```matlab
CDcovs{:}
an =
```
0.7353  -0.2148
-0.2148   0.6080
ans =
0.5743  -0.2912
-0.2912   0.8505
ans =
0.7130   0.1750
0.1750   0.6910

See Also  arrayfun, spfun, function_handle, cell2mat
cellplot

**Purpose**
Graphically display structure of cell array

**Syntax**
cellplot(c)
cellplot(c, 'legend')
handles = cellplot(c)

**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') places a colorbar next to the plot labelled to identify the data types in c.

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:

```matlab
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
**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 \( c \). Any trailing spaces in the rows of \( S \) are removed.
Use the `char` function to convert back to a string matrix.

**Examples**  
Given the string matrix

\[
S = \begin{bmatrix}
'abc';
'defg';
'hi'
\end{bmatrix}
\]

\[
S =
\begin{align*}
\text{abc} \\
\text{defg} \\
\text{hi}
\end{align*}
\]

\[
\text{whos } S
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>3x4</td>
<td>24</td>
<td>char array</td>
</tr>
</tbody>
</table>

The following command returns a 3-by-1 cell array.

\[
c = \text{cellstr}(S)
\]

\[
c =
\begin{bmatrix}
'abc'
'defg'
'hi'
\end{bmatrix}
\]

\[
\text{whos } c
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>3x1</td>
<td>294</td>
<td>cell array</td>
</tr>
</tbody>
</table>

**See Also**  
iscellstr, strings, char, isstrprop
**Purpose**
Conjugate gradients squared method

**Syntax**

```matlab
x = 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)
[x,flag] = cgs(A,b,...)
[x,flag,relres] = cgs(A,b,...)
[x,flag,relres,iter] = cgs(A,b,...)
[x,flag,relres,iter,resvec] = cgs(A,b,...)
```

**Description**

`x = cgs(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x)` returns `A*x`. See “Function Handles” in the MATLAB Programming documentation for more information.

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

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

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

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

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

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

[\( x,\text{flag} \) = cgs(\( A,b,\ldots \))] 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 ( \text{tol} ) within ( \text{maxit} ) iterations.</td>
</tr>
<tr>
<td>1</td>
<td>cgs iterated ( \text{maxit} ) times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>Preconditioner ( M ) was ill-conditioned.</td>
</tr>
<tr>
<td>3</td>
<td>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.

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

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

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

**Examples**

**Example**

\[
A = \text{gallery('wilk',21)}; \\
b = \text{sum}(A,2); \\
\]
tol = 1e-12; maxit = 15;
M1 = diag([10:-1:1 1 1:10]);
x = cgs(A,b,tol,maxit,M1);

displays the message

cgs converged at iteration 13 to a solution with relative residual
1.3e-016

Example 2

This example replaces the matrix A in Example 1 with a handle to a matrix-vector product function afun, and the preconditioner M1 with a handle to a backsolve function mfun. The example is contained in an M-file run_cgs that

- Calls cgs with the function handle @afun as its first argument.
- Contains afun as a nested function, so that all variables in run_cgs are available to afun and myfun.

The following shows the code for run_cgs:

```matlab
function x1 = run_cgs
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
tol = 1e-12; maxit = 15;
x1 = cgs(@afun,b,tol,maxit,@mfun);

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

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

2-571
When you enter

```matlab
x1 = run_cgs
```

MATLAB software returns

```
cgs converged at iteration 13 to a solution with relative residual 1.3e-016
```

**Example 3**

```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 `U1` has a zero on its diagonal, and `cgs` fails in the first iteration when it tries to solve a system such as `U1*y = 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 `cgs` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0) with

```matlab
semilogy(0:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
```
cgs

See Also
bicg, bicgstab, gmres, lsqr, luinc, minres, pcg, qmr, symmlq
function_handle (@), mldivide (\)

References

Purpose

Convert to character array (string)

Syntax

\[ S = \text{char}(X) \]
\[ S = \text{char}(C) \]
\[ S = \text{char}(t1, t2, t3, \ldots) \]

Description

\( S = \text{char}(X) \) converts the array \( X \) that contains nonnegative integers representing character codes into a MATLAB character array. The actual characters displayed depend on the character encoding scheme 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.

\( S = \text{char}(t1, t2, t3, \ldots) \) forms the character array \( S \) containing the text strings \( T1, T2, T3, \ldots \) as rows, automatically padding each string with blanks to form a valid matrix. Each text parameter, \( Ti \), can itself be a character array. This allows the creation of arbitrarily large character arrays. Empty strings are significant.

Examples

To print a 3-by-32 display of the printable ASCII characters,

\[
\text{ascii} = \text{char(reshape(32:127, 32, 3)')} \\
\text{ascii} = \\
!"#$%&'()*+,-./0123456789:;<=>? \\
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_ \\
`abcdefghijklmnopqrstuvwxyz{|}~
\]

See Also

ischar, isletter, isspace, isstrprop, cellstr, iscellstr, get, set, strings, strvcat, text
**Purpose**
Check files into source control system (UNIX platforms)

**GUI Alternatives**
As an alternative to the checkin function, use **File > Source Control > Check In** in the Editor, the Simulink® product, or the Stateflow® product, or in the context menu of the Current Directory browser. For more information, see “Checking Files Into the Source Control System on UNIX Platforms”.

**Syntax**

```matlab
checkin('filename','comments','comment_text')
checkin({'filename1','filename2'},'comments','comment_text')
checkin('filename','comments','comment_text','option','value')
```

**Description**

The `checkin` function checks in the file named `filename` to the source control system. Use the full path for `filename` and include the file extension. You must save the file before checking it in, but the file can be open or closed. The `comment_text` argument is a MATLAB string containing checkin comments for the source control system. You must supply `comments` and `comment_text`.

The `checkin` function checks in the files `filename1` through `filenamen` to the source control system. Use the full paths for the files and include file extensions. Comments apply to all files checked in.

The `checkin` function provides additional checkin options. For multiple file names, use an array of strings instead of `filename`, that is, `{'filename1','filename2',...}`. Options apply to all file names. The `option` and `value` arguments are shown in the following table.

<table>
<thead>
<tr>
<th>option Argument</th>
<th>value Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>'force'</td>
<td>'on'</td>
<td>filename is checked in even if the file has not changed since it was checked out.</td>
</tr>
</tbody>
</table>
**checkin**

<table>
<thead>
<tr>
<th>option Argument</th>
<th>value Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>'force'</td>
<td>'off'</td>
<td>filename is not checked in if there were no changes since checkout.</td>
</tr>
<tr>
<td></td>
<td>(default)</td>
<td></td>
</tr>
<tr>
<td>'lock'</td>
<td>'on'</td>
<td>filename is checked in with comments, and is automatically checked out.</td>
</tr>
<tr>
<td></td>
<td>'off'</td>
<td>filename is checked in with comments but does not remain checked out.</td>
</tr>
<tr>
<td></td>
<td>(default)</td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

**Check In a File**

Typing

```matlab
checkin('/myserver/mymfiles/clock.m','comments',... 'Adjustment for leapyear')
```

checks the file `/myserver/mymfiles/clock.m` into the source control system, with the comment *Adjustment for leapyear*.

**Check In Multiple Files**

Typing

```matlab
checkin({'/myserver/mymfiles/clock.m', ... '/myserver/mymfiles/calendar.m'},'comments',... 'Adjustment for leapyear')
```

checks the two files into the source control system, using the same comment for each.

**Check In a File and Keep It Checked Out**

Typing

```matlab
checkin('/myserver/mymfiles/clock.m','comments',... 'Adjustment for leapyear','lock','on')
```
checks the file /myserver/mymfiles/clock.m into the source control system and keeps the file checked out.

See Also

checkout, cmopts, undocheckout

For Microsoft Windows platforms, use verctrl.
**Purpose**
Check files out of source control system (UNIX platforms)

**GUI Alternatives**
As an alternative to the checkout function, select **Source Control > Check Out** from the **File** menu in the MATLAB Editor, the Simulink product, or the Stateflow product, or in the context menu of the Current Directory browser. For details, see “Checking Files Out of the Source Control System on UNIX”.

**Syntax**

```
checkout('filename')
checkout({'filename1','filename2', ...})
checkout('filename','option','value',...)
```

**Description**

`checkout('filename')` checks out the file named `filename` from the source control system. Use the full path for `filename` and include the file extension. The file can be open or closed when you use `checkout`.

`checkout({'filename1','filename2', ...})` checks out the files named `filename1` through `filenamen` from the source control system. Use the full paths for the files and include the file extensions.

`checkout('filename','option','value',...)` provides additional checkout options. For multiple file names, use an array of strings instead of `filename`, that is, `{'filename1','filename2', ...}`. Options apply to all file names. The `option` and `value` arguments are shown in the following table.

<table>
<thead>
<tr>
<th>option Argument</th>
<th>value Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>'force'</td>
<td>'on'</td>
<td>The checkout is forced, even if you already have the file checked out. This is effectively an undocheckout followed by a checkout.</td>
</tr>
</tbody>
</table>

2-578
<table>
<thead>
<tr>
<th>option</th>
<th>Argument</th>
<th>value</th>
<th>Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>'force'</td>
<td></td>
<td>'off'</td>
<td>(default)</td>
<td>Prevents you from checking out the file if you already have it checked out.</td>
</tr>
<tr>
<td>'lock'</td>
<td></td>
<td>'on'</td>
<td>(default)</td>
<td>The checkout gets the file, allows you to write to it, and locks the file so that access to the file for others is read only.</td>
</tr>
<tr>
<td>'lock'</td>
<td></td>
<td>'off'</td>
<td></td>
<td>The checkout gets a read-only version of the file, allowing another user to check out the file for updating. You do not have to check the file in after checking it out with this option.</td>
</tr>
<tr>
<td>'revision'</td>
<td></td>
<td>'version_num'</td>
<td></td>
<td>Checks out the specified revision of the file.</td>
</tr>
</tbody>
</table>

If you end the MATLAB session, the file remains checked out. You can check in the file from within the MATLAB desktop during a later session, or directly from your source control system.

**Examples**

**Check Out a File**

Typing

```matlab
checkout('/myserver/mymfiles/clock.m')
```

checks out the file `/myserver/mymfiles/clock.m` from the source control system.
Check Out Multiple Files

Typing

    checkout({'/myserver/mymfiles/clock.m', ...
    '/myserver/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

    checkout('/myserver/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

    checkout('/matlab/mymfiles/clock.m','revision','1.1')

checks out revision 1.1 of clock.m.

See Also

checkin, cmopts, undocheckout, customverctrl

For Microsoft Windows platforms, use verctrl.
**Purpose**  
Cholesky factorization

**Syntax**

- \( R = \text{chol}(A) \)
- \( L = \text{chol}(A, 'lower') \)
- \([R,p] = \text{chol}(A)\)
- \([L,p] = \text{chol}(A, 'lower')\)
- \([R,p,S] = \text{chol}(A)\)
- \([R,p,s] = \text{chol}(A, 'vector')\)
- \([L,p,s] = \text{chol}(A, 'lower', 'vector')\)

**Description**

\( R = \text{chol}(A) \) produces an upper triangular matrix \( R \) from the diagonal and upper triangle of matrix \( A \), satisfying the equation \( R' \cdot R = A \). The lower triangle is assumed to be the (complex conjugate) transpose of the upper triangle. Matrix \( A \) must be positive definite; otherwise, MATLAB software displays an error message.

\( L = \text{chol}(A, 'lower') \) produces a lower triangular matrix \( L \) from the diagonal and lower triangle of matrix \( A \), satisfying the equation \( L \cdot L' = A \). When \( A \) is sparse, this syntax of \texttt{chol} is typically faster. Matrix \( A \) must be positive definite; otherwise MATLAB displays an error message.

\([R,p] = \text{chol}(A)\) for positive definite \( A \), produces an upper triangular matrix \( R \) from the diagonal and upper triangle of matrix \( A \), satisfying the equation \( R' \cdot R = A \) and \( p \) is zero. If \( A \) is not positive definite, then \( p \) is a positive integer and MATLAB does not generate an error. When \( A \) is full, \( R \) is an upper triangular matrix of order \( q = p - 1 \) such that \( R' \cdot R = A(1:q,1:q) \). When \( A \) is sparse, \( R \) is an upper triangular matrix of size \( q \)-by-\( n \) so that the \( L \)-shaped region of the first \( q \) rows and first \( q \) columns of \( R' \cdot R \) agree with those of \( A \).

\([L,p] = \text{chol}(A, 'lower')\) for positive definite \( A \), produces a lower triangular matrix \( L \) from the diagonal and lower triangle of matrix \( A \), satisfying the equation \( L \cdot L' = A \) and \( p \) is zero. If \( A \) is not positive definite, then \( p \) is a positive integer and MATLAB does not generate an error. When \( A \) is full, \( L \) is a lower triangular matrix of order \( q = p - 1 \) such that \( L \cdot L' = A(1:q,1:q) \). When \( A \) is sparse, \( L \) is a lower triangular matrix of size \( q \)-by-\( n \) so that the \( L \)-shaped region of the first \( q \) rows and first \( q \) columns of \( L \cdot L' \) agree with those of \( A \).
[R,p,S] = chol(A), when A is sparse, returns a permutation matrix S. Note that the preordering S may differ from that obtained from amd since chol will slightly change the ordering for increased performance. When p=0, R is an upper triangular matrix such that R'*R=S'*A*S. When p is not zero, R is an upper triangular matrix of size q-by-n so that the L-shaped region of the first q rows and first q columns of R'*R agree with those of S'*A*S. The factor of S'*A*S tends to be sparser than the factor of A.

[R,p,s] = chol(A,'vector') returns the permutation information as a vector s such that A(s,s)=R'*R, when p=0. You can use the 'matrix' option in place of 'vector' to obtain the default behavior.

[L,p,s] = chol(A,'lower','vector') uses only the diagonal and the lower triangle of A and returns a lower triangular matrix L and a permutation vector s such that A(s,s)=L*L', when p=0. As above, you can use the 'matrix' option in place of 'vector' to obtain a permutation matrix.

For sparse A, CHOLMOD is used to compute the Cholesky factor.

**Note** Using chol is preferable to using eig for determining positive definiteness.

**Examples**

**Example 1**

The gallery function provides several symmetric, positive, definite matrices.

A=gallery('moler',5)

A =

```
1  -1  -1  -1  -1
-1  2   0   0   0
-1  0   3   1   1
-1  0   1   4   2
```
-1  0  1  2  5

C=chol(A)

ans =

    1   -1   -1   -1   -1
    0    1   -1   -1   -1
    0    0    1   -1   -1
    0    0    0    1   -1
    0    0    0    0    1

isequal(C'*C,A)

ans =

    1

For sparse input matrices, chol returns the Cholesky factor.

N = 100;
A = gallery('poisson', N);

N represents the number of grid points in one direction of a square
N-by-N grid. Therefore, A is N^2 by N^2.

L = chol(A, 'lower');
D = norm(A - L*L', 'fro');

The value of D will vary somewhat among different versions of MATLAB
but will be on order of 10^{-14}.

**Example 2**

The binomial coefficients arranged in a symmetric array create a
positive definite matrix.

n = 5;
X = pascal(n)
This matrix is interesting because its Cholesky factor consists of the same coefficients, arranged in an upper triangular matrix.

\[ R = \text{chol}(X) \]

\[ R = \]

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 \\
0 & 1 & 2 & 3 & 4 \\
0 & 0 & 1 & 3 & 6 \\
0 & 0 & 0 & 1 & 4 \\
0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

Destroy the positive definiteness (and actually make the matrix singular) by subtracting 1 from the last element.

\[ X(n,n) = X(n,n) - 1 \]

\[ X = \]

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 \\
1 & 3 & 6 & 10 & 15 \\
1 & 4 & 10 & 20 & 35 \\
1 & 5 & 15 & 35 & 69 \\
\end{array}
\]

Now an attempt to find the Cholesky factorization of \( X \) fails.

\[ \text{chol}(X) \]

??? Error using \( \text{===> chol} \)

Matrix must be positive definite.

**Algorithm**

For full matrices \( X \), \text{chol} uses the LAPACK routines listed in the following table.
For sparse matrices, MATLAB software uses CHOLMOD to compute the Cholesky factor.

**References**


**See Also**

cholinc, cholupdate
### Purpose
Sparse incomplete Cholesky and Cholesky-Infinity factorizations

### Syntax
- \( R = \text{cholinc}(X, \text{droptol}) \)
- \( R = \text{cholinc}(X, \text{options}) \)
- \( R = \text{cholinc}(X, '0') \)
- \([R, p] = \text{cholinc}(X, '0')\)
- \( R = \text{cholinc}(X, '\text{inf}') \)

### 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.

\( R = \text{cholinc}(X, \text{droptol}) \) performs the incomplete Cholesky factorization of \( X \), with drop tolerance \( \text{droptol} \).

\( R = \text{cholinc}(X, \text{options}) \) allows additional options to the incomplete Cholesky factorization. \text{options} is a structure with up to three fields:

- \( \text{droptol} \): Drop tolerance of the incomplete factorization
- \( \text{michol} \): Modified incomplete Cholesky
- \( \text{rdiag} \): Replace zeros on the diagonal of \( R \)

Only the fields of interest need to be set.

\( \text{droptol} \) 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 \( \text{droptol} \times \text{norm}(X(:,j)) \) (see \text{luinc}, the nonzero entries \( R(i,j) \) are bounded below by the local drop tolerance \( \text{droptol} \times \text{norm}(X(:,j))/R(i,i) \).
Setting `droptol = 0` produces the complete Cholesky factorization, which is the default.

`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.

\[ R = \text{cholinc}(X, '0') \]
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 `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.

\[ R = \text{cholinc}(X, 'inf') \]
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 `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 `Inf`. 
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.

\[
S = \text{delsq(numgrid('C',15))};
\]

S is the two-dimensional, five-point discrete negative Lapacian on the grid generated by 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.

\[
C = \text{chol}(S);
\]
\[
R0 = \text{cholinc}(S,'0');
\]
\[
S2 = S; S2(101,101) = 0;
\]
\[
[R,p] = \text{cholinc}(S2,'0');
\]

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.

\[
D1 = (R0'*R0).*spones(S)-S;
\]
\[
D2 = (R'*R).*spones(S2)-S2;
\]
D1 has elements of the order of \( \text{eps} \), showing that \( R0' \times R0 \) 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 \( \text{cholinc}(S,0) \), the incomplete Cholesky factor with a drop tolerance of 0, is the same as the Cholesky factor of \( S \). 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 $\frac{\text{norm}(R\cdot R - S, 1)}{\text{norm}(S, 1)}$ in the next figure.
Example 3

The Hilbert matrices have (i,j) entries $1/(i+j-1)$ and are theoretically positive definite:

\[ H3 = \text{hilb}(3) \]

\[
H3 = 
\begin{bmatrix}
1.0000 & 0.5000 & 0.3333 \\
0.5000 & 0.3333 & 0.2500 \\
0.3333 & 0.2500 & 0.2000 \\
\end{bmatrix}
\]

\[ R3 = \text{chol}(H3) \]

\[
R3 = 
\begin{bmatrix}
1.0000 & 0.5000 & 0.3333 \\
0 & 0.2887 & 0.2887 \\
0 & 0 & 0.0745 \\
\end{bmatrix}
\]

In practice, the Cholesky factorization breaks down for larger matrices:

\[ H20 = \text{sparse(hilb(20))}; \]
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 \texttt{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}
an =
Inf 0 0 0 0 0 0 0
 0 Inf 0 0 0 0 0 0
 0 0 Inf 0 0 0 0 0
 0 0 0 Inf 0 0 0 0
 0 0 0 0 Inf 0 0 0
 0 0 0 0 0 Inf 0 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**

\begin{verbatim}
R = cholinc(X,droptol) is obtained from [L,U] = luinc(X,options), where options.droptol = droptol and options.thresh = 0. The rows of the uppertriangular U are scaled by the square root of the diagonal in that row, and this scaled factor becomes R.

R = cholinc(X,options) is produced in a similar manner, except the rdiag option translates into the udiag option and the milu option takes the value of the michol option.
\end{verbatim}
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.

R = cholinc(X,'inf') is based on the algorithm in Zhang [2].

**See Also**

chol, ilu, luinc, pcg

**References**


**Purpose**
Rank 1 update to Cholesky factorization

**Syntax**

```matlab
R1 = cholupdate(R,x)
R1 = cholupdate(R,x,'+')
R1 = cholupdate(R,x,'-')
[R1,p] = cholupdate(R,x,'-')
```

**Description**

`R1 = cholupdate(R,x)` where `R = 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 = cholupdate(R,x,'+')` is the same as `R1 = cholupdate(R,x)`.

`R1 = 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] = 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**

```matlab
A = pascal(4)
A =

    1     1     1     1
    1     2     3     4
    1     3     6    10
    1     4    10    20

R = chol(A)
R =
```

2-594
This is called a rank one update to A since \( \text{rank}(xx') \) is 1:
\[
A + xx'
\]
\[
\text{ans} =
\]
\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & 1 & 2 & 3 \\
0 & 0 & 1 & 3 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
\[
x = [0 0 0 1]';
\]
Instead of computing the Cholesky factor with \( R1 = \text{chol}(A + xx') \), we can use \text{cholupdate}:
\[
R1 = \text{cholupdate}(R,x)
\]
\[
R1 =
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 \\
1 & 3 & 6 & 10 \\
1 & 4 & 10 & 21
\end{bmatrix}
\]
Next destroy the positive definiteness (and actually make the matrix singular) by subtracting 1 from the last element of A. The downdated matrix is:
\[
A - xx'
\]
\[
\text{ans} =
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4
\end{bmatrix}
\]
Compare chol with cholupdate:

\[ R1 = \text{chol}(A-x\cdot x') \]

??? Error using ==> chol
Matrix must be positive definite.
\[ R1 = \text{cholupdate}(R,x,'-') \]

??? 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:

\[
x = [0 \ 0 \ 0 \ 1/\sqrt{2}]'
\]
\[
R1 = \text{cholupdate}(R,x,'-')
\]

\[
R1 =
\begin{bmatrix}
1.0000 & 1.0000 & 1.0000 & 1.0000 \\
0 & 1.0000 & 2.0000 & 3.0000 \\
0 & 0 & 1.0000 & 3.0000 \\
0 & 0 & 0 & 0.7071
\end{bmatrix}
\]

**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**  
\[ B = \text{circshift}(A,\text{shiftsize}) \]

**Description**  
\( B = \text{circshift}(A,\text{shiftsize}) \) circularly shifts the values in the array, \( A \), by \( \text{shiftsize} \) elements. \( \text{shiftsize} \) 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 \( \text{shiftsize} \) 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.

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

\[
B = \text{circshift}(A,1)
\]

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

Circularly shift first dimension values down by 1 and second dimension values to the left by 1.

\[
B = \text{circshift}(A,[1 \ -1])
\]

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

**See Also**  
fftshift, shiftdim, permute, reshape
**Purpose**
Circumcenters of specified simplices

**Syntax**
CC = circumcenters(TR, SI)
[CC RCC] = circumcenters(TR, SI)

**Description**
CC = circumcenters(TR, SI) returns the coordinates of the
circumcenter of each specified simplex SI. CC is an m-by-n matrix, where
m is of length length(SI), the number of specified simplices, and n is
the dimension of the space where the triangulation resides.

[CC RCC] = circumcenters(TR, SI) returns the circumcenters and
the corresponding radii of the circumscribed circles or spheres.

**Inputs**
TR
Triangulation object.
SI
Column vector of simplex indices that index into the
triangulation matrix TR.Triangulation. If SI is not
specified the circumcenter information for the entire
triangulation is returned, where the circumcenter
associated with simplex i is the i’th row of CC.

**Outputs**
CC
m-by-n matrix. m is the number of specified simplices
and n is the dimension of the space where the
triangulation resides. Each row CC(i,:) represents
the coordinates of the circumcenter of simplex SI(i).
RCC
Vector of length length(SI), the number of specified
simplices containing radii of the circumscribed circles
or spheres.

**Definitions**
A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

**Examples**
Example 1
Load a 2-D triangulation.
load trimesh2d
trep = TriRep(tri, x, y)

Compute the circumcenters.

cc = circumcenters(trep);
triplot(trep);
axis([-50 350 -50 350]);
axis equal;
hold on;
plot(cc(:,1),cc(:,2),'*r');
hold off;

The circumcenters represent points on the medial axis of the polygon.
Example 2

Query a 3-D triangulation created with DelaunayTri. Compute the circumcenters of the first five tetrahedra.

```matlab
X = rand(10,3);
dt = DelaunayTri(X);
cc = circumcenters(dt, [1:5]')
```

See Also

incenters
DelaunayTri
Purpose

Clear current axes

GUI Alternatives

Remove axes and clear objects from them in plot edit mode. For details, see “Working in Plot Edit Mode” in the MATLAB Graphics documentation.

Syntax

cla
cla reset
cla(ax)
cla(ax,'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.

cla(ax) or cla(ax,'reset') clears the single axes with handle ax.

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” on page 1-103 for related functions
clabel

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(...)c
clabel(...,'PropertyName',propertyvalue,...)
clabel(...'LabelSpacing',points)

Description
The clabel function adds height labels to a 2-D 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).

```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,h,'LabelSpacing',72)
```
Label a contour plot with 15 point red text.

```
[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.

```matlab
[x,y,z] = peaks;
C = contour(x,y,z);
clabel(C)
```
See Also

contour, contourc, contourf

“Annotating Plots” on page 1-94 for related functions

“Drawing Text in a Box” for an example that illustrates the use of contour labels
Purpose
Create object or return class of object

Syntax
str = class(object)
obj = class(s,'class_name')
obj = class(s,'class_name',parent1,parent2,...)
obj = class(struct([]),'class_name',parent1,parent2,...)
obj_struct = class(struct_array,'class_name',parent_array)

Description
str = class(object) returns a string specifying the class of object.
The following table lists the class names that can be returned. All except the last one are MATLAB classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>Logical array of true and false values</td>
</tr>
<tr>
<td>char</td>
<td>Character array</td>
</tr>
<tr>
<td>int8</td>
<td>8-bit signed integer array</td>
</tr>
<tr>
<td>uint8</td>
<td>8-bit unsigned integer array</td>
</tr>
<tr>
<td>int16</td>
<td>16-bit signed integer array</td>
</tr>
<tr>
<td>uint16</td>
<td>16-bit unsigned integer array</td>
</tr>
<tr>
<td>int32</td>
<td>32-bit signed integer array</td>
</tr>
<tr>
<td>uint32</td>
<td>32-bit unsigned integer array</td>
</tr>
<tr>
<td>int64</td>
<td>64-bit signed integer array</td>
</tr>
<tr>
<td>uint64</td>
<td>64-bit unsigned integer array</td>
</tr>
<tr>
<td>single</td>
<td>Single-precision floating-point number array</td>
</tr>
<tr>
<td>double</td>
<td>Double-precision floating-point number array</td>
</tr>
<tr>
<td>cell</td>
<td>Cell array</td>
</tr>
<tr>
<td>struct</td>
<td>Structure array</td>
</tr>
<tr>
<td>function_handle</td>
<td>Array of values for calling functions indirectly</td>
</tr>
</tbody>
</table>
'class_name'       User–defined MATLAB class

'Java_class_name'  Java class

Using the class function within a class constructor (prior to MATLAB Version 7.6)

The following usage of the class function is restricted to pre MATLAB Version 7.6 class constructors (classes defined without a classdef statement). It can be used only within a function named class_name.m, which is in a directory named @class_name (where class_name is the same as the string passed to class and is the name of the class being constructed).

See “Class Constructor Methods” for information on implementing class constructor methods in MATLAB Version 7.6 and after.

obj = class(s,'class_name') creates an array of class class_name objects using the struct s as a pattern to determine the size of obj.

obj = class(s,'class_name',parent1,parent2,...) creates an array of class class_name objects that inherit the methods and fields of the parent objects parent1, parent2, and so on. The struct s is used as a pattern to determine the size of obj. The size of the parent objects must match the size of s or be a scalar (1-by-1), in which case, MATLAB performs scalar expansion.

obj = class(struct([]),'class_name',parent1,parent2,...) creates an array of class class_name objects 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. All parents must have the same, nonzero size, which determines the size of the returned object obj.

Arrays of objects

obj_struct = class(struct_array,'class_name',parent_array) struct_array is an array of structs and parent_array is an array
of parent objects. Every element of the `parent_array` is mapped to a corresponding element in the `struct_array` to produce the output array of objects, `obj_struct`. All arrays must be of the same size or, if either the `struct_array` or the `parent_array` is of size 1-by-1, then MATLAB performs scalar expansion to match the array sizes.

Note that you can create an object array of size 0-by-0 by setting the size of the `struct_array` and `parent_array` to 0-by-0.

**Examples**

To return in `nameStr` the class of Java object `j`,

```matlab
nameStr = class(j)
```

Obtain the full name of a package-based Java class,

```matlab
import java.lang.*;
obj = String('mystring');
class(obj)
```

**See Also**

`inferiorto`, `isa`, `struct`, `superiorto`

*Object-Oriented Programming*
**Purpose**  
Class definition key words

**Syntax**  
```matlab
classdef  
properties  
methods  
events  
```

**Description**  
classdef begins the class definition, which is terminated by an end key word. Only blank lines and comments can precede classdef. You must place a class definition in a file having the same name as the class, with a filename extension of \texttt{.m}. Class definition M-files can be in directories on the MATLAB path or in \texttt{@} directories whose parent directory is on the MATLAB path. See “Class Directories” for more information.

See “The Classdef Block” and “Defining Classes — Syntax” for more information on classes.

properties begins a property definition block, which is terminated by an end key word. Class definitions can contain multiple property definition blocks, each specifying different attribute settings that apply to the properties in that particular block.

See “Defining Properties” for more information.

methods begins a methods definition block, which is terminated by an end key word. This block contains functions that implement class methods. Class definitions can contain multiple method blocks, each specifying different attribute settings that apply to the methods in that particular block. It is possible for method functions to be defined in separate files.

See “Class Methods” for more information.

events begins an events definition block, which is terminated by an end key word. This block contains event names defined by the class. Class definitions can contain multiple event blocks, each specifying different attribute settings that apply to the events in that particular block.

See “Defining Events and Listeners — Syntax and Techniques” for more information.
properties, methods, and events are also the names of MATLAB functions used to query the respective class members for a given object or class name.

**Table of Attributes**

Display the attributes of all class component in a popup window, click this link: Attribute Tables

**Examples**

Here is the basic structure of a class definition.

```matlab
classdef class_name
    properties
        PropertyName
    end
    methods
        function obj = methodName(obj,arg2, ...)
            ...
        end
    end
    events
        EventName
    end
end
```

**See Also**

*Object-Oriented Programming*
**Purpose**
Clear Command Window

**GUI Alternatives**
As an alternative to the `clc` function, select **Edit > Clear Command Window** in the MATLAB desktop.

**Syntax**
`clc`

**Description**
`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.

**Examples**
Use `clc` in an M-file to always display output in the same starting position on the screen.

**See Also**
clear, clf, close, home
clear

**Purpose**
Remove items from workspace, freeing up system memory

**Graphical Interface**
As an alternative to the `clear` function, use **Edit > Clear Workspace** in the MATLAB desktop.

**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 <code>clear all</code> 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 Sun Microsystems Java packages import list.</td>
</tr>
<tr>
<td>classes</td>
<td>The same as <code>clear all</code>, 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. You should call <code>clear classes</code> whenever you change a class definition, including when the number or names of properties, methods, or events or any of their attributes change.</td>
</tr>
</tbody>
</table>
### clear

<table>
<thead>
<tr>
<th><strong>Keyword</strong></th>
<th><strong>Items Cleared</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>functions</td>
<td>Clears all the currently compiled M-functions and MEX-functions from memory. Using <code>clear function</code> 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>
<tr>
<td>java</td>
<td>The same as <code>clear all</code>, 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 <code>clear java</code> command after modifying any files on the Java dynamic class path.</td>
</tr>
<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 and global variables are cleared in both the function workspace and in your base workspace.
Limitations

On UNIX² systems, clear does not affect the amount of memory allocated to the MATLAB process.

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>
<tr>
<td>gbl2</td>
<td>1x1</td>
<td>8</td>
<td>double array (global)</td>
</tr>
<tr>
<td>xint</td>
<td>1x1</td>
<td>1</td>
<td>int8 array</td>
</tr>
</tbody>
</table>

You can clear a single variable, xint, by typing

    clear xint

To clear all global variables, type

    clear global

whos

    Name     Size    Bytes  Class          
    c        3x4     1200    cell array     
    frame    1x1     1200    java.awt.Frame 

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

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

    unlocked 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

See Also

clc, clearvars, close, import, inmem, load, mlock, munlock, pack, persistent, save, who, whos, workspace

“MATLAB Workspace” in the Desktop Tools and Development Environment documentation
clearvars

**Purpose**
Clear variables from memory

**Graphical Interface**
As an alternative to the clearvars function, in the Workspace browser, select variables to clear and then press **Delete**.

**Syntax**
clearvars v1 v2 ...
clearvars -global
clearvars -global v1 v2 ...
clearvars -regexp p1 p2 ...
clearvars -except v1 v2 ...
clearvars -except -regexp p1 p2 ...
clearvars v1 v2 ... -except -regexp p1 p2 ...
clearvars -regexp p1 p2 ... -except v1 v2 ...

**Description**
clearvars v1 v2 ... clears variables v1, v2, and so on from the currently active workspace. Each input must be an unquoted string specifying the variable to be cleared. This string may include the wildcard character (*) to clear all variables that match a pattern.
For example, clearvars X* clears all the variables in the current workspace that start with the letter X.

If any of the variables v1, v2, and so on, are global, clearvars removes these variables from the current workspace only, leaving them accessible to any functions that declare them as global.

clearvars -global removes all global variables, including those made global within functions.

clearvars -global v1 v2 ... completely removes the specified global variables.

The -global flag may be used with any of the following syntaxes. When used in this way, it must immediately follow the function name.

clearvars -regexp p1 p2 ... clears all variables that match regular expression patterns p1, p2, and so on.

clearvars -except v1 v2 ... clears all variables except for those specified following the -except flag. Use the wildcard character *
in a variable name to exclude variables that match a pattern from being cleared. `clearvars -except X*` clears all the variables in the current workspace, except for those that start with X, for instance. Use `clearvars -except` to keep the variables you want and remove all others.

`clearvars -except -regexp p1 p2 ...` clears all variables except those that regular expression patterns `p1`, `p2`. If used in this way, the `-regexp` flag must immediately follow the `-except` flag.

`clearvars v1 v2 ... -except -regexp p1 p2 ...` can be used to specify variables to clear that do not match specified regular expression patterns.

`clearvars -regexp p1 p2 ... -except v1 v2 ...` clears variables that match `p1`, `p2`, ..., except for variables `v1`, `v2`, ...

**Examples**

Clear variables starting with a, except for the variable `ab`:

```
clearvars a* -except ab
```

Clear all global variables except those starting with `x`:

```
clearvars -global -except x*
```

Clear variables that start with `b` and are followed by 3 digits, for the variable `b106`:

```
clearvars -regexp ^b\d{3}$ -except b106
```

Clear variables that start with `a`, except those ending with `a`:

```
clearvars a* -except -regexp a$
```

**See Also**

`clear`, `exist`, `global`, `persistent`, `save`, `who`, `whos` 

“MATLAB Workspace” in the Desktop Tools and Development Environment documentation
Purpose

Remove serial port object from MATLAB workspace

Syntax

`clear obj`

Description

clear obj removes obj from the MATLAB workspace, where obj is a serial port object or an array of serial port objects.

Remarks

If obj is connected to the device and it is cleared from the workspace, then obj remains connected to the device. You can restore obj to the workspace with the `instrfind` function. A serial port object connected to the device has a `Status` property value of `open`.

To disconnect obj from the device, use the `fclose` function. To remove obj from memory, use the `delete` function. You should remove invalid serial port objects from the workspace with `clear`.

Example

This example creates the serial port object s on a Windows platform, copies s to a new variable `scopy`, and clears s from the MATLAB workspace. s is then restored to the workspace with `instrfind` and is shown to be identical to `scopy`.

```matlab
s = serial('COM1');
scopy = s;
clear s
s = instrfind;
isequal(scory,s)
an =
   1
```

See Also

**Functions**
delete, fclose, instrfind, isvalid

**Properties**

Status
Purpose
Clear current figure window

GUI Alternatives
Use Clear Figure from the figure window’s File menu to clear the contents of a figure. You can also create a desktop shortcut to clear the current figure with one mouse click. See “Using MATLAB Shortcuts to Easily Run a Group of Statements” in the MATLAB Desktop Environment documentation.

Syntax
clf('reset')
clf(fig)
clf(fig,'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.

clf(fig) or clf(fig,'reset') clears the single figure with handle fig.

figure_handle = clf(...) returns the handle of the figure. This is useful when the figure IntegerHandle property is off because 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” on page 1-102 for related functions
Purpose
Copy and paste strings to and from system clipboard

Graphical Interface
As an alternative to clipboard, use the Import Wizard. To use the Import Wizard to copy data from the clipboard, select Paste to Workspace from the Edit menu.

Syntax
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 The clipboard function requires Sun Microsystems Java software.

See Also
load, uiimport
**Purpose**  
Current time as date vector

**Syntax**  
c = clock

**Description**  
c = clock returns a 6-element date vector containing the current date and time in decimal form:

\[\text{[year month day hour minute seconds]}\]

The sixth element of the date vector output (seconds) is accurate to several digits beyond the decimal point. The statement `fix(clock)` rounds to integer display format.

**Remarks**  
When timing the duration of an event, use the tic and toc functions instead of clock or etime. These latter two functions are based on the system time which can be adjusted periodically by the operating system and thus might not be reliable in time comparison operations.

**See Also**  
cputime, datenum, datevec, now, etime, tic, toc
close

Purpose
Remove 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 an error that terminates the execution of a CloseRequestFcn occurs, 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 to `on`), you 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 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.

When coding a `CloseRequestFcn` callback, make sure that it does not call `close`, because this sets up a recursion that results in a MATLAB warning. Instead, the callback should destroy the figure with `delete`. The `delete` function does not execute the figure’s `CloseRequestFcn`; it simply deletes the specified figure.

**See Also**

`delete`, `figure`, `gcf`

The figure `HandleVisibility` property

The root `ShowHiddenHandles` property

“Figure Windows” on page 1-102 for related functions
**close (avifile)**

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Close Audio/Video Interleaved (AVI) file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>aviobj = close(aviobj)</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>aviobj = close(aviobj) finishes writing and closes the AVI file associated with aviobj, which is an AVI file object created using the avifile function.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td>avifile, addframe, movie2avi</td>
</tr>
</tbody>
</table>
Purpose
Close connection to FTP server

Syntax
close(f)

Description
close(f) closes the connection to 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 by exiting MATLAB.

Examples
Connect to the MathWorks FTP server and then disconnect.

```
tmw=ftp('ftp.mathworks.com');
close(tmw)
```

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” on page 1-102 for related functions
Purpose
Name of source control system

GUI
Alternatives
As an alternative to cmopts, select
File > Preferences > General > Source Control to view the currently selected source control system.

Syntax
cmopts

Description
cmopts displays the name of the source control system you selected using preferences, which is one of the following:

- `clearcase` (UNIX platforms only)
- `customverctrl` (UNIX platforms only)
- `cvs` (UNIX platforms only)
- `pvcs` (UNIX platforms only, used for PVCS® and ChangeMan® software)
- `rcs` (UNIX platforms only)
- `sourcesafe` (Windows platforms only)

If you have not selected a source control system, cmopts displays `none`

For more information, see “Specify Source Control System with MATLAB Software” for PC platforms, and “Specifying the Source Control System on UNIX Platforms” for UNIX platforms in the MATLAB Desktop Tools and Development Environment documentation.

Examples
Type
cmopts

and MATLAB returns

`ans =`
Microsoft Visual SourceSafe

which is the source control system specified in preferences.

See Also

checkin, checkout, customverctrl, verctrl
Purpose
Rearrange colors in colormap

Syntax

\[
[Y,\text{newmap}] = \text{cmpermute}(X,\text{map}) \\
[Y,\text{newmap}] = \text{cmpermute}(X,\text{map},\text{index})
\]

Description

\[
[Y,\text{newmap}] = \text{cmpermute}(X,\text{map})
\]
randomly reorders the colors in map to produce a new colormap, newmap. The \text{cmpermute} function also modifies the values in X to maintain correspondence between the indices and the colormap, and returns the result in Y. The image Y and associated colormap, newmap, produce the same image as X and map.

\[
[Y,\text{newmap}] = \text{cmpermute}(X,\text{map},\text{index})
\]
uses an ordering matrix (such as the second output of sort) to define the order of colors in the new colormap.

Class Support
The input image X can be of class \text{uint8} or \text{double}. Y is returned as an array of the same class as X.

Examples
Order a colormap by luminance.

\begin{verbatim}
load trees
ntsc = rgb2ntsc(map);
[dum,index] = sort(ntsc(:,1));
[Y,newmap] = cmpermute(X,map,index);
figure, imshow(X,map)
figure, imshow(Y,newmap)
\end{verbatim}

See Also
randperm, sort
Purpose
Eliminate duplicate colors in colormap; convert grayscale or truecolor image to indexed image

Syntax
[Y,newmap] = cmunique(X,map)
[Y,newmap] = cmunique(RGB)
[Y,newmap] = cmunique(I)

Description
[Y,newmap] = cmunique(X,map) returns the indexed image Y and associated colormap, newmap, that produce the same image as (X,map) but with the smallest possible colormap. The cmunique function removes duplicate rows from the colormap and adjusts the indices in the image matrix accordingly.

[Y,newmap] = cmunique(RGB) converts the truecolor image RGB to the indexed image Y and its associated colormap, newmap. The return value newmap is the smallest possible colormap for the image, containing one entry for each unique color in RGB.

Note newmap might be very large, because the number of entries can be as many as the number of pixels in RGB.

[Y,newmap] = cmunique(I) converts the grayscale image I to an indexed image Y and its associated colormap, newmap. The return value, newmap, is the smallest possible colormap for the image, containing one entry for each unique intensity level in I.

Class Support
The input image can be of class uint8, uint16, or double. The class of the output image Y is uint8 if the length of newmap is less than or equal to 256. If the length of newmap is greater than 256, Y is of class double.

Examples
1 Use the magic function to create a sample 4-by-4 image that uses every value in the range between 1 and 16.

   X = magic(4)
2 Concatenate two 8-entry grayscale colormaps created using the gray function. The resultant colormap, map, has 16 entries. Entries 9 through 16 are duplicates of entries 1 through 8.

```matlab
map = [gray(8); gray(8)]
size(map)
```

```matlab
ans =
    16     3
```

3 Use cmunique to eliminate duplicate entries in the colormap.

```matlab
[Y, newmap] = cmunique(X, map);
size(newmap)
```

```matlab
ans =
    8     3
```

cmunique adjusts the values in the original image X to index the new colormap.

```matlab
Y =

    7     1     2     4
    4     2     1     7
    0     6     5     3
    3     5     6     0
```

4 View both images to verify that their appearance is the same.
figure, imshow(X, map, 'InitialMagnification', 'fit')
figure, imshow(Y, newmap, 'InitialMagnification', 'fit')

See Also

rgb2ind
Purpose

Column approximate minimum degree permutation

Syntax

\[ p = \text{colamd}(S) \]

Description

\[ p = \text{colamd}(S) \] 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 \( S(:,p)' \ast S(:,p) \) also tends to be sparser than that of \( S' \ast S \).

\( \text{knobs} \) is a two-element vector. If \( S \) is \( m \)-by-\( n \), then rows with more than \( (\text{knobs}(1)) \ast n \) entries are ignored. Columns with more than \( (\text{knobs}(2)) \ast m \) entries are removed prior to ordering, and ordered last in the output permutation \( p \). If the \( \text{knobs} \) parameter is not present, then \( \text{knobs}(1) = \text{knobs}(2) = \text{spparms('wh_frac')} \).

\( \text{stats} \) is an optional vector that provides data about the ordering and the validity of the matrix \( S \).

\begin{align*}
\text{stats}(1) & \quad \text{Number of dense or empty rows ignored by colamd} \\
\text{stats}(2) & \quad \text{Number of dense or empty columns ignored by colamd} \\
\text{stats}(3) & \quad \text{Number of garbage collections performed on the internal data structure used by colamd (roughly of size } 2.2 \ast \text{nnz}(S) + 4 \ast m + 7 \ast n \text{ integers)} \\
\text{stats}(4) & \quad 0 \text{ if the matrix is valid, or } 1 \text{ if invalid} \\
\text{stats}(5) & \quad \text{Rightmost column index that is unsorted or contains duplicate entries, or } 0 \text{ if no such column exists}
\end{align*}
colamd

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, colamd ignores the duplicate entries, continues processing, and provides information about the duplicate entries in stats(4:7).
- If row indices in a column are out of order, colamd sorts each column of its internal copy of the matrix S (but does not repair the input matrix S), continues processing, and provides information about the out-of-order entries in stats(4:7).
- If S is invalid in any other way, colamd cannot continue. It prints an error message, and returns no output arguments (p or stats).

The ordering is followed by a column elimination tree post-ordering.

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 colamd ordering scrambles this structure.

    load west0479
    A = west0479;
    p = colamd(A);
    subplot(1,2,1), spy(A,4), title('A')
    subplot(1,2,2), spy(A(:,p),4), title('A(:,p)')
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.

```
spy(lu(A),4)  
spy(lu(A(:,p)),4)
```
See Also  
colperm, spparms, symamd, symrcm

References  
[1] The authors of the code for “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

Colorbar showing color scale

GUI

Alternatives

Add a colorbar to a plot with the colorbar tool on the figure toolbar, or use Insert —> Colorbar from the figure menu. Use the Property Editor to modify the position, font and other properties of a legend. For details, see “Working in Plot Edit Mode” in the MATLAB Graphics documentation.

Syntax

colorbar

colorbar('off')
colorbar('hide')
colorbar('delete')
colorbar(...,'peer',axes_handle)
colorbar(...,'location')
colorbar(...,'PropertyName',propertyvalue)
cbar_axes = colorbar(...)  
colorbar(cbar_handle,'off')  
colorbar(cbar_handle,'hide')  
colorbar(cbar_handle,'delete')  
colorbar(cbar_handle, 'PropertyName',propertyvalue,...)

Description

The colorbar function displays the current colormap in the current figure and resizes the current axes to accommodate the colorbar.

colorbar adds a new vertical colorbar on the right side of the current axes. If a colorbar exists in that location, colorbar replaces it with a new one. If a colorbar exists at a nondefault location, it is retained along with the new colorbar.

colorbar('off'), colorbar('hide'), and colorbar('delete') delete all colorbars associated with the current axes.

colorbar(...,'peer',axes_handle) creates a colorbar associated with the axes axes_handle instead of the current axes.

colorbar(...,'location') adds a colorbar in the specified orientation with respect to the axes. If a colorbar exists at the location specified,
it is replaced. Any colorbars not occupying the specified location are retained. Possible values for location are

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>North Inside</td>
<td>Inside plot box near top</td>
</tr>
<tr>
<td>South Inside</td>
<td>Inside bottom</td>
</tr>
<tr>
<td>East Inside</td>
<td>Inside right</td>
</tr>
<tr>
<td>West Inside</td>
<td>Inside left</td>
</tr>
<tr>
<td>North Outside</td>
<td>Outside plot box near top</td>
</tr>
<tr>
<td>South Outside</td>
<td>Outside bottom</td>
</tr>
<tr>
<td>East Outside</td>
<td>Outside right</td>
</tr>
<tr>
<td>West Outside</td>
<td>Outside left</td>
</tr>
</tbody>
</table>

Using one of the ...Outside values for location ensures that the colorbar does not overlap the plot, whereas overlaps can occur when you specify any of the other four values.

`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. The location property applies only to colorbars and legends, not to axes.

`cbar_axes = colorbar(...)` returns a handle to a new colorbar object, which is a child of the current figure. If a colorbar exists, a new one is still created.

`colorbar(cbar_handle,'off'), colorbar(cbar_handle,'hide'), and colorbar(cbar_handle,'delete')` delete the colorbar specified by cbar_handle.

`colorbar(cbar_handle, PropertyName',propertyvalue,...)` sets properties for the existing colorbar having the handle cbar_handle. To obtain the handle to an existing colorbar, use the command

```
cbar_handle = findobj(figure_handle,'tag','Colorbar')
```
where `figure_handle` is the handle of the figure containing the colorbar you want to modify. If the figure contains more than one colorbar, `cbar_handle` is returned as a vector, and you must choose which of the handles to specify to `colorbar`.

**Backward-Compatible Version**

`h = colorbar('v6',...)` creates a colorbar compatible with MATLAB 6.5 and earlier. It returns the handles of patch objects instead of a colorbar object.

**Note** The v6 option enables MATLAB Version 7.x users to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

**Remarks**
You can use `colorbar` with 2-D and 3-D plots.

**Examples**

**Example 1**

Display a colorbar beside the axes and use descriptive text strings as y-tick labels. Note that labels will repeat cyclically when the number of y-ticks is greater than the number of labels, and not all labels will appear if there are fewer y-ticks than labels you have specified. Also note that when colorbars are horizontal, their ticks and labels are governed by the `XTick` property rather than the `YTick` property. For more information, see “Labeling Colorbar Ticks”.

```matlab
surf(peaks(30))
colorbar('YTickLabel',...{'Freezing','Cold','Cool','Neutral',... 'Warm','Hot','Burning','Nuclear'})
```
Example 2

Display a horizontal colorbar beneath the axes of a filled contour plot:

```matlab
contourf(peaks(60))
colormap cool
colorbar('location','southoutside')
```
See Also
colormap

"Color Operations" on page 1-105 for related functions
Purpose

Set default property values to display different color schemes

Syntax

```
colordef white
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 so that they contrast with 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. The most noticeable difference is that the axis background is set to 'none', making the axis background and figure background colors the same. The figure background color is set to black.

colordef(fig,color_option) sets the color scheme of the figure identified by the handle fig to one of the color options 'white', 'black', or 'none'. When you use this syntax to apply colordef to an existing figure, the figure must have no graphic content. If it does, you should first clear it (via clf) before using this form of the command.

```
h = colordef('new',color_option)
```
returns the handle to a new figure created with the specified color options (i.e., 'white', 'black', or 'none'). This form of the command is useful for creating GUIs when you may want to control the default environment. The figure is created with 'visible','off' to prevent flashing.

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, clf

“Color Operations” on page 1-105 for related functions
### Purpose
Set and get current colormap

### GUI Alternatives
Select a built-in colormap with the Property Editor. To modify the current colormap, use the Colormap Editor, accessible from Edit > Colormap on the figure menu.

### Syntax

- `colormap(map)`
- `colormap('default')`
- `cmap = colormap`
- `colormap(ax,...)`

### 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.

- `colormap(map)` sets the colormap to the matrix \( \text{map} \). If any values in \( \text{map} \) are outside the interval [0 1], you receive the error Colormap must have values in [0,1].
- `colormap('default')` sets the current colormap to the default colormap.
- `cmap = colormap` retrieves the current colormap. The values returned are in the interval [0 1].
- `colormap(ax,...)` uses the figure corresponding to axes \( \text{ax} \) instead of the current figure.

### Specifying Colormaps
M-files in the color directory generate a number of colormaps. Each M-file accepts the colormap size as an argument. For example,

```matlab
    colormap(hsv(128))
```

creates an hsv colormap with 128 colors. If you do not specify a size, a colormap the same size as the current colormap is created.
**Supported Colormaps**

The built-in MATLAB colormaps are illustrated and described below. In addition to specifying built-in colormaps programatically, you can use the Colormap menu in the Figure Properties pane of the Plot Tools GUI to select one interactively.

The named built-in colormaps are the following:

- **autumn** varies smoothly from red, through orange, to yellow.
- **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.
- **cubecol** contains as many regularly spaced colors in RGB color space as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
- **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. hsv(m) is the same as hsv2rgb([h 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 “Examples” on page 2-650 on page -3.
- 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
```
Algorithm

Each figure has its own colormap property. colormap is an M-file that sets and gets this property.

See Also

brightness, caxis, colorbar, colormapeditor, contrast, hsv2rgb, pcolor, rgbplot, rgb2hsv

The Colormap property of figure graphics objects

“Color Operations” on page 1-105 for related functions

“Coloring Mesh and Surface Plots” for 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. A color picker box appears, from which you can select a new color. After you select a new color at a node, the colors between nodes are reinterpolated.

<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. 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>
### Operation | How to Perform
---|---
Move multiple nodes | Select multiple nodes and use the left and right arrow keys to move nodes as a group. Movement stops when one of the selected nodes hits an unselected node or an end node.
Delete a node | Select the node and then press the **Delete** key, or select **Delete** from the **Edit** menu, or type **Ctrl+x**.
Delete multiple nodes | Select the nodes and then press the **Delete** key, or select **Delete** from the **Edit** menu, or type **Ctrl+x**.
Display color picker for a node | Double-click the node pointer.

#### 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 3,4,5...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).
1 Start the colormap editor using the `colormapeditor` command. The color map 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.
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.
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.
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.
The slice plane now has the desired range of colors for negative, zero, and positive data.
Green cell is now centered around zero.

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 (Color Specification)

Purpose
Color specification

Description
ColorSpec is not a function; it refers to the three ways in which you specify color for MATLAB graphics:

- 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.

Some high-level functions (for example, scatter) accept a colorspec as an input argument and use it to set the CData of graphic objects they
create. When using such functions, take care not to specify a colorspec in a property/value pair that sets CData; values for CData are always n-length vectors or n-by-3 matrices, where n is the length of XData and YData, never strings.

**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:

```matlab
whitebg('g')
whitebg('green')
whitebg([0 1 0]);
```

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” on page 1-105 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 \( \text{lu}(S(:,j)) \).

If \( S \) is symmetric, then \( j = \text{colperm}(S) \) 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 \( \text{chol}(S(j,j)) \).

Algorithm
The algorithm involves a sort on the counts of nonzeros in each column.

Examples
The \( n \)-by-\( n \) \textit{arrowhead} matrix

\[
A = [\text{ones}(1,n); \text{ones}(n-1,1) \ \text{speye}(n-1,n-1)]
\]

has a full first row and column. Its LU factorization, \( \text{lu}(A) \), is almost completely full. The statement

\[
j = \text{colperm}(A)
\]

returns \( j = [2:n 1] \). So \( A(j,j) \) sends the full row and column to the bottom and the rear, and \( \text{lu}(A(j,j)) \) has the same nonzero structure as \( A \) itself.

On the other hand, the Bucky ball example,

\[
B = \text{bucky}
\]

has exactly three nonzero elements in each row and column, so \( j = \text{colperm}(B) \) is the identity permutation and is no help at all for reducing fill-in with subsequent factorizations.
See Also

chol, colamd, lu, spparms, symamd, symrcm
**Purpose**

2-D comet plot

**GUI Alternatives**

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

**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 $p \times \text{length}(y)$. *p* defaults to 0.1.
- `comet(axes_handle,...)` plots into the axes with the handle *axes_handle* instead of into the current axes (*gca*).

**Remarks**

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).
Create a simple comet graph:

```matlab
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” on page 1-96 for related functions
Purpose

3-D comet plot

GUI Alternatives

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

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 3-D 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 the handle axes_handle instead of into the current axes (gca).

Remarks

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).
comet3

**Examples**

Create a 3-D 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” on page 1-96 for related functions
Purpose
Open Command History window, or select it if already open

GUI Alternatives
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 window when it is closed, and selects the Command History window when it is open. The Command History window presents a log of the statements most recently run in the Command Window.

See Also
diary, prefdir, startup

MATLAB Desktop Tools and Development Environment Documentation

- “Recalling Previous Lines in the Command Window”
- “Command History Window”
**commandwindow**

**Purpose**
Open Command Window, or select it if already open

**GUI Alternatives**
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 display in the Command Window, given its current size, use

```matlab
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**
commandhistory, input, inputdlg

MATLAB Desktop Tools and Development Environment documentation
- “Running Functions and Programs, and Entering Variables”
- “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

\[
\begin{bmatrix}
1 & 0 & -7 & 6 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
\end{bmatrix}
\]

The eigenvalues are the polynomial roots:

\[
\text{eig(compan(u))}
\]

\[
\text{ans} =
\begin{bmatrix}
-3.0000 \\
2.0000 \\
1.0000 \\
\end{bmatrix}
\]

This is also roots(u).

See Also      eig, poly, polyval, roots
**Purpose**

Plot arrows emanating from origin

**GUI Alternatives**

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

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

2-682
h = compass(...) returns handles to line objects.

**Examples**

Draw a compass graph of the eigenvalues of a matrix.

```matlab
Z = eig(randn(20,20));
compass(Z)
```

**See Also**

feather, LineSpec, quiver, rose

“Direction and Velocity Plots” on page 1-96 for related functions

“Compass Plots” for another example
Purpose

Construct complex data from real and imaginary components

Syntax

c = complex(a,b)

Description

c = 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.

\[ c = \text{complex}(a) \] 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 \( \text{isreal}(c) \) returns false.

The complex function provides a useful substitute for expressions such as

\[ a + i*b \quad \text{or} \quad a + j*b \]
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.

```matlab
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**

Information about computer on which MATLAB software is running

**Syntax**

\[
\begin{align*}
\text{str} &= \text{computer} \\
\text{archstr} &= \text{computer}('\text{arch}') \\
[\text{str}, \text{maxsize}] &= \text{computer} \\
[\text{str}, \text{maxsize}, \text{endian}] &= \text{computer}
\end{align*}
\]

**Description**

\(\text{str} = \text{computer}\) returns the string \(\text{str}\) with the computer type on which MATLAB is running.

\(\text{archstr} = \text{computer}('\text{arch}')\) returns the string \(\text{archstr}\) which is the architecture of the build platform. Use this string for the term \text{arch} in the \text{mex} command switch -arch.

\([\text{str}, \text{maxsize}] = \text{computer}\) returns the integer \text{maxsize}, the maximum number of elements allowed in an array with this version of MATLAB.

\([\text{str}, \text{maxsize}, \text{endian}] = \text{computer}\) returns either 'L' for little-endian byte ordering or 'B' for big-endian byte ordering.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Word Size</th>
<th>str</th>
<th>archstr</th>
<th>maxsize</th>
<th>endian</th>
<th>ispc</th>
<th>isunix</th>
<th>ismac</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Windows</td>
<td>32-bit PCWIN</td>
<td>win32</td>
<td>2^31 - 1</td>
<td>L</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64-bit PCWIN64</td>
<td>win64</td>
<td>2^48 - 1</td>
<td>L</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Linux®</td>
<td>32-bit GLNX86</td>
<td>glnx86</td>
<td>2^31 - 1</td>
<td>L</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64-bit GLNXA64</td>
<td>glnxa64</td>
<td>2^48 - 1</td>
<td>L</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
### Remarks

In some cases, both 32-bit and 64-bit versions of MATLAB can run on the same platform. In this case, the value returned by `computer` reflects which of these is running. For example, if you run a 32-bit version of MATLAB on a Windows x64 platform, `computer` returns `PCWIN`, indicating that the 32-bit version is running.

### See Also

`getenv`, `setenv`, `ispc`, `isunix`, `ismac`
**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.

$c = \text{cond}(X)$ returns the 2-norm condition number, the ratio of the largest singular value of $X$ to the smallest.

$c = \text{cond}(X,p)$ returns the matrix condition number in $p$-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
Purpose
1-norm condition number estimate

Syntax
\[ c = \text{condest}(A) \]
\[ c = \text{condest}(A,t) \]
\[ [c,v] = \text{condest}(A) \]

Description
\( c = \text{condest}(A) \) computes a lower bound \( C \) for the 1-norm condition number of a square matrix \( A \).

\( c = \text{condest}(A,t) \) 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.

\( [c,v] = \text{condest}(A) \) also computes a vector \( v \) which is an approximate null vector if \( c \) is large. \( v \) satisfies \( \text{norm}(A*v,1) = \text{norm}(A,1)*\text{norm}(v,1)/c \).

Note condest invokes rand. If repeatable results are required then invoke \text{rand}('state',j), for some \( j \), before calling this function.

This function is particularly useful for sparse matrices.

Algorithm
condest is based on the 1-norm condition estimator of Hager [1] and a block oriented generalization of Hager’s estimator given by Higham and Tisseur [2]. The heart of the algorithm involves an iterative search to estimate \( \|A^{-1}\|_1 \) without computing \( A^{-1} \). This is posed as the convex, but nondifferentiable, optimization problem

\[ \max \|A^{-1} x\|_1 \text{subject to } \|x\|_1 = 1 \]

See Also
cond, norm, normest
Reference


Purpose

Plot velocity vectors as cones in 3-D vector field

GUI

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

Syntax

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,'nointerp')
coneplot(axes_handle,...)
h = coneplot(...)

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 “Specifying 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 \([X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p)\), where \([m,n,p]=\text{size}(U)\).

coneplot(...) 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 \(s\), a value of 1 is used. 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 the handle
axes_handle instead of into 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.

\[
\text{daspect}([1,1,1])
\]
**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:

- 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 viewpoint, 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` to automatically 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;
[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 `xmin` and `xmax`, along the y-axis at `ymax`, and along the z-axis at `zmin`.

• 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).

    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” on page 1-108 for related functions
**Purpose**  
Complex conjugate

**Syntax**  
\( ZC = \text{conj}(Z) \)

**Description**  
\( ZC = \text{conj}(Z) \) returns the complex conjugate of the elements of \( Z \).

**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 next iteration of `for` or `while` loop

**Syntax**
```plaintext```
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. The same holds true for `continue` statements in nested loops. That is, execution continues at the beginning of the loop in which the `continue` statement was encountered.

**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.

```plaintext```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
fprintf('%d lines\n',count);
fclose(fid);
```

**See Also**
`for`, `while`, `end`, `break`, `return`
**Purpose**

Contour plot of matrix

**GUI Alternatives**

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

**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)
contour(axes_handle,...)
[C,h] = contour(...)
[C,h] = contour('v6',...)
```

**Description**

A contour plot 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 that contains at least two different values. The number of contour lines and the values of the contour lines 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 the monotonically increasing vector v. The number of contour levels is equal to `length(v)`. To draw a single
contour of level i, use \texttt{contour(Z,[i i])}. Specifying the vector v sets the LevelListMode to manual to allow user control over contour levels. See contourgroup properties for more information.

\texttt{contour(X,Y,Z)}, \texttt{contour(X,Y,Z,n)}, and \texttt{contour(X,Y,Z,v)} draw 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 and must be monotonically increasing.

\texttt{contour(...,LineSpec)} draws the contours using the line type and color specified by LineSpec. contour ignores marker symbols.

\texttt{contour(axes_handle,...)} plots into axes gerkaxes_handle instead of gca.

\texttt{[C,h] = contour(...)} returns a contour matrix, C, derived from the matrix returned by the low-level \texttt{contourc} function, and a handle, h, to a contourgroup object. \texttt{clabel} uses the contour matrix C to create the labels. (See descriptions of contourgroup properties.)

**Backward Compatibility**

\texttt{[C,h] = contour('v6',...)} returns the contour matrix C, as calculated by the function \texttt{contourc} and used by \texttt{clabel}, a vector of handles h to patch graphics objects instead of a contourgroup object, for compatibility with MATLAB Version 6.5 and earlier. When called with the 'v6' flag, contour creates patch graphics objects, unless you specify a LineSpec, in which case contour creates line graphics objects. 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. If you do not specify a LineSpec argument, the figure colormap and the color limits (caxis) control the color of the contour lines (patch objects).

**Note** The v6 option enables users of MATLAB Version 7.x to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.
### Remarks

Use `contourgroup` object properties to control the contour plot appearance.

If X or Y is irregularly spaced, `contour` calculates contours using a regularly spaced contour grid, and then transforms the data to X or Y.

### Examples

**Contour Plot of a Function**

Create a contour plot of the `peaks` function using the contour matrix and `contourgroup` object handle as output.

```matlab
[C,h] = contour(peaks(20),10);
colormap autumn
```
**Smoothing Contour Data**

Use `interp2` to create smoother contours. Also set the contour label text `BackgroundColor` to a light yellow and the `EdgeColor` to light gray.

```
Z = peaks;
[C,h] = contour(interp2(Z,4));
text_handle = clabel(C,h);
set(text_handle,'BackgroundColor',[1 1 .6],...
    'Edgecolor',[.7 .7 .7])
```

For more examples using `contour`, see “Contour Plots”.

**See Also**  
clabel, contourf, contour3, contourc, quiver
“Contour Plots” for related functions and more examples

countourgroup properties for related properties
**Purpose**

3-D contour plot

**GUI Alternatives**

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

**Syntax**

```
contour3(Z)
contour3(Z,n)
contour3(Z,v)
contour3(X,Y,Z)
contour3(X,Y,Z,n)
contour3(X,Y,Z,v)
contour3(...,LineSpec)
contour3(axes_handle,...)
[C,h] = contour3(...)
```

**Description**

`contour3` creates a 3-D contour plot of a surface defined on a rectangular grid.

`contour3(Z)` draws a contour plot of matrix `Z` in a 3-D view. `Z` is interpreted as heights with respect to the x-y plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The number of contour levels and the values of 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).

`contour3(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels in a 3-D 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]). Specifying the vector v sets the LevelListMode to manual to allow user control over contour levels. See contourgroup properties for more information.

contour3(X,Y,Z), contour3(X,Y,Z,n), and contour3(X,Y,Z,v) draw contour plots of Z using X and Y to determine 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 and must be monotonically increasing.

contour3(...) draws the contour lines using the line type and color specified by LineSpec. contour3 ignores marker symbols.

contour3(axes_handle,...) plots into the axes with the handle axes_handle instead of into the current axes (gca).

[C,h] = contour3(...) returns a contour matrix C, derived from the matrix returned by the low-level contourc function, and a handle, h, to a contourgroup object containing handles to graphics objects. contour3 creates patch graphics objects unless you specify LineSpec, in which case contour3 creates line graphics objects.

Remarks

If X or Y is irregularly spaced, contour3 calculates contours using a regularly spaced contour grid, and then transforms the data to X or Y. If you do not specify LineSpec, the functions colormap and caxis control the color.

Label the contour lines using clabel.

contour3(...) works the same as contour(...) with these exceptions:

- The contours are drawn at their corresponding Z level.
- Multiple patch objects are created instead of a contourgroup.
- Calling contour3 with trailing property-value pairs is not allowed.

Examples

Plot the three-dimensional contour of a function and superimpose a surface plot to enhance visualization of the function.
\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}

For more examples using contour3, see “Contour Plots”.

**See Also**
contour, contourc, contourf, meshc, meshgrid, surfc
“Contour Plots” section for more examples
contourgroup properties for related properties
**Purpose**

Low-level contour plot computation

**Syntax**

\[ C = \text{contourc}(Z) \]
\[ C = \text{contourc}(Z,n) \]
\[ C = \text{contourc}(Z,v) \]
\[ C = \text{contourc}(x,y,Z) \]
\[ C = \text{contourc}(x,y,Z,n) \]
\[ C = \text{contourc}(x,y,Z,v) \]

**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 = \text{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 = \text{contourc}(Z,n)` computes contours of matrix `Z` with `n` contour levels.

`C = \text{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 = \text{contourc}(x,y,Z)`, `C = \text{contourc}(x,y,Z,n)`, and `C = \text{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.

\[ C = [\text{value1} \ \text{xdata}(1) \ \text{xdata}(2) \ \ldots \ \text{xdata}(\text{dim1}) \ \text{value2} \ \text{xdata}(1) \ \text{xdata}(2) \]
Specifying irregularly spaced x and y vectors is not the same as contouring irregularly spaced data. If x or y is irregularly spaced, \texttt{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” on page 1-96 for related functions

“The Contouring Algorithm” for more information
contourf

Purpose

Filled 2-D contour plot

GUI

Alternatives

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

Syntax

contourf(Z)
ccontourf(Z,n)
contourf(Z,v)
contourf(X,Y,Z)
contourf(X,Y,Z,n)
contourf(X,Y,Z,v)
contourf(...,LineSpec)
contourf(axes_handle,...)
[C,h] = contourf(...)
[C,h,CF] = contourf('v6',...)

Description

A filled contour plot displays isolines calculated from matrix Z and fills the areas between the isolines using constant colors corresponding to the current figure’s colormap.

contourf(Z) draws a filled 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 that contains at least two different values. The number of contour lines and the values of the contour lines 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).

contourf(Z,n) draws a filled contour plot of matrix Z with n contour levels.
contourf(Z,v) draws a filled contour plot of matrix Z with contour lines at the data values specified in the monotonically increasing 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]). Specifying the vector v sets the LevelListMode to manual to allow user control over contour levels. See contourgroup properties for more information.

contourf(X,Y,Z), contourf(X,Y,Z,n), and contourf(X,Y,Z,v) draw filled 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 and must be monotonically increasing.

contourf(...,LineSpec) draws the contour lines using the line type and color specified by LineSpec. contourf ignores marker symbols.

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

[C,h] = contourf(...) returns a contour matrix C, derived from the matrix returned by the low-level contourc function, and a handle, h, to a contourgroup object containing the filled contours. clabel uses the contour matrix C to create the labels. (See descriptions of contourgroup properties.)

**Backward Compatibility**

[C,h,CF] = contourf('v6',...) returns the contour matrix C, as calculated by the function contourc and used by clabel, a vector of handles h to patch graphics objects (instead of a contourgroup object, for compatibility with MATLAB Version 6.5 and earlier) and a contour matrix CF for the filled areas. When called with the 'v6' flag, contourf creates patch graphics objects, unless you specify a LineSpec. In this case, contour creates line graphics objects and colors them using the colors defined in the axes ColorOrder property. If you do not specify a LineSpec argument, the figure colormap and the color limits (caxis) control the color of the contour lines (patch objects).
contourf

Note  The v6 option enables users of MATLAB Version 7.x to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

Use contourgroup object properties to control the filled contour plot appearance.

Label the contour lines using clabel.

NaNs in the Z-data leave white holes with black borders in the contour plot.

If X or Y is irregularly spaced, contourf calculates contours using a regularly spaced contour grid, and then transforms the data to X or Y.

Examples

Create a filled contour plot of the peaks function with contour matrix and contourgroup object handle as output and autumn colormap.

```matlab
[C,h] = contourf(peaks(20),10);
colormap autumn
```
For more examples using `contourf`, see “Contour Plots”.

**See Also**
- `clabel`, `contour`, `contour3`, `contourc`, `quiver`
- “Contour Plots” for related functions and more examples
- `contourgroup properties` for related properties
Contourgroup Properties

**Purpose**
Define 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.

**Annotation**
- **hg.Annotation object Read Only**

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

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

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

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Include the contourgroup object in a legend as one entry, but not its children objects</td>
</tr>
<tr>
<td>off</td>
<td>Do not include the contourgroup or its children in a legend (default)</td>
</tr>
<tr>
<td>children</td>
<td>Include only the children of the contourgroup as separate entries in the legend</td>
</tr>
</tbody>
</table>
Contourgroup Properties

Setting the IconDisplayStyle Property

These commands set the IconDisplayStyle of a graphics object with handle hobj to children, which causes each child object to have an entry in the legend:

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

Using the IconDisplayStyle Property

See “Controlling Legends” for more information and examples.

BeingDeleted

on | {off} Read Only

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

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

BusyAction

cancel | {queue}

*Callback routine interruption.* The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.
Contour group Properties

If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the Interruptible property is off, the BusyAction property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

ButtonDownFcn

string or function handle

*Button press callback function.* A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the HitTestArea property for information about selecting objects of this type.

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

This property can be

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

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

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

Children

array of graphics object handles
**Contourgroup Properties**

*Children of this object.* The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

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

**Clipping**

{on} | off

*Clipping mode.* MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

**ContourMatrix**

2-by-n matrix Read Only

*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:

\[
C = \begin{bmatrix}
\text{value1} & xdata(1) & xdata(2) & \ldots & \text{value2} & xdata(1) & xdata(2) & \ldots & \\
\text{dim1} & ydata(1) & ydata(2) & \ldots & \text{dim2} & ydata(1) & ydata(2) & \ldots & \\
\end{bmatrix}
\]

That is,

\[
C = [C(1) \ C(2) \ldots C(I) \ldots C(N)]
\]

where \(N\) is the number of contour levels, and

\[
C(i) = \begin{bmatrix}
\text{level(i)} & x(1) & x(2) & \ldots & x( \text{numel(i)})
\end{bmatrix}
\]
Contour group Properties

numel(i) y(1) y(2)...y( numel(i));

For further information, see The Contouring Algorithm.

CreateFcn
string or function handle

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

area(y,'CreateFcn',@CallbackFcn)

where @CallbackFcn is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

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

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

DeleteFcn
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a delete command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.
The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

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

See the `BeingDeleted` property for related information.

**DisplayName**

string (default is empty string)

*String used by legend for this contourgroup object.* The `legend` function uses the string defined by the `DisplayName` property to label this contourgroup object in the legend.  

- If you specify string arguments with the `legend` function, `DisplayName` is set to this contourgroup object’s corresponding string and that string is used for the legend.

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

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

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

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

See “Controlling Legends” for more examples.

**EraseMode**

{normal} | none | xor | background
Contourgroup Properties

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

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

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

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

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

Printing with Nonnormal Erase Modes

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

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

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

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 objects that you need to protect for some reason.

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

**Functions Affected by Handle Visibility**

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

**Properties Affected by Handle Visibility**

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

**Overriding Handle Visibility**

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

**Handle Validity**

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

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

HitTest

{on} | off

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

HitTestArea

on | {off}

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When HitTestArea is off, you must click the object’s lines or markers (excluding the baseline, if any) to select the object. When HitTestArea is on, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

Interruptible

{on} | off

Callback routine interruption mode. The Interruptible property controls whether an object’s callback can be interrupted by callbacks invoked subsequently.
Contourgroup Properties

Only callbacks defined for the ButtonDownFcn property are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback only when it encounters a drawnow, figure, getframe, or pause command in the routine. See the BusyAction property for related information.

Setting Interruptible to on allows any graphics object’s callback to interrupt callback routines originating from a 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.

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
**Contourgroup Properties**

*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`.

**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.
Contourgroup Properties

LineStyle
{}
{ } | – | : | -. | none

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

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

LineWidth
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = \(\frac{1}{72}\) inch). The defaultLineWidth is 0.5 points.

Parent
handle of parent axes, hgroup, or hgtransform

Parent of this object. This property contains the handle of the object’s parent. The parent is normally the axes, hgroup, or hgtransform object that contains the object.

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

Selected
on | {off}

Is object selected? When you set this property to on, MATLAB displays selection "handles" at the corners and midpoints if the
SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

**SelectionHighlight**

{on} | off

*Objects are highlighted when selected.* When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

**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” on page 2-702.

**Tag**

string

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

For example, you might create an areaseries object and set the Tag property.

```matlab
t = area(Y, 'Tag', 'area1')```
When you want to access objects of a given type, you can use \texttt{findobj} to find the object’s handle. The following statement changes the \texttt{FaceColor} property of the object whose \texttt{Tag} is \texttt{area1}.

\begin{verbatim}
set(findobj('Tag','area1'),'FaceColor','red')
\end{verbatim}

\textbf{TextList}

\texttt{vector of contour values}

\textit{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 the \texttt{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 plot:

\begin{verbatim}
[c,h]=contour(peaks);
clabel(c,h)
\end{verbatim}

You can get the \texttt{LevelList} property to see the contour line values:

\begin{verbatim}
get(h,'LevelList')
\end{verbatim}

Suppose you want to view the contour value \texttt{4.375} instead of the value of \texttt{4} that the contour function used. To do this, you need to set both the \texttt{LevelList} and \texttt{TextList} properties:

\begin{verbatim}
set(h,'LevelList',[-6 -4 -2 0 2 4.375 6 8],...
 'TextList',[-6 -4 -2 0 2 4.375 6 8])
\end{verbatim}

See the example “Contour Plot of a Function” on page 2-702 for additional information.

\textbf{TextListMode}

\texttt{\{auto\} | manual}
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
scalar

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” on page 2-702 for an example that uses the TextStep property.

TextStepMode
{auto} | manual

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.

Type
string (read only)

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 hgroup objects in the current axes.

    t = findobj(gca,'Type','hggroup');
Contourgroup Properties

UIContextMenu
handle of a uicontextmenu object

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

UserData
array

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

Visible
{on} | off

Visibility of this object and its children. By default, a new object’s visibility is on. This means all children of the object are visible unless the child object’s Visible property is set to off. Setting an object’s Visible property to off prevents the object from being displayed. However, the object still exists and you can set and query its properties.

XData
vector or matrix

The x-axis values for a graph. The x-axis values for graphs are specified by the X input argument. If XData is a vector, length(XData) must equal length(YData) and must be monotonic. If XData is a matrix, size(XData) must equal size(YData) and each column must be monotonic.
You can use XData to define meaningful coordinates for an underlying surface whose topography is being mapped. See for more information.

**XDataMode**

\{auto\} | manual

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

If you set XDataMode to auto after having specified XData, MATLAB resets the x-axis ticks to 1:size(YData,1) or to the column indices of the ZData, overwriting any previous values for XData.

**XDataSource**

string (MATLAB variable)

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

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

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

See the refreshdata reference page for more information.
Contourgroup Properties

**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

*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 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 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.
**Purpose**
Draw contours in volume slice planes

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

**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 the handle axes_handle instead of into 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 10-element vector of linearly spaced values from -8 to 2. This vector specifies that 10 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.

[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;
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
```
contourslice

[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” on page 1-108 for related functions
Purpose
Grayscale colormap for contrast enhancement

Syntax
- cmap = contrast(X)
- cmap = 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.

- `cmap = contrast(X)` returns a gray colormap that is the same length as the current colormap.
- `cmap = contrast(X,m)` returns an m-by-3 gray colormap.

Examples
Add contrast to the clown image defined by X.

```matlab
load clown;
cmap = contrast(X);
image(X);
colormap(cmap);
```

See Also
- `brighten`, `colormap`, `image`

for related functions
**Purpose**  
Convolution and polynomial multiplication

**Syntax**  
\[ w = \text{conv}(u,v) \]
\[ C = \text{conv}(..., 'shape') \]

**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 \( v \).

\[ C = \text{conv}(..., 'shape') \] 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 \( \text{length}(c) \) is \( \text{max}(\text{length}(a)-\text{max}(0,\text{length}(b)-1),0) \)

**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

\[ w(k) = \sum_j u(j)v(k-j) \]

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

\[
\begin{align*}
w(1) &= u(1) \cdot v(1) \\
w(2) &= u(1) \cdot v(2) + u(2) \cdot v(1) \\
w(3) &= u(1) \cdot v(3) + u(2) \cdot v(2) + u(3) \cdot v(1)
\end{align*}
\]
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**

2-D convolution

**Syntax**

\[
C = \text{conv2}(A,B)
\]
\[
C = \text{conv2}(\text{hcol},\text{hrow},A)
\]
\[
C = \text{conv2}(...,\text{shape'})
\]

**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 \([ma+mb-1,na+nb-1]\).

The indices of the center element of \(B\) are defined as \(\text{floor}(((mb\ nb)+1)/2)\).

\(C = \text{conv2}(\text{hcol},\text{hrow},A)\) convolves \(A\) first with the vector \(\text{hcol}\) along the rows and then with the vector \(\text{hrow}\) along the columns. If \(\text{hcol}\) is a column vector and \(\text{hrow}\) is a row vector, this case is the same as \(C = \text{conv2}(\text{hcol}\ast\text{hrow},A)\).

\(C = \text{conv2}(...,\text{shape'})\) returns a subsection of the two-dimensional convolution, as specified by the \text{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 \(\text{all(size}(A) >= \text{size}(B))\). Otherwise \text{conv2} returns \([]\).
**Note** If any of A, B, hcol, and hrow are empty, then C is an empty matrix [ ].

**Algorithm**

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, conv2 computes the convolution for finite intervals. Note that matrix indices in MATLAB software 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, 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.

\[
A = \text{rand}(3);
B = \text{rand}(4);
C = \text{conv2}(A,B) \quad \% \text{C is 6-by-6}
\]

\[
C =
\begin{bmatrix}
0.1838 & 0.2374 & 0.9727 & 1.2644 & 0.7890 & 0.3750 \\
0.6929 & 1.2019 & 1.5499 & 2.1733 & 1.3325 & 0.3096 \\
0.5627 & 1.5150 & 2.3576 & 3.1553 & 2.5373 & 1.0602
\end{bmatrix}
\]
conv2

\[
\begin{array}{cccccc}
0.9986 & 2.3811 & 3.4302 & 3.5128 & 2.4489 & 0.8462 \\
0.3089 & 1.1419 & 1.8229 & 2.1561 & 1.6364 & 0.6841 \\
0.3287 & 0.9347 & 1.6464 & 1.7928 & 1.2422 & 0.5423 \\
\end{array}
\]

\[
Cs = \text{conv2}(A,B,'same') \quad \% \text{Cs is the same size as A: 3-by-3}
\]

\[
Cs = \\
\begin{array}{ccc}
2.3576 & 3.1553 & 2.5373 \\
3.4302 & 3.5128 & 2.4489 \\
1.8229 & 2.1561 & 1.6364 \\
\end{array}
\]

**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 $A$.

$$V = \text{conv2}(A, s');$$
$$\text{figure, mesh}(V)$$
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.

[K,a] = convhull(...) also returns the area of the convex hull.

Visualization
Use plot to plot the output of convhull.

Examples

Example 1

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+')
Example 2

The following example illustrates the options input for convhull. The following commands

```matlab
X = [0 0 0 1];
Y = [0 1e-10 0 1];
K = convhull(X,Y)
```

return a warning.

```
Warning: qhull precision warning:
The initial hull is narrow (cosine of min. angle is 0.999999999999998).
A coplanar point may lead to a wide facet. Options 'QbB' (scale to unit box)
or 'Qbb' (scale last coordinate) may remove this warning. Use 'Pp' to skip
```
convhull

this warning.

To suppress this warning, use the option 'Pp'. The following command passes the option 'Pp', along with the default 'Qt', to convhull.

\[ K = \text{convhull}(X,Y,\{\text{'}Qt\text{'},\text{'}Pp\text{'}\}) \]

\[ K = \begin{array}{c}
2 \\
1 \\
4 \\
2 
\end{array} \]

Algorithm

convhull is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

See Also

convhulln, delaunay, plot, polyarea, voronoi

Reference


convhulln

Purpose
N-D 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 “Convex Hulls” in the MATLAB documentation.

- You cannot plot `convhulln` output for $n > 3$.

**Example**

The following example illustrates the options input for `convhulln`. The following commands

```matlab
X = [0 0; 0 1e-10; 0 0; 1 1];
K = convhulln(X)
```

return a warning.

Warning: qhull precision warning:
The initial hull is narrow
(cosine of min. angle is 0.9999999999999998).
A coplanar point may lead to a wide facet.
Options 'QbB' (scale to unit box) or 'Qbb'
(scaling last coordinate) may remove this warning.
Use 'Pp' to skip this warning.

To suppress the warning, use the option 'Pp'. The following command passes the option 'Pp', along with the default 'Qt', to `convhulln`.

```matlab
K = convhulln(X,{'Qt','Pp'})
```

```
K =

1  4
1  2
4  2
```

**Algorithm**

`convhulln` is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

**See Also**
`convhull`, `delaunayn`, `dsearchn`, `tsearchn`, `voronoin`
Reference

Purpose
N-D 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.
'velid' 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
Purpose
Copy file or directory

GUI Alternatives
As an alternative to the cd function, you can copy files and directories using the Current Directory browser.

Syntax
```matlab
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 path names for the directory or file. If source is a directory, destination also must be a directory. 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. To copy multiple files or directories, you can use one or more wildcard characters * after the last file separator in source. You cannot use a wildcard character in destination. You can rename when copying only when source is a single file. 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.

```matlab
[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 lasterror). Here, status is 1 for success and 0 for error. Only one output argument is required and the f input argument is optional.
**Remarks**

You can use a * (wildcard) in a path string. Current behavior of `copyfile` differs between the UNIX³ and Windows platforms when using the wildcard (*) or copying directories.

The timestamp given to the destination file is identical to that taken from the source file.

**Examples**

**Copying and Renaming a File**

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')
```

**Copying to Another Directory**

To copy `myfun.m` to the directory `d:/work/myfiles`, keeping the same filename, type

```matlab
    copyfile('myfun.m','d:/work/myfiles')
```

**Copying With 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

```matlab
    copyfile('myfiles/my*','../newprojects')
```

**Copying and Returning Status**

To copy all files and subdirectories in the current directory's `myfiles` directory to the directory `d:/work/myfiles`:

```matlab
    [s,mess,messid]=copyfile('myfiles','d:/work/myfiles')
    s =
    1
    mess =
```

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

The message returned indicates that `copyfile` was successful.

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.

**Copying to a Read-Only Directory**

Copy `myfile.m` from the current directory to `d:/work/restricted`, where `restricted` is a read-only directory:

```matlab
    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`
- “Managing Files and Working with the Current Directory”
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.

When programming a GUI, do not call copyobj or textwrap (which calls copyobj) inside a CreateFcn. The act of copying the uicontrol object fires the CreateFcn repeatedly, which raises a series of error messages after exceeding the root object’s RecursionLimit property.

Examples

Copy a surface to a new axes within a different figure.

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

“Graphics Object Identification” on page 1-100 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 \( R = \text{corrcoef}(X) \) is related to the covariance matrix \( C = \text{cov}(X) \) by

\[
R(i, j) = \frac{C(i, j)}{\sqrt{C(i, i)C(j, j)}}
\]

corrcoef(X) is the zeroth lag of the normalized covariance function, that is, the zeroth lag of \( x\text{cov}(x, 'coeff') \) packed into a square array.

R = corrcoef(x,y) where x and y are column vectors is the same as corrcoef([x y]). If x and y are not column vectors, corrcoef converts them to column vectors. For example, in this case \( R=\text{corrcoef}(x,y) \) is equivalent to \( R=\text{corrcoef}([x(:) \ y(:)]) \).

[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.
'alpha'  A number between 0 and 1 to specify a confidence level of 100*(1 - alpha)%. Default is 0.05 for 95% confidence intervals.

'rows'  Either 'all' (default) to use all rows, 'complete' to use rows with no NaN values, or 'pairwise' to compute $R(i,j)$ using rows with no NaN values in either column $i$ or $j$.

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 $0.5\times\log((1+R)/(1-R))$, 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.
```

```
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
   3   4

See Also

cov, mean, median, std, var
xcorr, xcov in the Signal Processing Toolbox
Purpose

Cosine of 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 \(-\pi \leq x \leq \pi\).

\[
\begin{align*}
x &= -\pi:0.01:\pi; \\
\text{plot}(x,\cos(x)), \text{ grid on}
\end{align*}
\]

The expression \( \cos(\pi/2) \) is not exactly zero but a value the size of the floating-point accuracy, \( \epsilon_p \), because \( \pi \) is only a floating-point approximation to the exact value of \( \pi \).
The cosine can be defined as

\[
\cos(x + iy) = \cos(x)\cosh(y) - i\sin(x)\sinh(y)
\]

\[
\cos(z) = \frac{e^{iz} + e^{-iz}}{2}
\]

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

**See Also** cosd, cOSH, acos, acosd, acosh
Purpose  
Cosine of 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 \( n \), \( \cosd(n*90) \) is exactly zero, whereas \( \cos(n*pi/2) \) reflects the accuracy of the floating point value of \( \pi \).

See Also  
\( \cos, \cosh, \acos, \acosd, \acosh \)
cosh

**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 $-5 \leq x \leq 5$.

```matlab
x = -5:0.01:5;
plot(x,cosh(x)), grid on
```

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

$$\cosh(z) = \frac{e^z + e^{-z}}{2}$$
**Algorithm**
cosh uses FDLIBM, which was developed at SunSoft, a Sun Microsystems business, by Kwok C. Ng, and others. For information about FDLIBM, see http://www.netlib.org.

**See Also**
acos, acosh, cos
Purpose

Cotangent of 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 $0 < x < \pi$.

```plaintext
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.

**See Also**
cotd, coth, acot, acotd, acoth
**Purpose**
Cotangent of 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 \( n \), \( \text{cotd}(n*180) \) is infinite, whereas \( \cot(n*pi) \) is large but finite, reflecting the accuracy of the floating point value of \( \pi \).

**See Also**
cot, coth, acot, acotd, acoth
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 \(0 < x < \pi\).

```matlab
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,coth(x1),x2,coth(x2)), grid on
```

Definition
The hyperbolic cotangent can be defined as
coth

\[ \text{coth}(z) = \frac{1}{\tanh(z)} \]

**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.

**See Also**

acot, acoth, cot
Purpose
Covariance matrix

Syntax
\text{cov}(x) \\
\text{cov}(x) \text{ or } \text{cov}(x,y) \\
\text{cov}(x,1) \text{ or } \text{cov}(x,y,1)

Description
\text{cov}(x)$, if $X$ is a vector, returns the variance. For matrices, where each row is an observation, and each column is a variable, $\text{cov}(X)$ is the covariance matrix. \text{diag}(\text{cov}(X)) is a vector of variances for each column, and \text{sqrt(diag(cov(X)))} is a vector of standard deviations. \text{cov}(X,Y)$, where $X$ and $Y$ are matrices with the same number of elements, is equivalent to $\text{cov}([X(:) \quad Y(:)])$.

$\text{cov}(x)$ or $\text{cov}(x,y)$ normalizes by $N-1$, if $N>1$, where $N$ is the number of observations. This makes $\text{cov}(X)$ the best unbiased estimate of the covariance matrix if the observations are from a normal distribution. For $N=1$, $\text{cov}$ normalizes by $N$.

$\text{cov}(x,1)$ or $\text{cov}(x,y,1)$ normalizes by $N$ and produces the second moment matrix of the observations about their mean. $\text{cov}(X,Y,0)$ is the same as $\text{cov}(X,Y)$ and $\text{cov}(X,0)$ is the same as $\text{cov}(X)$.

Remarks
$\text{cov}$ removes the mean from each column before calculating the result.

The \textit{covariance} function is defined as

$$
cov(x_1,x_2) = \mathbb{E}[(x_1 - \mu_1)(x_2 - \mu_2)]
$$

where $\mathbb{E}$ is the mathematical expectation and $\mu_i = \mathbb{E}x_i$.

Examples
Consider $A = \begin{bmatrix} -1 & 1 & 2 \\ -2 & 3 & 1 \\ 4 & 0 & 3 \end{bmatrix}$. To obtain a vector of variances for each column of $A$:

\begin{align*}
v &= \text{diag}(\text{cov}(A))' \\
v &= 10.3333 \quad 2.3333 \quad 1.0000
\end{align*}

Compare vector $v$ with covariance matrix $C$:
cov

C =
  10.3333  -4.1667   3.0000
 -4.1667   2.3333  -1.5000
  3.0000  -1.5000   1.0000

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, median, std, var
xcorr, xcov in the Signal Processing Toolbox
**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 $100\times\text{eps}$ relative to `abs(A(i))` determines which numbers are real and which elements are paired complex conjugates.

If `A` is a vector, `cplxpair(A)` returns `A` with complex conjugate pairs grouped together.

If `A` is a matrix, `cplxpair(A)` returns `A` with its columns sorted and complex conjugates paired.

If `A` is a multidimensional array, `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 `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, `cplxpair` generates the error message

```
Complex numbers can't be paired.
```
cputime

Purpose

Elapsed CPU time

Syntax

cputime

Description

cputime returns the total CPU time (in seconds) used by your MATLAB application from the time it was started. This number can overflow the internal representation and wrap around.

Remarks

Although it is possible to measure performance using the cputime function, it is recommended that you use the tic and toc functions for this purpose exclusively. See Using tic and toc Versus the cputime Function in the MATLAB Programming Fundamentals documentation for more information.

Examples

The following code returns the CPU time used to run surf(peaks(40)).

```matlab
  t = cputime; surf(peaks(40)); e = cputime-t
  e =
    0.4667
```

See Also

clock, etime, tic, toc
Purpose: Create random number streams

Class: @RandStream

Syntax:

\[
[s_1, s_2, \ldots] = \text{RandStream.create('gentype','NumStreams',n)} \\
\]

\[
s = \text{RandStream.create('gentype')} \\
\]

\[
[\ldots] = \text{RandStream.create(..., 'PARAM1',val1, 'PARAM2',val2, \ldots)} \\
\]

Description:

\[
[s_1, s_2, \ldots] = \text{RandStream.create('gentype','NumStreams',n)} \\
\]
creates \( n \) random number streams that use the uniform pseudorandom number generator algorithm specified by \( \text{gentype} \). The streams are independent in a pseudorandom sense. The streams are not necessarily independent from streams created at other times. \text{RandStream.list} returns all possible values for \( \text{gentype} \).

\textbf{Note} Multiple streams are not supported by all generator types. The multiplicative lagged Fibonacci generator (\text{mlfg6331_64}) and the combined multiple recursive generator (\text{mrg32k3a}) need to be active to use multiple stream creation.

\[
s = \text{RandStream.create('gentype')} \]
creates a single random stream.

\[
[\ldots] = \text{RandStream.create(..., 'PARAM1',val1, 'PARAM2',val2, \ldots)} \]
allows you to specify optional parameter name or value pairs to control creation of the stream(s). The parameters are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{NumStreams}</td>
<td>Total number of streams of this type that will be created across sessions or labs. Default is 1.</td>
</tr>
<tr>
<td>\text{StreamIndices}</td>
<td>Stream indices that should be created in this call. Default is 1:N, where N is the value given with the 'NumStreams' parameter.</td>
</tr>
</tbody>
</table>
create (RandStream)

<table>
<thead>
<tr>
<th>Seed</th>
<th>Nonnegative scalar integer with which to initialize all streams. Default is 0. Seeds must be an integer between 0 and .</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandnAlg</td>
<td>Algorithm that will be used by randn(S, ...) to generate normal pseudorandom values. Options are 'Ziggurat', 'Polar', or 'Inversion'.</td>
</tr>
<tr>
<td>CellOutput</td>
<td>Logical flag indicating whether or not to return the stream objects as elements of a cell array. Default is false.</td>
</tr>
</tbody>
</table>

**Examples**

Create three independent streams.

```matlab
[s1,s2,s3] = RandStream.create('mrg32k3a','NumStreams',3);
r1 = rand(s1,100000,1); r2 = rand(s2,100000,1); r3 = rand(s3,100000,1);
corrcoef([r1,r2,r3])
```

Create one stream from a set of three independent streams and designate it as the default stream.

```matlab
s2 = RandStream.create('mrg32k3a','NumStreams',3,'StreamIndices',2);
RandStream.setDefaultStream(s2);
```

**See Also**

@RandStream, RandStream (RandStream), list (RandStream), getDefaultStream (RandStream), setDefaultStream (RandStream), rand (RandStream), randi (RandStream), randn (RandStream).
**createClassFromWsdl**

**Purpose**
Create MATLAB class based on WSDL document

**Syntax**
classFromWsdl(source)

**Description**
classFromWsdl(source) creates a MATLAB class based on source, a Web Services Description Language (WSDL) document. The source argument is a string that specifies a URL, full path, or relative path to a WSDL document located on a server. The WSDL document defines the operations, arguments, and transactions for a service. MATLAB is a client, using the WSDL document to access and run APIs on the server via Web services. classFromWsdl(source) returns the name of the class it created, servicename.

Based on the WSDL document, the classFromWsdl function creates a new class directory, @servicename, in the current directory. The class directory contains a method M-file for each Web service operation you can use. In addition, classFromWsdl creates two default M-files: the class's display method (display.m) and its constructor (servicename.m). MATLAB uses the methods to exchange data with the server via SOAP messages.

For example, if myWebService offers two operations (method1 and method2), the classFromWsdl function creates the following:

- @myWebService folder in the current directory.
- method1.m — M-file for method1, in the @myWebService directory.
- method2.m — M-file for method2, in the @myWebService directory.
- display.m — Default M-file for display method, in the @myWebService directory. Running display(myWebService) shows the endpoint and WSDL document location for the Web service used to create the class.
- myWebService.m — Default M-file that is the constructor method for the myWebService class, in the @myWebService directory.
createClassFromWsdl

Examples

Given the WSDL document for the test scores Web service, http://examplestandardtests.com/scoreswebservicewsdl, the following statements create the class, show the methods, and then retrieve a student name.

Note The example does not use an actual WSDL document; therefore, you cannot run it. The example only illustrates how to use the function.

```matlab
createClassFromWsdl('http://examplestandardtests.com/scoreswebservicewsdl');
obj = TestScoreWebService;
methods(obj)
students = StudentNames(obj);
students.StudentInfo(1)
```

MATLAB returns

```matlab
StudentNameLast: 'Benjamin'
StudentNameFirst: 'Ali'
```

See Also

callSoapService, createSoapMessage, parseSoapResponse, xmlread

“Using Web Services with MATLAB” in the MATLAB External Interfaces documentation
createCopy (inputParser)

**Purpose**
Create copy of inputParser object

**Syntax**
```matlab
p.createCopy
createCopy(p)
```

**Description**
p.createCopy creates a copy of inputParser object p. Because the inputParser class uses handle semantics, a normal assignment statement does not create a copy.

createCopy(p) is functionally the same as the syntax above.

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

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

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

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

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

    Make a copy of object p, assigning it to variable x. Use the Parameters property of inputParser to list the arguments belonging to each object:

    disp(' ')
    disp 'The input parameters for object p are'
```
disp(p.Parameters')
x = p.createCopy;

disp(' ')
disp('The input parameters for the copy of object p are')
disp(x.Parameters')

Save the M-file using the **Save** option on the MATLAB **File** menu, and then run it:

```
publish_ip('ipscript.m', 'ppt', 'maxWidth', 500, 'MAXHeight', 300);
```

The input parameters for object p are

- 'format'
- 'maxHeight'
- 'maxWidth'
- 'outputDir'
- 'script'

The input parameters for the copy of object p are

- 'format'
- 'maxHeight'
- 'maxWidth'
- 'outputDir'
- 'script'

**See Also**

- inputParser
- addRequired(inputParser)
- addOptional(inputParser)
- addParamValue(inputParser)
- parse(inputParser)
Purpose
Create SOAP message to send to server

Syntax
message = createSoapMessage(namespace, method, values, names, types, style)

Description
message = createSoapMessage(namespace, method, values, names, types, style) creates a SOAP message based on the values you provide for the arguments. message is a Sun Java document object model (DOM). To send message to the Web service, use it with callSoapService.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
<td>Location of the Web service in the form of a valid Uniform Resource Identifier (URI).</td>
</tr>
<tr>
<td>method</td>
<td>Name of the Web service operation you want to run.</td>
</tr>
<tr>
<td>values</td>
<td>Cell array of input you need to provide for the method.</td>
</tr>
<tr>
<td>names</td>
<td>Cell array of parameters for method.</td>
</tr>
<tr>
<td>types</td>
<td>Cell array defining the XML data types for values. Specifying style is optional; when you do not include the argument, MATLAB uses unspecified.</td>
</tr>
<tr>
<td>style</td>
<td>Style for structuring the SOAP message, either 'document' or 'rpc'. Specifying style is optional; when you do not include the argument, MATLAB uses rpc. Use a style supported by the service you specified in namespace.</td>
</tr>
</tbody>
</table>

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

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

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

MATLAB returns:

```
author = Kate Alvin
```

where author is a char class (type).

**See Also**

callSoapService, createClassFromWsdl, parseSoapResponse, urlread, xmlread

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

```matlab
a = [1 2 3];
b = [4 5 6];
c = cross(a,b)

C =
    -3     6    -3

d = dot(a,b)

d =
    32
```

**See Also**
dot
**Purpose**
Cosecant of argument in radians

**Syntax**
\[ Y = \text{csc}(x) \]

**Description**
The `csc` function operates element-wise on arrays. The function’s domains and ranges include complex values. All angles are in radians.

\[ Y = \text{csc}(x) \]
returns the cosecant for each element of \( x \).

**Examples**
Graph the 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,csc(x1),x2,csc(x2)), grid on
```
**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 business, by Kwok C. Ng, and others. For information about FDLIBM, see [http://www.netlib.org](http://www.netlib.org).

**See Also**

cscd, csch, acsc, acscd, acsch
cscd

**Purpose**
Cosecant of 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**
csc, csch, acsc, acscd, acsch
Purpose

Hyperbolic cosecant

Syntax

\[ Y = \text{csch}(x) \]

Description

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

\[ Y = \text{csch}(x) \]

returns the hyperbolic cosecant for each element of \( x \).

Examples

Graph the hyperbolic cosecant over the domains \(-\pi < x < 0\) and \(0 < x < \pi\).

\[
x1 = -\pi + 0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,csch(x1),x2,csch(x2)), grid on
\]

Definition

The hyperbolic cosecant can be defined as
\[
\text{csch}(z) = \frac{1}{\sinh(z)}
\]

**Algorithm**

\text{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**

\text{acsc}, \text{acsch}, \text{csc}
**Purpose**
Read 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 `filename` input is a string enclosed in single quotes. 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 non-space delimiter, 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;real&gt;–&lt;imag&gt;i</td>
<td>5.7-3.1i</td>
</tr>
<tr>
<td>–&lt;imag&gt;i</td>
<td>-7j</td>
</tr>
</tbody>
</table>

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:

- 02, 04, 06, 08, 10, 12
- 03, 06, 09, 12, 15, 18
- 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')
```

```plaintext
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)
```

```plaintext
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])
```

```plaintext
m =
```
csvread

5 10 15 20
7 14 21 28

See Also  csvwrite, dlmread, textscan, wk1read, file formats, importdata, uiimport
**Purpose**
Write 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. The filename input is a string enclosed in single quotes.
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.

**Remarks**
csvwrite terminates each line with a line feed character and no carriage return.
csvwrite writes a maximum of five significant digits. If you need greater precision, use dlmwrite with a precision argument.
csvwrite does not accept cell arrays for the input matrix M. To export a cell array that contains only numeric data, use cell2mat to convert the cell array to a numeric matrix before calling csvwrite. To export cell arrays with mixed alphabetic and numeric data, where each cell contains a single element, you can create an Excel spreadsheet (if your system has Excel installed) using xlswrite. For all other cases, you must use low-level export functions to write your data.

**Examples**
The following example creates a comma-separated value file from the matrix m.

```matlab
m = [3 6 9 12 15; 5 10 15 20 25; ...
   7 14 21 28 35; 11 22 33 44 55];

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.

```matlab
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, xlswrite, file formats, importdata, uiimport
Purpose

Transpose timeseries object

Syntax

ts1 = ctranspose(ts)

Description

ts1 = ctranspose(ts) returns a new timeseries object ts1 with IsTimeFirst value set to the opposite of what it is for ts. For example, if ts has the first data dimension aligned with the time vector, ts1 has the last data dimension aligned with the time vector as a result of this operation.

Remarks

The ctranspose function that is overloaded for timeseries objects does not transpose the data. Instead, this function changes whether the first or the last dimension of the data is aligned with the time vector.

Note

To transpose the data, you must transpose the Data property of the timeseries object. For example, you can use the syntax ctranspose(ts.Data) or (ts.Data)'. Data must be a 2-D array.

Consider a timeseries object with 10 samples with the property IsTimeFirst = True. When you transpose this object, the data size is changed from 10-by-1 to 1-by-1-by-10. Note that the first dimension of the Data property is shown explicitly.

The following table summarizes the size for Data property of the timeseries object (up to three dimensions) before and after transposing.

<table>
<thead>
<tr>
<th>Size of Original Data</th>
<th>Size of Transposed Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-by-1</td>
<td>1-by-1-by-N</td>
</tr>
<tr>
<td>N-by-M</td>
<td>M-by-1-by-N</td>
</tr>
<tr>
<td>N-by-M-by-L</td>
<td>M-by-L-by-N</td>
</tr>
</tbody>
</table>
Examples
Suppose that a timeseries object ts has ts.data size 10-by-3-by-2 and its time vector has a length of 10. The IsTimeFirst property of ts is set to true, which means that the first dimension of the data is aligned with the time vector. ctranspose(ts) modifies ts such that the last dimension of the data is now aligned with the time vector. This permutes the data such that the size of ts.Data becomes 3-by-2-by-10.

See Also
transpose (timeseries), tsprops
cumprod

**Purpose**
Cumulative product

**Syntax**

\[
\begin{align*}
B &= \text{cumprod}(A) \\
B &= \text{cumprod}(A,\text{dim})
\end{align*}
\]

**Description**

\(B = \text{cumprod}(A)\) returns the cumulative product along different dimensions of an array.

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

If \(A\) is a matrix, \(\text{cumprod}(A)\) returns a matrix the same size as \(A\) containing the cumulative products for each column of \(A\).

If \(A\) is a multidimensional array, \(\text{cumprod}(A)\) works on the first nonsingleton dimension.

\(B = \text{cumprod}(A,\text{dim})\) returns the cumulative product of the elements along the dimension of \(A\) specified by scalar \(\text{dim}\). For example, \(\text{cumprod}(A,1)\) increments the column index, thus working along the columns of \(A\). Thus, \(\text{cumprod}(A,1)\) and \(\text{cumprod}(A)\) will return the same thing. To increment the row index, use \(\text{cumprod}(A,2)\).

**Examples**

\[
\begin{align*}
\text{cumprod}(1:5) \\
\text{ans} &= \\
&= 1 \ 2 \ 6 \ 24 \ 120
\end{align*}
\]

\[
\begin{align*}
A &= [1 \ 2 \ 3; \ 4 \ 5 \ 6]; \\
\text{cumprod}(A,1) \\
\text{ans} &= \\
&= 1 \ 2 \ 3 \\
&= 4 \ 10 \ 18
\end{align*}
\]

\[
\begin{align*}
\text{cumprod}(A,2) \\
\text{ans} &= \\
&= 1 \ 2 \ 6 \\
&= 4 \ 20 \ 120
\end{align*}
\]
See Also  
cumsum, prod, sum
**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 along the first dimension (the columns); cumsum(A,2) works along the second dimension (the rows).

**Examples**
- cumsum(1:5)
  ans =
  [ 1  3  6  10  15]

- A = [1 2 3; 4 5 6];

  cumsum(A,1)
  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**

- $Z = \text{cumtrapz}(Y)$
- $Z = \text{cumtrapz}(X,Y)$
- $Z = \text{cumtrapz}(X,Y,\text{dim})$ or $\text{cumtrapz}(Y,\text{dim})$

**Description**
$Z = \text{cumtrapz}(Y)$ 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. Input $Y$ can be complex.

For vectors, $\text{cumtrapz}(Y)$ is a vector containing the cumulative integral of $Y$.

For matrices, $\text{cumtrapz}(Y)$ is a matrix the same size as $Y$ with the cumulative integral over each column.

For multidimensional arrays, $\text{cumtrapz}(Y)$ works across the first nonsingleton dimension.

$Z = \text{cumtrapz}(X,Y)$ 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 $\text{length}(X)$. $\text{cumtrapz}$ operates across this dimension. Inputs $X$ and $Y$ can be complex.

If $X$ is a column vector and $Y$ an array whose first nonsingleton dimension is $\text{length}(X)$, $\text{cumtrapz}(X,Y)$ operates across this dimension.

$Z = \text{cumtrapz}(X,Y,\text{dim})$ or $\text{cumtrapz}(Y,\text{dim})$ integrates across the dimension of $Y$ specified by scalar $\text{dim}$. The length of $X$ must be the same as $\text{size}(Y,\text{dim})$.

**Example 1**

\[
Y = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix};
\]

\[
\text{cumtrapz}(Y,1)
\]
\[
\text{ans} =
\begin{bmatrix}
0 & 0 & 0
\end{bmatrix}
\]
1.5000  2.5000  3.5000

cumtrapz(Y,2)
ans =
  0   0.5000   2.0000
  0   3.5000   8.0000

Example 2
This example uses two complex inputs:

z = exp(1i*pi*(0:100)/100);

ct = cumtrapz(z,1./z);
ct(end)
ans =
  0.0000 + 3.1411i

See Also  cumsum, trapz
Purpose

Compute curl and angular velocity of vector field

Syntax

\[ \text{[curlx,curly,curlz,cav]} = \text{curl}(X,Y,Z,U,V,W) \]
\[ \text{[curlx,curly,curlz,cav]} = \text{curl}(U,V,W) \]
\[ \text{[curlz,cav]} = \text{curl}(X,Y,U,V) \]
\[ \text{[curlz,cav]} = \text{curl}(U,V) \]
\[ \text{[curlx,curly,curlz]} = \text{curl}(\ldots), \text{[curlx,curly]} = \text{curl}(\ldots) \]
\[ \text{cav} = \text{curl}(\ldots) \]

Description

\[ \text{[curlx,curly,curlz,cav]} = \text{curl}(X,Y,Z,U,V,W) \] computes the curl \((\text{curlx, curly, curlz})\) and angular velocity \((\text{cav})\) 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 \text{meshgrid}).

\[ \text{[curlx,curly,curlz,cav]} = \text{curl}(U,V,W) \] assumes \(X, Y,\) and \(Z\) are determined by the expression

\[ [X \ Y \ Z] = \text{meshgrid}(1:n,1:m,1:p) \]

where \([m,n,p] = \text{size}(U)\).

\[ \text{[curlz,cav]} = \text{curl}(X,Y,U,V) \] 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 \text{meshgrid}).

\[ \text{[curlz,cav]} = \text{curl}(U,V) \] assumes \(X\) and \(Y\) are determined by the expression

\[ [X \ Y] = \text{meshgrid}(1:n,1:m) \]

where \([m,n] = \text{size}(U)\).

\[ \text{[curlx,curly,curlz]} = \text{curl}(\ldots), \text{[curlx,curly]} = \text{curl}(\ldots) \]
returns only the curl.

\(\text{cav} = \text{curl}(\ldots)\) 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.

```matlab
load wind
cav = curl(x,y,z,u,v,w);
slice(x,y,z,cav,[90 134],[59],[0]);
shading interp
daspect([1 1 1]); axis tight
colormap hot(16)
camlight
```

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" on page 1-108 for related functions

"Example – Displaying Curl with Stream Ribbons" for another example
customverctrl

**Purpose**
Allow custom source control system (UNIX platforms)

**Syntax**
customerverctrl

**Description**
customerverctrl function is for customers who want to integrate a source control system that is not supported for use with MATLAB software. When using this function, conform to the structure of one of the supported version control systems, for example, RCS. For examples, see the files clearcase.m, cvs.m, pvcs.m, and rcs.m in matlabroot\toolbox\matlab\verctrl.

**See Also**
checkin, checkout, cmopts, undocheckout

For Microsoft Windows platforms, use verctrl.
cylinder

Purpose

Generate cylinder

Syntax

[X,Y,Z] = cylinder
[X,Y,Z] = cylinder(r)
[X,Y,Z] = cylinder(r,n)
cylinder(axes_handle,...)
cylinder(...)

Description

cylinder generates x-, y-, and z-coordinates of a unit cylinder. You can draw the cylindrical object using surf or mesh, or draw it immediately by not providing output arguments.

[X,Y,Z] = 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] = cylinder(r) returns the x-, y-, and z-coordinates of a cylinder using r to define a profile curve. 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] = 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.

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

cylinder(...), with no output arguments, plots the cylinder using surf.
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.

```matlab
cylinder
axis square
h = findobj('Type','surface');
set(h,'CData',rand(size(get(h,'CData')))))
```

Generate a cylinder defined by the profile function $2+\sin(t)$.

```matlab
t = 0:pi/10:2*pi;
```
cylinder

\[ [X,Y,Z] = \text{cylinder}(2+\cos(t)); \]
\[ \text{surf}(X,Y,Z) \]
\[ \text{axis square} \]

**See Also**

sphere, surf

“Polygons and Surfaces” on page 1-97 for related functions
Purpose
Read Data Acquisition Toolbox (.daq) file

Syntax
data = daqread('filename')
[data, time] = daqread(...)  
[data, time, abstime] = daqread(...)  
[data, time, abstime, events] = daqread(...)  
[data, time, abstime, events, daqinfo] = daqread(...)  
data = daqread(...,'Param1', Val1,...)  
daqinfo = daqread('filename','info')

Description
data = daqread('filename') reads all the data from the Data Acquisition Toolbox (.daq) file specified by filename. daqread returns data, an \( m \)-by-\( n \) data matrix, where \( m \) is the number of samples and \( n \) is the number of channels. If data includes data from multiple triggers, the data from each trigger is separated by a NaN. If you set the OutputFormat property to tscollection, daqread returns a time series collection object. See below for more information.

[data, time] = daqread(...) returns time/value pairs. time is an \( m \)-by-1 vector, the same length as data, that contains the relative time for each sample. Relative time is measured with respect to the first trigger that occurs.

[data, time, abstime] = daqread(...) returns the absolute time of the first trigger. abstime is returned as a clock vector.

[data, time, abstime, events] = daqread(...) returns a log of events. events is a structure containing event information. If you specify either the Samples, Time, or Triggers parameters (see below), the events structure contains only the specified events.

[data, time, abstime, events, daqinfo] = daqread(...) returns a structure, daqinfo, that contains two fields: ObjInfo and HwInfo. ObjInfo is a structure containing property name/property value pairs and HwInfo is a structure containing hardware information. The entire event log is returned to daqinfo.ObjInfo.EventLog.
data = daqread(...,'Param1', Val1,...) specifies the amount of data returned and the format of the data, using the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samples</td>
<td>Specify the sample range.</td>
</tr>
<tr>
<td>Time</td>
<td>Specify the relative time range.</td>
</tr>
<tr>
<td>Triggers</td>
<td>Specify the trigger range.</td>
</tr>
<tr>
<td>Channels</td>
<td>Specify the channel range. Channel names can be specified as a cell array.</td>
</tr>
<tr>
<td>DataFormat</td>
<td>Specify the data format as doubles (default) or native.</td>
</tr>
<tr>
<td>TimeFormat</td>
<td>Specify the time format as vector (default) or matrix.</td>
</tr>
<tr>
<td>OutputFormat</td>
<td>Specify the output format as matrix (the default) or tscollection. When you specify tscollection, daqread only returns data.</td>
</tr>
</tbody>
</table>

The Samples, Time, and Triggers properties are mutually exclusive; that is, either Samples, Triggers or Time can be defined at once.

daqinfo = daqread('filename','info') returns metadata from the file in the daqinfo structure, without incurring the overhead of reading the data from the file as well. The daqinfo structure contains two fields:

daqinfo.ObjInfo
   a structure containing parameter/value pairs for the data acquisition object used to create the file, filename. Note: The UserData property value is not restored.

daqinfo.HwInfo
   a structure containing hardware information. The entire event log is returned to daqinfo.ObjInfo.EventLog.
Remarks  More About .daq Files

- The format used by `daqread` to return data, relative time, absolute time, and event information is identical to the format used by the `getdata` function that is part of Data Acquisition Toolbox. For more information, see the Data Acquisition Toolbox documentation.

- If data from multiple triggers is read, then the size of the resulting data array is increased by the number of triggers issued because each trigger is separated by a NaN.

- `ObjInfo.EventLog` always contains the entire event log regardless of the value specified by `Samples`, `Time`, or `Triggers`.

- The `UserData` property value is not restored when you return device object (`ObjInfo`) information.

- When reading a `.daq` file, the `daqread` function does not return property values that were specified as a cell array.

- Data Acquisition Toolbox (.daq) files are created by specifying a value for the `LogFileName` property (or accepting the default value), and configuring the `LoggingMode` property to Disk or Disk&Memory.

More About Time Series Collection Object Returned

When `OutputFormat` is set to `tscollection`, `daqread` returns a time series collection object. This times series collection object contains an absolute time series object for each channel in the file. The following describes how `daqread` sets some of the properties of the times series collection object and the time series objects.

- The `time` property of the time series collection object is set to the value of the `InitialTriggerTime` property specified in the file.

- The `name` property of each time series object is set to the value of the `Name` property of a channel in the file. If this name cannot be used as a time series object name, `daqread` sets the name to `'Channel'` with the `HwChannel` property of the channel appended.
The value of the Units property of the time series object depends on the value of the DataFormat parameter. If the DataFormat parameter is set to 'double', daqread sets the DataInfo property of each time series object in the collection to the value of the Units property of the corresponding channel in the file. If the DataFormat parameter is set to 'native', daqread sets the Units property to 'native'. See the Data Acquisition Toolbox documentation for more information on these properties.

Each time series object will have tsdata.event objects attached corresponding to the log of events associated with the channel.

If daqread returns data from multiple triggers, the data from each trigger is separated by a NaN in the time series data. This increases the length of data and time vectors in the time series object by the number of triggers.

Examples

Use Data Acquisition Toolbox to acquire data. The analog input object, ai, acquires one second of data for four channels, and saves the data to the output file data.daq.

```matlab
ai = analoginput('nidaq','Dev1');
chans = addchannel(ai,0:3);
set(ai,'SampleRate',1000)
ActualRate = get(ai,'SampleRate');
set(ai,'SamplesPerTrigger',ActualRate)
set(ai,'LoggingMode','Disk&Memory')
set(ai,'LogFileName','data.daq')
start(ai)
```

After the data has been collected and saved to a disk file, you can retrieve the data and other acquisition-related information using daqread. To read all the sample-time pairs from data.daq:

```matlab
[data,time] = daqread('data.daq');
```

To read samples 500 to 1000 for all channels from data.daq:
data = daqread('data.daq','Samples',[500 1000]);

To read only samples 1000 to 2000 of channel indices 2, 4 and 7 in native format from the file, data.daq:

data = daqread('data.daq', 'Samples', [1000 2000],...
     'Channels', [2 4 7], 'DataFormat', 'native');

To read only the data which represents the first and second triggers on all channels from the file, data.daq:

[data, time] = daqread('data.daq', 'Triggers', [1 2]);

To obtain the channel property information from data.daq:

daqinfo = daqread('data.daq','info');
chaninfo = daqinfo.ObjInfo.Channel;

To obtain a list of event types and event data contained by data.daq:

daqinfo = daqread('data.daq','info');
events = daqinfo.ObjInfo.EventLog;
event_type = {events.Type};
event_data = {events.Data};

To read all the data from the file data.daq and return it as a time series collection object:

data = daqread('data.daq','OutputFormat','tscollection');

See Also

Functions

timeseries, tscollection

For more information about using this function, see the Data Acquisition Toolbox documentation.
**Purpose**
Set or query 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, the data aspect ratio adjusts 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 fit the window). 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)
```

![3D surface plot](image)
Querying the data aspect ratio shows how the surface is drawn.

```
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

“Aspect Ratio and Axis Limits” on page 1-107 for related functions

“Understanding Axes Aspect Ratio” for more information
**Purpose**
Enable or disable interactive data cursor mode

**GUI Alternatives**
Use the Data Cursor tool to label x, y, and z values on graphs and surfaces. For details, see Data Cursor — Displaying Data Values Interactively in the MATLAB Graphics documentation.

**Syntax**
datacursormode on
datacursormode off
datacursormode
datacursormode(figure_handle,...)
dcm_obj = datacursormode(figure_handle)

**Description**
ddatacursormode 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” on page 2-820.

A *data cursor* is a small black square with a white border that you interactively position on a graph in data cursor mode. When you do this, a *datatip* appears. Datatips are small text boxes or windows that float within an axes that display data values at data cursor locations. The default style is a text box. Datatips list x-, y- and (where appropriate) z-values for one data point at a time. See “Examples” on page 2-822 for an illustration of these two styles.

Most types of graphs support data cursor mode, but several do not (pareto, for example). Polar plots support datatips, but display Cartesian rather than polar coordinates on them. Histograms created with hist display specialized datatips that itemize the observation counts, lower and upper limits and center point for histogram bins.
The data cursor mode object has properties that enable you to control 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**

**Enable**

- **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.

- **window** displays cursor information in a floating window within the figure.

**Figure**

- **handle**

Handle of the figure associated with the data cursor mode object.

**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 passed to the function when it 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 Object containing event data structure
% output_txt Data cursor text (string or cell array of string)
```

`event_obj` is an object that contains a struct having the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Handle of the object the data cursor is referencing (the object on which the user clicked)</td>
</tr>
<tr>
<td>Position</td>
<td>An array specifying the x, y, (and z for 3-D graphs) coordinates of the cursor</td>
</tr>
</tbody>
</table>

You can query these properties within your function. For example,

```matlab
pos = get(event_obj,'Position');
```

returns the coordinates of the cursor. Another way of accessing that data is to obtain the struct and query its `Position` field.

```matlab
eventdata = get(event_obj);
pos = eventdata.Position;
```

See Function Handles for more information on creating a function handle.

See “Change Data Cursor Text” on page 2-826 for an example.

**Querying Data Cursor Mode**

The `getCursorInfo` function queries the data cursor mode object (`dcm_obj` in the above syntax) to obtain information about the data cursor. For example,
info_struct = getCursorInfo(dcm_obj);

returns a vector of structures, one for each data cursor on the graph. Each structure has the following fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>The handle of the graphics object containing the data point</td>
</tr>
<tr>
<td>Position</td>
<td>An array specifying the x, y, (and z) coordinates of the cursor</td>
</tr>
</tbody>
</table>

Line and lineseries objects have an additional field.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataIndex</td>
<td>A scalar index into the data arrays that correspond to the nearest data point. The value is the same for each array.</td>
</tr>
</tbody>
</table>

**Note** Do not change figure callbacks within an interactive mode. While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure’s callbacks and the operation will not succeed. The one exception to this rule is the figure WindowButtonMotionFcn callback, which can be changed from within a mode. Therefore, if you are creating a GUI that updates a figure’s callbacks, the GUI should some keep track of which interactive mode is active, if any, before attempting to do this.

**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
```

Selecting a point on the surface opens a datatip displaying its x-, y-, and z-coordinates.
You change the datatip display style to be a window instead of a text box using the Tools > Options > Display cursor in window, or use the context menu Display Style > Window inside figure to view the datatip in a floating window that you can move around inside the axes.
You can position multiple text box datatips on the same graph, the window style of datatip displays only one value at a time. For more information on interacting with data cursors, including point selection options and exporting datatips to the workspace, see “Data Cursor — Displaying Data Values Interactively” in the MATLAB Graphics documentation.

**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','Enable','on')

% 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:”

Note  Save the following functions in you current directory or any writable directory on the MATLAB path before running them. As they are functions, you cannot highlight them and then evaluate the selection to make them work.

% After saving both these functions as M-files, % execute the following one first by typing % >> doc_datacursormode

function doc_datacursormode
    fig = figure;
    a = -16; t = 0:60;
    plot(t,sin(a*t))
    dcm_obj = datacursormode(fig);
    set(dcm_obj,'UpdateFcn',@myupdatefcn)

% Now click on line to select data point to use the update function

function txt = myupdatefcn(empt,event_obj)
    pos = get(event_obj,'Position');
    txt = {{'Time: ',num2str(pos(1))},... 
           {'Amplitude: ',num2str(pos(2))}};

See Also  brush, pan, zoom

“Example — Visually Exploring Demographic Statistics” for a further example of a data cursor update function
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:

\[
A = \text{rand}(5); \\
\text{datatipinfo}(A) \\
A: 5x5 double = \\
\begin{array}{ccccc}
0.4445 & 0.3567 & 0.7458 & 0.0767 & 0.4400 \\
0.7962 & 0.6575 & 0.3918 & 0.8289 & 0.9746 \\
0.5641 & 0.9808 & 0.0265 & 0.4838 & 0.6722 \\
0.9099 & 0.9653 & 0.2508 & 0.4859 & 0.4054 \\
0.2857 & 0.5198 & 0.7383 & 0.9301 & 0.9604
\end{array}
\]

Get datatip information for a 50-by-50 matrix. For this larger matrix, datatipinfo displays just the size and data type:

\[
A = \text{rand}(50); \\
\text{datatipinfo}(A) \\
A: 50x50 double
\]

Also for multidimensional matrices, datatipinfo displays just the size and data type:

\[
A = \text{rand}(5); \\
A(:,:,2) = A(:,:,1); \\
\text{datatipinfo}(A) \\
A: 5x5x2 double
\]

See Also

data
Purpose
Current date string

Syntax
str = date

Description
str = date returns a string containing the date in dd-mmm-yyyy format.

See Also
clock, datestr, datenum, now
Purpose

Convert date and time to serial date number

Syntax

N = datenum(V)
N = datenum(S, F)
N = datenum(S, F, P)
N = datenum([S, P, F])
N = datenum(Y, M, D)
N = datenum(Y, M, D, H, MN, S)
N = datenum(S)
N = datenum(S, P)

Description

datenum is one of three conversion functions that enable you to express dates and times in any of three formats in your MATLAB application: a string (or date string), a vector of date and time components (or date vector), or as a numeric offset from a known date in time (or serial date number). Here is an example of a date and time expressed in the three MATLAB formats:

Date String:       '24-Oct-2003 12:45:07'
Date Vector:       [2003 10 24 12 45 07]
Serial Date Number: 7.3188e+005

A serial date number represents the whole and fractional number of days from a specific date and time, where datenum('Jan-1-0000 00:00:00') returns the number 1. (The year 0000 is merely a reference point and is not intended to be interpreted as a real year in time.)

N = datenum(V) converts one or more date vectors V to serial date numbers N. Input V can be an m-by-6 or m-by-3 matrix containing m full or partial date vectors respectively. A full date vector has six elements, specifying year, month, day, hour, minute, and second, in that order. A partial date vector has three elements, specifying year, month, and day, in that order. Each element of V must be a positive double-precision number. datenum returns a column vector of m date numbers, where m is the total number of date vectors in V.

N = datenum(S, F) converts one or more date strings S to serial date numbers N using format string F to interpret each date string. Input S
can be a one-dimensional character array or cell array of date strings. All date strings in S must have the same format, and that format must match one of the date string formats shown in the help for the datetick function. datenum returns a column vector of m date numbers, where m is the total number of date strings in S. MATLAB considers date string years that are specified with only two characters (e.g., '79') to fall within 100 years of the current year.

See the datetick reference page to find valid string values for F. These values are listed in Table 1 in the column labeled “Dateform String.” You can use any string from that column except for those that include the letter Q in the string (for example, 'QQ-YYYY'). Certain formats may not contain enough information to compute a date number. In these cases, hours, minutes, seconds, and milliseconds default to 0, the month defaults to January, the day to 1, and the year to the current year.

N = datenum(S, F, P) converts one or more date strings S to date numbers N using format F and pivot year P. The pivot year is used in interpreting date strings that have the year specified as two characters. It is the starting year of the 100-year range in which a two-character date string year resides. The default pivot year is the current year minus 50 years.

N = datenum([S, P, F]) is the same as the syntax shown above, except the order of the last two arguments are switched.

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) of type double. You can also specify the input arguments as a date vector, [Y M D].

For this and the following syntax, values outside the normal range of each array are automatically carried to the next unit. 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.
N = datenum(Y, M, D, H, MN, S) returns the serial date numbers for corresponding elements of the Y, M, D, H, MN, and S (year, month, day, hour, minute, and second) array values. datenum does not accept milliseconds in a separate input, but as a fractional part of the seconds (S) input. Inputs Y, M, D, H, MN, and S must be arrays of the same size (or any can be a scalar) of type double. You can also specify the input arguments as a date vector, [Y M D H MN S].

N = datenum(S) converts date string S into a serial date number. String S must be in one of the date formats 0, 1, 2, 6, 13, 14, 15, 16, or 23, as defined in the reference page for the datestr function. MATLAB considers date string years that are specified with only two characters (e.g., '79') to fall within 100 years of the current year. If the format of date string S is known, use the syntax N = datenum(S, F).

N = datenum(S, P) converts date string S, using pivot year P. If the format of date string S is known, use the syntax N = datenum(S, F, P).

**Note** The last two calling syntaxes are provided for backward compatibility and are significantly slower than the syntaxes that include a format argument F.

**Examples**

Convert a date string to a serial date number:

```matlab
n = datenum('19-May-2001', 'dd-mmm-yyyy')
n = 730990
```

Specifying year, month, and day, convert a date to a serial date number:

```matlab
n = datenum(2001, 12, 19)
n = 731204
```
Convert a date vector to a serial date number:

```matlab
format bank
datenum('March 28, 2005 3:37:07.033 PM')
ans =
    732399.65
```

Convert a date string to a serial date number using the default pivot year:

```matlab
n = datenum('12-jun-17', 'dd-mmm-yy')
n =
    736858
```

Convert the same date string to a serial date number using 1400 as the pivot year:

```matlab
n = datenum('12-jun-17', 'dd-mmm-yy', 1400)
n =
    517712
```

Specify format 'dd.mm.yyyy' to be used in interpreting a nonstandard date string:

```matlab
n = datenum('19.05.2000', 'dd.mm.yyyy')
n =
    730625
```

See Also

datestr, datevec, date, clock, now, datetick
Purpose
Convert date and time to string format

Syntax
S = datestr(V)
S = datestr(N)
S = datestr(D, F)
S = datestr(S1, F, P)
S = datestr(..., 'local')

Description
datestr is one of three conversion functions that enable you to express dates and times in any of three formats in your MATLAB application: a string (or date string), a vector of date and time components (or date vector), or as a numeric offset from a known date in time (or serial date number). Here is an example of a date and time expressed in the three MATLAB formats:

Date String: '24-Oct-2003 12:45:07'
Date Vector: [2003 10 24 12 45 07]
Serial Date Number: 7.3188e+005

A serial date number represents the whole and fractional number of days from 1-Jan-0000 to a specific date. The year 0000 is merely a reference point and is not intended to be interpreted as a real year in time.

S = datestr(V) converts one or more date vectors V to date strings S. Input V must be an m-by-6 matrix containing m full (six-element) date vectors. Each element of V must be a positive double-precision number. datestr returns a column vector of m date strings, where m is the total number of date vectors in V.

S = datestr(N) converts one or more serial date numbers N to date strings S. Input argument N can be a scalar, vector, or multidimensional array of positive double-precision numbers. datestr returns a column vector of m date strings, where m is the total number of date numbers in N.

S = datestr(D, F) converts one or more date vectors, serial date numbers, or date strings D into the same number of date strings S.
Input argument \( F \) is a format number or string that determines the format of the date string output. Valid values for \( F \) are given in the table Standard MATLAB Date Format Definitions on page 2-834, below. Input \( F \) may also contain a free-form date format string consisting of format tokens shown in the table Free-Form Date Format Specifiers on page 2-837, below.

Date strings with 2-character years are interpreted to be within the 100 years centered around the current year.

\[
S = \text{datestr}(S1, F, P)
\]

converts date string \( S1 \) to date string \( S \), applying format \( F \) to the output string, and using pivot year \( P \) 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. All date strings in \( S1 \) must have the same format.

\[
S = \text{datestr}(..., '\text{local}')
\]

returns the date string in the localized format that you currently have selected by means of your computer’s operating system. You cannot select a nondefault format using the \text{datestr} function. The default is US English (‘en_US’).

The local argument must come last in the argument sequence. When you specify the \text{local} keyword with datestr, MATLAB returns the date string in a format

\[\text{Example}\]

<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>
</tbody>
</table>
### Standard MATLAB Date Format Definitions (Continued)

<table>
<thead>
<tr>
<th>dateform (number)</th>
<th>dateform (string)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<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>
<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>
<tr>
<td>21</td>
<td>'mmm.dd,yyyy HH:MM:SS'</td>
<td>Mar.01,2000 15:45:17</td>
</tr>
<tr>
<td>22</td>
<td>'mmm.dd,yyyy'</td>
<td>Mar.01,2000</td>
</tr>
</tbody>
</table>
### Standard MATLAB Date Format Definitions (Continued)

<table>
<thead>
<tr>
<th>dateform (number)</th>
<th>dateform (string)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>'mm/dd/yyyy'</td>
<td>03/01/2000</td>
</tr>
<tr>
<td>24</td>
<td>'dd/mm/yyyy'</td>
<td>01/03/2000</td>
</tr>
<tr>
<td>25</td>
<td>'yy/mm/dd'</td>
<td>00/03/01</td>
</tr>
<tr>
<td>26</td>
<td>'yyyy/mm/dd'</td>
<td>2000/03/01</td>
</tr>
<tr>
<td>27</td>
<td>'QQ-YYYY'</td>
<td>Q1-2001</td>
</tr>
<tr>
<td>28</td>
<td>'mmm/yyyy'</td>
<td>Mar2000</td>
</tr>
<tr>
<td>29 (ISO 8601)</td>
<td>'yyyy-mm-dd'</td>
<td>2000-03-01</td>
</tr>
<tr>
<td>30 (ISO 8601)</td>
<td>'yyyyymmdTHMMSS'</td>
<td>20000301T154517</td>
</tr>
<tr>
<td>31</td>
<td>'yyyy-mm-dd HH:MM:SS'</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.

**Note** For date formats that specify only a time (i.e., dateform numbers 13, 14, 15, and 16), MATLAB sets the date to January 1 of the current year.

Time formats like 'h:m:s', 'h:m:s.s', 'h:m pm', ... can also be part of the input array S. If you do not specify a format string F, or if you specify F as -1, the date string format defaults to the following:
If \( S \) contains date information only, e.g., 01-Mar-1995

If \( S \) contains time information only, e.g., 03:45 PM

If \( S \) 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.

**Note** You cannot use more than one format specifier for any date or time field. For example, \texttt{datestr(n, 'dddd dd mmmm')} specifies two formats for the day of the week, and thus returns an error.

### 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>
</tbody>
</table>
Free-Form Date Format Specifiers (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Interpretation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<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>FFF</td>
<td>Show millisecond in three digits.</td>
<td>.057</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.

**Remarks**

A vector of three or six numbers could represent either a single date vector, or a vector of individual serial date numbers. For example, the vector [2000 12 15 11 45 03] could represent either 11:45:03 on December 15, 2000 or a vector of date numbers 2000, 12, 15, etc.. MATLAB uses the following general rule in interpreting vectors associated with dates:
• A 3- or 6-element vector having a first element within an approximate range of 500 greater than or less than the current year is considered by MATLAB to be a date vector. Otherwise, it is considered to be a vector of serial date numbers.

To specify dates outside of this range as a date vector, first convert the vector to a serial date number using the `datenum` function as shown here:

```matlab
datestr(datenum([1400 12 15 11 45 03]), ...
    'mmm.dd,yyyy HH:MM:SS')
ans =
    Dec.15,1400 11:45:03
```

### Examples

Return the current date and time in a string using the default format, 0:

```matlab
datestr(now)
ans =
    28-Mar-2005 15:36:23
```

Reformat the date and time, and also show milliseconds:

```matlab
dt = datestr(now, 'mmmm dd, yyyy HH:MM:SS.FFF AM')
dt =
    March 28, 2005 3:37:07.952 PM
```

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 =
    03/28/05
```

Display the returned date string using your own format made up of symbols shown in the Free-Form Date Format Specifiers on page 2-837 table above.
Convert a nonstandard date form into a standard MATLAB date form by first converting to a date number and then to a string:

```
datestr(datenum('28.03.2005', 'dd.mm.yyyy'), 2)
```

```
ans =
03/28/05
```

**See Also**
datenum, datevec, date, clock, now, datetick
**Purpose**  
Date formatted tick labels

**Syntax**

```matlab
datetick(tickaxis)
datetick(tickaxis, dateformat)
datetick(tickaxis, dateformnum)
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. The axis data values should be generated by or be compatible with the output of the `datenum` function.

`datetick(tickaxis, dateformat)` formats the labels according to the string `dateformat`. A date format string can consist of the following elements (or combinations of them), identified by the format symbols in the left-hand column.

<table>
<thead>
<tr>
<th>Date Format</th>
<th>Interpretation of Format Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>yyyy</td>
<td>Full year, e.g., 1990, 2001, or 2008</td>
</tr>
<tr>
<td>yy</td>
<td>Partial year, e.g. 90, 01, or 08</td>
</tr>
<tr>
<td>mmmm</td>
<td>Full name of the month, according to the calendar locale, e.g., &quot;March&quot; or &quot;April&quot; in the UK and USA English locales</td>
</tr>
<tr>
<td>mm</td>
<td>First three letters of the month, according to the calendar locale, e.g., &quot;Mar&quot; or &quot;Apr&quot; in the UK and USA English</td>
</tr>
<tr>
<td>mm</td>
<td>Numeric month of year, padded with leading zeros, e.g., ../03/.. or ../12/..</td>
</tr>
<tr>
<td>Date Format</td>
<td>Interpretation of Format Symbol</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>m</td>
<td>Capitalized first letter of the month, according to the calendar locale; for backwards compatibility, e.g., “D” for December</td>
</tr>
<tr>
<td>dddd</td>
<td>Full name of the weekday, according to the calendar locale, e.g., &quot;Monday&quot; or &quot;Tuesday&quot;, for the UK and USA calendar locales</td>
</tr>
<tr>
<td>ddd</td>
<td>First three letters of the weekday, according to the calendar locale, e.g., &quot;Mon&quot; or &quot;Tue&quot;, for the UK and USA calendar locales</td>
</tr>
<tr>
<td>dd</td>
<td>Numeric day of the month, padded with leading zeros, e.g., 05/.. or 20/..</td>
</tr>
<tr>
<td>d</td>
<td>Capitalized first letter of the weekday, e.g., “M” for Monday; for backwards compatibility</td>
</tr>
<tr>
<td>HH</td>
<td>Hour of the day, according to the time format. In case the time format AM</td>
</tr>
<tr>
<td>MM</td>
<td>Minutes of the hour, padded with leading zeros, e.g., 10:05 or 10:05 AM</td>
</tr>
<tr>
<td>SS</td>
<td>Second of the minute, padded with leading zeros, e.g., 10:15:30, 10:05:30, 10:05:30 AM</td>
</tr>
<tr>
<td>FFF</td>
<td>Milliseconds field, padded with leading zeros, e.g., 10:15:30.015</td>
</tr>
<tr>
<td>PM</td>
<td>Setting the time format to morning or afternoon by appending AM or PM to the date string, as appropriate, without separating symbols</td>
</tr>
</tbody>
</table>

You can mix format symbols to create customized data symbols. For example:
datetick('x','dd (ddd)')

generates ticks along the x-axis that display the day of the month followed by the three-letter abbreviation of the day of the week in parentheses, for example, 01 (Wed). To preface each date tick with an abbreviated month name, you could specify

datetick('x','mmm-dd (ddd)')

to yield ticks such as Apr-01 (Wed).
datetick(tickaxis, dateformnum) formats the labels according to the integer dateformnum, a date format index (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>Date Format Number</th>
<th>dateformat (string)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>'dd-mmm-yyyy HH:MM:SS'</td>
<td>01-Mar-2008 15:45:17</td>
</tr>
<tr>
<td>1</td>
<td>'dd-mmm-yyyy'</td>
<td>01-Mar-2008</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>
</tbody>
</table>
### Date Format Number | `dateformat` (string) | Example
--- | --- | ---
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
29 | (ISO 8601) 'yyyy-mm-dd' | 2000-03-01
30 | (ISO 8601) 'yyyyymmddTHHMMSS' | 20000301T154517
31 | 'yyyy-mm-dd HH:MM:SS' | 2000-03-01 15:45:17

`datetick(...,'keeplimits')` changes the tick labels to date-based labels while preserving the axis limits.
datetick(...) 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.

Calling datetick sets the TickMode of the specified axis to 'manual'. This means that after zooming, panning or otherwise changing axis limits, you should call datetick again to update the ticks and labels.

**Examples**

**Example 1 — Plot US Population Across the 20th Century**

Graph population data for the 20th Century taken from the 1990 US census.

```matlab
% Create time data by decade
t = (1900:10:1990)';
% Enter total population counts for the USA
p = [75.995 91.972 105.711 123.203 131.669 ... 150.697 179.323 203.212 226.505 249.633]';
% Convert years to date numbers and plot
plot(datenum(t,1,1),p)
grid on
% Replace x-axis ticks with 2-digit years using date format 11
datetick('x',11)
```
Example 2 — Plot Hourly Traffic Counts by AM and PM

Plot traffic count data against date ticks for hours of the day showing AM and PM.

% Get traffic count data
load count.dat
% Create arrays for an arbitrary date, here April 18, 1995
n = length(count);
year = 1990 * ones(1,n);
month = 4 * ones(1,n);
day = 18 * ones(1,n);
% Create arrays for each of 24 hours;
hour = 1:n;
min = zeros(1,n);
% Get the datenums for the data (only hours change)
xdate = datenum(year,month,day,hour,min,min);
% Plot the traffic data against datenums
plot(xdate,count)
% Update the graph's x-axis with date ticks
datetick('x','HH:MM')

See Also

The axes properties: XTick, YTick, and ZTick
datenum, datestr
“Annotating Plots” on page 1-94 for related functions
Purpose

Convert date and time to vector of components

Syntax

V = datevec(N)
V = datevec(S, F)
V = datevec(S, F, P)
V = datevec(S, P, F)
[Y, M, D, H, MN, S] = datevec(...)  
V = datevec(S)
V = datevec(S, P)

Description
datevec is one of three conversion functions that enable you to express
dates and times in any of three formats in your MATLAB application:
a string (or date string), a vector of date and time components (or date
vector), or as a numeric offset from a known date in time (or serial date
number). Here is an example of a date and time expressed in the three
MATLAB formats:

Date String:  '24-Oct-2003 12:45:07'
Date Vector:  [2003 10 24 12 45 07]
Serial Date Number:  7.3188e+005

A serial date number represents the whole and fractional number
of days from 1-Jan-0000 to a specific date. The year 0000 is merely
a reference point and is not intended to be interpreted as a real year
in time.

V = datevec(N) converts one or more date numbers N to date vectors V.
Input argument N can be a scalar, vector, or multidimensional array of
positive date numbers. datevec returns an m-by-6 matrix containing m
date vectors, where m is the total number of date numbers in N.

V = datevec(S, F) converts one or more date strings S to date vectors
V using format string F to interpret the date strings in S. Input argument
S can be a cell array of strings or a character array where each row
corresponds to one date string. All of the date strings in S must have the
same format which must be composed of date format symbols according
to the table “Free-Form Date Format Specifiers” in the datestr help.
Formats with 'Q' are not accepted by `datevec`. `datevec` returns an m-by-6 matrix of date vectors, where m is the number of date strings in S.

Certain formats may not contain enough information to compute a date vector. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. Date strings with two character years are interpreted to be within the 100 years centered around the current year.

\[ V = \text{datevec}(S, F, P) \] converts the date string S to a date vector V using date format F and pivot year P. The pivot year is 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.

\[ V = \text{datevec}(S, P, F) \] is the same as the syntax shown above, except the order of the last two arguments are switched.

\[ [Y, M, D, H, MN, S] = \text{datevec}(\ldots) \] takes any of the two syntaxes shown above and returns the components of the date vector as individual variables. `datevec` does not return milliseconds in a separate output, but as a fractional part of the seconds (S) output.

\[ V = \text{datevec}(S) \] converts date string S to date vector V. Input argument S must be in one of the date formats 0, 1, 2, 6, 13, 14, 15, 16, or 23 as defined in the reference page for the `datenum` function. This calling syntax is provided for backward compatibility, and is significantly slower than the syntax which specifies the format string. If the format is known, the \[ V = \text{datevec}(S, F) \] syntax is recommended.

\[ V = \text{datevec}(S, P) \] converts the date string S using pivot year P. If the format is known, the \[ V = \text{datevec}(S, F, P) \] or \[ V = \text{datevec}(S, P, F) \] syntax should be used.

**Note** If more than one input argument is used, the first argument must be a date string or array of date strings.

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.

Note The vectorized calling syntax can offer significant performance
improvement for large arrays.

Examples

Obtain a date vector using a string as input:

```matlab
format short g
datevec('March 28, 2005 3:37:07.952 PM')
ans =
    2005   3    28    15    37    7.952
```

Obtain a date vector using a serial date number as input:

```matlab
t = datenum('March 28, 2005 3:37:07.952 PM')
t =
    7.324e+005
datevec(t)
ans =
    2005   3    28    15    37    7.952
```

Assign elements of the returned date vector:

```matlab
[y, m, d, h, mn, s] = datevec('March 28, 2005 3:37:07.952 PM');
sprintf('Date: %d/%d/%d    Time: %d:%d:%2.3f\n', m, d, y, h, mn, s)
ans =
    Date: 3/28/2005    Time: 15:37:7.952
```
Use free-form date format 'dd.mm.yyyy' to indicate how you want a nonstandard date string interpreted:

```matlab
datevec('28.03.2005', 'dd.mm.yyyy')
```

```
ans = 2005 3 28 0 0 0
```

**See Also**
datenum, datestr, date, clock, now, datetick
**Purpose**
Clear breakpoints

**GUI Alternatives**
In the Editor, click \( \text{Enter} \) to clear a breakpoint, or \( \text{Ctrl} + \text{Alt} + \text{Break} \) to clear all breakpoints. For details, see “Disabling and Clearing Breakpoints”.

**Syntax**
- `dbclear all`
- `dbclear in mfile ...`
- `dbclear if error ...`
- `dbclear if warning ...`
- `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 ...` formats are listed here:

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbclear in mfile</code></td>
<td>Removes all breakpoints in <code>mfile</code>. <code>mfile</code> must be the name of an M-file, and can include a MATLAB partialpath. If the command includes the <code>-completenames</code> option, then <code>mfile</code> need not be on the path, as long as it is a fully qualified file name. (On Microsoft Windows platforms, this is a file name that begins with <code>\</code> or with a drive <code>%</code> letter followed by a colon. On UNIX platforms, this is a file name that begins with <code>/</code> or <code>-</code>). <code>mfile</code> can include a filemarker to specify the path to a particular subfunction or to a nested function within an M-file.</td>
</tr>
<tr>
<td><code>dbclear in mfile at lineneno</code></td>
<td>Removes the breakpoint set at line number <code>lineno</code> in <code>mfile</code>.</td>
</tr>
<tr>
<td><code>dbclear in mfile at lineneno@</code></td>
<td>Removes the breakpoint set in the anonymous function at line number <code>lineno</code> in <code>mfile</code>.</td>
</tr>
</tbody>
</table>
**dbclear**

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbclear in mfile at lineno</code></td>
<td>Removes the breakpoint set in the anonymous function at line number lineno in mfile.</td>
</tr>
<tr>
<td><code>dbclear in mfile at subfun</code></td>
<td>Removes all breakpoints in subfunction subfun in mfile.</td>
</tr>
</tbody>
</table>

**dbclear if error** ... formats are listed here:

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbclear if error</code></td>
<td>Removes the breakpoints set using the dbstop if error and dbstop if error identifier statements.</td>
</tr>
<tr>
<td><code>dbclear if error identifier</code></td>
<td>Removes the breakpoint set using dbstop if error identifier for the specified identifier. Running this produces an error if dbstop if error or dbstop if error all is set.</td>
</tr>
<tr>
<td><code>dbclear if caught error</code></td>
<td>Removes the breakpoints set using the dbstop if caught error and dbstop if caught error identifier statements.</td>
</tr>
<tr>
<td><code>dbclear if caught error identifier</code></td>
<td>Removes the breakpoints set using the dbstop if caught error identifier statement for the specified identifier. Running this produces an error if dbstop if caught error or dbstop if caught error all is set.</td>
</tr>
</tbody>
</table>

**dbclear if warning** ... formats are listed here:

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbclear if warning</code></td>
<td>Removes the breakpoints set using the dbstop if warning and dbstop if warning identifier statements.</td>
</tr>
<tr>
<td><code>dbclear if warning identifier</code></td>
<td>Removes the breakpoint set using dbstop if warning identifier for the specified identifier. Running this produces an error if dbstop if warning or dbstop if warning all is set.</td>
</tr>
</tbody>
</table>

**dbclear if naninf** removes the breakpoint set by dbstop if naninf or dbstop if infnan.
dbclear  if infnan  removes the breakpoint set by dbstop  if infnan
or dbstop  if naninf.

Remarks

The at and in keywords are optional.

In the syntax, mfile can be an M-file, or the path to a function within
a file. For example

    dbclear in foo>myfun

clears the breakpoint at the myfun function in the file foo.m on Windows
platforms.

See Also

dbcont, dbdown, dbquit, dbstack, dbstatus, dbstep, dbstop, dbtype,
dbup, filemarker, partialpath
### dbcont

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Resume execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GUI Alternatives</strong></td>
<td>Select <strong>Debug &gt; Continue</strong> from most desktop tools, or in the Editor, click ![Debug].</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>dbcont</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>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 software returns to the base workspace prompt.</td>
</tr>
</tbody>
</table>

**Note** If you want to edit an M-file as a result of debugging, it is best to first quit debug mode and then edit and save changes to the M-file. If you edit an M-file while paused in debug mode, you can get unexpected results when you resume execution of the file and the results might not be reliable.

| See Also | dbclear, dbdown, dbquit, dbstack, dbstatus, dbstep, dbstop, dbtype, dbup |

2-856
Purpose
Reverse workspace shift performed by dbup, while in debug mode

GUI Alternatives
Use the Stack field in the Editor or in the Workspace browser.

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 over a rectangle

**Syntax**

\[
q = \text{dblquad} \left( \text{fun}, x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}} \right)
\]

\[
q = \text{dblquad} \left( \text{fun}, x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}, \text{tol} \right)
\]

\[
q = \text{dblquad} \left( \text{fun}, x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}, \text{tol}, \text{method} \right)
\]

**Description**

\[
q = \text{dblquad} \left( \text{fun}, x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}} \right)
\]
calls the \text{quad} function to evaluate the double integral \( \text{fun}(x,y) \) over the rectangle \( x_{\text{min}} \leq x \leq x_{\text{max}}, y_{\text{min}} \leq y \leq y_{\text{max}} \). \text{fun} is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. \text{fun}(x,y) must accept a vector \( x \) and a scalar \( y \) and return a vector of values of the integrand.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function \( \text{fun} \), if necessary.

\[
q = \text{dblquad} \left( \text{fun}, x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}, \text{tol} \right)
\]
uses a tolerance \( \text{tol} \) instead of the default, which is \( 1.0 \times 10^{-6} \).

\[
q = \text{dblquad} \left( \text{fun}, x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}, \text{tol}, \text{method} \right)
\]
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}:

\[
Q = \text{dblquad}\left(\@\text{integrnd},\pi,2*\pi,0,\pi\right);
\]

where the M-file \text{integrnd.m} is

\[
\begin{align*}
\text{function } z &= \text{integrnd}(x, y) \\
z &= y \text{*sin}(x) + x \text{*cos}(y);
\end{align*}
\]

Pass anonymous function handle \( F \) to \text{dblquad}:

\[
\begin{align*}
F &= @(x,y)y \text{*sin}(x) + x \text{*cos}(y) \\
Q &= \text{dblquad}(F,\pi,2*\pi,0,\pi);
\end{align*}
\]
The integrand function integrates \( y \sin(x) + x \cos(y) \) over the square \( \pi \leq x \leq 2\pi, 0 \leq y \leq \pi \). 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

\[
\texttt{dblquad(@(x,y)sqrt(max(1-(x.^2+y.^2),0)), -1, 1, -1, 1)}
\]

or

\[
\texttt{dblquad(@(x,y)sqrt(1-(x.^2+y.^2)).*(x.^2+y.^2<=1)}, -1, 1, -1, 1)
\]

**See Also**
quad2d, quad, quadgk, quadl, triplequad, function_handle (@), “Anonymous Functions”
Purpose
Enable MEX-file debugging (on UNIX platforms)

Syntax
\texttt{dbmx\ on}
\texttt{dbmx\ off}
\texttt{dbmx\ stop}

Description
\texttt{dbmx\ on} enables MEX-file debugging for UNIX\textsuperscript{4} platforms. It is not supported on the Sun Solaris platform.

To use this option, first start the MATLAB software from a debugger by typing \texttt{matlab -Ddebugger}, where \texttt{debugger} is the name of the debugger.

\texttt{dbmx\ off} disables MEX-file debugging.
\texttt{dbmx\ stop} returns to the debugger prompt.

Remarks
On Solaris, \texttt{dbmx} is not supported. See the Technical Support solution 1-17Z0R at \url{http://www.mathworks.com/support/solutions/data/1-17Z0R.html} for an alternative method of debugging.

See Also
\texttt{dbclear, dbcont, dbdown, dbquit, dbstack, dbstatus, dbstep, dbstop, dbtype, dbup}

4. UNIX is a registered trademark of The Open Group in the United States and other countries.
### Purpose
Quit debug mode

### GUI Alternative
From most desktop tools, select **Debug > Exit Debug Mode**, or in the Editor, click 

### Syntax
```
dbquit
```
```
dbquit('all')
```
```
dbquit all
```

### Description
The `dbquit` function terminates debug mode. The Command Window then displays the standard prompt (>>). The M-file being processed is *not* completed and no results are returned. All breakpoints remain in effect. As an alternative to `dbquit`, press **Shift+F5**.

If you debug `file1` and step into `file2`, running `dbquit` terminates debugging for both files. However, if you debug `file3` and also debug `file4`, running `dbquit` terminates debugging for `file4`, but `file3` remains in debug mode until you run `dbquit` again.

```
dbquit('all')
```

or the command form, `dbquit all`, ends debugging for all files at once.

### Examples
This example illustrates the use of `dbquit` relative to `dbquit('all')`. Set breakpoints in and run `file1` and `file2`:
```
>> dbstop in file1
>> dbstop in file2
>> file1
K>> file2
K>> dbstack
```

MATLAB software returns
```
K>> dbstack
  In file1 at 11
  In file2 at 22
```

If you use the `dbquit` syntax
K>> dbquit

MATLAB ends debugging for file2 but file1 is still in debug mode as shown here

K>> dbstack
   in file1 at 11

Run dbquit again to exit debug mode for file1.
Alternatively, dbquit('all') ends debugging for both files at once:

K>> dbstack
   In file1 at 11
   In file2 at 22
   dbquit('all')
   dbstack

returns no result.

See Also
dbclear, dbcont, dbdown, dbstack, dbstatus, dbstep, dbstop, dbtype, dbup
**Purpose**

Function call stack

**GUI Alternative**

Use the Stack field in the Editor or in the Workspace browser.

**Syntax**

dbstack

dbstack(n)

dbstack('-completenames')

[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 display lists the line number of the most recently executed function call (at which the current breakpoint occurred) first, followed by its calling function, which is followed by its calling function, and so on. This continues until the tomost M-file function is reached. Each line number is a hyperlink you can click to go directly to that line in the Editor. The notation `functionname>subfunctionname` is used to describe the subfunction location.

dbstack(n) omits the first n frames from the display. This is useful when issuing a dbstack from within an error handler, for example.

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 n and '-completenames' can 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 is 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, `dbstack` returns a negative line number value to identify that special case. For example, if the last line to be executed is line 15, then the `dbstack` line number is 15 before you execute that line and -15 afterwards.

**Remarks**

In addition to using `dbstack` while debugging, you can also use `dbstack` within an M-file outside the context of debugging. In this case, to get and analyze information about the current M-file stack. For example, to get the name of the calling M-file, use `dbstack` with an output argument within the file being called. For example:

```
    st=dbstack;
```

**Examples**

This example shows the information returned when you issue `dbstack` while debugging an M-file:

```
    dbstack

    In /usr/local/matlab/toolbox/matlab/cond.m at line 13
    In test1.m at line 2
    In test.m at line 3
```

This example shows the information returned when you issue `dbstack` while debugging `lengthofline.m` to get the complete name of the file, the function name, and line number in which the function appears:

```
    [ST,I] = dbstack('-completenames')
    ST =
    file: 'I:\MATLABFiles\mymfiles\lengthofline.m'
    name: 'lengthofline'
    line: 28
    I =
    1
```

**See Also**

`dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`, `dbup`, `evalin`, `mfilename`, `whos`
MATLAB Desktop Tools and Development Environment Documentation

- “Editing and Debugging M-Files”
- “Examining Values”
**Purpose** List all breakpoints

**GUI Alternative** Breakpoint line numbers are displayed graphically via the breakpoint icons when the file is open in the Editor.

**Syntax**

```
dbstatus
dbstatus mfile
dbstatus('-completenames')
s = dbstatus(...)
```

**Description**

`dbstatus` lists all the breakpoints in effect including errors, caught errors, warnings, and `nan`/`inf`s.

`dbstatus mfile` displays a list of the line numbers for which breakpoints are set in the specified M-file, where `mfile` is an M-file function name or a MATLAB relative partial path. Each line number is a hyperlink you can click to go directly to that line in the Editor.

`dbstatus('-completenames')` displays, for each breakpoint, the absolute file name and the sequence of functions that nest the function containing the breakpoint.

`s = dbstatus(...)` returns breakpoint information in an `m`-by-1 structure with the fields listed in the following table. Use this syntax to save breakpoint status and restore it at a later time using `dbstop(s)`—see `dbstop` for an example.

<table>
<thead>
<tr>
<th>name</th>
<th>Function name.</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Full path for file containing breakpoints.</td>
</tr>
<tr>
<td>line</td>
<td>Vector of breakpoint line numbers.</td>
</tr>
<tr>
<td>anonymous</td>
<td>Vector of integers representing the anonymous functions in the <code>line</code> field. For example, 2 means the second anonymous function in that line. A value of 0 means the breakpoint is at the start of the line, not in an anonymous function.</td>
</tr>
</tbody>
</table>
Cell vector of breakpoint conditional expression strings corresponding to lines in the line field.

Condition string ('error', 'caught error', 'warning', or 'naninf').

When cond is '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 forms you can further qualify the function name with a subfunction name, as in `dbstatus function>subfunction`.

**Remarks**

In the syntax, `mfile` can be an M-file, or the path to a function within a file. For example

```
Breakpoint for foo>mfun is on line 9
```

means there is a breakpoint at the `myfun` subfunction, which is line 9 in the file `foo.m`.

**See Also**

dbclear, dbcont, dbdown, dbquit, dbstack, dbstep, dbstop, dbtype, dbup, error, partialpath, warning
Purpose

Execute one or more lines from current breakpoint

GUI Alternatives

As an alternative to `dbstep`, you can select **Debug > Step** or **Step In** in most desktop tools, or click the Step or Step In buttons on the Editor 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` 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 software also stops execution at any breakpoint it encounters.
Note If you want to edit an M-file as a result of debugging, it is best to first quit debug mode and then edit and save changes to the M-file. If you edit an M-file while paused in debug mode, you can get unexpected results when you resume execution of the file and the results might not be reliable.

See Also dbclear, dbcont, dbdown, dbquit, dbstack, dbstatus, dbstop, dbtype, dbup
Purpose  Set breakpoints

GUI Alternative  Use the Debug menu in most desktop tools, or the context menu in Editor. See “Setting Breakpoints”.

Syntax  

```
dbstop in mfile ...
dbstop in nonmfile
dbstop if error ...
dbstop if warning ...
dbstop if naninf
dbstop if infnan
dbstop(s)
```

Description  

`dbstop in mfile ...` formats are listed here:
<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbstop in mfile</code></td>
<td>Temporarily stops execution of the running <code>mfile</code> at the first executable line, putting MATLAB software in debug mode. <code>mfile</code> must be the name of an M-file, and can include a MATLAB partial path. If the command includes the <code>-completenames</code> option, then <code>mfile</code> need not be on the path, as long as it is a fully qualified file name. (On Microsoft Windows, this is a file name that begins with <code>\</code> or with a drive <code>%</code> letter followed by a colon. On UNIX platforms, this is a file name that begins with <code>/</code> or <code>.</code>) <code>mfile</code> can include a file marker to specify the path to a particular subfunction or to a nested function within an M-file. The <code>in</code> keyword is optional.</td>
<td>If you have graphical debugging enabled, the MATLAB Debugger opens with a breakpoint at the first executable line of <code>mfile</code>. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use <code>dbcont</code> or <code>dbstep</code> to resume execution of <code>mfile</code>. Use <code>dbquit</code> to exit from debug mode.</td>
</tr>
</tbody>
</table>
### dbstop

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbstop in mfile at lineno</code></td>
<td>Temporarily stops execution of running <code>mfile</code> just prior to execution of the line whose number is <code>lineno</code>, putting MATLAB in debug mode. If that line is not executable, execution stops and the breakpoint is set at the next executable line following <code>lineno</code>. <code>mfile</code> must be in a directory that is on the search path, or in the current directory. The <code>at</code> keyword is optional.</td>
<td>If you have graphical debugging enabled, MATLAB opens <code>mfile</code> with a breakpoint at line <code>lineno</code>. When execution stops, you can use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use <code>dbcont</code> or <code>dbstep</code> to resume execution of <code>mfile</code>. Use <code>dbquit</code> to exit from debug mode.</td>
</tr>
<tr>
<td><code>dbstop in mfile at lineno@</code></td>
<td>Stops just after any call to the first anonymous function in the specified line number in <code>mfile</code>.</td>
<td></td>
</tr>
<tr>
<td><code>dbstop in mfile at lineno@n</code></td>
<td>Stops just after any call to the <code>nth</code> anonymous function in the specified line number in <code>mfile</code>.</td>
<td></td>
</tr>
<tr>
<td><code>dbstop in mfile at subfun</code></td>
<td>Temporarily stops execution of running <code>mfile</code> just prior to execution of the subfunction <code>subfun</code>, putting MATLAB in debug mode. <code>mfile</code> must be in a directory that is on the search path, or in the current directory.</td>
<td>If you have graphical debugging enabled, MATLAB opens <code>mfile</code> with a breakpoint at the subfunction <code>subfun</code>. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use <code>dbcont</code> or <code>dbstep</code> to resume execution of <code>mfile</code>. Use <code>dbquit</code> to exit from debug mode.</td>
</tr>
<tr>
<td>Format</td>
<td>Action</td>
<td>Additional Information</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------------------</td>
</tr>
<tr>
<td><strong>dbstop in mfile at lineno if expression</strong></td>
<td>Temporarily stops execution of running <code>mfile</code>, just prior to execution of the line whose number is <code>lineno</code>, putting MATLAB in debug mode. Execution stops only if expression evaluates to true. expression is evaluated (as if by <code>eval</code>), in <code>mfile</code>'s workspace when the breakpoint is encountered, and must evaluate to a scalar logical value (1 or 0 for true or false). If that line is not executable, execution stops and the breakpoint is set at the next executable line following <code>lineno</code>. <code>mfile</code> must be in a directory that is on the search path, or in the current directory.</td>
<td>If you have graphical debugging enabled, MATLAB opens <code>mfile</code> with a breakpoint at line <code>lineno</code>. When execution stops, you can use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use <code>dbcont</code> or <code>dbstep</code> to resume execution of <code>mfile</code>. Use <code>dbquit</code> to exit from debug mode.</td>
</tr>
<tr>
<td><strong>dbstop in mfile at lineno@ if expression</strong></td>
<td>Stops just after any call to the first anonymous function in the specified line number in <code>mfile</code> if expression evaluates to logical 1 (true).</td>
<td></td>
</tr>
<tr>
<td><strong>dbstop in mfile at lineno@n if expression</strong></td>
<td>Stops just after any call to the nth anonymous function in the specified line number in <code>mfile</code> if expression evaluates to logical 1 (true).</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Action</td>
<td>Additional Information</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>------------------------</td>
</tr>
<tr>
<td><code>dbstop in mfile if expression</code></td>
<td>Temporarily stops execution of running <code>mfile</code>, at the first executable line, putting MATLAB in debug mode. Execution stops only if <code>expression</code> evaluates to logical <code>1</code> (true). <code>expression</code> is evaluated (as if by <code>eval</code>), in <code>mfile</code>'s workspace when the breakpoint is encountered, and must evaluate to a scalar logical value (<code>0</code> or <code>1</code> for true or false). <code>mfile</code> must be in a directory on the search path, or in the current directory.</td>
<td>If you have graphical debugging enabled, MATLAB opens <code>mfile</code> with a breakpoint at the first executable line of <code>mfile</code>. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use <code>dbcont</code> or <code>dbstep</code> to resume execution of <code>mfile</code>. Use <code>dbquit</code> to exit from debug mode.</td>
</tr>
<tr>
<td><code>dbstop in mfile at subfun if expression</code></td>
<td>Temporarily stops execution of running <code>mfile</code>, just prior to execution of the subfunction <code>subfun</code>, putting MATLAB in debug mode. Execution stops only if <code>expression</code> evaluates to logical <code>1</code> (true). <code>expression</code> is evaluated (as if by <code>eval</code>), in <code>mfile</code>'s workspace when the breakpoint is encountered, and must evaluate to a scalar logical value (<code>0</code> or <code>1</code> for true or false). <code>mfile</code> must be in a directory on the search path, or in the current directory.</td>
<td>If you have graphical debugging enabled, MATLAB opens <code>mfile</code> with a breakpoint at the subfunction specified by <code>subfun</code>. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use <code>dbcont</code> or <code>dbstep</code> to resume execution of <code>mfile</code>. Use <code>dbquit</code> to exit from debug mode.</td>
</tr>
</tbody>
</table>

`dbstop in nonmfile` temporarily stops execution of the running M-file at the point where `nonmfile` is called. This puts MATLAB in debug mode, where `nonmfile` is, for example, a built-in or MDL-file. MATLAB issues a warning because it cannot actually stop in the file;
rather MATLAB stops prior to the file's execution. Once stopped, you can examine values and code around that point in the execution. Use `dbstop in nonmfile` with caution because the debugger stops in M-files it uses for running and debugging if they contain `nonmfile`. As a result, some debugging features do not operate as expected, such as typing `help functionname` at the `K>>` prompt.

`dbstop if error` ... formats are listed here:

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbstop if error</code></td>
<td>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 errors that stop execution do not include run-time errors that are detected within a <code>try...catch</code> block. You cannot resume execution after an uncaught run-time error. Use <code>dbquit</code> to exit from debug mode.</td>
</tr>
<tr>
<td><code>dbstop if error identifier</code></td>
<td>Stops execution when any M-file you subsequently run produces a run-time error whose message identifier is <code>identifier</code>, putting MATLAB in debug mode, paused at the line that generated the error. The errors that stop execution do not include run-time errors that are detected within a <code>try...catch</code> block. You cannot resume execution after an uncaught run-time error. Use <code>dbquit</code> to exit from debug mode.</td>
</tr>
<tr>
<td><code>dbstop if caught error</code></td>
<td>Stops execution when any M-file you subsequently run produces a run-time error, putting MATLAB in debug mode, paused at the line in the <code>try</code> portion of the block that generated the error. The errors that stop execution are those detected within a <code>try...catch</code> block.</td>
</tr>
<tr>
<td><code>dbstop if caught error identifier</code></td>
<td>Stops execution when any M-file you subsequently run produces a run-time error whose message identifier is <code>identifier</code>, putting MATLAB in debug mode, paused at the line in the <code>try</code> portion of the block that generated the error. The errors that stop execution are those detected within a <code>try...catch</code> block.</td>
</tr>
</tbody>
</table>

`dbstop if warning` ... formats are listed here:
dbstop

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbstop if warning</td>
<td>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. Use dbcont or dbstep to resume execution.</td>
</tr>
<tr>
<td>dbstop if warning identifier</td>
<td>Stops execution when any M-file you subsequently run produces a runtime warning whose message identifier is identifier, putting MATLAB in debug mode, paused at the line that generated the warning. Use dbcont or dbstep to resume execution.</td>
</tr>
</tbody>
</table>

**Remarks**

Note that MATLAB could become nonresponsive if it stops at a breakpoint while displaying a modal dialog box or figure that your M-file creates. In that event, use Ctrl+C to go the MATLAB prompt.

To open the M-file in the Editor when execution reaches a breakpoint, select Debug > Open M-Files When Debugging.

To stop at each pass through a for loop, do not set the breakpoint at the for statement. For example, in
MATLAB executes the for statement only once, which is efficient. Therefore, when you set a breakpoint at the for statement and step through the file, you only stop at the for statement once. Instead place the breakpoint at the next line, m=n+1 to stop at each pass through the loop.

**Examples**

The file buggy, used in these examples, consists of three lines.

```matlab
function z = buggy(x)
    n = length(x);
    z = (1:n)./x;
end
```

**Stop at First Executable Line**

The statements

```matlab
dbstop in buggy
buggy(2:5)
```

stop execution at the first executable line in buggy:

```matlab
n = length(x);
```

The function

```matlab
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

```matlab
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.

Stop if InfNaN

In buggy, if any of the elements of the input x is zero, a division by zero occurs. The statements

    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.

Stop at Function in File

In this example, MATLAB stops at the newTemp function in the M-file yearlyAvgs:

    dbstop in yearlyAvgs>newTemp

Stop at Non M-File

In this example, MATLAB stops at the built-in function clear when you run myfile.m.

    dbstop in clear; myfile

MATLAB issues a warning, but permits the stop:
Warning: MATLAB debugger can only stop in M-files, and "m_interpreter>clear" is not an M-file. Instead, the debugger will stop at the point right before "m_interpreter>clear" is called.

Execution stops in myfile at the point where the clear function is called.

**Restore Saved Breakpoints**

1. Set breakpoints in myfile as follows:

   ```matlab
dbstop at 12 in myfile
   dbstop if error
   ```

2. Running dbstatus shows

   Breakpoint for myfile is on line 12.
   Stop if error.

3. Save the breakpoints to the structure s, and then save s to the MAT-file myfilebrkpnts.

   ```matlab
   s = dbstatus
   save myfilebrkpnts s
   ```

   Use `s=dbstatus('completenames')` to save absolute paths and the breakpoint function nesting sequence.

4. At this point, you can end the debugging session and clear all breakpoints, or even end the MATLAB session.

   When you want to restore the breakpoints, be sure all of the files containing the breakpoints are on the search path or in the current directory. Then load the MAT-file, which adds s to the workspace, and restore the breakpoints as follows:

   ```matlab
   load myfilebrkpnts
   dbstop(s)
   ```
5 Verify the breakpoints by running `dbstatus`, which shows

```
  dbstop at 12 in myfile
  dbstop if error
```

If you made changes to `myfile` after saving the breakpoints, the results from restoring the breakpoints are not predictable. For example, if you added a new line prior to line 12 in `myfile`, the breakpoint will now be set at the new line 12.

**See Also**

`assignin`, `break`, `dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbtype`, `dbup`, `evalin`, `filemarker`, `keyboard`, `partialpath`, `return`, `whos`
Purpose
List M-file with line numbers

GUI Alternatives
As an alternative to the dbtype function, you can see an M-file with line numbers by opening it in the Editor.

Syntax
```
dbtype mfilename
dbtype mfilename start:end
```

Description
The dbtype command is used to list an M-file with line numbers, which is helpful when setting breakpoints with dbstop.

`dbtype mfilename` displays the contents of the specified M-file, with the line number preceding each line. `mfilename` must be the full path name of an M-file, or a MATLAB relative `partialpath`.

`dbtype mfilename start:end` displays the portion of the M-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, use the syntax

```
dbtype mfilename 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**  
Shift current workspace to workspace of caller, while in debug mode

**GUI Alternative**  
As an alternative to the `dbup` function, you can select a different workspace from the **Stack** field in the Editor toolbar.

**Syntax**  
dbup

**Description**  
This function allows you to examine the calling M-file to determine what caused the arguments to be 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.)

**Remarks**  
If your receive an error message such as the following, it means that the parent workspace is under construction so that the value of `x` is unavailable:

```plaintext
??? Reference to a called function result under construction x
```

For more information, see “Problems Viewing Variable Values from Parent Workspace”.

**See Also**  
dbclear, dbcont, dbdown, dbquit, dbstack, dbstatus, dbstep, dbstop, dbtype
**Purpose**  
Solve delay differential equations (DDEs) with constant delays

**Syntax**  
\[
sol = \text{dde23}(\text{ddefun}, \text{lags}, \text{history}, \text{tspan})
\]
\[
sol = \text{dde23}(\text{ddefun}, \text{lags}, \text{history}, \text{tspan}, \text{options})
\]

**Arguments**

- **\text{ddefun}**  
  Function handle that evaluates the right side of the differential equations
  \[
y'(t) = f(t, y(t), y(t - \tau_1), \ldots, y(t - \tau_k))
  \]
  The function must have the form
  \[
dydt = \text{ddefun}(t, y, Z)
  \]
  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 \(\tau_j = \text{lags}(j)\). The output is a column vector corresponding to \(\tilde{f}(t, y(t), y(t - \tau_1), \ldots, y(t - \tau_k))\).

- **\text{lags}**  
  Vector of constant, positive delays \(\tau_1, \ldots, \tau_k\).

- **\text{history}**  
  Specify \text{history} in one of three ways:
  - A function of \(t\) such that \(y = \text{history}(t)\) 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 \(\text{sol}\) from a previous integration, if this call continues that integration


**Description**

```markdown
sol = dde23(ddefun, lags, history, tspan) integrates the system of DDEs

\[ y'(t) = f(t, y(t), y(t - \tau_1), \ldots, y(t - \tau_k)) \]

on the interval \([t_0, t_f]\), where \(\tau_1, \ldots, \tau_k\) are constant, positive delays and \(t_0 < t_f\). ddefun is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information.

“Parametrizing Functions” in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function 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 = deval(sol, 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 = dde23(ddefun, lags, history, tspan, options) solves as above with default integration properties replaced by values in options, an argument created with ddeset. See ddeset and “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 \( g(t, y(t), y(t - \tau_1), \ldots, y(t - \tau_k)) \) vanish. This function must be of the form

\[
[value, isterminal, direction] = events(t, y, Z)
\]

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:
**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 \(t \leq 0\).

```matlab
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.

```matlab
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`.

**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**

`ddesd`, `ddeget`, `ddeset`, `deval`, `function_handle (@)`
References


Purpose

Extract properties from delay differential equations options structure

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, ddesd, ddeset
**Purpose**  Solve delay differential equations (DDEs) with general delays

**Syntax**  

```matlab
sol = ddesd(ddefun,delays,history,tspan)
sol = ddesd(ddefun,delays,history,tspan,options)
```

**Arguments**

- **ddefun**  Function handle that evaluates the right side of the differential equations  
  
  $$y'(t) = f(t, y(t), y(d(1)), \ldots, y(d(k))).$$

  The function must have the form

  ```matlab
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 \( y(d(j)) \) for delay \( d(j) \) given as component \( j \) of \( \text{delays}(t,y) \). The output is a column vector corresponding to \( f(t, y(t), y(d(1)), \ldots, y(d(k))) \).

- **delays**  Function handle that returns a column vector of delays \( d(j) \). The delays can depend on both \( t \) and \( y(t) \). **ddesd** imposes the requirement that \( d(j) \leq t \) by using \( \min(d(j), t) \).

  If all the delay functions have the form \( d(j) = t - \tau j \), you can set the argument delays to a constant vector \( \text{delays}(j) = \tau j \). With delay functions of this form, **ddesd** is used exactly like **dde23**.
history

Specify history in one of three ways:

- A function of \( t \) such that \( y = \text{history}(t) \)
  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 \( \text{sol} \) from a previous integration,
  if this call continues that integration

\( \text{tspan} \)

Interval of integration from \( t_0 = \text{tspan}(1) \) to \( t_f = \text{tspan}(\text{end}) \) with \( t_0 < t_f \).

\( \text{options} \)

Optional integration argument. A structure you create using the \( \text{ddeset} \) function. See \( \text{ddeset} \) for details.

Description

\( \text{sol} = \text{ddesd}(\text{ddefun}, \text{delays}, \text{history}, \text{tspan}) \) integrates the system of DDEs

\[
y'(t) = f(t, y(t), y(d(1)), \ldots, y(d(k)))
\]

on the interval \([t_0, t_f]\), where delays \( d(j) \) can depend on both \( t \) and \( y(t) \), and \( t_0 < t_f \). Inputs \( \text{ddefun} \) and \( \text{delays} \) are function handles. See “Function Handles” in the MATLAB Programming documentation for more information.

“Parametrizing Functions” in the MATLAB Mathematics documentation, explains how to provide additional parameters to the functions \( \text{ddefun}, \text{delays}, \) and \( \text{history} \), if necessary.

\( \text{ddesd} \) returns the solution as a structure \( \text{sol} \). Use the auxiliary function \( \text{deval} \) and the output \( \text{sol} \) to evaluate the solution at specific points \( \text{tint} \) in the interval \( \text{tspan} = [t_0, t_f] \).

\[
y\text{int} = \text{deval}() \text{sol}, \text{tint})
\]

The structure \( \text{sol} \) returned by \( \text{ddesd} \) has the following fields.
sol.x  Mesh selected by ddesd
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, 'ddesd'

\[
sol = \text{ddesd}(\text{ddefun}, \text{delays}, \text{history}, \text{tspan}, \text{options})
\]
solves as above with default integration properties replaced by values in options, an argument created with ddeset. See ddeset and “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 'Events' option to specify a function that ddesd calls to find where functions \( g(t, y(t), y(d(1)), ..., y(d(k))) \) vanish. This function must be of the form

\[
[\text{value}, \text{isterminal}, \text{direction}] = \text{events}(t, y, Z)
\]

and contain an event function for each event to be tested. For the \( k \)th event function in events:

- \( \text{value}(k) \) is the value of the \( k \)th event function.
- \( \text{isterminal}(k) = 1 \) if you want the integration to terminate at a zero of this event function and 0 otherwise.
- \( \text{direction}(k) = 0 \) if you want ddesd 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:
**Examples**

The equation

```matlab
sol = ddesd(@ddex1de,@ddex1delays,@ddex1hist,[0,5]);
```

solves a DDE on the interval \([0,5]\) with delays specified by the function `ddex1delays` and differential equations computed by `ddex1de`. The history is evaluated for \(t \leq 0\) by the function `ddex1hist`. The solution is evaluated at 100 equally spaced points in \([0,5]\):

```matlab
tint = linspace(0,5);
yint = deval(sol,tint);
```

and plotted with

```matlab
plot(tint,yint);
```

This problem involves constant delays. The delay function has the form

```matlab
function d = ddex1delays(t,y)

\%DDEX1DELAYS Delays for using with DDEX1DE.

d = [ t - 1
      t - 0.2];
```

The problem can also be solved with the syntax corresponding to constant delays

```matlab
 delays = [1, 0.2];
 sol = ddesd(@ddex1de,delays,@ddex1hist,[0, 5]);
```

or using `dde23`:
sol = dde23(@ddex1de, delays, @ddex1hist, [0, 5]);

For more examples of solving delay differential equations see ddex2 and ddex3.

See Also

dde23, ddeget, ddeset, deval, function_handle (@)

References

**Purpose**
Create or alter delay differential equations options structure

**Syntax**

```matlab
options = ddeset('name1',value1,'name2',value2,...)
options = ddeset(oldopts,'name1',value1,...)
options = ddeset(oldopts,newopts)
```

**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. `ddeset` ignores case for property names.

`options = ddeset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This overwrites any values in `oldopts` that are specified using name/value pairs and returns the modified structure as the output argument.

`options = ddeset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `oldopts`.

`ddeset` with no input arguments displays all property names and their possible values, indicating defaults with braces `{}`.

You can use the function `ddeget` to query the `options` structure for the value of a specific property.

**DDE Properties**
The following sections describe the properties that you can set using `ddeset`. There are several categories of properties:

- Error control
- Solver output
- Step size
- Event location
- Discontinuities
Error Control Properties

At each step, solvers dde23 and ddesd estimate an error $e$. dde23 estimates the local truncation error, and ddesd estimates the residual. In either case, this error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, $\text{RelTol}$, and the specified absolute tolerance, $\text{AbsTol}$.

$$|e(i)| \leq \max(\text{RelTol} \cdot \text{abs}(y(i)), \text{AbsTol}(i))$$

For routine problems, dde23 and ddesd deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over “long” intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust $\text{RelTol}$. For the absolute error tolerance, the scaling of the solution components is important: if $|y|$ is somewhat smaller than $\text{AbsTol}$, the solver is not constrained to obtain any correct digits in $y$. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want $\text{RelTol}$ correct digits in all solution components except those smaller than thresholds $\text{AbsTol}(i)$. Even if you are not interested in a component $y(i)$ when it is small, you may have to specify $\text{AbsTol}(i)$ small enough to get some correct digits in $y(i)$ so that you can accurately compute more interesting components.

The following table describes the error control properties.
## DDE Error Control Properties

<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 error tolerance that applies to all components of the solution vector (y). It is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components except those smaller than thresholds (\text{AbsTol}(i)). The default, (1e-3), corresponds to 0.1% accuracy. 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 tolerances that apply to the individual components of the solution vector. (\text{AbsTol}(i)) is a threshold below which the value of the (i)th solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero. Even if you are not interested in a component (y(i)) when it is small, you may have to specify (\text{AbsTol}(i)) small enough to get some correct digits in (y(i)) so that you can accurately compute more interesting components. If (\text{AbsTol}) is a vector, the length of (\text{AbsTol}) must be the same as the length of the solution vector (y). If (\text{AbsTol}) is a scalar, the value applies to all components of (y).</td>
</tr>
<tr>
<td>NormControl</td>
<td>on</td>
<td>{off}</td>
</tr>
</tbody>
</table>
Solver Output Properties

You can use the solver output properties to control the output that the solvers generate.

DDE Solver Output Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputFcn</td>
<td>Function handle <code>@odeplot</code></td>
<td>The output function is a function that the solver calls after every successful integration step. To specify an output function, set 'OutputFcn' to a function handle. For example,</td>
</tr>
</tbody>
</table>

```matlab
options = ddeset('OutputFcn',... @myfun)
```

sets 'OutputFcn' to @myfun, a handle to the function myfun. See “Function Handles” in the MATLAB Programming documentation for more information.

The output function must be of the form

```matlab
status = myfun(t,y,flag)
```

“Parametrizing Functions” in the MATLAB Mathematics documentation, explains how to provide additional parameters to myfun, if necessary.

The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:
### DDE Solver Output Properties (Continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>myfun(tspan,y0,'init') before beginning the integration to allow the output function to initialize. tspan is the input argument to solvers dde23 and ddesd. y0 is the initial value of the solution, either from history(t0) or specified in the initialY option.</td>
<td></td>
</tr>
<tr>
<td>{none}</td>
<td>status = myfun(t,y) after each integration step on which output is requested. t contains points where output was generated during the step, and y is the numerical solution at the points in t. If t is a vector, the ith column of y corresponds to the ith element of t. myfun must return a status output value of 0 or 1. If literal &gt; status, the solver halts integration. You can use this mechanism, for instance, to implement a Stop button.</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>myfun([],[],'done') when integration is complete to allow the output function to perform any cleanup chores.</td>
<td></td>
</tr>
</tbody>
</table>

You can use these general purpose output functions or you can edit them to create your own. Type help functionname at the command line for more information.

- odeplot – time series plotting (default when you call the solver with no output argument and you have not specified an output function)
- odephas2 – two-dimensional phase plane plotting
- odephas3 – three-dimensional phase plane plotting
- odeprint – print solution as the solver computes it
DDE Solver Output Properties (Continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputSel</td>
<td>Vector of indices</td>
<td>Vector of indices specifying which components of the solution vector the dde23 or ddesd solver passes to the output function. For example, if you want to use the odeplot output function, but you want to plot only the first and third components of the solution, you can do this using</td>
</tr>
<tr>
<td></td>
<td></td>
<td>options = ddeset... ('OutputFcn',@odeplot,... 'OutputSel',[1 3]); By default, the solver passes all components of the solution to the output function.</td>
</tr>
<tr>
<td>Stats</td>
<td>on</td>
<td>{off}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The number of successful steps</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The number of failed attempts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The number of times the DDE function was called</td>
</tr>
</tbody>
</table>

**Step Size Properties**

The step size properties let you specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step size properties.
## DDE Step Size Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitialStep</td>
<td>Positive scalar</td>
<td>Suggested initial step size. InitialStep sets an upper bound on the magnitude of the first step size the solver tries. If you do not set InitialStep, the solver bases the initial step size on the slope of the solution at the initial time tspan(1). The initial step size is limited by the shortest delay. If the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable InitialStep.</td>
</tr>
</tbody>
</table>
### DDE Step Size Properties (Continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| MaxStep  | Positive scalar $\{0.1 \cdot \text{abs}(t0-tf)\}$ | Upper bound on solver step size. If the differential equation has periodic coefficients or solutions, it may be a good idea to set MaxStep to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce MaxStep:  
- When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance RelTol, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector AbsTol. (See “Error Control Properties” on page 2-895 for a description of the error tolerance properties.)  
- To make sure that the solver doesn’t step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solver (dde23 or ddesd) twice. If you do not know the time at which the change occurs, try reducing the error tolerances RelTol and AbsTol. Use MaxStep as a last resort. |

### Event Location Property

In some DDE problems, the times of specific events are important. While solving a problem, the dde23 and ddesd solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the Events property.
### DDE Events Property

<table>
<thead>
<tr>
<th>String</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| Events  | Function handle | Handle to a function that includes one or more event functions. See “Function Handles” in the MATLAB Programming documentation for more information. The function is of the form  
\[
[value, isterminal, direction] = events(t,y,Z)
\]

value, isterminal, and direction are vectors for which the \( i \)th element corresponds to the \( i \)th event function:
DDE Events Property (Continued)

<table>
<thead>
<tr>
<th>String</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
|        | • `value(i)` is the value of the $i$th event function.  
• `isterminal(i) = 1` if you want the integration to terminate at a zero of this event function, and 0 otherwise.  
• `direction(i) = 0` if you want the solver (`dde23` or `ddesd`) to locate all zeros (the default), +1 if only zeros where the event function is increasing, and -1 if only zeros where the event function is decreasing.  
If you specify an events function and events are detected, the solver returns three additional fields in the solution structure `sol`:  
• `sol.xe` is a row vector of times at which events occur.  
• `sol.ye` is a matrix whose columns are the solution values corresponding to times in `sol.xe`.  
• `sol.ie` is a vector containing indices that specify which event occurred at the corresponding time in `sol.xe`. |

For examples that use an event function while solving ordinary differential equation problems, see “Event Location” (`ballode`) and “Advanced Event Location” (`orbitode`), in the MATLAB Mathematics documentation.

Discontinuity Properties

Solvers `dde23` and `ddesd` can solve problems with discontinuities in the history or in the coefficients of the equations. The following properties enable you to provide these solvers with a different initial value, and, for `dde23`, locations of known discontinuities. See “Discontinuities” in the MATLAB Mathematics documentation for more information.

The following table describes the discontinuity properties.
DDE Discontinuity Properties

<table>
<thead>
<tr>
<th>String</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jumps</td>
<td>Vector</td>
<td>Location of discontinuities. Points ( t ) where the history or solution may have a jump discontinuity in a low-order derivative. This applies only to the dde23 solver.</td>
</tr>
<tr>
<td>InitialY</td>
<td>Vector</td>
<td>Initial value of solution. By default the initial value of the solution is the value returned by history at the initial point. Supply a different initial value as the value of the InitialY property.</td>
</tr>
</tbody>
</table>

**Example**

To create an options structure that changes the relative error tolerance of the solver from the default value of \( 1e^{-3} \) to \( 1e^{-4} \), enter

```matlab
options = ddeset('RelTol', 1e-4);
```

To recover the value of 'RelTol' from options, enter

```matlab
ddeget(options, 'RelTol')
```

```matlab
ans =

1.0000e-004
```

**See Also**

dde23, ddesd, ddeget, function_handle (@)
**Purpose**
Distribute inputs to outputs

**Note** Beginning with MATLAB Version 7.0 software, you can access the contents of cell arrays and structure fields without using the `deal` function. See Example 3, below.

**Syntax**
- `[Y1, Y2, Y3, ...] = deal(X)`
- `[Y1, Y2, Y3, ...] = deal(X1, X2, X3, ...)`
- `[S.field] = deal(X)`
- `[X{:}] = deal(A.field)`
- `[Y1, Y2, Y3, ...] = deal(X{:})`
- `[Y1, Y2, Y3, ...] = deal(S.field)`

**Description**
- `[Y1, Y2, Y3, ...] = deal(X)` copies the single input to all the requested outputs. It is the same as `Y1 = X; Y2 = X; Y3 = X; ...`
- `[Y1, Y2, Y3, ...] = deal(X1, X2, X3, ...)` is the same as `Y1 = X1; Y2 = X2; Y3 = X3; ...`

**Remarks**
deal is most useful when used with cell arrays and structures via comma-separated list expansion. Here are some useful constructions:
- `[S.field] = 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).field] = deal(X)`.
- `[X{:}] = deal(A.field)` copies the values of the field with name `field` to the cell array `X`. If `X` doesn’t exist, use `[X{1:m}] = deal(A.field)`.
- `[Y1, Y2, Y3, ...] = deal(X{:})` copies the contents of the cell array `X` to the separate variables `Y1, Y2, Y3, ...`
- `[Y1, Y2, Y3, ...] = deal(S.field)` copies the contents of the fields with the name `field` to separate variables `Y1, Y2, Y3, ...`
Examples

Example 1 — Assign Data From a Cell Array

Use `deal` to copy the contents of a 4-element cell array into four separate output variables.

```matlab
C = {rand(3) ones(3,1) eye(3) zeros(3,1)};
[a,b,c,d] = deal(C{:})
```

```matlab
a =
    0.9501    0.4860    0.4565
   0.2311    0.8913    0.0185
   0.6068    0.7621    0.8214

b =
    1
    1
    1

c =
    1     0     0
    0     1     0
    0     0     1

d =
    0
    0
    0
```

Example 2 — Assign Data From Structure Fields

Use `deal` to obtain the contents of all the `name` fields in a structure array:

```matlab
A.name = 'Pat'; A.number = 176554;
A(2).name = 'Tony'; A(2).number = 901325;
[name1,name2] = deal(A(:).name)
```

```matlab
name1 =
    Pat
```
name2 =
    Tony

Example 3 — Doing the Same Without deal

Beginning with MATLAB Version 7.0 software, you can, in most cases, access the contents of cell arrays and structure fields without using the deal function. The two commands shown below 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 end of string

**Syntax**
```matlab
str = deblank(str)
c = deblank(c)
```

**Description**
`str = deblank(str)` removes all trailing whitespace and null characters from the end of character string `str`. A whitespace is any character for which the `isspace` function returns logical 1 (true).

`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**

**Example 1 – Removing Trailing Blanks From a String**

Compose a string `str` that contains space, tab, and null characters:

```matlab
NL = char(0); TAB = char(9);
str = [NL 32 TAB NL 'AB' 32 NL 'CD' NL 32 TAB NL 32];
```

Display all characters of the string between `|` symbols:

```matlab
['|' str '|']
ans =
   |   AB   CD   |
```

Remove trailing whitespace and null characters, and redisplay the string:

```matlab
newstr = deblank(str);
['|' newstr '|']
ans =
   |   AB   CD   |
Example 2– Removing Trailing Blanks From a Cell Array of Strings

A{1,1} = 'MATLAB   ';  
A{1,2} = 'SIMULINK   ';  
A{2,1} = 'Toolboxes   ';  
A{2,2} = 'The MathWorks   ';  
A =   
'MATLAB   '   'SIMULINK   '   
'Toolboxes   '   'The MathWorks   '   

debblank(A)  
ans =   
'MATLAB'   'SIMULINK'   
'Toolboxes'   'The MathWorks'   

See Also  strjust, strtrim
**debug**

**Purpose**
List M-file debugging functions

**GUI Alternatives**
Use the **Debug** menu in most desktop tools, or use the Editor.

**Syntax**
debug

**Description**
debug lists M-file debugging functions.

Use debugging functions (listed in the See Also section) to help you identify problems in your M-files. Set breakpoints using **dbstop**. When MATLAB software encounters a breakpoint during execution, it enters debug mode, the Editor becomes active, and the prompt in the Command Window changes to a **K>>**. Any MATLAB command is allowed at the prompt. To resume execution, use **dbcont** or **dbstep**. To exit from debug mode, use **dbquit**.

To open the M-File in the Editor when execution reaches a breakpoint, select **Debug > Open M-Files When Debugging**.

**See Also**
dbclear, dbcont, dbdown, dbquit, dbstack, dbstatus, dbstep, dbstop, dbtype, dbup, evalin, whos

“Editing and Debugging M-Files” in the MATLAB Desktop Tools and Development Environment documentation
**Purpose**  Convert decimal to base N number in string

**Syntax**

\[
\text{str} = \text{dec2base}(d, \text{base})  \\
\text{str} = \text{dec2base}(d, \text{base}, n)
\]

**Description**  \( \text{str} = \text{dec2base}(d, \text{base}) \) converts the nonnegative integer \( d \) to the specified base. \( d \) must be a nonnegative integer smaller than \( 2^{52} \), and \( \text{base} \) must be an integer between 2 and 36. The returned argument \( \text{str} \) is a string.

\( \text{str} = \text{dec2base}(d, \text{base}, n) \) produces a representation with at least \( n \) digits.

**Examples**  The expression \( \text{dec2base}(23, 2) \) converts \( 23_{10} \) to base 2, returning the string '10111'.

**See Also**  base2dec
Purpose
Convert decimal to binary number in string

Syntax
str = dec2bin(d)
str = dec2bin(d,n)

Description
returns the
str = dec2bin(d) 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
Decimal 23 converts to binary 010111:

dec2bin(23)
ans =
10111

See Also
bin2dec, dec2hex
Purpose
Convert decimal to hexadecimal number in string

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**

```matlab
[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,resnrm] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0...)
```

**Description**

```matlab
[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. `odefun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. The function `decic` changes as few components of the guesses as possible. You can specify that `decic` holds certain components fixed by setting `fixed_y0(i) = 1` if no change is permitted in the guess for `y0(i)` and 0 otherwise. `decic` interprets `fixed_y0 = []` as allowing changes in all entries. `fixed yp0` is handled similarly.

“Parametrizing Functions” in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `odefun`, if necessary.

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.

```matlab
[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.

```matlab
[y0mod,yp0mod,resnrm] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0...)
```

returns the
norm of $\text{odefun}(t_0, y_{0\text{mod}}, y_{p\text{mod}})$ as $\text{resnrm}$. If the norm seems unduly large, use options to decrease $\text{RelTol}$ (1e-3 by default).

**Examples**

These demos provide examples of the use of `decic` in solving implicit ODEs: `ihb1dae`, `iburgersode`.

**See Also**

`ode15i`, `odeget`, `odeset`, `function_handle (@)`
**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 \( u \):

\[
[q,r] = \text{deconv}(c,u)
\]

\[
\begin{align*}
q &= 10 \ 20 \ 30 \\
r &= 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0
\end{align*}
\]

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**

\[ L = \text{del2}(U) \]
\[-L = \text{del2}(U) \]
\[ L = \text{del2}(U, h) \]
\[ L = \text{del2}(U, hx, hy) \]
\[ L = \text{del2}(U, hx, hy, hz, \ldots) \]

**Definition**

If the matrix \( U \) is regarded as a function \( u(x, y) \) evaluated at the point on a square grid, then \( 4 \times \text{del2}(U) \) is a finite difference approximation of Laplace’s differential operator applied to \( u \), that is:

\[
L = \frac{\nabla^2 u}{4} = \frac{1}{4} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} \right)
\]

where:

\[
l_{ij} = \frac{1}{4} (u_{i+1, j} + u_{i-1, j} + u_{i, j+1} + u_{i, j-1} - u_{ij})
\]

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,

\[
l = \frac{\nabla^2 u}{2N} = \frac{1}{2N} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} + \frac{d^2 u}{dz^2} + \ldots \right)
\]

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
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 a multidimensional array, returns an approximation of

$$\nabla^2 u \quad \frac{2N}{2N}$$

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.

$L = \text{del2}(U,hx,hy,hz,\ldots)$ where $U$ is multidimensional uses the spacing given by $hx$, $hy$, $hz$, ...
has

\[ \nabla^2 u = 4 \]

For this function, \( 4 \times \text{del2}(U) \) is also 4.

\[ [x,y] = \text{meshgrid}(-4:4,-3:3); \]
\[ U = x.*x+y.*y \]
\[ U = \]
\[ \begin{array} {cccccccccc}
  25 & 18 & 13 & 10 & 9 & 10 & 13 & 18 & 25 \\
  20 & 13 & 8 & 5 & 4 & 5 & 8 & 13 & 20 \\
  17 & 10 & 5 & 2 & 1 & 2 & 5 & 10 & 17 \\
  16 & 9 & 4 & 1 & 0 & 1 & 4 & 9 & 16 \\
  17 & 10 & 5 & 2 & 1 & 2 & 5 & 10 & 17 \\
  20 & 13 & 8 & 5 & 4 & 5 & 8 & 13 & 20 \\
  25 & 18 & 13 & 10 & 9 & 10 & 13 & 18 & 25 \\
\end{array} \]

\[ V = 4 \times \text{del2}(U) \]
\[ V = \]
\[ \begin{array} {cccccccccccc}
  4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
  4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
  4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
  4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
  4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
  4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
  4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
  4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
\end{array} \]

See Also

diff, gradient
DelaunayTri class

**Superclasses**
TriRep

**Purpose**
Delaunay triangulation in 2-D and 3-D

**Description**
DelaunayTri creates a Delaunay triangulation object from a set of points. You can incrementally modify the triangulation by adding or removing points. In 2-D triangulations you can impose edge constraints. You can perform topological and geometric queries, and compute the Voronoi diagram and convex hull.

**Definitions**
The 2-D Delaunay triangulation of a set of points is the triangulation in which no point of the set is contained in the circumcircle for any triangle in the triangulation. The definition extends naturally to higher dimensions.

**Construction**
DelaunayTri

**Methods**

- `convexHull` Convex hull
- `inOutStatus` Status of triangles in 2-D constrained Delaunay triangulation
- `nearestNeighbor` Point closest to specified location
- `pointLocation` Simplex containing specified location
- `voronoiDiagram` Voronoi diagram

**Contract**
Delaunay triangulation
## Inherited methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>baryToCart</td>
<td>Converts point coordinates from barycentric to Cartesian</td>
</tr>
<tr>
<td>cartToBary</td>
<td>Convert point coordinates from cartesian to barycentric</td>
</tr>
<tr>
<td>circumcenters</td>
<td>Circumcenters of specified simplices</td>
</tr>
<tr>
<td>edgeAttachments</td>
<td>Simplices attached to specified edges</td>
</tr>
<tr>
<td>edges</td>
<td>Triangulation edges</td>
</tr>
<tr>
<td>faceNormals</td>
<td>Unit normals to specified triangles</td>
</tr>
<tr>
<td>featureEdges</td>
<td>Sharp edges of surface triangulation</td>
</tr>
<tr>
<td>freeBoundary</td>
<td>Facets referenced by only one simplex</td>
</tr>
<tr>
<td>incenters</td>
<td>Incenters of specified simplices</td>
</tr>
<tr>
<td>isEdge</td>
<td>Test if vertices are joined by edge</td>
</tr>
<tr>
<td>neighbors</td>
<td>Simplex neighbor information</td>
</tr>
<tr>
<td>size</td>
<td>Size of triangulation matrix</td>
</tr>
<tr>
<td>vertexAttachments</td>
<td>Return simplices attached to specified vertices</td>
</tr>
</tbody>
</table>
### DelaunayTri class

<table>
<thead>
<tr>
<th>Properties</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>Constraints is a numc-by-2 matrix that defines the constrained edge data in the triangulation, where numc is the number of constrained edges. Each constrained edge is defined in terms of its endpoint indices into X. The constraints can be specified when the triangulation is constructed or can be imposed afterwards by directly editing the constraints data. This feature is only supported for 2-D triangulations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>The dimension of X is mpts-by-ndim, where mpts is the number of points and ndim is the dimension of the space where the points reside. If column vectors of x,y or x,y,z coordinates are used to construct the triangulation, the data is consolidated into a single matrix X.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties</th>
<th>Triangulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangulation</td>
<td>Triangulation is a matrix representing the set of simplices (triangles or tetrahedra etc.) that make up the triangulation. The matrix is of size mtri-by-nv, where mtri is the number of simplices and nv is the number of vertices per simplex. The triangulation is represented by standard simplex-vertex format; each row specifies a simplex defined by indices into X, where X is the array of point coordinates.</td>
</tr>
</tbody>
</table>
DelaunayTri class

Instance Hierarchy

DelaunayTri is a subclass of TriRep.

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also

“Triangulation Representations”—How to query triangulation data

TriScatteredInterp
DelaunayTri

Purpose
Contract Delaunay triangulation

Syntax
DT = DelaunayTri()
DT = DelaunayTri(X)
DT = DelaunayTri(x,y)
DT = DelaunayTri(x,y,z)
DT = DelaunayTri(..., C)

Description
DT = DelaunayTri() creates an empty Delaunay triangulation.
DT = DelaunayTri(X), DT = DelaunayTri(x,y) and DT = DelaunayTri(x,y,z) create a Delaunay triangulation from a set of points. The points can be specified as an mpts-by-ndim matrix X, where mpts is the number of points and ndim is the dimension of the space where the points reside, ndim>= 2 . Alternatively, the points can be specified as column vectors (x,y) or (x,y,z) for 2-D and 3-D input.

DT = DelaunayTri(..., C) creates a constrained Delaunay triangulation. The edge constraints C are defined by an numc-by-2 matrix, numc being the number of constrained edges. Each row of C defines a constrained edge in terms of its endpoint indices into the point set X. This feature is only supported for 2-D triangulations.

Definitions
The 2-D Delaunay triangulation of a set of points is the triangulation in which no point of the set is contained in the circumcircle for any triangle in the triangulation. The definition extends naturally to higher dimensions.

Example
Compute the Delaunay triangulation of twenty random points located within a unit square.

```matlab
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y);
triplot(dt);
```
For more examples, type `help demoDelaunayTri` at the MATLAB command-line prompt.

**References**

DelaunayTri uses CGAL—The Computational Geometry Algorithms Library. (http://www.cgal.org)

**See Also**

“Triangulation Representations”—How to query triangulation data

TriScatteredInterp
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.

![Delaunay triangulation and Voronoi polygon diagram]

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

  \[
  \text{trisurf} \left( \text{TRI}, x, y, \text{zeros(size(x))} \right)
  \]

  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,

  \[
  \text{trimesh} \left( \text{TRI}, x, y, \text{zeros(size(x))} \right)
  \]

  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);
```
Compare the Voronoi diagram of the same points:

```matlab
[vx, vy] = voronoi(x,y,TRI);
```

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.

```matlab
[x,y] = meshgrid(1:15,1:15);
tri = delaunay(x,y);
```
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)`
Example 3

The following example illustrates the options input for delaunay.

\[
x = [-0.5 \quad -0.5 \quad 0.5 \quad 0.5];
\]
\[
y = [-0.5 \quad 0.5 \quad 0.5 \quad -0.5];
\]

The command

\[
T = \text{delaunay}(X);
\]

returns the following error message.

```
??? qhull input error: can not scale last coordinate. Input is
cocircular
or cospherical. Use option 'Qz' to add a point at infinity.
```

The error message indicates that you should add 'Qz' to the default
Qhull options.
tri = delaunay(x,y,{'Qt','Qbb','Qc','Qz'})

tri =

3   2   1
3   4   1

Algorithm
delaunay is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

See Also
delaunay3, delaunay, dsearch, griddata, plot, triplot, trimesh, trisurf, tsearch, voronoi

References
Purpose

3-D Delaunay tessellation

Syntax

\[ T = \text{delaunay3}(x,y,z) \]
\[ T = \text{delaunay3}(x,y,z,\text{options}) \]

Description

\[ T = \text{delaunay3}(x,y,z) \] returns an array \( T \), each row of which contains the indices of the points in \( (x,y,z) \) that make up a tetrahedron in the tessellation of \( (x,y,z) \). \( T \) is a numtes-by-4 array where 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.

\[ T = \text{delaunay3}(x,y,z,\text{options}) \] specifies a cell array of strings options to be used in Qhull via \text{delaunay3}. 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.

Visualization

Use \text{tetramesh} to plot \text{delaunay3} output. \text{tetramesh} displays the tetrahedrons defined in \( T \) as mesh. \text{tetramesh} uses the default transparency parameter value 'FaceAlpha' = 0.9.

Examples

Example 1

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.

\[
\begin{align*}
    d &= [-1 \ 1] \\
    [x,y,z] &= \text{meshgrid}(d,d,d); \quad \% \text{A cube} \\
    x &= [x(:);0] \\
    y &= [y(:);0] \\
    z &= [z(:);0]
\end{align*}
\]
% [x,y,z] are corners of a cube plus the center.
Tes = delaunay3(x,y,z)

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

X = [x(:) y(:) z(:)];
tetramesh(Tes,X);camorbit(20,0)
Example 2

The following example illustrates the options input for `delaunay3`.

\[
X = [-0.5 \ -0.5 \ -0.5 \ -0.5 \ 0.5 \ 0.5 \ 0.5 \ 0.5];
Y = [-0.5 \ -0.5 \ 0.5 \ 0.5 \ -0.5 \ -0.5 \ 0.5 \ 0.5];
Z = [-0.5 \ 0.5 \ -0.5 \ 0.5 \ -0.5 \ 0.5 \ -0.5 \ 0.5];
\]

The command

\[ T = delaunay3(X); \]

returns the following error message.

```plaintext
??? qhull input error: can not scale last coordinate. Input is cocircular
```
or cospherical. Use option 'Qz' to add a point at infinity.

The error message indicates that you should add 'Qz' to the default Qhull options.

\[ T = \text{delaunay3}(X, Y, Z, \{\text{'Qt'}, \text{'Qbb'}, \text{'Qc'}, \text{'Qz'}\}) \]

\[ T = \]

\[ \begin{array}{cccc}
4 & 3 & 5 & 1 \\
4 & 2 & 5 & 1 \\
4 & 7 & 3 & 5 \\
4 & 7 & 8 & 5 \\
4 & 6 & 2 & 5 \\
4 & 6 & 8 & 5 \\
\end{array} \]

Algorithm
delaunay3 is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

See Also
delaunay, delaunayn

Reference
Purpose
N-D Delaunay tessellation

Syntax
T = delaunayn(X)
T = delaunayn(X, options)

Description
T = 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 = delaunayn(X, options) specifies a cell array of strings 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 options is [], the default options 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.

Visualization
Plotting the output of 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.

- You cannot plot delaunayn output for n > 3.
Examples

Example 1

This example generates an n-dimensional Delaunay tessellation, where \( n = 3 \).

\[
\begin{align*}
d &= [-1 \ 1]; \\
[x,y,z] &= \text{meshgrid}(d,d,d); \quad \% \text{A cube} \\
x &= [x(:);0]; \\
y &= [y(:);0]; \\
z &= [z(:);0]; \\
\% [x,y,z] \text{ are corners of a cube plus the center.} \\
X &= [x(:) \ y(:) \ z(:)]; \\
Tes &= \text{delaunayn}(X)
\end{align*}
\]

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

You can use \texttt{tetramesh} to visualize the tetrahedrons that form the corresponding simplex. \texttt{camorbit} rotates the camera position to provide a meaningful view of the figure.

\[
\text{tetramesh}(Tes,X); \text{camorbit}(20,0)
\]
Example 2

The following example illustrates the options input for delaunayn.

\[ X = \begin{bmatrix} -0.5 & -0.5 & -0.5; \\ & -0.5 & 0.5 & 0.5; \\ & -0.5 & 0.5 & -0.5; \\ & -0.5 & 0.5 & 0.5; \\ & 0.5 & -0.5 & -0.5; \\ & 0.5 & -0.5 & 0.5; \\ & 0.5 & 0.5 & -0.5; \\ & 0.5 & 0.5 & 0.5 \end{bmatrix}; \]

The command

\[ T = \text{delaunayn}(X); \]
returns the following error message.

```matlab
??? qhull input error: cannot scale last coordinate. Input is cocircular or cospherical. Use option 'Qz' to add a point at infinity.
```

This suggests that you add 'Qz' to the default options.

```matlab
T = delaunayn(X,{'Qt','Qbb','Qc','Qz'});
```

To visualize this answer you can use the `tetramesh` function:

```matlab
tetramesh(T,X)
```
Algorithm  
delaunayn is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

See Also  
convhulln, delaunayn, delaunay3, tetramesh, voronoin

Reference  
### Purpose
Remove 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(handle_array)`
- `delete('filename')`

### Description
- **`delete filename`**: deletes the named file from the disk. The `filename` can include an absolute path or a path relative to the current directory. The `filename` can 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(handle_array)`**: is a method of the `handle` class. It removes from memory the handle objects referenced by `handle_array`.

  Once deleted, any references to the objects in `handle_array` become invalid. You can remove the handle variables using the `clear` function.

- **`delete('filename')`**: is the function form of `delete`. Use this form when the file name is stored in a string.

### Remarks

**Note** The MATLAB software does not ask for confirmation when you use `delete`. To avoid accidentally losing files or graphics objects, make sure you have accurately specified the items you want deleted, or use the recycle preference and the `recycle` function.

The action that the `delete` function takes on deleted files depends upon the setting of the recycle state in MATLAB. 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 preferences—select **File > Preferences > General**. To enable or disable recycling, use **Move files to the Recycle Bin or Delete files permanently**. See “General Preferences” 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*

“Managing Files and Working with the Current Directory”
Purpose  Remove COM control or server

Syntax  h.delete
delete(h)

Description  h.delete releases all interfaces derived from the specified COM server or control, and then deletes the server or control itself. This is different from releasing an interface, which releases and invalidates only that interface.

delete(h) is an alternate syntax.

Remarks  COM functions are available on Microsoft Windows systems only.

Examples  Create a Microsoft Calendar application. Then create a TitleFont interface and use it to change the appearance of the font of the calendar’s title:

```
f = figure('position',[300 300 500 500]);
cal = actxcontrol('mscal.calendar', [0 0 500 500], f);
TFont = cal.TitleFont
```

MATLAB software displays information similar to:

```
TFont =
    Interface.Microsoft_Forms_2.0_Object_Library.Font
```

Make the following changes and observe the results:

```
TFont.Name = 'Viva BoldExtraExtended';
TFont.Bold = 0;
```

When you’re finished working with the title font, release the TitleFont interface:

```
TFont.release;
```
Now create a GridFont interface and use it to modify the size of the calendar's date numerals:

```matlab
GFont = cal.GridFont
```

MATLAB displays:

```matlab
GFont =
    Interface.Microsoft_Forms_2.0_Object_Library.Font
```

Make the following changes and observe the results:

```matlab
GFont.Size = 16;
```

When you're done, delete the cal object and the figure window. Deleting the cal object also releases all interfaces to the object (for example, GFont):

```matlab
cal.delete;
delete(f);
clear f;
```

Note that, although the object and interfaces themselves have been destroyed, the variables assigned to them still reside in the MATLAB workspace until you remove them with clear:

```matlab
whos
```

MATLAB displays (in part):

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFont</td>
<td>1x1</td>
<td>0</td>
<td>handle</td>
</tr>
<tr>
<td>TFont</td>
<td>1x1</td>
<td>0</td>
<td>handle</td>
</tr>
<tr>
<td>cal</td>
<td>1x1</td>
<td>0</td>
<td>handle</td>
</tr>
</tbody>
</table>

**See Also**

release, save (COM), load (COM), actxcontrol, actxserver
**Purpose**
Remove 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**  
Handle object destructor function

**Syntax**  
delete(h)

**Description**  
delete(h) optional method you can implement to perform cleanup tasks just before the handle object is destroyed. The MATLAB runtime calls the `delete` method of any handle object (if it exists) when the object is destroyed. `h` is a scalar handle object.

A `delete` method should not generate errors or create new handles to the object being destroyed. If the `delete` method has a different signature (having output arguments or more than one input argument) it is not called when the handle objects is destroyed.

See “Handle Class Delete Methods” for more information.

**See Also**  
handle, isvalid
serial.delete

**Purpose**
Remove serial port object from memory

**Syntax**
delete(obj)

**Description**
delete(obj) removes obj from memory, where obj is a serial port object or an array of serial port objects.

**Remarks**
When you delete obj, it becomes an *invalid* object. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the clear command. If multiple references to obj exist in the workspace, then deleting one reference invalidates the remaining references.

If obj is connected to the device, it has a Status property value of open. If you issue delete while obj is connected, then the connection is automatically broken. You can also disconnect obj from the device with the fclose function.

If you use the help command to display help for delete, then you need to supply the pathname shown below.

```
help serial/delete
```

**Example**
This example creates the serial port object s on a Windows platform, connects s to the device, writes and reads text data, disconnects s from the device, removes s from memory using delete, and then removes s from the workspace using clear.

```
s = serial('COM1');
ofopen(s)
fprintf(s, '*IDN?')
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```
See Also

Functions

clear, fclose, isvalid

Properties

Status
**Purpose**
Remove 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), timer
Purpose
Remove custom property from COM object

Syntax
h.deleteproperty('propertyname')
deleteproperty(h, 'propertyname')

Description
h.deleteproperty('propertyname') deletes the property specified in the string propertyname from the custom properties belonging to object or interface, h.
deleteproperty(h, 'propertyname') is an alternate syntax.

Note You can only delete properties that have been created with addproperty.

Remarks
COM functions are available on Microsoft Windows systems only.

Examples
Create an mwsamp control and display its properties:

    f = figure('position', [100 200 200 200]);
    h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
    h.get

MATLAB software displays:

    Label: 'Label'
    Radius: 20

Add a new property named Position to the control. Assign an array value to the property:

    h.addproperty('Position');
    h.Position = [200 120];
    h.get

MATLAB displays (in part):
Delete the custom `Position` property:

```matlab
h.deleteproperty('Position');
h.get
```

MATLAB displays:

```
Label: 'Label'
Radius: 20
```

**See Also**

`addproperty`, `get (COM)`, `set (COM)`, `inspect`
**Purpose**
Remove tsdata.event objects from timeseries object

**Syntax**
- \( ts = \text{delevent}(ts, \text{event}) \)
- \( ts = \text{delevent}(ts, \text{events}) \)
- \( ts = \text{delevent}(ts, \text{event}, n) \)

**Description**
- \( ts = \text{delevent}(ts, \text{event}) \) removes the tsdata.event object from the ts.events property, where event is an event name string.
- \( ts = \text{delevent}(ts, \text{events}) \) removes the tsdata.event object from the ts.events property, where events is a cell array of event name strings.
- \( ts = \text{delevent}(ts, \text{event}, n) \) removes the \( n \)th tsdata.event object from the ts.events property. event is the name of the tsdata.event object.

**Examples**
The following example shows how to remove an event from a timeseries object:

1. Create a time series.
   \[
   ts = \text{timeseries}(\text{rand}(5,4))
   \]
2. Create an event object called 'test' such that the event occurs at time 3.
   \[
   e = \text{tsdata}.\text{event}(\text{'test'},3)
   \]
3. Add the event object to the time series ts.
   \[
   ts = \text{addevent}(ts,e)
   \]
4. Remove the event object from the time series ts.
   \[
   ts = \text{delevent}(ts,\text{'test'})
   \]

**See Also**
addevent, timeseries, tsdata.event, tsprops
**Purpose**
Remove sample from timeseries object

**Syntax**

\[
\begin{align*}
\text{ts} &= \text{delsample}(\text{ts},'\text{Index}',N) \\
\text{ts} &= \text{delsample}(\text{ts},'\text{Value}',\text{Time})
\end{align*}
\]

**Description**

\[
\text{ts} = \text{delsample}(\text{ts},'\text{Index}',N)
\]
deletes samples from the timeseries object \( \text{ts} \). \( N \) specifies the indices of the \( \text{ts} \) time vector that correspond to the samples you want to delete.

\[
\text{ts} = \text{delsample}(\text{ts},'\text{Value}',\text{Time})
\]
deletes samples from the timeseries object \( \text{ts} \). \( \text{Time} \) specifies the time values that correspond to the samples you want to delete.

**See Also**
addsample
**Purpose**  
Remove sample from ts_collection object

**Syntax**

```matlab
 tsc = delsamplefromcollection(tsc,'Index',N)
 tsc = delsamplefromcollection(tsc,'Value',Time)
```

**Description**

`tsc = delsamplefromcollection(tsc,'Index',N)` deletes samples from the `ts_collection` object `tsc`. `N` specifies the indices of the `tsc` time vector that correspond to the samples you want to delete.

`tsc = delsamplefromcollection(tsc,'Value',Time)` deletes samples from the `ts_collection` object `tsc`. `Time` specifies the time values that correspond to the samples you want to delete.

**See Also**

`addsampletocollection`, `ts_collection`
Purpose
Access product demos via Help browser

GUI Alternatives
As an alternative to the demo function, you can select Help > Demos from any desktop tool, 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 pane in the Help browser, listing demos for all installed products that are selected in the Help browser product filter preference. To access demos from the Demos 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 Mathematics). Select a specific demo from the list (for example, Square Wave from Sine Waves). In the right pane, view instructions for using the demo. For more information, see the topic “Viewing and Running Demos” in the MATLAB Desktop Tools and Development Environment documentation. To run a demo from the command line, type the demo name. To run an M-file demo, open it in the Editor and run it using Cell > Evaluate Current Cell and Advance, or run echodemo followed by the demo name.

demo 'subtopic' opens the Demos pane in the Help browser with the specified subtopic expanded. Subtopics are matlab, toolbox, simulink, blockset, and links and targets. If no products in subtopic are installed, or if none are selected in the Help browser product filter preference, an error page appears.

demo 'subtopic category' opens the Demos pane in the Help browser to the specified product or category within the subtopic. The demo function uses the full name displayed in the Demo pane for category. If the product specified by category is not installed, or is not selected in the Help browser product filter preference, an error page appears.
demo('subtopic', 'category') is the function form of the syntax. This illustration shows the result of running

    demo matlab graphics

and then selecting the Square Wave from Sine Waves example.
Square Wave from Sine Waves

The Fourier series expansion for a square-wave is made up of a sum of odd harmonics. We show this graphically using MATLAB®.

We start by forming a time vector running from 0 to 10 in steps of 0.1, and take the sine of all the points. Let's plot this fundamental frequency.

t = 0:.1:10;
y = sin(t);
plot(t,y);
Examples

**Accessing Toolbox Demos**

To find the demos relating to Communications Toolbox™ product, type

```matlab
demo toolbox communications
```

The Help browser opens to the **Demos** pane with the Toolbox subtopic expanded and with the Communications entry highlighted and expanded to show the available demos.

**Accessing Simulink Demos**

To access the demos within the Simulink product, type

```matlab
demo simulink automotive
```

The **Demos** pane opens with the subtopic for Simulink open and the Automotive category expanded.

**Function Form of demo**

To access the Simulink® Control Design™ demos, run

```matlab
demo('simulink', 'simulink control design')
```

which displays
Running a Demo from the Command Line

Type

vibes

to run a visualization demonstration showing an animated L-shaped membrane.
Running an M-File Demo from the Command Line

Type

\texttt{quake}

to run an earthquake data demo. Not much appears to happen because \texttt{quake} is an M-file demo and executes from start to end without stopping.

It displays a link in the Command Window: \texttt{View the published version of this demo}. Click the link to view and run the demo from the Help browser.

You can view the M-file, \texttt{quake.m}, by typing

\texttt{edit quake}

The first line, that is, the H1 line for \texttt{quake}, is

\texttt{%% Loma Prieta Earthquake}

The \texttt{%%} indicates that \texttt{quake} is an M-file demo. You can step through the demo cell-by-cell, from the Editor—select \texttt{Cell > Evaluate Current Cell and Advance}.

Alternatively, run

\texttt{echodemo quake}

and the \texttt{quake} demo runs step-by-step in the Command Window.

\textbf{See Also}

\texttt{echodemo, grabcode, help, helpbrowser}

“Viewing and Running Demos”
Purpose
List dependent directories of M-file or P-file

Syntax
list = depdir('file_name')
[list, prob_files, prob_sym, prob_strings] = depdir('file_name')
[param1, param2, ..., paramn] = 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.

[param1, param2, ..., paramn] = 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
list = depdir('mesh')

See Also
depfun
**Purpose**
List dependencies of M-file or P-file

**Syntax**

```matlab
list = depfun('fun')
[list, builtins, classes] = depfun('fun')
[list, builtins, classes, prob_files, prob_sym, eval_strings,
     ... called_from, java_classes] = depfun('fun')
[... ] = depfun('fun1', 'fun2', ...)
[... ] = depfun({'fun1', 'fun2', ...})
[... ] = depfun('fig_file')
[... ] = depfun(..., options)
```

**Description**
The `depfun` function lists the paths of all files a specified M-file or P-file needs to operate.

**Note**
It cannot be guaranteed that `depfun` will find every dependent file. Some dependent files can be hidden in callbacks, or can be constructed dynamically for evaluation, for example. Also note that the list of functions returned by `depfun` often includes extra files that would never be called if the specified function were actually evaluated.

`list = depfun('fun')` creates a cell array of strings containing the paths of all the files that function `fun` uses. This includes the second-level files that are called directly by `fun`, and the third-level files that are called by the second-level files, and so on.

Function `fun` must be on the MATLAB path, as determined by the `which` function. If the MATLAB path contains any relative directories, then files in those directories will also have a relative path.

**Note**
If MATLAB returns a parse error for any of the input functions, 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.
[list, builtins, classes] = depfun('fun') creates three cell arrays containing information about dependent functions. list contains the paths of all the files that function fun and its subordinates use. builtins contains the built-in functions that fun and its subordinates use. classes contains the MATLAB classes that fun and its subordinates use.

[list, builtins, classes, prob_files, prob_sym, eval_strings,... called_from, java_classes] = depfun('fun') 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** — 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 having these fields:
  - name (path to the file)
  - listindex (index of the file in list)
  - errmsg (problems encountered)

- **unused** — This is a placeholder for an output argument that is not fully implemented at this time. MATLAB returns an empty structure array for this output.

- **called_from** — Cell array of the same length as list that indicates which functions call other functions. This cell array is arranged so that the following statement returns all functions in function fun that invoke the function list{\textit{i}}:

  \[
  \text{list}(\text{called_from}\{\textit{i}\})
  \]

- **java_classes** — Cell array of Java class names used by fun and its subordinate functions.
[...] = depfun('fun1', 'fun2',...) performs the same operation for multiple functions. The dependent functions of all files are listed together in the output arrays.

[...] = depfun({'fun1', 'fun2', ...}) performs the same operation, but on a cell array of functions. The dependent functions of all files are listed together in the output array.

[...] = depfun('fig_file') looks for dependent functions among the callback strings of the GUI elements that are defined in the figure file named fig_file.

[...] = depfun(..., options) modifies the depfun operation according to the options specified (see table below).

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-all'</td>
<td>Computes all possible left-side arguments and displays the results in the report(s). Only the specified arguments are returned.</td>
</tr>
<tr>
<td>'-calltree'</td>
<td>Returns a call list in place of a called_from list. This is derived from the called_from list as an extra step.</td>
</tr>
<tr>
<td>'-expand'</td>
<td>Includes both indices and full paths in the call or called_from list.</td>
</tr>
<tr>
<td>'-print', 'file'</td>
<td>Prints a full report to file.</td>
</tr>
<tr>
<td>'-quiet'</td>
<td>Displays only error and warning messages, and not a summary report.</td>
</tr>
<tr>
<td>'-toponly'</td>
<td>Examines only the files listed explicitly as input arguments. It does not examine the files on which they depend.</td>
</tr>
<tr>
<td>'-verbose'</td>
<td>Outputs additional internal messages.</td>
</tr>
</tbody>
</table>

**Examples**

list = depfun('mesh'); % Files mesh.m depends on
list = depfun('mesh','-toponly') % Files mesh.m depends on directly
depfun

[li]\li\text{st}, \text{bu}l\text{t}i\text{ns}, \text{c}l\text{ass}\text{s}] = \text{depfun('(gca')};

\textbf{See Also}

\text{deppd}\text{ir}
Purpose
Matrix determinant

Syntax
d = det(X)

Description
d = 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 det(X) == 0 as a test for matrix singularity is appropriate only for matrices of modest order with small integer entries. Testing singularity using abs(det(X)) <= tolerance is not recommended as it is difficult to choose the correct tolerance. The function cond(X) can check for singular and nearly singular matrices.

Algorithm
The determinant is computed from the triangular factors obtained by Gaussian elimination

\[
\begin{align*}
[L, U] &= \text{lu}(A) \\
s &= \det(L) \quad \text{\% This is always +1 or -1} \\
\det(A) &= s \times \text{prod(diag(U))}
\end{align*}
\]

Examples
The statement \( A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \) produces

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

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
cond, condest, inv, lu, 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.

\[
\text{detrend}(x, 'linear'), \text{ with no breakpoint vector specified, is the same as } \text{detrend}(x).
\]

**Example**

\[
sig = [0 \ 1 \ -2 \ 1 \ 0 \ 1 \ -2 \ 1 \ 0]; \quad \% \text{signal with no linear trend}
\]

\[
trend = [0 \ 1 \ 2 \ 3 \ 4 \ 3 \ 2 \ 1 \ 0]; \quad \% \text{two-segment linear trend}
\]
x = sig+trend;                           % signal with added trend
y = detrend(x,'linear',5)               % breakpoint at 5th element

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*
detrend (timeseries)

**Purpose**
Subtract mean or best-fit line and all NaNs from time series

**Syntax**
```matlab
ts = detrend(ts1,method)
ts = detrend(ts1,Method,Index)
```

**Description**
`ts = detrend(ts1,method)` subtracts either a mean or a best-fit line from time-series data, usually for FFT processing. Method is a string that specifies the detrend method and has two possible values:

- `'constant'` — Subtracts the mean
- `'linear'` — Subtracts the best-fit line

`ts = detrend(ts1,Method,Index)` uses the optional `Index` integer array to specify the columns or rows to detrend. When `ts.IsTimeFirst` is true, `Index` specifies one or more data columns. When `ts.IsTimeFirst` is false, `Index` specifies one or more data rows.

**Remarks**
You cannot apply `detrend` to time-series data with more than two dimensions.
### Purpose
Evaluate solution of differential equation problem

### Syntax
- \( \text{sxint} = \text{deval}(\text{sol}, \text{xint}) \)
- \( \text{sxint} = \text{deval}(\text{xint}, \text{sol}) \)
- \( \text{sxint} = \text{deval}(\text{sol}, \text{xint}, \text{idx}) \)
- \( \text{sxint} = \text{deval}(\text{xint}, \text{sol}, \text{idx}) \)
- \( [\text{sxint}, \text{spxint}] = \text{deval}(\ldots) \)

### Description
- \( \text{sxint} = \text{deval}(\text{sol}, \text{xint}) \) and \( \text{sxint} = \text{deval}(\text{xint}, \text{sol}) \) evaluate the solution of a differential equation problem. \( \text{sol} \) is a structure returned by one of these solvers:
  - An initial value problem solver (ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb, ode15i)
  - A delay differential equations solver (dde23 or ddesd),
  - The boundary value problem solver (bvp4c or bvp5c).

\( \text{xint} \) 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 \( i \), \( \text{sxint}( :, i) \) is the solution at \( \text{xint}(i) \).

- \( \text{sxint} = \text{deval}(\text{sol}, \text{xint}, \text{idx}) \) and \( \text{sxint} = \text{deval}(\text{xint}, \text{sol}, \text{idx}) \) evaluate as above but return only the solution components with indices listed in the vector \( \text{idx} \).

- \( [\text{sxint}, \text{spxint}] = \text{deval}(\ldots) \) also returns \( \text{spxint} \), the value of the first derivative of the polynomial interpolating the solution.

### Note
For multipoint boundary value problems, the solution obtained by bvp4c or bvp5c might be discontinuous at the interfaces. For an interface point \( \text{xc} \), \( \text{deval} \) returns the average of the limits from the left and right of \( \text{xc} \). To get the limit values, set the \( \text{xint} \) argument of \( \text{deval} \) to be slightly smaller or slightly larger than \( \text{xc} \).
Example

This example solves the system $y' = \text{vdp1}(t, y)$ using \texttt{ode45}, and evaluates and plots the first component of the solution at 100 points in the interval $[0,20]$.

```matlab
sol = ode45(@vdp1,[0 20],[2 0]);
x = linspace(0,20,100);
y = deval(sol,x,1);
plot(x,y);
```

See Also

ODE solvers: \texttt{ode45}, \texttt{ode23}, \texttt{ode113}, \texttt{ode15s}, \texttt{ode23s}, \texttt{ode23t}, \texttt{ode23tb}, \texttt{ode15i}

DDE solvers: \texttt{dde23}, \texttt{ddesd}

BVP solver: \texttt{bvp4c}, \texttt{bvp5c}
**Purpose**
Diagonal matrices and diagonals of 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.

\[
\begin{array}{cc}
k = 0 & k > 0 \\
\end{array}
\]

\[
\begin{array}{cc}
k < 0 & \\
\end{array}
\]

\(X = \text{diag}(v)\) 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 \(k = 0\).

**Remarks**
\(\text{diag}(\text{diag}(X))\) is a diagonal matrix.

\(\text{sum}(\text{diag}(X))\) is the trace of \(X\).

\(\text{diag}([])\) generates an empty matrix, ([]).

\(\text{diag}(m\text{-by-}1,k)\) generates a matrix of size \(m+\text{abs}(k)\)-by-\(m+\text{abs}(k)\).
diag(1-by-n,k) generates a matrix of size n+abs(k)-by-n+abs(k).

**Examples**

The statement

```
diag(-m:m)+diag(ones(2*m,1),1)+diag(ones(2*m,1),-1)
```

produces a tridiagonal matrix of order 2*m+1.

**See Also**

spdiags, tril, triu, blkdiag
Purpose
Create and display empty dialog box

Syntax
h = dialog('PropertyName', PropertyValue, ...)

Description
h = dialog('PropertyName', PropertyValue, ...) returns a handle to a dialog box. The dialog box is a figure graphics object with properties recommended for dialog boxes. You can specify any valid figure property value except DockControls, which is always off.

The properties that dialog sets and their values are

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'BackingStore'</td>
<td>'off'</td>
</tr>
<tr>
<td>'ButtonDownFcn'</td>
<td>'if isempty(allchild(gcf)), close(gcf), end'</td>
</tr>
<tr>
<td>'Colormap'</td>
<td>[]</td>
</tr>
<tr>
<td>'Color'</td>
<td>DefaultUicontrolBackgroundColor</td>
</tr>
<tr>
<td>'DockControls'</td>
<td>'off'</td>
</tr>
<tr>
<td>'HandleVisibility'</td>
<td>'callback'</td>
</tr>
<tr>
<td>'IntegerHandle'</td>
<td>'off'</td>
</tr>
<tr>
<td>'InvertHardcopy'</td>
<td>'off'</td>
</tr>
<tr>
<td>'MenuBar'</td>
<td>'none'</td>
</tr>
<tr>
<td>'NumberTitle'</td>
<td>'off'</td>
</tr>
<tr>
<td>'PaperPositionMode'</td>
<td>'auto'</td>
</tr>
<tr>
<td>'Resize'</td>
<td>'off'</td>
</tr>
<tr>
<td>'Visible'</td>
<td>'on'</td>
</tr>
<tr>
<td>'WindowStyle'</td>
<td>'modal'</td>
</tr>
</tbody>
</table>
**Note** By default, the dialog box is modal. A modal dialog box prevents the user from interacting with other windows before responding. For more information, see **WindowStyle** in the MATLAB Figure Properties.

The default ButtonDownFcn, if `isempty(allchild(gcf))`, `close(gcf)`, end, causes the dialog to terminate itself when clicked as long as it contains no child objects. Replace it with another callback or an empty callback if you do not want this behavior. You can do this only from a script or function if the dialog is modal (the default WindowStyle).

Any parameter from the **figure** function is valid for this function.

**Examples**

```MATLAB
out = dialog;
out = dialog('WindowStyle', 'normal', 'Name', 'My Dialog');
```

**See Also**

`errordlg`, `helpdlg`, `inputdlg`, `listdlg`, `msgbox`, `questdlg`, `warndlg`

`figure`, `uiwait`, `uiresume`

“Predefined Dialog Boxes” on page 1-110 for related functions
Purpose
Save session to file

Syntax
```
diary
diary('filename')
diary off
diary on
```

Description
The `diary` function creates a log of keyboard input and the resulting text output, with some exceptions (see “Remarks” on page 2-977 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`, the MATLAB software 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
str = sprintf('%s%s', ... 
   '<a href="matlab:magic(4)">', ... 
   'Generate magic square</a>'); 
disp(str)
```

MATLAB displays

`Generate magic square`

However, the diary file, when viewed in a text editor, shows

```matlab
str = sprintf('%s%s', ... 
   '<a href="matlab:magic(4)">', ... 
   'Generate magic square</a>'); 
disp(str)
<a href="matlab:magic(4)">Generate magic square</a>
```

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**

`evalc`

“Command History Window” in the MATLAB Desktop Tools and Development Environment 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:

\[
\begin{bmatrix}
X(2)-X(1) & X(3)-X(2) & \ldots & X(n)-X(n-1)
\end{bmatrix}
\]

If \( X \) is a matrix, then \( \text{diff}(X) \) returns a matrix of row differences:

\[
\begin{bmatrix}
X(2:m,:)-X(1:m-1,:)
\end{bmatrix}
\]

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.

\[
\begin{align*}
x &= [1 \ 2 \ 3 \ 4 \ 5] \\
y &= \text{diff}(x) \\
y &= \\
&\begin{bmatrix}
1 & 1 & 1 & 1 & 1
\end{bmatrix}
\end{align*}
\]
\[ z = \text{diff}(x, 2) \]
\[ z = \]
\[ 0 \quad 0 \quad 0 \]

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
Calculate diffuse reflectance

Syntax
R = diffuse(Nx,Ny,Nz,S)

Description
R = diffuse(Nx,Ny,Nz,S) returns the reflectance of a surface with normal vector components [Nx,Ny,Nz]. S specifies the direction to the light source. You can specify these directions as three vectors [x,y,z] or two vectors [Theta Phi (in spherical coordinates)].

Lambert’s Law: R = cos(PSI) where PSI is the angle between the surface normal and light source.

See Also
specular, surfnorm, surfl

“Lighting as a Visualization Tool”
**Purpose**
Directory listing

**GUI Alternatives**
As an alternative to the `dir` function, you can use the Current Directory browser to view directory contents.

**Syntax**

```
dir
dir name
files = dir('dirname')
```

**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 path or file name, or can include both. You can use absolute and relative path names and wildcards (*).

`files = dir('dirname')` 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 listed here.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>File name</td>
<td>char array</td>
</tr>
<tr>
<td>date</td>
<td>Modification date timestamp</td>
<td>char array</td>
</tr>
<tr>
<td>bytes</td>
<td>Number of bytes allocated to the file</td>
<td>double</td>
</tr>
<tr>
<td>isdir</td>
<td>1 if name is a directory; 0 if not</td>
<td>logical</td>
</tr>
<tr>
<td>datenum</td>
<td>Modification date as serial date number. This value is locale-dependent.</td>
<td>double</td>
</tr>
</tbody>
</table>
Remarks

Listing Drives

On Microsoft Windows platforms, you can obtain a list of available drives using the DOS `net use` command. In the Command Window, run

```matlab
dos('net use')
```

Or run

```matlab
[s,r] = dos('net use')
```

to return the results to the character array `r`.

DOS File Names

The MATLAB `dir` function is consistent with the Microsoft Windows operating system `dir` command in that both support short file names generated by DOS. For example, both of the following commands are equivalent in both Windows and MATLAB:

```matlab
dir long_matlab_mfile_name.m
```

```matlab
dir long_m~1.m
```

Structure Results for Nonexistent Files

When you run `dir` with an output argument and the results include a nonexistent file or a file that `dir` cannot query for some other reason, `dir` returns the following default values:

```matlab
date: ''
bytes: []
isdir: 0
datenum: []
```
The most common occurrence is on UNIX® platforms when dir queries a file that is a symbolic link and the symbolic link points to a nonexistent target. A nonexistent target is when a target has been moved, removed, or renamed. For example, if my_file in my_dir is a symbolic link to another file that has been deleted, then running

\[ r = \text{dir('my_dir')} \]

includes this result for my_file:

\[
\begin{align*}
\text{r(n)} = \\
\text{name: 'my_file'} \\
\text{date: ''} \\
\text{bytes: []} \\
\text{isdir: 0} \\
\text{datenum: []}
\end{align*}
\]

where \( n \) is the index for my_file, found by searching \( r \) by the name field. See also the example “Excluding Files That Cannot Be Queried” on page 2-986

**Examples**

**Listing Directory Contents**

To view the contents of the matlab/audiovideo directory, type

\[ \text{dir(fullfile(matlabroot, 'toolbox/matlab/audiovideo'))} \]

**Using Wildcard and File Extension**

To view the MAT-files in your current working directory that include the term java, type

\[ \text{dir *java*.mat} \]

MATLAB returns all file names that match this specification:

\[ \text{java_array.mat javafrmobj.mat testjava.mat} \]

5. UNIX is a registered trademark of The Open Group in the United States and other countries.
Using a Relative Path Name

To view the M-files in the MATLAB audiovideo directory, type

```
dir(fullfile(matlabroot,'toolbox/matlab/audiovideo/*.m'))
```

MATLAB returns

```
Contents.m    aviinfo.m    render_uimgraudiotoolbar.m
audiodevinfo.m aviread.m    sound.m
audioplayerreg.m lin2mu.m    soundsc.m
audiorecorderreg.m mmcompinfo.m wavinfo.m
audiouniquename.m mmfileinfo.m wavplay.m
aufinfo.m     movie2avi.m    wavread.m
auread.m      mu2lin.m       wavrecord.m
auwrite.m     prefspanel.m   wavwrite.m
avifinfo.m    render_fullaudiotoolbar.m
```

Returning File List to Structure

To return the list of files to the variable `av_files`, type

```
av_files = dir(fullfile(matlabroot, ...'
    'toolbox/matlab/audiovideo/*.m'))
```

MATLAB returns the information in a structure array:

```
av_files =
24x1 struct array with fields:
    name
    date
    bytes
    isdir
    datenum
```

Index into the structure to access a particular item. For example:

```
av_files(3).name
ans =
    audioplayerreg.m
```
Excluding Files That Cannot Be Queried

To return the list of files excluding those that cannot be queried, run the following:

```matlab
y = dir;
y = y(find(~cellfun(@isempty,{y(:).date})));```

See Also

cd, copyfile, delete, fileattrib, filebrowser, fileparts, genpath, isdir, ls, matlabroot, mkdir, mfilename, movefile, rmdir, type, what

“Managing Files and Working with the Current Directory”
Purpose
Directory contents on FTP server

Syntax
\[ \text{dir}(f, 'dirname') \]
\[ d = \text{dir}(...) \]

Description
\text{dir}(f, 'dirname') lists the files in the specified directory, \textit{dirname}, on the FTP server \textit{f}, where \textit{f} was created using \texttt{ftp}. If \textit{dirname} is unspecified, \text{dir} lists the files in the current directory of \textit{f}.

\[ d = \text{dir}(...) \] returns the results in an m-by-1 structure with the following fields for each file:

<table>
<thead>
<tr>
<th>Fieldname</th>
<th>Description</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Filename</td>
<td>char array</td>
</tr>
<tr>
<td>date</td>
<td>Modification date timestamp</td>
<td>char array</td>
</tr>
<tr>
<td>bytes</td>
<td>Number of bytes allocated to the file</td>
<td>double</td>
</tr>
<tr>
<td>isdir</td>
<td>1 if name is a directory; 0 if not</td>
<td>logical</td>
</tr>
<tr>
<td>datenum</td>
<td>Modification date as serial date number</td>
<td>char array</td>
</tr>
</tbody>
</table>

Examples
Connect to the MathWorks FTP server and view the contents.

\[ \text{tmw}=\text{ftp}('ftp.mathworks.com'); \]
\[ \text{dir(tmw)} \]

\[ \text{README incoming matlab outgoing pub pubs} \]

Change to the directory pub/pentium.

\[ \text{cd(tmw, 'pub/pentium')} \]
View the contents of that directory.

```matlab
dir(tmw)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Bytes</th>
<th>Isdir</th>
<th>DataNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel_resp.txt</td>
<td>NYT_2.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel_support.txt</td>
<td>NYT_Dec14.uu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Andy_Grove.txt</td>
<td>Intel_white.ps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Associated_Press.txt</td>
<td>MathWorks_press.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNN.html</td>
<td>Mathisen.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coe.txt</td>
<td>Moler_1.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cygnus.txt</td>
<td>Moler_2.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EE_Times.txt</td>
<td>Moler_3.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAQ.txt</td>
<td>Moler_4.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM_study.txt</td>
<td>Moler_5.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel_FAX.txt</td>
<td>Moler_6.ps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel_fix.txt</td>
<td>Moler_7.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel_replace.txt</td>
<td>Myths.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p87test.txt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p87test.zip</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Or return the results to the structure `m`.

```matlab
m=dir(tmw)
```

```
m =
37x1 struct array with fields:
    name
    date
    bytes
    isdir
    datanum
```

View element 17.

```matlab
m(17)
```

```
ans =
    name: 'Moler_1.txt'
```
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

Example 1 — Display a matrix with column labels

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

<table>
<thead>
<tr>
<th>Corn</th>
<th>Oats</th>
<th>Hay</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2113</td>
<td>0.8474</td>
<td>0.2749</td>
</tr>
<tr>
<td>0.0820</td>
<td>0.4524</td>
<td>0.8807</td>
</tr>
<tr>
<td>0.7599</td>
<td>0.8075</td>
<td>0.6538</td>
</tr>
<tr>
<td>0.0087</td>
<td>0.4832</td>
<td>0.4899</td>
</tr>
<tr>
<td>0.8096</td>
<td>0.6135</td>
<td>0.7741</td>
</tr>
</tbody>
</table>

Example 2 — Display a hyperlink in the Command Window

You also can use the disp command to display a hyperlink in the Command Window. Include the full hypertext string on a single line as input to disp:

```
disp('<a href = "http://www.mathworks.com">The MathWorks Web Site</a>')
```

which generates this hyperlink in the Command Window:

The MathWorks Web Site
Click the link to display The MathWorks home page in a MATLAB Web browser.

**Example 3 — Display multiple items on the same line**

Use concatenation to display multiple items using `disp`. For example:

```matlab
x = [1 2 3];
disp(['The values of x are: ', num2str(x)]);
```

displays

The values of x are: 1 2 3

If you want to display text without a trailing newline character, use `fprintf`. For example,

```matlab
fprintf('%s %d %d %d ', 'The values of x are:', x(:));
```

displays text similar to the above, but does not include a newline.

**See Also**

`format`, `int2str`, `matlabcolon`, `num2str`, `rats`, `sprintf`, `fprintf`
Purpose
Information about memmapfile object

Syntax
disp(obj)

Description
disp(obj) displays all properties and their values for memmapfile object obj.

The MATLAB software also displays this information when you construct a memmapfile object or set any of the object’s property values, provided you do not terminate the command to do so with a semicolon.

Examples
Construct an object m of class memmapfile:

```
m = memmapfile('records.dat', ... 
    'Offset', 2048, 
    'Format', { 
        'int16' [2 2] 'model'; 
        'uint32' [1 1] 'serialno'; 
        'single' [1 3] 'expenses'});
```

Use disp to display all the object’s current properties:

```
disp(m)
    Filename: 'd:\matlab\mfiles\records.dat'
   Writable: false
     Offset: 2048
      Format: {'int16' [2 2] 'model'
                'uint32' [1 1] 'serialno'
                'single' [1 3] 'expenses'}
     Repeat: Inf
       Data: 753x1 struct array with fields:
               model
               serialno
               expenses
```

See Also
memmapfile, get(memmapfile)
Purpose
Display MException object

Syntax
disp(ME)
disp(ME.property)

Description
disp(ME) displays all properties (fields) of MException object ME.
disp(ME.property) displays the specified property of MException object ME.

Examples
Using the surf command without input arguments throws an exception. Use disp to display the identifier, message, stack, and cause properties of the MException object:

```matlab
try
    surf
catch ME
    disp(ME)
end
```

MException object with properties:

- identifier: 'MATLAB:nargchk:notEnoughInputs'
- message: 'Not enough input arguments.'
- stack: [1x1 struct]
- cause: {}

Display only the stack property:

```matlab
disp(ME.stack)
```

file: 'X:\bat\Akern\perfect\matlab\toolbox\matlab\graph3d\surf.m'
- name: 'surf'
- line: 54

See Also
try, catch, error, assert, MException, getReport(MException), throw(MException), rethrow(MException),
disp (MException)

throwAsCaller(MException), addCause(MException),
isequal(MException), eq(MException), ne(MException),
last(MException),

2-994
### Purpose
Serial port object summary information

### Syntax

```
obj
disp(obj)
```

### Description

`obj` or `disp(obj)` displays summary information for `obj`, a serial port object or an array of serial port objects.

### Remarks
In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when:

- Creating a serial port object
- Configuring property values using the dot notation

Use the display summary to quickly view the communication settings, communication state information, and information associated with read and write operations.

### Example
The following commands display summary information for the serial port object `s` on a Windows platform

```
s = serial('COM1')
s.BaudRate = 300
s
```
**Purpose**  
Information about timer object

**Syntax**  
disp(obj)

**Description**  
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.

obj, that is, typing the object name alone, does the same as disp(obj)

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.

```matlab
  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.

```matlab
t2 = timer;
disp(timerfind)
```

Timer Object Array

<table>
<thead>
<tr>
<th>Index</th>
<th>ExecutionMode</th>
<th>Period</th>
<th>TimerFcn</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>singleShot</td>
<td>1</td>
<td>''</td>
<td>timer-1</td>
</tr>
<tr>
<td>2</td>
<td>singleShot</td>
<td>1</td>
<td>''</td>
<td>timer-2</td>
</tr>
</tbody>
</table>

See Also

`timer`, `get(timer)`
Purpose
Display text or array (overloaded method)

Syntax
display(X)

Description
display(X) prints the value of a variable or expression, X. The MATLAB software 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 \( \text{magic}(3) \), with no terminating semicolon, calls this function as \( \text{display(magic(3))} \).

```matlab
magic(3)
```

```plaintext
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.

```matlab
function display(p)
    % POLYNOM/DISPLAY Command window display of a polynom
    disp(' ');  
    disp([inputname(1),' = ']);
    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
**Purpose**
Convert image, increasing apparent color resolution by dithering

**Syntax**

\[
X = \text{dither}(\text{RGB}, \text{map}) \\
X = \text{dither}(\text{RGB}, \text{map}, Qm, Qe) \\
\text{BW} = \text{dither}(\text{I})
\]

**Description**

\[
X = \text{dither}(\text{RGB}, \text{map}) \text{ creates an indexed image approximation of the RGB image in the array RGB by dithering the colors in the colormap map. The colormap cannot have more than } 65,536 \text{ colors.}
\]

\[
X = \text{dither}(\text{RGB}, \text{map}, Qm, Qe) \text{ creates an indexed image from RGB, where } Qm \text{ specifies the number of quantization bits to use along each color axis for the inverse color map, and } Qe \text{ specifies the number of quantization bits to use for the color space error calculations. If } Qe < Qm, \text{ dithering cannot be performed, and an undithered indexed image is returned in } X. \text{ If you omit these parameters, } \text{dither} \text{ uses the default values } Qm = 5, Qe = 8.
\]

\[
\text{BW} = \text{dither}(\text{I}) \text{ converts the grayscale image in the matrix } \text{I to the binary (black and white) image } \text{BW by dithering.}
\]

**Class Support**

RGB can be uint8, uint16, single, or double. I can be uint8, uint16, int16, single, or double. All other input arguments must be double. BW is logical. X is uint8, if it is an indexed image with 256 or fewer colors; otherwise, it is uint16.

**Algorithm**

dither increases the apparent color resolution of an image by applying Floyd-Steinberg’s error diffusion dither algorithm.

**Examples**

Convert intensity image to binary using dithering.

\[
\begin{align*}
\text{I} &= \text{imread}('\text{cameraman.tif}'); \\
\text{BW} &= \text{dither}(\text{I}); \\
\text{imshow}(&\text{I}), \text{ figure, imshow}(\text{BW})
\end{align*}
\]
References


See Also
rgb2ind
divergence

Purpose
Compute divergence of vector field

Syntax
\[
\text{div} = \text{divergence}(X,Y,Z,U,V,W) \\
\text{div} = \text{divergence}(U,V,W) \\
\text{div} = \text{divergence}(X,Y,U,V) \\
\text{div} = \text{divergence}(U,V)
\]

Description
\[
\text{div} = \text{divergence}(X,Y,Z,U,V,W) \text{ computes the divergence of a 3-D vector field } U, V, W. \text{ The arrays } X, Y, Z \text{ define the coordinates for } U, V, W \text{ and must be monotonic and 3-D plaid (as if produced by } \text{meshgrid}).
\]
\[
\text{div} = \text{divergence}(U,V,W) \text{ assumes } X, Y, \text{ and } Z \text{ are determined by the expression}
\]
\[
[X \ Y \ Z] = \text{meshgrid}(1:n,1:m,1:p)
\]
\[
\text{where } [m,n,p] = \text{size}(U).
\]
\[
\text{div} = \text{divergence}(X,Y,U,V) \text{ computes the divergence of a 2-D vector field } U, V. \text{ The arrays } X, Y \text{ define the coordinates for } U, V \text{ and must be monotonic and 2-D plaid (as if produced by } \text{meshgrid}).
\]
\[
\text{div} = \text{divergence}(U,V) \text{ assumes } X \text{ and } Y \text{ are determined by the expression}
\]
\[
[X \ Y] = \text{meshgrid}(1:n,1:m)
\]
\[
\text{where } [m,n] = \text{size}(U).
\]

Examples
This example displays the divergence of vector volume data as slice planes, using color to indicate divergence.

\[
\text{load wind} \\
\text{div} = \text{divergence}(x,y,z,u,v,w); \\
\text{slice}(x,y,z,\text{div},[90 \ 134],[59],[0]); \\
\text{shading interp} \\
\text{daspect([1 1 1])} \\
\text{camlight}
\]
See Also

streamtube, curl, isosurface

“Volume Visualization” on page 1-108 for related functions

“Example – Displaying Divergence with Stream Tubes” for another example
Purpose
Read ASCII-delimited file of numeric data into 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 from the ASCII-delimited numeric data file filename to output matrix M. The filename input is a string enclosed in single quotes. The delimiter separating data elements is 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 \t to specify a tab delimiter.

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.
**Note** `dlmread` reads numeric data only. The file being read may contain nonnumeric data, but this nonnumeric data cannot be within the range being imported.

\[ M = dlmread(filename, delimiter, range) \]

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'`.

**Remarks**

If you want to specify an \(R, C,\) or range input, but not a `delimiter`, set the `delimiter` argument to the empty string, (two consecutive single quotes with no spaces in between, `''`). For example,

\[ M = dlmread('myfile.dat', '', 5, 2) \]

Using this syntax enables you to specify the starting row and column or range to read while having `dlmread` treat repeated whitespaces as a single delimiter.

`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;real&gt;–&lt;imag&gt;i</td>
<td>5.7-3.1i</td>
</tr>
<tr>
<td>–&lt;imag&gt;i</td>
<td>-7j</td>
</tr>
</tbody>
</table>

Embedded white-space in a complex number is invalid and is regarded as a field delimiter.
Examples

Example 1

Export the 5-by-8 matrix \( M \) to a file, and read it with \texttt{dlmread}, first with no arguments other than the filename:

\[
\begin{align*}
\text{rand('state', 0); } & \quad \text{M = rand(5,8); } \quad \text{M = floor(M * 100);} \\
\text{dlmwrite('myfile.txt', M, 'delimiter', '\t')} & \\
\text{dlmread('myfile.txt')} \\
\text{ans = } & \\
& \begin{bmatrix}
  95 & 76 & 61 & 40 & 5 & 20 & 1 & 41 \\
  23 & 45 & 79 & 93 & 35 & 19 & 74 & 84 \\
  60 & 1 & 92 & 91 & 81 & 60 & 44 & 52 \\
  48 & 82 & 73 & 41 & 0 & 27 & 93 & 20 \\
  89 & 44 & 17 & 89 & 13 & 19 & 46 & 67
\end{bmatrix}
\end{align*}
\]

Now read a portion of the matrix by specifying the row and column of the upper left corner:

\[
\begin{align*}
\text{dlmread('myfile.txt', '\t', 2, 3)} & \\
\text{ans = } & \\
& \begin{bmatrix}
  91 & 81 & 60 & 44 & 52 \\
  41 & 0 & 27 & 93 & 20 \\
  89 & 13 & 19 & 46 & 67
\end{bmatrix}
\end{align*}
\]

This time, read a different part of the matrix using a range specifier:

\[
\begin{align*}
\text{dlmread('myfile.txt', '\t', 'C1..G4')} & \\
\text{ans = } & \\
& \begin{bmatrix}
  61 & 40 & 5 & 20 & 1 \\
  79 & 93 & 35 & 19 & 74 \\
  92 & 91 & 81 & 60 & 44 \\
  73 & 41 & 0 & 27 & 93
\end{bmatrix}
\end{align*}
\]

Example 2

Export matrix \( M \) to a file, and then append an additional matrix to the file that is offset one row below the first:

\[
\begin{align*}
M = \text{magic(3)};
\end{align*}
\]
dlmwrite('myfile.txt', [M*5 M/5], ' ')

dlmwrite('myfile.txt', rand(3), '-append', ...
     '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

When dlmread imports these two matrices from the file, it pads the smaller matrix with zeros:

dlmand('myfile.txt')
    40.0000  5.0000  30.0000  1.6000  0.2000  1.2000
    15.0000  25.0000  35.0000  0.6000  1.0000  1.4000
    20.0000  45.0000  10.0000  0.8000  1.8000  0.4000
    0.6038  0.0153  0.9318  0  0  0
    0.2722  0.7468  0.4660  0  0  0
    0.1988  0.4451  0.4187  0  0  0

See Also

dlmwrite, textscan, csvread, csvwrite, wk1read, wk1write
dlmwrite

**Purpose**
Write matrix to ASCII-delimited file

**Syntax**
- `dlmwrite(filename, M)`
- `dlmwrite(filename, M, 'D')`
- `dlmwrite(filename, M, 'D', R, C)`
- `dlmwrite(filename, M, 'attrib1', value1, 'attrib2', 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`. The `filename` input is a string enclosed in single quotes.

`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>Attribute</td>
<td>Value</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</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>
<tr>
<td>coffset</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>precision</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>

This table shows which values you can use when setting the **newline** attribute.

<table>
<thead>
<tr>
<th>Line Terminator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'pc'</td>
<td>PC terminator (implies carriage return/line feed (CR/LF))</td>
</tr>
<tr>
<td>'unix'</td>
<td>UNIX terminator (implies line feed (LF))</td>
</tr>
</tbody>
</table>

dlmwrite(filename, M, '-append') appends the matrix to the file. If you do not specify '-append', dlmwrite overwrites any existing data in the file.

dlmwrite(filename, M, '-append', attribute-value list) is the same as the syntax shown above, but accepts a list of attribute-value pairs. You can place the '-append' flag in the argument list anywhere between attribute-value pairs, but not in between an attribute and its value.

**Remarks**

The resulting file is readable by spreadsheet programs.
The `dlmwrite` function does not accept cell arrays for the input matrix `M`. To export a cell array that contains only numeric data, use `cell2mat` to convert the cell array to a numeric matrix before calling `csvwrite`. To export cell arrays with mixed alphabetic and numeric data, where each cell contains a single element, you can create an Excel spreadsheet (if your system has Excel installed) using `xlswrite`. For all other cases, you must use low-level export functions to write your data.

**Examples**

**Example 1**

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', '	', ...
'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
```

**Example 2**

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
```

**Example 3**

Export matrix `M` to a file, and then append an additional matrix to the file that is offset one row below the first:
M = magic(3);
dlmwrite('myfile.txt', [M*5 M/5], ' ')

dlmwrite('myfile.txt', rand(3), '-append', ...
     'roffset', 1, 'delimiter', ' ')

type myfile.txt

40 5 30 1.6 0.2 1.2
15 25 35 0.6 1 1.4
20 45 10 0.8 1.8 0.4

0.81472 0.91338 0.2785
0.90579 0.63236 0.54688
0.12699 0.09754 0.95751

When dlmread imports these two matrices from the file, it pads the smaller matrix with zeros:

dlmread('myfile.txt')
40.0000  5.0000  30.0000  1.6000  0.2000  1.2000
15.0000  25.0000  35.0000  0.6000  1.0000  1.4000
20.0000  45.0000  10.0000  0.8000  1.8000  0.4000
0.8147  0.9134  0.2785  0  0  0
0.9058  0.6324  0.5469  0  0  0
0.1270  0.0975  0.9575  0  0  0

See Also
dlmread, csvwrite, csvread, fileformats
**Purpose**
Dulmage-Mendelsohn decomposition

**Syntax**
\[ p = \text{dmperm}(A) \]
\[ [p, q, r, s, cc, rr] = \text{dmperm}(A) \]

**Description**
\[ p = \text{dmperm}(A) \] finds a vector \( p \) such that \( p(j) = i \) if column \( j \) is matched to row \( i \), or zero if column \( j \) is unmatched. If \( A \) is a square matrix with full structural rank, \( p \) is a maximum matching row permutation and \( A(p,:) \) has a zero-free diagonal. The structural rank of \( A \) is \( \text{sprank}(A) = \text{sum}(p>0) \).

\[ [p, q, r, s, cc, rr] = \text{dmperm}(A) \] where \( A \) need not be square or full structural rank, finds the Dulmage-Mendelsohn decomposition of \( A \). \( p \) and \( q \) are row and column permutation vectors, respectively, such that \( A(p,q) \) has a block upper triangular form. \( r \) and \( s \) are index vectors indicating the block boundaries for the fine decomposition. \( cc \) and \( rr \) are vectors of length five indicating the block boundaries of the coarse decomposition.

\( C = A(p,q) \) is split into a 4-by-4 set of coarse blocks:

\[
\begin{bmatrix}
A_{11} & A_{12} & A_{13} & A_{14} \\
0 & 0 & A_{23} & A_{24} \\
0 & 0 & 0 & A_{34} \\
0 & 0 & 0 & A_{44}
\end{bmatrix}
\]

where \( A_{12} \), \( A_{23} \), and \( A_{34} \) are square with zero-free diagonals. The columns of \( A_{11} \) are the unmatched columns, and the rows of \( A_{44} \) are the unmatched rows. Any of these blocks can be empty. In the coarse decomposition, the \((i,j)\)th block is \( C(\text{rr}(i):\text{rr}(i+1)-1,\text{cc}(j):\text{cc}(j+1)-1) \). For a linear system,

- \([A_{11} \ A_{12}]\) is the underdetermined part of the system—it is always rectangular and with more columns and rows, or 0-by-0,
- \(A_{23}\) is the well-determined part of the system—it is always square, and
[A34 ; A44] is the overdetermined part of the system—it is always rectangular with more rows than columns, or 0-by-0.

The structural rank of $A$ is $\text{sprank}(A) = \text{rr}(4) - 1$, which is an upper bound on the numerical rank of $A$. $\text{sprank}(A) = \text{rank}(\text{full}(\text{sprand}(A)))$ with probability 1 in exact arithmetic.

The $A_{23}$ submatrix is further subdivided into block upper triangular form via the fine decomposition (the strongly connected components of $A_{23}$). If $A$ is square and structurally nonsingular, $A_{23}$ is the entire matrix.

$C(r(i):r(i+1)-1,s(j):s(j+1)-1)$ is the $(i,j)$th block of the fine decomposition. The $(1,1)$ block is the rectangular block $[A_{11} A_{12}]$, unless this block is 0-by-0. The $(b,b)$ block is the rectangular block $[A_{34} ; A_{44}]$, unless this block is 0-by-0, where $b = \text{length}(r) - 1$. All other blocks of the form $C(r(i):r(i+1)-1,s(i):s(i+1)-1)$ are diagonal blocks of $A_{23}$, and are square with a zero-free diagonal.

**Remarks**

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].

$\text{dmperm}$ uses CSparse [2].

**References**


See Also  
sprank
Purpose
Reference page in Help browser

GUI Alternatives
As an alternative to the `doc` function, use the Help browser search field. Type the function name and press Enter.

Syntax
```
doc
doc functionname
doc toolboxdirname
doc toolboxdirname/functionname
doc classname.methodname
doc userclassname
```

Description
doc opens the Help browser, if it is not already running, or brings the window to the top, displaying the Contents pane when the Help browser 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. Here `functionname` can be a function, block, property, method, or object. If `functionname` is overloaded, that is, if `functionname` appears in multiple directories on the search path MATLAB uses, `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. Overloaded functions within the same product are not listed — use the overloaddirectory form of the syntax. If a reference page for `functionname` does not exist, `doc` displays its M-file help in the Help browser. The `doc` function is intended only for help files supplied by The MathWorks, and is not supported for use with HTML files you create yourself; to display HTML files for functions you create, use the `web` function.

`doc toolboxdirname` displays the roadmap page for `toolboxdirname` in the Help browser, which provides a summary of the most pertinent documentation for that product located in `toolboxdirname`. `toolboxdirname` is the directory name for a product in `matlabroot/toolbox`. If you do not know `toolboxdirname` for a
product, run which functionname, where functionname is the name of a function in that product; MATLAB returns the full path to functionname, and toolboxdirname is the directory after matlabroot/toolbox.

doc toolboxdirname/functionname displays the reference page for the functionname that belongs to the specified toolboxdirname, in the Help browser. This is useful for overloaded functions.

doc classname.methodname displays the reference page for the methodname that is a member of classname.

doc userclassname displays the help comments from the user-created class definition file, userclassname.m, in an HTML format in the Help browser. userclassname.m must have a help comment following the classdef userclassname statement or following the constructor method for userclassname. To directly view the help for any method, property, or event of userclassname, use dot notation, as in doc userclassname.methodname. For more information, see “Help for User-Created Classes”.

**Note** If there is a function called name as well as a toolbox directory called name, the roadmap page for the toolbox directory called name displays. To see the reference page for the function called name, use doc toolboxdirname/name, where toolboxdirname is the name of the toolbox directory in which the function name resides. For example, doc matlab displays the roadmap page for MATLAB (that is, the matlab toolbox directory), while doc matlab/matlabunix displays the reference page for the matlab startup function for UNIX® platforms, which is in the MATLAB product.

**Examples**

Run doc abs to display the reference page for the abs function. If the Simulink and Signal Processing Toolbox™ products are installed and

6. UNIX is a registered trademark of The Open Group in the United States and other countries.
on the search path, the Command Window lists hyperlinks for the \texttt{abs} function in those products:

\begin{verbatim}
doc signal/abs
doc simulink/abs
\end{verbatim}

Run \texttt{doc signal/abs} to display the reference page for the \texttt{abs} function in the Signal Processing Toolbox product.

Run \texttt{doc signal} to display the roadmap page for Signal Processing Toolbox product.

Run \texttt{doc serial.get} to display the reference page for the \texttt{get} method located in the \texttt{serial} directory of MATLAB. This syntax is required because there is at least one other \texttt{get} function in MATLAB.

Run \texttt{doc sads} to display the help comments in the \texttt{sads.m} class definition file for the user-created \texttt{sads} class. Run \texttt{doc sads.steer} to go directly to help for the \texttt{steer} method of the user-created \texttt{sads} class. Run \texttt{sads.Spacing} to go directly to help for the \texttt{Spacing} property of the user-created \texttt{sads} class.

\section*{See Also}

\texttt{docopt, docsearch, help, helpbrowser, lookfor, type, web}

Related topics in the MATLAB Desktop Tools and Development Environment documentation:

- “Assistance While Entering Statements”, including function hints and the Function Browser
- “Help, Demos, and Related Resources”
- “Help for the Files You and Other Users Create”
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Web browser for UNIX platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Note</strong></td>
<td>docopt produces a warning and will be removed in a future version. Use Web preferences instead, by selecting <code>File &gt; Web &gt; Preferences</code>. For more information, see “Web Preferences”.</td>
</tr>
<tr>
<td>Syntax</td>
<td><code>docopt</code> <code>doccmd = docopt</code></td>
</tr>
</tbody>
</table>
| Description | `docopt` displays the Web browser used with the MATLAB software when running on UNIX\(^7\) platforms, except for the Apple Macintosh platform, with the default being *netscape* (for the Netscape Navigator\(^\circledast\) application). For UNIX platforms (other than the Macintosh platform), you can modify the `docopt.m` file to specify the Web browser that 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. (To locate `docopt.m`, run `which docopt`.) Here is an example of changing `docopt.m`. Initially, the file contains: 

```
50 elseif isunix % UNIX
51 %  doccmd = '';
```

Remove the comment symbol in line 51. Inside the quotation marks, enter the command that starts your Web browser, and save the file. For example, 

\(^7\) UNIX is a registered trademark of The Open Group in the United States and other countries.
doccmd = 'mozilla';

specifies Mozilla® as the Web browser MATLAB uses.

See Also  doc, edit, helpbrowser, web
Purpose

Open Help browser and search for specified term

GUI Alternatives

As an alternative to the `docsearch` function, select Desktop > Help. In the search field, type the words you want to find, and press Enter.

Syntax

docsearch

docsearch word

docsearch word1 word2 ...

docsearch "word1 word2" ...

docsearch wo*rd ...

docsearch word1 word2 BOOLEANOP word3

docsearch('word1 word2')

docsearch(charvar)

Description

docsearch opens the Help browser to the Search Results pane, or if the Help browser is already open to that pane, brings it to the top.

docsearch word executes a Help browser full-text search for word, displaying results in the Help browser Search Results pane. If word is a functionname or blockname, the first entry in Search Results is its reference page.

docsearch word1 word2 ... executes a Help browser full-text search for pages containing word1, word2, and any other specified words, displaying results in the Help browser Search Results pane.

docsearch "word1 word2" ... executes a Help browser full-text search for pages containing the exact phrase word1 word2 and any other specified words, displaying results in the Help browser Search Results pane.

docsearch wo*rd ... executes a Help browser full-text search for pages containing words that begin with wo and end with rd, and any other specified words, displaying results in the Help browser Search Results pane. This is also called a wildcard or partial word search. You can use a wildcard symbol (*) multiple times within a word. You cannot use the wildcard symbol within an exact phrase. You must use at least two letters or digits with a wildcard symbol.
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 refine the search. docsearch evaluates NOTs first, then ORs, and finally ANDs. Results display in the Help browser Search Results pane.

docsearch('word1 word2') is the function form of the syntax. The function form supports all options.

docsearch(charvar) finds all pages containing the string defined in charvar, where charvar is a variable of the char class.

Examples

docsearch plot finds all pages that contain the word plot.

docsearch plot tools finds all pages that contain the words plot and the word tools anywhere in the page.

docsearch "plot tools" finds all pages that contain the exact phrase plot tools.

docsearch plot* tools finds all pages that contain the word tools and the word plot or variations of plot, such as plotting, and plots.

docsearch "plot tools" NOT "time series" finds all pages that contain the exact phrase plot tools, but only if the pages do not contain the exact phrase time series.

docsearch(m), where m='plot tools', finds all pages that contain the word plot and the word tools anywhere in the page.

docsearch('plot tools'), finds all pages that contain the word plot and the word tools anywhere in the page.

See Also

builddocsearchdb, doc, helpbrowser, lookfor

Related topics in the MATLAB Desktop Tools and Development Environment documentation:

- “”
- “Finding Functions Using the Function Browser”
• “Help, Demos, and Related Resources”
**Purpose**
Execute DOS command and return result

**Syntax**

dos command
status = dos('command')
[status,result] = dos('command')
[status,result] = dos('command','-echo')

**Description**
dos command calls upon the shell to execute the given command for Microsoft Windows platforms.

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, the MATLAB software always waits 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.
Note Running dos with a command that relies upon the current directory will fail when the current directory is specified using a UNC pathname. This is because DOS does not support UNC pathnames. In that event, MATLAB returns this error: ??? Error using ==> dos DOS commands may not be executed when the current directory is a UNC pathname. To work around this limitation, change the directory to a mapped drive prior to running dos or a function that calls dos.

Examples

The following example performs a directory listing, returning a zero (success) in s and the string containing the listing in w.

```matlab
[s, w] = dos('dir');
```

To open the DOS 5.0 editor in a DOS window

```matlab
dos('edit &')
```

To open the Microsoft Notepad editor and return control immediately to MATLAB, run

```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

“Running External Programs” in the MATLAB Desktop Tools and Development Environment documentation
**Purpose**
Vector dot product

**Syntax**

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

**Description**

\[ C = \text{dot}(A,B) \]
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, \( \text{dot}(A,B) \) is the same as \( A'*B \).

For multidimensional arrays \( A \) and \( B \), \( \text{dot} \) returns the scalar product along the first non-singleton dimension of \( A \) and \( B \). \( A \) and \( B \) must have the same size.

\[ C = \text{dot}(A,B,\text{dim}) \]
returns the scalar product of \( A \) and \( B \) in the dimension \( \text{dim} \).

**Examples**
The dot product of two vectors is calculated as shown:

\[
\begin{align*}
a &= [1 \ 2 \ 3]; \\
b &= [4 \ 5 \ 6]; \\
c &= \text{dot}(a,b)
\end{align*}
\]

\[
\begin{align*}
c &= 32
\end{align*}
\]

**See Also**
cross
Purpose
Convert to double precision

Syntax
double(x)

Description
double(x) returns the double-precision value for X. If X is already a double-precision array, double has no effect.

Remarks
double is called for the expressions in for, if, and while loops if the expression isn’t already double-precision. double should be overloaded for any object when it makes sense to convert it to a double-precision value.
Drag rectangles with mouse

**Syntax**

\[
[\text{finalrect}] = \text{dragrect}(\text{initialrect})
\]

\[
[\text{finalrect}] = \text{dragrect}(\text{initialrect}, \text{stepsize})
\]

**Description**

\[
[\text{finalrect}] = \text{dragrect}(\text{initialrect})
\]

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.

\[
[\text{finalrect}] = \text{dragrect}(\text{initialrect}, \text{stepsize})
\]

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.

\[
[\text{finalrect}] = \text{dragrect}(\ldots)
\]

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 a ButtonDownFcn, 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.

**Note** You cannot use normalized figure units with dragrect.
**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`

“Region of Interest” on page 1-107 for related functions
Purpose
Flush event queue and update figure window

Syntax
drawnow
drawnow expose
drawnow update

Description
drawnow causes figure windows and their children to update, and flushes the system event queue. Any callbacks generated by incoming events (e.g., mouse or key events) are dispatched before drawnow returns.
drawnow expose causes only graphics objects to refresh, if needed. It does not allow callbacks to execute and does not process other events in the queue.
drawnow update causes only non graphics objects to refresh, if needed. It does not allow callbacks to execute and does not process other events in the queue.
You can combine the expose and update options to obtain both effects:

    drawnow expose update

Other Events That Cause Event Queue Processing
Other events that cause the MATLAB software to flush the event queue and draw the figure include:

- Returning to the MATLAB prompt
- Executing the following functions:
  - figure
  - getframe
  - input
  - keyboard
  - pause
- Functions that wait for user input (i.e., waitforbuttonpress, waitfor, ginput)
Any code that causes one of the above functions to execute. For example, suppose \( h \) is the handle of an axes. Calling \( \text{axes}(h) \) causes its parent figure to be made the current figure and brought to the front of all displayed figures, which results in the event queue being flushed.

**Examples**

Using `drawnow` in a loop causes the display to update while the loop executes:

```matlab
t = 0:pi/20:2*pi;
y = exp(sin(t));
h = plot(t,y,'YDataSource','y');
for k = 1:.1:10
    y = exp(sin(t.*k));
    refreshdata(h,'caller') % Evaluate y in the function workspace
    drawnow; pause(.1)
end
```

**See Also**

`snapnow`, `waitfor`, `waitforbuttonpress`
**Purpose**  
Search Delaunay triangulation for nearest point

**Syntax**  
\[ K = \text{dsearch}(x,y,\text{TRI},x_i,y_i) \]  
\[ K = \text{dsearch}(x,y,\text{TRI},x_i,y_i,S) \]

**Description**  
\[ K = \text{dsearch}(x,y,\text{TRI},x_i,y_i) \] returns the index into \( x \) and \( y \) of the nearest point to the point \((x_i,y_i)\). \text{dsearch} requires a triangulation \( \text{TRI} \) of the points \( x,y \) obtained using \text{delaunay}. If \( x_i \) and \( y_i \) are vectors, \( K \) is a vector of the same size.

\[ K = \text{dsearch}(x,y,\text{TRI},x_i,y_i,S) \] uses the sparse matrix \( S \) instead of computing it each time:

\[ S = \text{sparse}(\text{TRI}(:,[1 1 2 2 3 3]),\text{TRI}(:,[2 3 1 3 1 2]),1,nxy,nxy) \]

where \( nxy = \text{prod(size}(x)) \).

**See Also**  
delaunay, tsearch, voronoi
**Purpose**
N-D nearest point search

**Syntax**

```matlab
k = dsearchn(X,T,XI)
k = dsearchn(X,T,XI,outval)
k = dsearchn(X,XI)
[k,d] = dsearchn(X,...)
```

**Description**

`k = 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 `delaunayn`. The output `k` is a column vector of length `p`.

`k = dsearchn(X,T,XI,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 `outval`, a scalar double. `Inf` is often used for `outval`. If `outval` is `[]`, then `k` is the same as in the case `k = dsearchn(X,T,XI)`.

`k = 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] = dsearchn(X,...)` also returns the distances `d` to the closest points. `d` is a column vector of length `p`.

**Algorithm**

`dsearchn` is based on Qhull [1]. 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**
tsearch, dsearch, tsearchn, griddatan, delaunayn

**Reference**

**Purpose**
Abstract class used to derive handle class with dynamic properties

**Syntax**
```matlab
classdef myclass < dynamicprops
```

**Description**
classdef myclass < dynamicprops makes myclass a subclass of the dynamicprops class, which is a subclass of the handle class.

Use the dynamicprops class to derive classes that can define dynamic properties (instance properties), which are associated with a specific objects, but have no effect on the objects class definition. Dynamic properties are useful for attaching temporary data to one or more objects.

**dynamicprops Methods**
This class defines one method addprop and, as a subclass of the handle class, inherits all the handle class methods.

- addprop — adds the named property to the specified handle objects. See “Dynamic Properties — Adding Properties to an Instance” for more information.

**See Also**
handle
Purpose  
Echo M-files during execution

Syntax  
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
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>echo on all</td>
<td>Sets echoing on for all function files</td>
</tr>
<tr>
<td>echo off all</td>
<td>Sets echoing off for all function files</td>
</tr>
</tbody>
</table>

**See Also**

function
**Purpose**
Run M-file demo step-by-step in Command Window

**GUI Alternatives**
As an alternative to the `echodemo` function, select the demo in the Help browser Demos tab and click the Run in the Command Window link.

**Syntax**
```
echodemo filename
```
```
echodemo('filename', cellindex)
```

**Description**
`echodemo filename` runs the M-file demo `filename` step-by-step in the Command Window. At each step, follow links in the Command Window to proceed. Depending on the size of the Command Window, you might have to scroll up to see the links. The script `filename` was created in the Editor using cells. (The associated HTML demo file for `filename` that appears in the Help browser Demos pane was created using the MATLAB cell publishing feature.) The link to `filename` also shows the current cell number, n, and the total number of cells, m, as n/m, and when clicked, opens `filename` in the Editor. To end the demo, click the Stop link.

`echodemo('filename', cellindex)` runs the M-file type demo `filename`, starting with the cell number specified by `cellindex`. Because steps prior to `cellindex` are not run, this statement might produce an error or unexpected result, depending on the demo.

**Note** M-file demos run as scripts. Therefore, the variables are part of the base workspace, which could result in problems if you have any variables of the same name. For more information, see “Running Demos and Base Workspace Variables” in the Desktop Tools and Development Environment documentation.

**Examples**
echodemo quake runs the MATLAB Loma Prieta Earthquake demo.
echodemo ('quake', 6) runs the MATLAB Loma Prieta Earthquake demo, starting at cell 6.
echodemo ('intro', 3) produces an error because cell 3 of the MATLAB demo intro requires data created when cells 1 and 2 run.

See Also
demo, helpbrowser
**TriRep.edgeAttachments**

**Purpose**
Simplices attached to specified edges

**Syntax**

```
SI = edgeAttachments(TR, V1, V2)
SI = edgeAttachments(TR, EDGE)
```

**Description**

```
SI = edgeAttachments(TR, V1, V2) returns the simplices SI
attached to the edges specified by (V1, V2). (V1, V2) represents the
start and end vertices of the edges to be queried.
```

```
SI = edgeAttachments(TR, EDGE) specifies edges in matrix format.
```

**Inputs**

```
TR
V1, V2
EDGE
```

- **TR**
  Triangulation representation.
- **V1, V2**
  Column vectors of vertex indices into the array of
  points representing the vertex coordinates.
- **EDGE**
  Matrix specifying edge start and end points. EDGE is
  of size m-by-2, m being the number of edges to query.

**Outputs**

```
SI
```

- **SI**
  Vector cell array of indices into the triangulation
  matrix. SI is a cell array because the number of
  simplices associated with each edge can vary.

**Definitions**

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

**Examples**

**Example 1**

Load a 3-D triangulation to compute the tetrahedra attached to an edge.

```
load tetmesh
trep = TriRep(tet, X);
v1 = [15 21]';
v2 = [936 716]';
t1 = edgeAttachments(trep, v1, v2);
```

You can also specify the input as edges.
```
e = [v1 v2];
t2 = edgeAttachments(trep, e);
isequal(t1,t2);
```

**Example 2**

Create a triangulation with DelaunayTri.

```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
dt = DelaunayTri(x,y);
```

Query the triangles attached to edge (1,5).

```
t = edgeAttachments(dt, 1,5);
t{:};
```

**See Also**

“Triangulation Representations”—How to query triangulation data. TriRep.edges
TriRep.edges

**Purpose**
Triangulation edges

**Syntax**

\[ E = \text{edges}(TR) \]

**Description**

\[ E = \text{edges}(TR) \]

returns the edges in the triangulation in an \( n \)-by-2 matrix. \( n \) is the number of edges. The vertices of the edges index into \( TR.X \), the array of points representing the vertex coordinates.

**Inputs**

\( TR \)

Triangulation representation.

**Outputs**

\( E \)

Edge matrix.

**Examples**

**Example 1**
Load a 2-D triangulation.

\[
\text{load trimesh2d}
\]

\[
\text{trep} = \text{TriRep(tri, x,y)};
\]

Return all edges.

\[
e = \text{edges(trep)};
\]

**Example 2**
Query a 2-D DelaunayTri-generated triangulation.

\[
X = \text{rand}(10,2);
\]

\[
\text{dt} = \text{DelaunayTri}(X);
\]

\[
e = \text{edges(dt)};
\]

**See Also**

“Triangulation Representations”—How to query triangulation data.

TriRep.edgeAttachments
**Purpose**

Edit or create M-file

**GUI Alternatives**

As an alternative to the `edit` function, select **File > New** or **Open** in the MATLAB desktop or any desktop tool.

**Syntax**

```matlab
edit
edit fun.m
edit file.ext
edit fun1 fun2 fun3 ...
edit classname/fun
edit private/fun
edit classname/private/fun
edit +packagename/classname/fun
edit('my file.m')
```

**Description**

`edit` opens a new editor window.

`edit fun.m` opens the M-file `fun.m` in the default editor. The `fun.m` file specification can include a MATLAB partial path, complete path, relative path, or no path. Be aware of the following:

- If you do not specify a path, the current directory is the default.
- If you specify a path, the directory must exist; otherwise MATLAB returns an error.
- If you specify a path and the directory exits, but the specified file does not, a prompt opens such as shown in the following image:

![MATLAB Editor Prompt](image-url)
edit

To create a blank file named fun.m in the specified directory, click Yes. To suppress the prompt, select Do not show this prompt again. To reinstate the prompt after suppressing it, open the Preferences dialog box by selecting File > Preferences > General > Confirmation Dialogs and then selecting Prompt when editing files that do not exist in the pane on the right.

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 classname/fun, or edit private/fun, or edit classname/private/fun opens a method, private function, or private method for the named class.

edit +packagename/classname/fun opens a method for the named class in the named package.

edit('my file.m') opens the M-file my file.m in the default editor. This form of the edit function is useful when a file name contains a space; you cannot use the command form in such a case.

Remarks

To specify the default editor for MATLAB, select Preferences from the File menu. On the Editor/Debugger pane, select MATLAB Editor or specify another editor.

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, 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 file name 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
Eigenvalues and eigenvectors

### Syntax

- `d = eig(A)`
- `d = eig(A,B)`
- `[V,D] = eig(A)`
- `[V,D] = eig(A,'nobalance')`
- `[V,D] = eig(A,B)`
- `[V,D] = eig(A,B,flag)`

### Description

- `d = eig(A)` returns a vector of the eigenvalues of matrix `A`.
- `d = 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 = eig(S)` to return the eigenvalues of `S`. If `S` is sparse but not symmetric, or if you want to return the eigenvectors of `S`, use the function `eigs` instead of `eig`.

- `[V,D] = 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] = eig(A.'); W = conj(W)` to compute the left eigenvectors.

- `[V,D] = eig(A,'nobalance')` finds eigenvalues and eigenvectors without a preliminary balancing step. This may give more accurate results for certain problems with unusual scaling. 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 \( A*V = B*V*D \).

\[ [V,D] = \text{eig}(A,B,\text{flag}) \] specifies the algorithm used to compute
eigenvalues and eigenvectors. \( \text{flag} \) 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 \( B \).

\textbf{Note} For \text{eig}(A), the eigenvectors are scaled so that the norm of each
is 1.0. For \text{eig}(A,B), \text{eig}(A,'\text{nobalance}'), and \text{eig}(A,B,\text{flag}), the
eigenvectors are not normalized.

Also note that if \( A \) is symmetric, \text{eig}(A,'\text{nobalance}') ignores the
\text{nobalance} option since \( A \) is already balanced.

\textbf{Remarks} The eigenvalue problem is to determine the nontrivial solutions of the
equation

\[ A x = \lambda x \]

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 \textit{eigenvalues},
and the corresponding values of \( x \) are the \textit{right eigenvectors}.
The \text{MATLAB} function \text{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 \( n \times n \) 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 \text{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 \text{nobalance} option is necessary to compute the eigenvectors correctly. Try the statements

\[
[\text{VB},\text{DB}] = \text{eig}(B) \\
B*\text{VB} - \text{VB}*\text{DB} \\
[\text{VN},\text{DN}] = \text{eig}(B, '\text{nobalance}')
\]
**Algorithm**

**Inputs of Type Double**

For inputs of type double, MATLAB software 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>
</tbody>
</table>
| • With preliminary balance step                       | DGEEV (with the scaling factor SCLFAC = 2 in DGEBA,
|                                                        | instead of the LAPACK default value of 8)       |
| • d = eig(A,'nobalance')                               | DGEHRD, DHSEQR                                  |
| • [V,D] = eig(A,'nobalance')                           | DGEHRD, DORGHR, DHSEQR, DTREVC                  |
| Hermitian A                                            | ZHEEV                                            |
| Non-Hermitian A:                                       |                                                  |
| • With preliminary balance step                       | ZGEEV (with SCLFAC = 2 instead of 8 in ZGEBAL)   |
| • d = eig(A,'nobalance')                               | ZGEHRD, ZHSEQR                                  |
| • [V,D] = eig(A,'nobalance')                           | ZGEHRD, ZUNGHR, ZHSEQR, ZTREVC                  |
| Real symmetric A, symmetric positive definite B.       | DSYGV                                            |
| Special case: eig(A,B,'qz') for real A, B (same as real nonsymmetric A, real general B) | DGGEV                                            |
| Real nonsymmetric A, real general B                    | DGGEV                                            |
| Complex Hermitian A, Hermitian positive definite B.    | ZHEGV                                            |
### Case Routine

<table>
<thead>
<tr>
<th>Case</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special case: <code>eig(A, B, 'qz')</code> for complex A or B (same as complex non-Hermitian A, complex B)</td>
<td>ZGGEV</td>
</tr>
<tr>
<td>Complex non-Hermitian A, complex B</td>
<td>ZGGEV</td>
</tr>
</tbody>
</table>

### Inputs of Type Single

For inputs of type `single`, MATLAB software 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>SSYEV</td>
</tr>
<tr>
<td>Real nonsymmetric A:</td>
<td></td>
</tr>
<tr>
<td>• With preliminary balance step</td>
<td>SGEEV (with the scaling factor <code>SCLFAC = 2</code> in <code>SGEBAL</code>, instead of the LAPACK default value of <code>8</code>)</td>
</tr>
<tr>
<td>• <code>d = eig(A, 'nobalance')</code></td>
<td>SGEHRD, SHSEQR</td>
</tr>
<tr>
<td>• <code>[V,D] = eig(A, 'nobalance')</code></td>
<td>SGEHRD, SORGHR, SHSEQR, STREVC</td>
</tr>
<tr>
<td>Hermitian A</td>
<td>CHEEV</td>
</tr>
<tr>
<td>Non-Hermitian A:</td>
<td></td>
</tr>
<tr>
<td>• With preliminary balance step</td>
<td>CGEEV</td>
</tr>
<tr>
<td>• <code>d = eig(A, 'nobalance')</code></td>
<td>CGEHRD, CHSEQR</td>
</tr>
<tr>
<td>• <code>[V,D] = eig(A, 'nobalance')</code></td>
<td>CGEHRD, CUNGHR, CHSEQR, CTREVC</td>
</tr>
<tr>
<td>Real symmetric A, symmetric positive definite B.</td>
<td>CSYGV</td>
</tr>
<tr>
<td>Special case: <code>eig(A, B, 'qz')</code> for real A, B (same as real nonsymmetric A, real general B)</td>
<td>SGGEV</td>
</tr>
<tr>
<td>Case</td>
<td>Routine</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Real nonsymmetric A, real general B</td>
<td>SGGEV</td>
</tr>
<tr>
<td>Complex Hermitian A, Hermitian positive definite B.</td>
<td>CHEGV</td>
</tr>
<tr>
<td>Special case: eig(A,B,'qz') for complex A or B (same as complex non-Hermitian A, complex B)</td>
<td>CGGEV</td>
</tr>
<tr>
<td>Complex non-Hermitian A, complex B</td>
<td>CGGEV</td>
</tr>
</tbody>
</table>

**See Also**

balance, condeig, eigs, hess, qz, schur

**References**

Purpose

Largest eigenvalues and eigenvectors of matrix

Syntax

d = eigs(A)
[V,D] = eigs(A)
[V,D,flag] = eigs(A)
eigs(A,B)
eigs(A,k)
eigs(A,B,k)
eigs(A,k,sigma)
eigs(A,B,k,sigma)
eigs(A,K,sigma,opts)
eigs(A,B,k,sigma,opts)
eigs(Afun,n,...)

Description

d = eigs(A) returns a vector of A’s six largest magnitude eigenvalues. A must be a square matrix. A should be large and sparse, though eigs will work on full matrices as well. See “Remarks” below.

[V,D] = eigs(A) returns a diagonal matrix D of A’s six largest magnitude eigenvalues and a matrix V whose columns are the corresponding eigenvectors.

[V,D,flag] = eigs(A) also returns a convergence flag. If flag is 0 then all the eigenvalues converged; otherwise not all converged.

eigs(A,B) solves the generalized eigenvalue problem A*V == B*V*D. B must be symmetric (or Hermitian) positive definite and the same size as A. eigs(A,[],...) indicates the standard eigenvalue problem A*V == V*D.

eigs(A,k) and eigs(A,B,k) 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:
The eigenvalues closest to sigma. If A is a function, Afun must return \( Y = (A - \sigma \cdot B) \backslash x \) (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': Formerly largest algebraic ('lr')
- 'sa': Formerly smallest algebraic ('sr')
- '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 syntax `eigs(A,k,...)` is not valid when A is scalar. To pass a value for k, you must specify B as the second argument and k as the third (`eigs(A,B,k,...)`). If necessary, you can set B equal to [], the default.

`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 Lanczos basis vectors. p &gt;= 2k (p &gt;= 2k+1 real nonsymmetric) advised. p must satisfy k &lt; p &lt;= n for real symmetric, k+1 &lt; p &lt;= n otherwise. Note: If you do not specify a p value, the default algorithm uses at least 20 Lanczos vectors.</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>
eigs(Afun,n,...) accepts the function handle Afun instead of the matrix A. See “Function Handles” in the MATLAB Programming documentation for more information. Afun must accept an input vector of size n.

\[ y = Afun(x) \] should return:

- \[ A \times x \] if \( \sigma \) is not specified, or is a string other than 'sm'
- \[ A \backslash x \] if \( \sigma \) is 0 or 'sm'
- \[ (A-\sigma \cdot I) \backslash x \] if \( \sigma \) is a nonzero scalar (standard eigenvalue problem). I is an identity matrix of the same size as A.
- \[ (A-\sigma \cdot B) \backslash x \] if \( \sigma \) is a nonzero scalar (generalized eigenvalue problem)

“Parametrizing Functions” in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function Afun, if necessary.

The matrix A, A-\( \sigma \cdot I \) or A-\( \sigma \cdot B \) 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,...).

**Remarks**

d = eigs(A,k) is not a substitute for

\[
\begin{align*}
d &= \text{eig(full(A))} \\
d &= \text{sort(d)} \\
d &= d(\text{end-k+1:end})
\end{align*}
\]

but is most appropriate for large sparse matrices. If the problem fits into memory, it may be quicker to use \text{eig(full(A))}.

**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

```matlab
A = delsq(numgrid('C',15));
d1 = eigs(A,5,'sm')
```

returns

Iteration 1: a few Ritz values of the 20-by-20 matrix:

0
0
0
0
0
0
0

Iteration 2: a few Ritz values of the 20-by-20 matrix:

1.8117
2.0889
2.8827
3.7374
7.4954

Iteration 3: a few Ritz values of the 20-by-20 matrix:

1.8117
2.0889
2.8827
3.7374
7.4954

d1 =

0.5520
0.4787
0.3469
0.2676
0.1334
Example 2
This example replaces the matrix A in example 1 with a handle to a function dnRk. The example is contained in an M-file run_eigs that

- Calls eigs with the function handle @dnRk as its first argument.
- Contains dnRk as a nested function, so that all variables in run_eigs are available to dnRk.

The following shows the code for run_eigs:

```matlab
function d2 = run_eigs
    n = 139;
    opts.issym = 1;
    R = 'C';
    k = 15;
    d2 = eigs(@dnRk,n,5,'sm',opts);

    function y = dnRk(x)
        y = (delsq(numgrid(R,k))) \ x;
    end
end
```

Example 3
west0479 is a real 479-by-479 sparse matrix with both real and pairs of complex conjugate eigenvalues. eig computes all 479 eigenvalues. eigs easily picks out the largest magnitude eigenvalues.

This plot shows the 8 largest magnitude eigenvalues of west0479 as computed by eig and eigs.

```matlab
load west0479
d = eig(full(west0479))
dlm = eigs(west0479,8)
[dum,ind] = sort(abs(d));
plot(dlm,'k+')
hold on
plot(d(ind(end-7:end)),'ks')
```
Example 4

A = delsq(numgrid('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 `eig` function computes all 632 eigenvalues. It computes and plots the six largest and smallest magnitude eigenvalues of A successfully with:

```
A = delsq(numgrid('C',30));
d = eig(full(A));
[dum,ind] = sort(abs(d));
dlm = eigs(A);
dsm = eigs(A,6,'sm');
```
```matlab
subplot(2,1,1)
plot(dlm,'k+')
hold on
plot(d(ind(end:-1:end-5)),'ks')
hold off
legend('eigs(A)','eig(full(A))',3)
set(gca,'XLim',[0.5 6.5])

subplot(2,1,2)
plot(dsm,'k+')
hold on
plot(d(ind(1:6)),'ks')
hold off
legend('eigs(A,6,'sm')','eig(full(A))',2)
set(gca,'XLim',[0.5 6.5])
```
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 $A - 4.0*I$. This involves divisions of the form $1/(\lambda - 4.0)$, 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.

$$\sigma = 4 - 1e-6$$

$$[V,D] = \text{eigs}(A,18,\sigma)$$

The plot shows the 20 eigenvalues closest to 4 that were computed by `eig`, along with the 18 eigenvalues closest to $4 - 1e-6$ that were computed by `eigs`. 
See Also
eig, svds, function_handle (@)

References


ellipj

**Purpose**
Jacobi elliptic functions

**Syntax**
```matlab
[SN,CN,DN] = ellipj(U,M)
[SN,CN,DN] = ellipj(U,M,tol)
```

**Definition**
The Jacobi elliptic functions are defined in terms of the integral:

\[ u = \int_{0}^{\phi} \frac{d\theta}{(1 - m \sin^2 \theta)^{1/2}} \]

Then

\[ sn(u) = \sin \phi, \quad cn(u) = \cos \phi, \quad dn(u) = (1 - m \sin^2 \phi)^{1/2}, \quad am(u) = \phi \]

Some definitions of the elliptic functions use the modulus \( k \) instead of the parameter \( m \). They are related by

\[ k^2 = m = \sin^2 \alpha \]

where \( \alpha \) is the modular angle.

The Jacobi elliptic functions obey many mathematical identities; for a good sample, see [1].

**Description**
\[ [SN,CN,DN] = ellipj(U,M) \] 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).

\[ [SN,CN,DN] = ellipj(U,M,tol) \] 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**
`ellipj` computes the Jacobi elliptic functions using the method of the arithmetic-geometric mean [1]. It starts with the triplet of numbers:
\[ a_0 = 1, \quad b_0 = (1 - m)^{\frac{1}{2}}, \quad c_0 = (m)^{\frac{1}{2}} \]

ellipj computes successive iterates with

\[a_i = \frac{1}{2}(a_{i-1} + b_{i-1})
\]
\[b_i = (a_{i-1}b_{i-1})^{\frac{3}{2}}
\]
\[c_i = \frac{1}{2}(a_{i-1} - b_{i-1})
\]

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:

\[sn(u) = \sin \phi_0\]
\[en(u) = \cos \phi_0\]
\[dn(u) = (1 - m \cdot sn(u)^2)^{\frac{3}{2}}\]

**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**

ellipke

Purpose
Complete elliptic integrals of 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

\[ K(m) = \int_0^1 [(1-t^2)(1-mt^2)]^{-1/2} \, d\theta = \int_0^{\pi/2} (1-m\sin^2 \theta)^{-1/2} \, d\theta \]

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

\[ k^2 = m = \sin^2 \alpha \]

where \( \alpha \) is the modular angle.

Description
\( K = \text{ellipke}(M) \) returns the complete elliptic integral of the first kind for the elements of \( M \).

\( [K,E] = \text{ellipke}(M) \) returns the complete elliptic integral of the first and second kinds.
ellipke(M, to1) computes the complete elliptic integral to
accuracy to1. The default is eps; increase this for a less accurate but
more quickly computed answer.

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 \( cn \approx 0 \), within the tolerance specified by
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

References
ellipsoid

**Purpose**
Generate ellipsoid

![Ellipsoid Image](image.png)

**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 a surface mesh described by three n+1-by-n+1 matrices, enabling `surf(x,y,z)` to plot an ellipsoid with center `(xc,yc,zc)` and semi-axis lengths `(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 plots the ellipsoid as a surface.

**Algorithm**

ellipsoid generates the data using the following equation:

\[
\frac{(x-xc)^2}{xr^2} + \frac{(y-yc)^2}{yr^2} + \frac{(z-zc)^2}{zr^2}
\]

Note that `ellipsoid(0,0,0, .5,.5,.5)` is equivalent to a unit sphere.
**Example**

Generate ellipsoid with size and proportions of a standard U.S. football:

```matlab
[x, y, z] = ellipsoid(0,0,0,5.9,3.25,3.25,30);
surf(x, y, z)
colormap copper
axis equal
```

**See Also**
cylinder, sphere, surf

“Polygons and Surfaces” on page 1-97 for related functions
else

**Purpose**
Execute statements if condition is false

**Syntax**
```matlab
if expression, statements1, else statements2, end
```

**Description**
`if expression, statements1, else statements2, end` evaluates `expression` and, if the evaluation yields logical 1 (`true`) or a nonzero result, executes one or more MATLAB commands denoted here as `statements1` or, if the evaluation yields logical 0 (`false`), executes the commands in `statements2`. `else` is used to delineate the alternate block of statements.

A true expression has either a logical 1 (`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 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 “Program Control Statements” in the MATLAB Programming Fundamentals documentation for more information on controlling the flow of your program code.

**Examples**
In this example, if both of the conditions are not satisfied, then the student fails the course.

```matlab
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**
Execute statements if additional condition is true

**Syntax**
```
if expression1, statements1, elseif expression2, statements2, end
```

**Description**
if expression1, statements1, elseif expression2, statements2, end evaluates expression1 and, if the evaluation yields logical 1 (true) or a nonzero result, executes one or more MATLAB commands denoted here as statements1. If expression1 is false, MATLAB evaluates the elseif expression, expression2. If expression2 evaluates to true or a nonzero result, executes the commands in statements2.

A true expression has either a logical 1 (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 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 “Program Control Statements” in the MATLAB Programming Fundamentals documentation for more information on controlling the flow of your program code.

**Remarks**
The commands else and if, with a space or line break between them, differ from elseif, with no space. The former introduces a new, nested if that requires a matching end statement. 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.

```
if A
    x = a
elseif A
    x = a
```

```
elseif

else
    elseif B
        x = b
    elseif C
        x = c
    else
        elseif B
            x = b
        elseif C
            x = c
        else
            x = d
        end
    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
Enable, disable, or report status of Automation server

Syntax
state = enableservice('AutomationServer',enable)
state = enableservice('AutomationServer')

Description
state = enableservice('AutomationServer',enable) enables or disables the MATLAB Automation server.

If enable is logical 1 (true), enableservice converts an existing MATLAB session into an Automation server. If enable is logical 0 (false), enableservice disables the MATLAB Automation server.

state indicates the previous state of the Automation server. If state = 1, MATLAB was an Automation server. If state is logical 0 (false), MATLAB was not an Automation server.

state = enableservice('AutomationServer') returns the current state of the Automation server. If state is logical 1 (true), MATLAB is an Automation server.

Remarks
COM functions are available on Microsoft Windows systems only.

Examples
Enable an Automation Server Example
Enable the Automation server in the current MATLAB session:

    state = enableservice('AutomationServer',true);

Next, show the current state of the MATLAB session:

    state = enableservice('AutomationServer')

MATLAB displays state = 1 (true), showing that MATLAB is an Automation server.

Finally, enable the Automation server and show the previous state by typing

    state = enableservice('AutomationServer',true)
enableservice

MATLAB displays state = 1 (true), showing that MATLAB previously was an Automation server.

Note the previous state may be the same as the current state. As seen in this case, state = 1 shows MATLAB was, and still is, an Automation server.

See Also

actxserver
**Purpose**

Terminate block of code, or indicate last array index

**Syntax**

```
end
```

**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.

`end` also marks the termination of an M-file function, although in most cases, it is optional. `end` statements are required only in M-files that employ one or more nested functions. Within such an M-file, *every* function (including primary, nested, private, and subfunctions) must be terminated with an `end` statement. You can terminate any function type with `end`, but doing so is not required unless the M-file contains a nested function.

The `end` function 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(end-1,:)
```

The MATLAB software calls the `end` method defined for `A` using the syntax

```
end(A,1,2)
```

**Examples**

This example shows `end` used with the `for` and `if` statements.

```
for k = 1:n
    if a(k) == 0
```
a(k) = a(k) + 2;
end
end

In this example, end is used in an indexing expression.

A = magic(5)

A =

17  24   1   8  15
23   5   7  14  16
 4   6  13  20  22
10  12  19  21   3
11  18  25   2   9

B = A(end,2:end)

B =

18  25   2   9

See Also break, for, if, return, switch, try, while
Purpose
Last day 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 the twentieth century, try:

```matlab
y = 1900:1999;
E = eomday(y, 2);
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
eps
d = eps(X)
eps('double')
eps('single')

Description
eps returns the distance from 1.0 to the next largest double-precision number, that is \( \text{eps} = 2^{-52} \).

d = eps(X) is the positive distance from \( \text{abs}(X) \) 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 \( X \),

\[
\text{eps}(X) = \text{eps}(-X) = \text{eps}(\text{abs}(X))
\]

eps('double') is the same as eps or eps(1.0).
eps('single') is the same as eps(single(1.0)) or single(2^-23).

Except for numbers whose absolute value is smaller than realmin, if \( 2^E \leq \text{abs}(X) < 2^{(E+1)} \), then

\[
\text{eps}(X) = 2^{(E-23)} \text{ if } \text{isa}(X, \text{'single'})
\]
\[
\text{eps}(X) = 2^{(E-52)} \text{ if } \text{isa}(X, \text{'double'})
\]

For all \( X \) of class double such that \( \text{abs}(X) \leq \text{realmin} \), \( \text{eps}(X) = 2^{(-1074)} \). Similarly, for all \( X \) of class single such that \( \text{abs}(X) \leq \text{realmin('single')} \), \( \text{eps}(X) = 2^{(-149)} \).

Replace expressions of the form

\[
\text{if } Y < \text{eps} \times \text{ABS}(X)
\]

with

\[
\text{if } Y < \text{eps}(X)
\]

Examples
double precision
eps(1/2) = 2^{(-53)}
\[ \begin{align*}
\text{eps}(1) &= 2^\left(-52\right) \\
\text{eps}(2) &= 2^\left(-51\right) \\
\text{eps}(\text{realmax}) &= 2^\left(971\right) \\
\text{eps}(0) &= 2^\left(-1074\right) \\
\text{if}(\text{abs}(x)) \leq \text{realmin}, \text{eps}(x) &= 2^\left(-1074\right) \\
\text{eps}(\text{realmin}/2) &= 2^\left(-1074\right) \\
\text{eps}(\text{realmin}/16) &= 2^\left(-1074\right) \\
\text{eps}(\text{Inf}) &= \text{NaN} \\
\text{eps}(\text{NaN}) &= \text{NaN} \\
\text{single precision} \\
\text{eps}(\text{single}(1/2)) &= 2^\left(-24\right) \\
\text{eps}(\text{single}(1)) &= 2^\left(-23\right) \\
\text{eps}(\text{single}(2)) &= 2^\left(-22\right) \\
\text{eps}(\text{realmax}('\text{single}')}) &= 2^\left(104\right) \\
\text{eps}(\text{single}(0)) &= 2^\left(-149\right) \\
\text{eps}(\text{realmin}('\text{single}')/2) &= 2^\left(-149\right) \\
\text{eps}(\text{realmin}('\text{single}')/16) &= 2^\left(-149\right) \\
\text{if}(\text{abs}(x)) \leq \text{realmin}('\text{single}'), \text{eps}(x) &= 2^\left(-149\right) \\
\text{eps}(\text{single}(\text{Inf})) &= \text{single}(\text{NaN}) \\
\text{eps}(\text{single}(\text{NaN})) &= \text{single}(\text{NaN})
\end{align*} \]

**See Also**
realmax, realmin
Purpose
Test for equality

Syntax
A == B
eq(A, B)

Description
A == B compares each element of array A for equality with the corresponding element of array B, and returns an array with elements set to logical 1 (true) where A and B are equal, or logical 0 (false) where they are not equal. Each input of the expression can be an array or a scalar value.

If both A and B are scalar (i.e., 1-by-1 matrices), then the MATLAB software returns a scalar value.

If both A and B are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as A and B.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input A is the number 100, and B is a 3-by-5 matrix, then A is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

eq(A, B) is called for the syntax A == B when either A or B is an object.

Examples
Create two 6-by-6 matrices, A and B, and locate those elements of A that are equal to the corresponding elements of B:

```
A = magic(6);
B = repmat(magic(3), 2, 2);
A == B
ans =
     0     1     1     0     0     0
     1     0     1     0     0     0
     0     1     1     0     0     0
     1     0     0     0     0     0
```
See Also

ne, le, ge, lt, gt, relational operators
**Purpose**

Compare MException objects for equality

**Syntax**

eObj1 == eObj2

**Description**

eObj1 == eObj2 tests scalar MException objects eObj1 and eObj2 for equality, returning logical 1 (true) if the two objects are identical, otherwise returning logical 0 (false).

**See Also**

try, catch, error, assert, MException, isequal(MException), ne(MException), getReport(MException), disp(MException), throw(MException), rethrow(MException), throwAsCaller(MException), addCause(MException), last(MException)
**Purpose**

Error functions

**Syntax**

\[
Y = \text{erf}(X) \\
Y = \text{erfc}(X) \\
Y = \text{erfcx}(X) \\
X = \text{erfinv}(Y) \\
X = \text{erfcinv}(Y)
\]

**Definition**

The error function \( \text{erf}(x) \) is twice the integral of the Gaussian distribution with 0 mean and variance of \( \frac{1}{2} \).

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt
\]

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 \( \left( \frac{1}{\sqrt{\pi}} \right) \frac{1}{x} \).

**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 = 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 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 erfc and the standard normal probability distribution returned by the Statistics Toolbox function normcdf is

\[
\text{normcdf}(x) = 0.5 * \text{erfc}(-x/\sqrt{2})
\]

The relationship between the inverse complementary error function erfcinv and the inverse standard normal probability distribution returned by the Statistics Toolbox function norminv is

\[
\text{norminv}(p) = -\sqrt{2} * \text{erfcinv}(2p)
\]

**Examples**

erfinv(1) is Inf
erfinv(-1) is -Inf.
For abs(Y) > 1, 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**

**Purpose**
Display message and abort function

**Syntax**
- `error('msgID', 'errmsg', v1, v2 ...)`
- `error('errmsg', v1, v2, ...)`
- `error('errmsg')`
- `error(msgStruct)`

**Description**
`error('msgID', 'errmsg', v1, v2 ...)` displays a descriptive message `errmsg` when the currently-running M-file program encounters an error condition. Depending on how the program code responds to the error, MATLAB either enters a `catch` block to handle the error condition, or exits the program.

The `msgID` argument is a unique *message identifier* string that MATLAB attaches to the error message when it throws the error. A message identifier has the format **component**:mnemonic. Its purpose is to better identify the source of the error (see Message Identifiers in the MATLAB Programming Fundamentals documentation for more information).

The `errmsg` argument is a character string that informs the user about the cause of the error and can also suggest how to correct the faulty condition. The `errmsg` string can include predefined escape sequences, such as `\n` for newline, and conversion specifiers, such as `%%d` for a decimal number.

The `v1, v2, ...` arguments represent values or substrings that are to replace conversion specifiers used in the `errmsg` string. The format is the same as that used with the `sprintf` function. For example, if `errmsg` is “Error on line %d, command %%s”, then `v1` is the line number at which the error was detected, and `v2` is the command that failed. The `vN` arguments replace the conversion specifiers at the time of execution.

Valid escape sequences for the `errmsg` string are `\b`, `\f`, `\n`, `\r`, `\t`, and `\x` or `\` when followed by a valid hexadecimal or octal number, respectively. Following a backslash in the `errmsg` with any other character causes MATLAB to issue a warning. Conversion specifiers are similar to those used in the C programming language and in the `sprintf` function.
All string input arguments must be enclosed in single quotation marks. If `errMsg` is an empty string, the `error` command has no effect.

`error('errMsg', v1, v2, ...)` reports an error without including a message identifier in the error report. Although including a message identifier in an error report is recommended, it is not required.

`error('errMsg')` is the same as the above syntax, except that the `errMsg` string contains no conversion specifiers, no escape sequences, and no substitution value (v1, v2, ...) arguments. All characters in `errMsg` are interpreted exactly as they appear in the `errMsg` argument. MATLAB displays the \	 in 'C:\testdir' for example, as a backslash character followed by the letter t, and not as a horizontal tab.

`error(msgStruct)` accepts a scalar error structure input `msgStruct` with at least one of the fields `message`, `identifier`, and `stack`.

(See the help for `lasterror` for more information on these fields.) When the `msgStruct` input includes a `stack` field, the stack field of the error will be set according to the contents of the `stack` input. When specifying a `stack` input, use the absolute file name and the entire sequence of functions that nests the function in the stack frame. (This is the same as the string returned by `dbstack('-completenames')`). If `msgStruct` is an empty structure, no action is taken and `error` returns without exiting from the M-file.

**Remarks**

The `error` function also determines where the error occurred and provides this information in the `stack` field of the structure returned by `MException.last`. This field contains a structure array that has the same format as the output of the `dbstack` function. This stack points to the line where the `error` function was called.

**Examples**

**Example 1 — Simple Error Message, lasterror**

Write a short M-file `errtest1` that throws an error when called with an incorrect number of input arguments. Include a message identifier 'myApp:argChk' and error message:

```matlab
function errtest1(x, y)
if nargin ~= 2
```
Call the function with an incorrect number of inputs. The call to 
nargin, a function that checks the number of inputs, fails and the 
program calls error:

```
errtest1(pi)
```

```?? Error using ==> errtest1 at 3
Wrong number of input arguments
```

You can use lasterror to get information on the last error thrown:

```
err = lasterror
err =
  
  message: [1x120 char]
  identifier: 'myApp:argChk'
  stack: [1x1 struct]

err.stack
ans =
  
  file: 'c:\work\errtest1.m'
  name: 'errtest1'
  line: 3
```

**Example 2 — Formatted Message String, lasterror**

Specify a message identifier and formatted error message string with 
error:

```
function plotshape(newAngle)
 maxAngle = 90;
    check_angles(newAngle, maxAngle)

function check_angles(newAngle, maxAngle)
 if newAngle > maxAngle
   error('MyToolbox:angleTooLarge', ...
       'Specified angle must be less than %d degrees.', ...
maxAngle)
end

Run the function, which then displays the error message:

plotshape(100)
??? Error using ==> plotshape>check_angles at 14
The angle specified must be less than 90 degrees.

Error in ==> plotshape at 4
    check_angles(newAngle, maxAngle)

Use the lasterror function to obtain more information about the error:

err = lasterror
err =
    message: [1x157 char]
    identifier: 'MyToolbox:angleTooLarge'
    stack: [2x1 struct]

Show the message string and identifier:

err.message
ans =
    Error using ==> plotshape>check_angles at 14
    The angle specified must be less than 90 degrees.

err.identifier
ans =
    MyToolbox:angleTooLarge

Show the most recent entry on the stack:

stk = err.stack(1)
stk =
    file: 'd:\mytools\plotshape.m'
    name: 'check_angles'
    line: 14
Example 3 — Special Characters
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:

    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:

    error('ErrorTests:convertTest', ...
          'In this case, the newline \n is converted.')
    ??? In this case, the newline
    is converted.

See Also
lasterror, rethrow, assert, errordlg, warning, lastwarn, warndlg,
dbstop, disp, sprintf
**errorbar**

**Purpose**
Plot error bars along curve

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

**Syntax**

```matlab
errorbar(Y,E)
errorbar(X,Y,E)
errorbar(X,Y,L,U)
errorbar(...,LineSpec)
```

`h = errorbar(...)`

`hlines = errorbar('v6',...)`

**Description**
Error bars show the confidence intervals 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))`.  

2-1086
errorbar(...,LineSpec) uses the color and line style specified by the string 'LineSpec'. The color is applied to the data line and error bars. The linestyle and marker are applied to the data line only. See linespec for examples of styles.

h = errorbar(...) returns handles to the errorbarseries objects created. errorbar creates one object for vector input arguments and one object per column for matrix input arguments. See errorbarseries 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.

---

**Note** The v6 option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See “Plot Objects and Backward Compatibility” for more information.

**Remarks**

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)
```
Plot the computed average traffic volume and computed standard deviations for three street locations over the course of a day using red 'x' markers:

```matlab
load count.dat;
y = mean(count,2);
e = std(count,1,2);
figure
errorbar(y,e,'xr')
```
See Also corrcoef, linespec, plot, std

“Basic Plots and Graphs” on page 1-93 and ConfidenceBounds for related functions

Errorbarseries Properties for property descriptions
Errorbarseries Properties

**Purpose**
Define errorbarseries properties

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

Note that you cannot define default property values for errorbarseries objects. See “Plot Objects” for more information on errorbarseries objects.

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

**Annotation**
`hg.Annotation` object Read Only

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

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

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

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Include the errorbarseries object in a legend as one entry, but not its children objects</td>
</tr>
</tbody>
</table>
Errorbar series Properties

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>Do not include the errorbarseries or its children in a legend (default)</td>
</tr>
<tr>
<td>children</td>
<td>Include only the children of the errorbarseries as separate entries in the legend</td>
</tr>
</tbody>
</table>

Setting the IconDisplayStyle Property

These commands set the IconDisplayStyle of a graphics object with handle hobj to children, which causes each child object to have an entry in the legend:

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

Using the IconDisplayStyle Property

See “Controlling Legends” for more information and examples.

BeingDeleted

on  |  {off} Read Only

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

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

**BusyAction**
cancel | {queue}

*Callback routine interruption.* The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the Interruptible property is off, the BusyAction property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

**ButtonDownFcn**
string or function handle

*Button press callback function.* A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the HitTestArea property for information about selecting objects of this type.

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

This property can be
- A string that is a valid MATLAB expression
• The name of an M-file
• A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

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

Children
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object’s HandleVisibility property is set to callback or off, its handle does not show up in this object’s Children property unless you set the root ShowHiddenHandles property to on:

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

Clipping
{on} | off

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set Clipping to off, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set hold to on, freeze axis scaling (axis manual), and then create a larger plot object.

Color
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object’s color.
**Errorbarseries Properties**

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

**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 this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

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

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

See the `BeingDeleted` property for related information.

**DisplayName**
string (default is empty string)

*String used by legend for this errorbarseries object.* The `legend` function uses the string defined by the `DisplayName` property to label this errorbarseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this errorbarseries object’s corresponding string and that string is used for the legend.
• If `DisplayName` is empty, `legend` creates a string of the form, `[ 'data' n]`, where `n` is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.

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

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

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

See “Controlling Legends” for more examples.

**EraseMode**

{normal} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

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

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

• **xor** — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing
Errorbarseries Properties

the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes Color property is set to none). That is, it isn’t erased correctly if there are objects behind it.

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

Printing with Nonnormal Erase Modes

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

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

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

HandleVisibility
{on} | callback | off

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

preventing command-line users from accidentally accessing objects that you need to protect for some reason.

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

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

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

HitTest

{on} | off

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

HitTestArea

on | {off}

*Select the object by clicking lines or area of extent.* This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When HitTestArea is off, you must click the object’s lines or markers (excluding the baseline, if any) to select the object. When
HitTestArea is on, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

**Interruptible**

{on} | off

*Callback routine interruption mode.* The **Interruptible** property controls whether an object’s callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the ButtonDownFcn property are affected by the **Interruptible** property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the BusyAction property for related information.

Setting **Interruptible** to on allows any graphics object’s callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

**LData**

array equal in size to `XData` and `YData`

*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**

string (MATLAB variable)

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

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.

**LineStyle**

{ - | -- | : | - | none

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

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

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

**LineWidth**

scalar

*The width of linear objects and edges of filled areas.* Specify this value in points (1 point = 1/72 inch). The default **LineWidth** is 0.5 points.
Marker character (see table)

Marker symbol. The Marker property specifies the type of markers that are displayed at plot vertices. You can set values for the Marker property independently from the LineStyle property. Supported markers include those shown in the following table.

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

MarkerEdgeColor
ColorSpec | none | {auto}

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none
Errorbarseries Properties

specifies no color, which makes nonfilled markers invisible. auto
sets MarkerEdgeColor to the same color as the Color property.

MarkerFaceColor
    ColorSpec | {none} | auto

    Marker face color. The fill color for markers that are closed shapes
    (circle, square, diamond, pentagram, hexagram, and the four
    triangles). ColorSpec defines the color to use. none makes the
    interior of the marker transparent, allowing the background to
    show through. auto sets the fill color to the axes color, or to the
    figure color if the axes Color property is set to none (which is the
    factory default for axes objects).

MarkerSize
    size in points

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

Parent
    handle of parent axes, hggroup, or hgtransform

    Parent of this object. This property contains the handle of the
    object’s parent. The parent is normally the axes, hggroup, or
    hgtransform object that contains the object.

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

Selected
    on | {off}

    Is object selected? When you set this property to on, MATLAB
displays selection "handles" at the corners and midpoints if the
SelectionHighlight property is also on (the default). You
can, for example, define the ButtonDownFcn callback to set this
property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

**SelectionHighlight**

{on} | off

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

**Type**

string (read only)
**Errorbarseries Properties**

*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.

```
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 open error dialog box

**Syntax**

```matlab
h = errordlg
h = errordlg(errorstring)
h = errordlg(errorstring,dlgname)
h = errordlg(errorstring,dlgname,createmode)
```

**Description**

`h = errordlg` creates and displays a dialog box with title `Error Dialog` that contains the string `This is the default error string`. The `errordlg` function returns the handle of the dialog box in `h`.

`h = errordlg(errorstring)` displays a dialog box with title `Error Dialog` that contains the string `errorstring`.

`h = errordlg(errorstring,dlgname)` displays a dialog box with title `dlgname` that contains the string `errorstring`.

`h = errordlg(errorstring,dlgname,createmode)` specifies whether the error dialog box is modal or nonmodal. Optionally, it can also specify an interpreter for `errorstring` and `dlgname`. The `createmode` argument can be a string or a structure.

If `createmode` is a string, it must be one of the values shown in the following table.

<table>
<thead>
<tr>
<th><code>createmode Value</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>modal</td>
<td>Replaces the error dialog box having the specified Title, that was last created or clicked on, with a modal error dialog box as specified. All other error dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.</td>
</tr>
<tr>
<td><code>createMode</code> Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>non-modal (default)</td>
<td>Creates a new nonmodal error dialog box with the specified parameters. Existing error dialog boxes with the same title are not deleted.</td>
</tr>
<tr>
<td>replace</td>
<td>Replaces the error dialog box having the specified Title, that was last created or clicked on, with a nonmodal error dialog box as specified. All other error dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.</td>
</tr>
</tbody>
</table>

**Note** A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function.

If you open a dialog with `errordlg`, `msgbox`, or `warndlg` using `'CreateMode','modal'` and a non-modal dialog created with any of these functions is already present and has the same name as the modal dialog, the non-modal dialog closes when the modal one opens.

For more information about modal dialog boxes, see `WindowStyle` in the Figure Properties.

If `CreateMode` is a structure, it can have fields `WindowStyle` and `Interpreter`. `WindowStyle` must be one of the options shown in the table above. `Interpreter` is one of the strings 'tex' or 'none'. The default value for `Interpreter` is 'none'.

**Remarks** MATLAB sizes the dialog box to fit the string 'errorstring'. The error dialog box has an OK push button 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 platform you use.

**Examples**

The function

```matlab
errordlg('File not found','File Error');
```

displays this dialog box:

![File Error](image)

**See Also**

dialog, helpdlg, inputdlg, listdlg, msgbox, questdlg, warndlg
figure, uwait, uiresume

“Predefined Dialog Boxes” on page 1-110 for related functions
**Purpose**

Time elapsed between date vectors

**Syntax**

e = etime(t2, t1)

**Description**

`e = etime(t2, t1)` returns the number of seconds between vectors `t1` and `t2`. The two vectors must be six elements long, in the format returned by `clock`:

\[ T = [\text{Year \ Month \ Day \ Hour \ Minute \ Second}] \]

**Remarks**

etime does not account for the following:

- Leap seconds.
- Daylight savings time adjustments.
- Differences in time zones.

When timing the duration of an event, use the tic and toc functions instead of clock and etime. `clock` uses the system time, which might be adjusted periodically by the operating system and thus might not be reliable in time comparison operations.

**Examples**

This example shows two ways to calculate how long a particular FFT operation takes. Using tic and toc is preferred, as it can be more reliable for timing the duration of an event:

```matlab
x = rand(800000, 1);

t1 = tic; fft(x); toc(t1) % Recommended
Elapsed time is 0.097665 seconds.

t = clock; fft(x); etime(clock, t)
ans =
     0.1250
```

**See Also**

tic, toc, cputime, clock, now
etree

**Purpose**
Elimination tree

**Syntax**

- `p = etree(A)`
- `p = etree(A, 'col')`
- `p = etree(A, 'sym')`
- `[p, q] = etree(...)`

**Description**
`p = 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 = etree(A, 'col')` returns the elimination tree of `A'*A`.
- `p = etree(A, 'sym')` is the same as `p = etree(A)`.
- `[p, q] = etree(...)` also returns a postorder permutation `q` of the tree.

**See Also**
treelayout, treeplot, 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
 eval

**Purpose**
Execute string containing MATLAB expression

**Syntax**
eval(expression)

[a1, a2, a3, ...] = eval('myfun(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('myfun(b1, b2, b3, ...)') executes function myfun 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. Use the second syntax instead:

% Not recommended
    eval('[a1, a2, a3, ...] = function(var)')

% Recommended syntax
    [a1, a2, a3, ...] = eval('function(var)')

**Examples**

**Example 1 – Working with a Series of Files**
Load MAT-files August1.mat to August10.mat into the MATLAB workspace:

```matlab
for d=1:10
    s = ['load August' int2str(d) '.mat']
    eval(s)
end
```

These are the strings being evaluated:
Example 2 – Assigning to Variables with Generated Names

Generate variable names that are unique in the MATLAB workspace and assign a value to each using `eval`:

```matlab
for k = 1:5
    t = clock;
    pause(uint8(rand * 10));
    v = genvarname('time_elapsed', who);
    eval([v ' = etime(clock,t)'])
end
```

As this code runs, `eval` creates a unique statement for each assignment:

- `time_elapsed = 5.0070`
- `time_elapsed1 = 2.0030`
- `time_elapsed2 = 7.0010`
- `time_elapsed3 = 8.0010`
- `time_elapsed4 = 3.0040`

Example 3 – Evaluating a Returned Function Name

The following command removes a figure by evaluating its `CloseRequestFcn` property as returned by `get`.

```matlab
eval(get(h,'CloseRequestFcn'))
```
See Also

evalc, evalin, assignin, feval, catch, lasterror, try
Purpose  Evaluate MATLAB expression with capture

Syntax  

\[
T = \text{evalc}(S) \\
[T, X, Y, Z, ...] = \text{evalc}(S)
\]

Description  

\(T = \text{evalc}(S)\) is the same as \(\text{eval}(S)\) except that anything that would normally be written to the command window, except for error messages, is captured and returned in the character array \(T\) (lines in \(T\) are separated by \(\backslash n\) characters).

\([T, X, Y, Z, ...] = \text{evalc}(S)\) is the same as \([X, Y, Z, ...] = \text{eval}(S)\) except that any output is captured into \(T\).

Remark  When you are using \text{evalc}, \text{diary}, \text{more}, and \text{input} are disabled.

See Also  \text{eval}, \text{evalin}, \text{assignin}, \text{feval}, \text{diary}, \text{input}, \text{more}
evalin

**Purpose**
Execute MATLAB expression in specified workspace

**Syntax**

```
evalin(ws, expression)
[a1, a2, a3, ...] = evalin(ws, expression)
```

**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.

**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.

```
evalin('caller', ...) finds only variables in the caller’s workspace; it does not find functions in the caller. For this reason, you cannot use evalin to construct a handle to a function that is defined in the caller.
```

If you use `evalin('caller', ws)` in the MATLAB debugger after having changed your local workspace context with `dbup` or `dbdown`, MATLAB evaluates the expression in the context of the function that is one level up in the stack from your current workspace context.
Examples
This example extracts the value of the variable `var` in the MATLAB base workspace and captures the value in the local variable `v`:

\[
    v = \text{evalin}('base', 'var');
\]

Limitation
`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`, `eval`, `evalc`, `feval`, `catch`, `lasterror`, `try`
**Purpose**

Base class for all data objects passed to event listeners

**Description**

The `event` package contains the `event.EventData` class, which defines the data objects passed to event listeners. If you want to provide additional information to event listeners, you can do so by subclassing `event.EventData`. See “Defining Event-Specific Data” for more information.

**Properties**

The `event.EventData` class defines two properties and no methods:

- **eventName** — The name of the event described by this data object.
- **source** — The source object whose class defines the event described by the data object.

**See Also**

`event.PropertyEvent`

“Events — Sending and Responding to Messages”
event.PropertyEvent

**Purpose**
Listener for property events

**Description**
The `event.PropertyEvent` class defines the data objects passed to listeners of the `meta.property` events `PreGet`, `PostGet`, `PreSet`, and `PostSet`. `event.PropertyEvent` is a sealed subclass of `event.EventData` (i.e., you cannot subclass `event.PropertyEvent`).

**Properties**
`event.PropertyEvent` inherits the `EventName` and `Source` properties from `event.EventData` and defines one new property:

- **AffectedObject** — The instance of the class to which this event refers.

**See Also**
`event.EventData`, `meta.property`
“Listening for Changes to Property Values”
**Purpose**
Class defining listener objects

**Syntax**

```
lh = event.listener(Hobj, 'EventName', @CallbackFunction)
```

**Description**

```
lh = event.listener(Hobj, 'EventName', @CallbackFunction)
```
creates a listener object for the named event on the specified object.

Listener objects respond to the specified event, which is broadcast by the source object Hobj. When the listener object lh receives the notification that the named event has occurred, the specified callback function executes.

The event.listener class is a handle class.

**Limiting Listener Lifecycle**

Generally, you create a listener object using addlistener. However, you can call the event.listener constructor directly to create a listener. However, when you do not use addlistener, the listener’s lifecycle is not tied to the object(s) being listened to—once the listener object goes out of scope, the listener no longer exists. See “Ways to Create Listeners” for more information on creating listener objects.

**Removing a Listener**

If you call delete(lh) on the listener object, the listener ceases to exist, which means the event no longer causes the listener callback function to execute.

**Disabling a Listener**

You can enable or disable a listener by setting the value of the listener’s Enabled property (see Properties table below).

**More Information on Events and Listeners**

See “Events — Sending and Responding to Messages” for more information and examples of how to use events and listeners.
Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Cell array of source objects</td>
</tr>
<tr>
<td>EventName</td>
<td>Name of the event</td>
</tr>
<tr>
<td>Callback</td>
<td>Function to execute when the event is triggered and the Enabled property is set to true</td>
</tr>
<tr>
<td>Enabled</td>
<td>callback executes when the event occurs if and only if Enabled is set to true (the default).</td>
</tr>
<tr>
<td>Recursive</td>
<td>When this property is set to true (the default), a listener can cause the same event that triggered the callback. This can lead to infinite recursion and the MATLAB recursion limit eventually triggers an error to end the recursion. When set to false, this listener does not execute recursively. Therefore, if the callback triggers its own event, the listener does not execute again.</td>
</tr>
</tbody>
</table>

See Also
addlistener, delete, event.proplistener
**Purpose**
Define listener object for property events

**Syntax**
\[ lh = \text{event.proplistener}(\text{Hobj}, \text{Properties}, 'PropEvent', \@\text{CallbackFunction}) \]

**Description**
\[ lh = \text{event.proplistener}(\text{Hobj}, \text{Properties}, 'PropEvent', \@\text{CallbackFunction}) \]
creates a property listener object for one or more properties on the specified object.

- **Hobj** — handle of object whose property or properties are to be listened to. If Hobj is an array, the listener responds to the named event on all objects in the array.

- **Properties** — an object array or a cell array of meta.property object handles representing the properties to which you want to listen.

- **PropEvent** — must be one of the strings: PresSet, PostSet, PreGet, PostGet

- **@CallbackFunction** — function handle to the callback function that executes when the event occurs.

The `event.proplistener` class defines property event listener objects. It is a subclass of the `event.listener` class and adds one property to those defined by `event.listener`:

- **Object** — Cell array of objects whose property events are being listened to.

You can call the `event.proplistener` constructor instead of calling `addlistener` to create a property listener. However, when you do not use `addlistener`, the listener’s lifecycle is not tied to the object(s) being listened to.

See “Listening for Changes to Property Values”.

See “Obtaining Information About Classes with Meta-Classes” for more information on using `meta.property` objects.
See Also

event.listener, addlistener
eventlisteners

**Purpose**
List all event handler functions registered for COM object

**Syntax**
C = h.eventlisteners
C = eventlisteners(h)

**Description**
C = h.eventlisteners lists any events, along with their event handler routines, that have been registered with COM object, h. The function returns a cell array of strings C, with each row containing the name of a registered event and the handler routine for that event. If the object has no registered events, then eventlisteners returns an empty cell array.

Events and their event handler routines must be registered in order for the control to respond to them. You can register events either when you create the control, using actxcontrol, or at any time afterwards, using registerevent.

C = eventlisteners(h) is an alternate syntax for the same operation.

**Remarks**
COM functions are available on Microsoft Windows systems only.

**Examples**

**Control Example**
Create an mwsamp control, registering only the Click event. eventlisteners returns the name of the event and its event handler routine, myclick:

```matlab
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', ...
                  [0 0 200 200], f, ... 
                  {'Click' 'myclick'});
h.eventlisteners
```

MATLAB software displays:

```matlab
ans =
    'Click'    'myclick'
```
Register two more events: DblClick and MouseDown. `eventlisteners` returns the names of the three registered events along with their respective handler routines:

```matlab
h.registerevent({'DblClick', 'my2click'; 'MouseDown' 'mymoused'});
```

`MATLAB` displays:

```matlab
ans =
    'Click'    'myclick'
    'Dblclick' 'my2click'
    'Mousedown' 'mymoused'
```

Now unregister all events for the control. `eventlisteners` returns an empty cell array, indicating that no events have been registered for the control:

```matlab
h.unregisterallevents
h.eventlisteners
```

`MATLAB` displays:

```matlab
ans =
    {}
```

**Microsoft Excel Workbook Example**

```matlab
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
w = wbs.Add;
w.registerevent({'Activate' 'EvtActivateHandler'});
w.eventlisteners
```

`MATLAB` displays:

```matlab
ans =
    'Activate'    'EvtActivateHandler'
```
See Also

events (COM), registerevent, unregisterevent, unregisterallevents, isevent
**Purpose**
Display class event names

**Syntax**
```matlab
events('classname')
events(obj)
e = events(...)
```

**Description**
`events('classname')` displays the names of the public events for the MATLAB class `classname`, including events inherited from superclasses.

`events(obj)` displays the names of the public events for the class of the object `obj`, where `obj` is an instance of a MATLAB class. `obj` can be either a scalar object or an array of objects.

`e = events(...)` returns the event names in a cell array of strings.

An event is public when its `ListenAccess` attribute is set to `public` and its `Hidden` attribute is set to `false` (default values for both attributes). See “Event Attributes” for a complete list of attributes.

**Note** `events` is also a keyword used in MATLAB class definition. See `classdef` for more information on class definition keywords.

See “Events — Sending and Responding to Messages” for information on using events and listeners.

**Examples**
Get the names of the public events for the `handle` class:

```matlab
events('handle')
Events for class handle:

ObjectBeingDestroyed
```

**See Also**
`properties`, `methods`
**Purpose**
List of events COM object can trigger

**Syntax**

\[
S = h.\text{events} \\
S = \text{events}(h)
\]

**Description**

\( S = h.\text{events} \) returns structure array \( S \) containing all events, both registered and unregistered, known to the COM object, and the function prototype used when calling the event handler routine. For each array element, the structure field is the event name and the contents of that field is the function prototype for that event’s handler.

\( S = \text{events}(h) \) is an alternate syntax.

**Remarks**
COM functions are available on Microsoft Windows systems only.

**Examples**

**List Control Events Example**
Create an \texttt{mwsamp} control and list all events:

\[
\begin{align*}
f &= \text{figure}('\text{position}', [100 200 200 200]); \\
h &= \text{actxcontrol}('\text{mwsamp.mwsampctrl1.2}', [0 0 200 200], f); \\
h.\text{events}
\end{align*}
\]

MATLAB software displays information similar to:

\[
\begin{align*}
\text{Click} &= \text{void} \text{Click}() \\
\text{Db1Click} &= \text{void} \text{Db1Click}() \\
\text{MouseDown} &= \text{void} \text{MouseDown}(\text{int16} \ \text{Button}, \ \text{int16} \ \text{Shift}, \\
& \quad \text{Variant} \ x, \ \text{Variant} \ y) \\
\text{Event_Args} &= \text{void} \text{Event_Args}(\text{int16} \ \text{typeshort}, \ \text{int32} \ \text{typelong}, \\
& \quad \text{double} \ \text{typedouble}, \ \text{string} \ \text{typestring}, \ \text{bool} \ \text{typebool})
\end{align*}
\]

Assign the output to a variable and get one field of the returned structure:

\[
\begin{align*}
ev &= h.\text{events}; \\
ev.\text{MouseDown}
\end{align*}
\]
MATLAB displays:

```
ans = 
    void MouseDown(int16 Button, int16 Shift, Variant x, Variant y)
```

**List Workbook Events Example**

Open a Microsoft Excel application and list all events for a Workbook object:

```
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
wb = wbs.Add;
wb.events
```

The MATLAB software displays all events supported by the Workbook object.

```
Open = void Open()
Activate = void Activate()
Deactivate = void Deactivate()
BeforeClose = void BeforeClose(bool Cancel)
```

**See Also**
isevent, eventlisteners, registerevent, unregisterevent, unregisterallevents
Execute MATLAB command in Automation server

Purpose

Syntax

MATLAB Client

result = h.Execute('command')
result = Execute(h, 'command')
result = invoke(h, 'Execute', 'command')

Method Signature

BSTR Execute([in] BSTR command)

Microsoft® Visual Basic® Client

Execute(command As String) As String

Description

The Execute function executes the MATLAB statement specified by the string command in the MATLAB Automation server attached to handle h.

The server returns output from the command in the string, result. The result string also contains any warning or error messages that might have been issued by MATLAB software as a result of the command.

Note that if you terminate the MATLAB command string with a semicolon and there are no warnings or error messages, result might be returned empty.

Remarks

If you want to be able to display output from Execute in the client window, you must specify an output variable (i.e., result in the above syntax statements).

Server function names, like Execute, are case sensitive when used with dot notation (the first syntax shown).

All three versions of the MATLAB client syntax perform the same operation.

COM functions are available on Microsoft Windows systems only.

Examples

Execute the MATLAB version function in the server and return the output to the MATLAB client.
MATLAB Client

h = actxserver('matlab.application');
server_version = h.Execute('version')
server_version =
ans =
    6.5.0.180913a (R13)

Visual Basic® .NET Client

Dim Matlab As Object
Dim server_version As String
Matlab = CreateObject("matlab.application")
server_version = Matlab.Execute("version")

See Also
Feval, PutFullMatrix, GetFullMatrix, PutCharArray, GetCharArray
Purpose

Read EXIF information from JPEG and TIFF image files

Syntax

output = exifread(filename)

Description

output = exifread(filename) reads the Exchangeable Image File Format (EXIF) data from the file specified by the string filename. filename must specify a JPEG or TIFF image file. output is a structure containing metadata values about the image or images in imagefile.

Note

exifread returns all EXIF tags and does not process them in any way.

EXIF is a standard used by digital camera manufacturers to store information in the image file, such as, the make and model of a camera, the time the picture was taken and digitized, the resolution of the image, exposure time, and focal length. For more information about EXIF and the meaning of metadata attributes, see http://www.exif.org/.

See Also

imfinfo, imread
**Purpose**
Check existence of variable, function, directory, or Java programming language class

**Graphical Interface**
As an alternative to the `exist` function, use the Workspace Browser or the Current Directory Browser.

**Syntax**
```
exist name
exist name kind
A = exist('name','kind')
```

**Description**
`exist name` returns the status of `name`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>If <code>name</code> does not exist.</td>
</tr>
<tr>
<td>1</td>
<td>If <code>name</code> is a variable in the workspace.</td>
</tr>
<tr>
<td>2</td>
<td>If <code>name</code> is an M-file on your MATLAB search path. It also returns 2 when <code>name</code> is the full pathname to a file or the name of an ordinary file on your MATLAB search path.</td>
</tr>
<tr>
<td>3</td>
<td>If <code>name</code> is a MEX- or DLL-file on your MATLAB search path.</td>
</tr>
<tr>
<td>4</td>
<td>If <code>name</code> is an MDL-file on your MATLAB search path.</td>
</tr>
<tr>
<td>5</td>
<td>If <code>name</code> is a built-in MATLAB function.</td>
</tr>
<tr>
<td>6</td>
<td>If <code>name</code> is a P-file on your MATLAB search path.</td>
</tr>
<tr>
<td>7</td>
<td>If <code>name</code> is a directory.</td>
</tr>
<tr>
<td>8</td>
<td>If <code>name</code> is a Java class. (<code>exist</code> returns 0 if you start MATLAB with the <code>-nojvm</code> option.)</td>
</tr>
</tbody>
</table>

`exist name kind` returns the status of `name` for the specified `kind`. If `name` of type `kind` does not exist, it returns 0. The `kind` argument may be one of the following:

<table>
<thead>
<tr>
<th><code>kind</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>builtin</td>
<td>Checks only for built-in functions.</td>
</tr>
<tr>
<td>class</td>
<td>Checks only for Java classes.</td>
</tr>
</tbody>
</table>
exist

<table>
<thead>
<tr>
<th>dir</th>
<th>Checks only for directories.</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Checks only for files or directories.</td>
</tr>
<tr>
<td>var</td>
<td>Checks only for variables.</td>
</tr>
</tbody>
</table>

If \textit{name} belongs to more than one category (e.g., if there are both an M-file and variable of the given name) and you do not specify a \textit{kind} argument, \texttt{exist} returns one value according to the order of evaluation shown in the table below. For example, if \textit{name} matches both a directory and M-file name, \texttt{exist} returns 7, identifying it as a directory.

<table>
<thead>
<tr>
<th>Order of Evaluation</th>
<th>Return Value</th>
<th>Type of Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Variable</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Built-in</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>Directory</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>MEX or DLL-file</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>MDL-file</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>P-file</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>M-file</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>Java class</td>
</tr>
</tbody>
</table>

\texttt{A = exist('name','kind')} is the function form of the syntax.

\textbf{Remarks}

If \textit{name} specifies a filename, that filename may include an extension to preclude conflicting with other similar filenames. For example, \texttt{exist('file.ext')}.

If \textit{name} 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.
You can specify a partial path to a directory or file. A partial pathname is a path name relative to the MATLAB path that contains only the trailing one or more components of the full pathname. For example, both of the following commands return 2, identifying `mkdir.m` as an M-file. The first uses a partial pathname:

```
exist('matlab/general/mkdir.m')
exist([matlabroot '/toolbox/matlab/general/mkdir.m'])
```

If a file or directory is not on the search path, then `name` must specify either a full pathname, a partial pathname relative to `MATLABPATH`, a partial pathname relative to your current directory, or the file or directory must reside in your current working directory.

If `name` is a Java class, then `exist('name')` returns an 8. However, if `name` is a Java class file, then `exist('name')` returns a 2.

**Remarks**

To check for the existence of more than one variable, use the `ismember` function. For example,

```
a = 5.83;
c = 'teststring';
ismember({'a','b','c'},who)
```

```
ans =
     1     0     1
```

**Examples**

This example uses `exist` to check whether a MATLAB function is a built-in function or a file:

```
type = exist('plot')
type =
     5
```

This indicates that `plot` is a built-in function.

In the next example, `exist` returns 8 on the Java class, `Welcome`, and returns 2 on the Java class file, `Welcome.class`:
exist Welcome
ans =
  8

eexist 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:

exist('testresults', 'var')
an =
  1
exist('testresults', 'dir')
an =
  7

See Also assignin, computer, dir, evalin, help, inmem, isfield, isempty, lookfor, mfilename, partialpath, what, which, who
### Purpose
Terminate MATLAB program (same as quit)

### GUI Alternatives
As an alternative to the `exit` function, select **File > Exit MATLAB** or click the Close box in the MATLAB desktop.

### Syntax

**exit**

### Description
`exit` terminates the current session of MATLAB after running `finish.m`, if the file `finish.m` exists. It performs the same as `quit` and takes the same termination options, such as `force`. For more information, see `quit`.

### See Also
`quit`, `finish`
**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**  
\[ Y = \text{expint}(X) \]

**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

\[ E_i(x) = \int_{-\infty}^{x} \frac{e^t}{t} dt \]

which, for real positive \( x \), is related to \( \text{expint} \) as

\[ E_1(-x) = -E_i(x) - i\pi \]

**Description**  
\( Y = \text{expint}(X) \) 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.

Although it is not computed this way, if \( X \) has a full set of eigenvectors \( V \) with corresponding eigenvalues \( D \), then

\[
[V,D] = \text{EIG}(X) \quad \text{and} \quad \text{EXPM}(X) = V \times \text{diag}(\exp(\text{diag}(D))) / V
\]

Use \( \exp \) for the element-by-element exponential.

Algorithm

\( \text{expm} \) uses the Padé approximation with scaling and squaring. See reference [3], below.

Note

The \( \text{expmdemo1} \), \( \text{expmdemo2} \), and \( \text{expmdemo3} \) 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.

Examples

This example computes and compares the matrix exponential of \( A \) and the exponential of \( A \).

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

\[
\text{expm}(A) \\
\text{ans} = \\
2.7183 & 1.7183 & 1.0862 \\
0 & 1.0000 & 1.2642 \\
0 & 0 & 0.3679
\]

2-1142
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, expm1, funm, logm, eig, 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**
\( \text{exp} \), \( \text{expm} \), \( \text{log1p} \)
**Purpose**
Export variables to workspace

**Syntax**

```matlab
export2wsdlg(checkboxlabels, defaultvariablenames, itemstoexport)
export2wsdlg(checkboxlabels, defaultvariablenames, itemstoexport, title)
export2wsdlg(checkboxlabels, defaultvariablenames, itemstoexport, title, selected)
export2wsdlg(checkboxlabels, defaultvariablenames, itemstoexport, title, selected, helpfunction)
export2wsdlg(checkboxlabels, defaultvariablenames, itemstoexport, title, selected, helpfunction, functionlist)
hdialog = export2wsdlg(...)
[hdialog, ok_pressed] = export2wsdlg(...)
```

**Description**

`export2wsdlg(checkboxlabels, defaultvariablenames, itemstoexport)` creates a dialog with a series of checkboxes and edit fields. `checkboxlabels` is a cell array of labels for the checkboxes. `defaultvariablenames` is a cell array of strings that serve as a basis for variable names that appear in the edit fields. `itemstoexport` is a cell array of the values to be stored in the variables. If there is only one item to export, `export2wsdlg` creates a text control instead of a check box.

**Note**
By default, the dialog box is modal. A modal dialog box prevents the user from interacting with other windows before responding.
**export2wsdlg**

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstosexport,title,selected,helpfunction)` creates the dialog with a help button. `helpfunction` is a callback that displays help.

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstosexualg,title,selected,helpfunction,functionlist)` creates a dialog that enables the user to pass in `functionlist`, a cell array of functions and optional arguments that calculate, then return the value to export. `functionlist` should be the same length as `checkboxlabels`.

`hdialog = export2wsdlg(...)` returns the handle of the dialog.

`[hdialog,ok_pressed] = export2wsdlg(...)` sets `ok_pressed` to true if the OK button is pressed, or false otherwise. If two return arguments are requested, `hdialog` is `[]` and the function does not return until the dialog is closed.

The user can edit the text fields to modify the default variable names. If the same name appears in multiple edit fields, `export2wsdlg` creates a structure using that name. It then uses the `defaultvariablenames` as fieldnames for that structure.

The lengths of `checkboxlabels`, `defaultvariablenames`, `itemstosexualg` and `selected` must all be equal.

The strings in `defaultvariablenames` must be unique.

### Examples

This example creates a dialog box that enables the user to save the variables `sumA` and/or `meanA` to the workspace. The dialog box title is Save Sums to Workspace.

```matlab
A = randn(10,1);
checkLabels = {'Save sum of A to variable named:' ...
               'Save mean of A to variable named:'};
varNames = {'sumA','meanA'};
items = {sum(A),mean(A)};
export2wsdlg(checkLabels,varNames,items,...
             'Save Sums to Workspace');
```
**Purpose**

Identity matrix

**Syntax**

\[
Y = \text{eye}(n)
\]

\[
Y = \text{eye}(m,n)
\]

\[
\text{eye}([m \ n])
\]

\[
Y = \text{eye(size}(A))
\]

\[
\text{eye}(m, n, \text{classname})
\]

\[
\text{eye}([m,n],\text{classname})
\]

**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.

**Note** The size inputs \(m\) and \(n\) should be nonnegative integers. Negative integers are treated as 0.

\(Y = \text{eye(size}(A))\) 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', 'uint32', 'int64', or 'uint64'.

**Example:**

\[
x = \text{eye}(2,3,\text{'}int8\text{'})
\]

**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(fun)`
- `ezcontour(fun,domain)`
- `ezcontour(...,n)`
- `ezcontour(axes_handle,...)`
- `h = ezcontour(...)`

Description

- `ezcontour(fun)` plots the contour lines of `fun(x,y)` using the `contour` function. `fun` is plotted over the default domain: `-2\pi < x < 2\pi`, `-2\pi < y < 2\pi`.
- `fun` can be a function handle for an M-file function or an anonymous function (see “Function Handles” and “Anonymous Functions”) or a string (see Remarks).

- `ezcontour(fun,domain)` plots `fun(x,y)` 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`).

- `ezcontour(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

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

- `h = ezcontour(...)` returns the handles to contour objects in `h`. `ezcontour` automatically adds a title and axis labels.

Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezcontour`. For example, the MATLAB syntax for a contour plot of the expression

```matlab
sqrt(x.^2 + y.^2)
```
is written as

\[
\text{ezcontour('sqrt(x^2 + y^2)'})
\]

That is, \( x^2 \) is interpreted as \( x.^2 \) in the string you pass to \text{ezcontour}.

If the function to be plotted is a function of the variables \( u \) and \( v \) (rather than \( x \) and \( y \)), the domain endpoints \( u_{\text{min}}, u_{\text{max}}, v_{\text{min}}, \) and \( v_{\text{max}} \) are sorted alphabetically. Thus, \text{ezcontour('u^2 - v^3',[0,1],[3,6])} plots the contour lines for \( u^2 - v^3 \) over \( 0 < u < 1, 3 < v < 6 \).

**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{ezcontour}.

\[
\text{fh} = @(x,y) \text{sqrt}(x.^2 + y.^2);
\]

\[
\text{ezcontour(fh)}
\]

When using function handles, you must use the array power, array multiplication, and array division operators (\( .^, .*, ./ \)) since \text{ezcontour} does not alter the syntax, as in the case with string inputs.

**Passing Additional Arguments**

If your function has additional parameters, for example, \( k \) in \text{myfun}:

\[
\begin{aligned}
\text{function} & \quad z = \text{myfun}(x,y,k) \\
& \quad z = x.^k - y.^k - 1;
\end{aligned}
\]

then use an anonymous function to specify that parameter:

\[
\text{ezcontour(@(x,y)myfun(x,y,2))}
\]

**Examples**

The following mathematical expression defines a function of two variables, \( x \) and \( y \).

\[
f(x, y) = 3(1-x)^9 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right) e^{-x^2-y^2} - \frac{1}{3} e^{-(x+1)^2-y^2}
\]
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.

\[
f = @(x, y) 3*(1-x)^2.*exp(-(x^2) - (y+1)^2) \ldots \
- 10*(x/5 - x^3 - y^5).*exp(-x^2-y^2) \ldots 
- 1/3*exp(-(x+1)^2 - y^2);\]

For convenience, this function is written on three lines. The MATLAB peaks function evaluates this expression for different sizes of grids.

Pass the function handle \(f\) to \texttt{ezcontour} along with a domain ranging from -3 to 3 in both \(x\) and \(y\) and specify a computational grid of 49-by-49:

\[
\texttt{ezcontour}(f, [-3,3], 49)\]
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, ezcontourf, ezmesh, ezmeshc, ezplot, ezplot3, ezpolar, ezsurf, ezsurfc, function_handle

“Contour Plots” on page 1-96 for related functions
ezcontourf

**Purpose**
Easy-to-use filled contour plotter

**Syntax**
```
ezcontourf(fun)
ezcontourf(fun,domain)
ezcontourf(...,n)
ezcontourf(axes_handle,...)
h = ezcontourf(...)
```

**Description**
ezcontourf(fun) plots the contour lines of $f(x,y)$ using the contourf function..fun is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

fun can be a function handle for an M-file function or an anonymous function (see “Function Handles” and Anonymous Functions) or a string (see Remarks).

ezcontourf(fun,domain) plots $f(x,y)$ 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.

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

h = ezcontourf(...) returns the handles to contour 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 string expression you pass to ezcontourf. For example, the MATLAB syntax for a filled contour plot of the expression

```matlab
sqrt(x.^2 + y.^2);
```
is written as

```matlab
ezcontourf('sqrt(x^2 + y^2)')
```

That is, \(x^2\) is interpreted as \(x.^2\) in the string you pass to `ezcontourf`.

If the function to be plotted 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, `ezcontourf('u^2 - v^3',[0,1],[3,6])` plots the contour lines for \(u^2 - v^3\) over \(0 < u < 1, 3 < v < 6\).

**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 `ezcontourf`.

```matlab
fh = @(x,y) sqrt(x.^2 + y.^2);
ezcontourf(fh)
```

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.

**Passing Additional Arguments**

If your function has additional parameters, for example, \(k\) in `myfun`:

```matlab
function z = myfun(x,y,k)
    z = x.^k - y.^k - 1;

then you can use an anonymous function to specify that parameter:

```matlab
ezcontourf(@(x,y)myfun(x,y,2))
```

**Examples**

The following mathematical expression defines a function of two variables, \(x\) and \(y\).

\[
f(x, y) = 3(1-x)^2 e^{-x^2 -(y+1)^2} - 10 \left( \frac{x}{5} - x^3 - y^5 \right) e^{-x^2 - y^2} - \frac{1}{3} e^{-(x+1)^2 - y^2}
\]
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

```matlab
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:

```matlab
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**

contourf, ezcontour, ezmesh, ezmeshc, ezplot, ezplot3, ezpolar, ezsurf, ezsurf, function_handle

“Contour Plots” on page 1-96 for related functions
**ezmesh**

**Purpose**
Easy-to-use 3-D mesh plotter

**Syntax**

```matlab
ezmesh(fun)
ezmesh(fun,domain)
ezmesh(funx,funy,funz)
ezmesh(funx,funy,funz,[smin,smax,tmin,tmax])
ezmesh(funx,funy,funz,[min,max])
ezmesh(...,n)
ezmesh(...,'circ')
ezmesh(axes_handle,...)
h = ezmesh(...)
```

**Description**
ezmesh(fun) creates a graph of \( f(x,y) \) using the mesh function. \( f(x,y) \) is plotted over the default domain: \(-2\pi < x < 2\pi, -2\pi < y < 2\pi\).

\( f(x,y) \) 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(fun,domain) plots \( f(x,y) \) 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 \( \text{min} < x < \text{max}, \text{min} < y < \text{max} \)).

ezmesh(funx,funy,funz) plots the parametric surface \( f_{\text{funx}}(s,t), f_{\text{funy}}(s,t), \) and \( f_{\text{funz}}(s,t) \) over the square: \(-2\pi < s < 2\pi, -2\pi < t < 2\pi\).

ezmesh(funx,funy,funz,[smin,smax,tmin,tmax]) or ezmesh(funx,funy,funz,[min,max]) plots the parametric surface using the specified domain.

ezmesh(...,n) plots \( f(x,y) \) over the default domain using an \( n \)-by-\( n \) grid. The default value for \( n \) is 60.

ezmesh(...,'circ') plots \( f(x,y) \) 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 handle to a surface object in h.

Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string 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.

If the function to be plotted 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 \(0 < u < 1, 3 < v < 6\).

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.

\[ \text{fh = @(x,y) sqrt(x.^2 + y.^2);} \]
\[ \text{ezmesh(fh)} \]

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.

Passing Additional Arguments

If your function has additional parameters, for example \(k\) in myfun:

\[ \text{function z = myfun(x,y,k)} \]
\[ \text{z = x.^k - y.^k - 1;} \]
then you can use an anonymous function to specify that parameter:

\[
\text{ezmesh(@(x,y)myfun(x,y,2))}
\]

**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:

\[
fh = @(x,y) x.*exp(-x.^2-y.^2);
nzhfhfhzfhfhfhf
\]

```
fh = @(x,y) x.*exp(-x.^2-y.^2);
ezmesh(fh,40)
colormap([0 0 1])
```

![Graph of the function](image)
See Also

ezmeshc, function_handle, mesh

“Function Plots” on page 1-96 for related functions
ezmeshc

**Purpose**
Easy-to-use combination mesh/contour plotter

**Syntax**
- `ezmeshc(fun)`
- `ezmeshc(fun,domain)`
- `ezmeshc(funx,funy,funz)`
- `ezmeshc(funx,funy,funz,[smin,smax,tmin,tmax])`
- `ezmeshc(funx,funy,funz,[min,max])`
- `ezmeshc(...,n)`
- `ezmeshc(...,'circ')`
- `ezmesh(axe_handle,...)`
- `h = ezmeshc(...)`

**Description**
- `ezmeshc(fun)` creates a graph of `fun(x,y)` using the `meshc` function. `fun` is plotted over the default domain `-2\pi < x < 2\pi, -2\pi < y < 2\pi`.
- `fun` 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(fun,domain)` plots `fun` 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`).
- `ezmeshc(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: `-2\pi < s < 2\pi, -2\pi < t < 2\pi`.
- `ezmeshc(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezmeshc(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.
- `ezmeshc(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.
- `ezmeshc(...,'circ')` plots `fun` over a disk centered on the domain.
- `ezmesh(axe_handle,...)` plots into the axes with handle `axe_handle` instead of the current axes (`gca`).
- `h = ezmeshc(...)` returns the handle to a surface object in `h`. 
Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string 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

\[ \text{ezmeshc('sqrt(x^2 + y^2)')} \]

That is, \( x^2 \) is interpreted as \( x \cdot x \) in the string you pass to `ezmeshc`.

If the function to be plotted 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, `ezmeshc('u^2 - v^3',[0,1],[3,6])` plots \( u^2 - v^3 \) over \( 0 < u < 1, 3 < v < 6 \).

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 `ezmeshc`.

\[ \text{fh = @(x,y) sqrt(x^2 + y^2);} \]
\[ \text{ezmeshc(fh)} \]

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.

Passing Additional Arguments

If your function has additional parameters, for example \( k \) in `myfun`:

\[ \text{function z = myfun(x,y,k)} \]
\[ \text{z = x.^k - y.^k - 1;} \]

then you can use an anonymous function to specify that parameter:

\[ \text{ezmeshc(@(x,y)myfun(x,y,2))} \]
Examples

Create a mesh/contour graph of the expression

\[ f(x, y) = \frac{y}{1 + x^2 + y^2} \]

over the domain \(-5 < x < 5, -2\pi < y < 2\pi\):

\[ \text{ezmeshc('y/(1 + x^2 + y^2)',[-5,5,-2*pi,2*pi])} \]

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, function_handle, meshc
“Function Plots” on page 1-96 for related functions
**Purpose**

Easy-to-use function plotter

![Plot Example]

**Syntax**

```matlab
ezplot(fun)
ezplot(fun,[min,max])
ezplot(fun2)
ezplot(fun2,[xmin,xmax,ymin,ymax])
ezplot(fun2,[min,max])
ezplot(funx,funy)
ezplot(funx,funy,[tmin,tmax])
ezplot(...,figure_handle)
ezplot(axes_handle,...)
h = ezplot(...)
```

**Description**

`ezplot(fun)` plots the expression `fun(x)` over the default domain `-2\pi < x < 2\pi`, where `fun(x)` is not an implicit function of only one variable.

`fun` 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(fun,[min,max])` plots `fun(x)` over the domain: `min < x < max`.

For implicitly defined functions, `fun2(x,y)`: 

`ezplot(fun2)` plots `fun2(x,y) = 0` over the default domain `-2\pi < x < 2\pi`, `-2\pi < y < 2\pi`.

`ezplot(fun2,[xmin,xmax,ymin,ymax])` plots `fun2(x,y) = 0` over `xmin < x < xmax` and `ymin < y < ymax`.

`ezplot(fun2,[min,max])` plots `fun2(x,y) = 0` over `min < x < max` and `min < y < max`.

`ezplot(funx,funy)` plots the parametrically defined planar curve `funx(t)` and `funy(t)` over the default domain `0 < t < 2\pi`. 
ezplot(funx,funy,[tmin,tmax]) plots funx(t) and funy(t) over tmin < t < 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 handle 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`,

\[
\begin{align*}
\text{fh} &= @(x,y) \sqrt{x^2 + y^2 - 1}; \\
\text{ezplot(fh)} \\
\text{axis} \quad \text{equal}
\end{align*}
\]

which plots a circle. 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.
Passing Additional Arguments

If your function has additional parameters, for example k in myfun:

```matlab
function z = myfun(x,y,k)
    z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```matlab
ezplot(@(x,y)myfun(x,y,2))
```

Examples

This example plots the implicitly defined function

\[ x^2 - y^4 = 0 \]

over the domain \([-2\pi, 2\pi]\):

```matlab
ezplot('x^2-y^4')
```
See Also  

```
ezplot3, ezpolar, function_handle, plot
```

“Function Plots” on page 1-96 for related functions
ezplot3

**Purpose**

Easy-to-use 3-D parametric curve plotter

```

Syntax

```ezplot3(funx,funy,funz)```  
`ezplot3(funx,funy,funz,[tmin,tmax])`  
`ezplot3(...,'animate')`  
`ezplot3(axes_handle,...)`  
`h = ezplot3(...)`

**Description**

`ezplot3(funx,funy,funz)` plots the spatial curve `funx(t)`, `funy(t)`, and `funz(t)` over the default domain `0 < t < 2π`.

`funx`, `funy`, and `funz` can be function handles for M-file functions or an anonymous functions (see “Function Handles” and “Anonymous Functions”) or strings (see the Remarks section).

`ezplot3(funx,funy,funz,[tmin,tmax])` plots the curve `funx(t)`, `funy(t)`, and `funz(t)` over the domain `tmin < t < tmax`.

`ezplot3(...,'animate')` produces an animated trace of the spatial curve.

`ezplot3(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezplot3(...)` returns the handle to the plotted objects in `h`.

**Remarks**

**Passing the Function as a String**

Array multiplication, division, and exponentiation are always implied in the expression you pass to `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

```
ezplot3('s/2','2*s','s^2')
```
That is, s/2 is interpreted as s ./2 in the string you pass to 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 fh to ezplot3.

```matlab
fh1 = @(s) s./2; fh2 = @(s) 2.*s; fh3 = @(s) s.^2;
edzplot3(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.

**Passing Additional Arguments**

If your function has additional parameters, for example k in myfuntk:

```matlab
function s = myfuntk(t,k)
s = t.^k.*sin(t);
```

then you can use an anonymous function to specify that parameter:

```matlab
edzplot3(@(cos,@(t)myfuntk(t,1),@sqrt)
```

**Examples**

This example plots the parametric curve

\[ x = \sin(t), \quad y = \cos(t), \quad z = t \]

over the domain [0,6\pi]:

```matlab
edzplot3('sin(t)','cos(t)','t',[0,6*pi])
```
**ezplot3**

$x = \sin(t), \ y = \cos(t), \ z = t$

**See Also**
ezplot, ezpolar, function_handle, plot3

“Function Plots” on page 1-96 for related functions
**Purpose**

Easy-to-use polar coordinate plotter

**Syntax**

```plaintext
ezpolar(fun)
ezpolar(fun,[a,b])
ezpolar(axes_handle,...)
h = ezpolar(...)
```

**Description**

`ezpolar(fun)` plots the polar curve $\rho = \text{fun}(\theta)$ over the default domain $0 < \theta < 2\pi$.

`fun` can be a function handle for an M-file function or an anonymous function (see “Function Handles” and “Function Handles”) or a string (see the Remarks section).

`ezpolar(fun,[a,b])` plots `fun` for $a < \theta < b$.

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

`h = ezpolar(...)` returns the handle to a line object in `h`.

**Remarks**

**Passing the Function as a String**

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

\[ t^2 \cos(t) \]

which represents an implicitly defined function, is written as

```matlab
ezpolar('t^2*cos(t)')
```

That is, `t^2` is interpreted as `t.^2` in the string you pass to `ezpolar`. 
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 ezpolar.

\[
fh = @(t) t.^2.*\cos(t);
\]
\[
ezpolar(fh)
\]

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.

Passing Additional Arguments

If your function has additional parameters, for example \(k1\) and \(k2\) in myfun:

\[
function \ s = myfun(t,k1,k2)
\]
\[
s = \sin(k1*t).*\cos(k2*t);
\]

then you can use an anonymous function to specify the parameters:

\[
ezpolar(@(t)myfun(t,2,3))
\]

Examples

This example creates a polar plot of the function

\[1 + \cos(t)\]

over the domain \([0, 2\pi]\):

\[
ezpolar('1+\cos(t)')
\]
See Also  

ezplot, ezplot3, function_handle, plot, plot3, polar  

“Function Plots” on page 1-96 for related functions
Purpose

Easy-to-use 3-D colored surface plotter

Syntax

ezsurf(fun)
ezsurf(fun,domain)
ezsurf(funx,funy,funz)
ezsurf(funx,funy,funz,[smin,smax,tmin,tmax])
ezsurf(funx,funy,funz,[min,max])
ezsurf(...,n)
ezsurf(...,'circ')
ezsurf(axes_handle,...)
h = ezsurf(...)

Description

ezsurf(fun) creates a graph of fun(x,y) using the surf function. fun is plotted over the default domain: -2π < x < 2π, -2π < y < 2π.
fun 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(fun,domain) plots fun 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).

ezsurf(funx,funy,funz) plots the parametric surface funx(s,t), funy(s,t), and funz(s,t) over the square: -2π < s < 2π, -2π < t < 2π.

ezsurf(funx,funy,funz,[smin,smax,tmin,tmax]) or ezsurf(funx,funy,funz,[min,max]) plots the parametric surface using the specified domain.

ezsurf(...,n) plots fun over the default domain using an n-by-n grid. The default value for n is 60.

ezsurf(...,'circ') plots fun 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 = ezsurf(...) returns the handle to a surface object in h.

Remarks

ezsurf and ezsurfc do not accept complex inputs.

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.

If the function to be plotted 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, \( \text{ezsurf('u^2 - v^3',[0,1],[3,6])} \) plots \( u^2 - v^3 \) over \( 0 < u < 1, 3 < v < 6 \).

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 ezsurf.

\[
\begin{align*}
\text{fh} &= @(x,y) \sqrt{x.^2 + y.^2}; \\
\text{ezsurf(fh)}
\end{align*}
\]

Note that when using function handles, you must use the array power, array multiplication, and array division operators (\( .^*, .^/, \)) since ezsurf does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example \( k \) in myfun:

\[
\begin{align*}
\text{function} \ z &= \text{myfun}(x,y,k1,k2,k3) \\
z &= x.*(y.^k1)./(x.^k2 + y.^k3);
\end{align*}
\]
then you can use an anonymous function to specify that parameter:

```matlab
ezsurf(@(x,y)myfun(x,y,2,2,4))
```

**Examples**

ezsurf does not graph points where the mathematical function is not defined (these data points are set to NaNs, which do not plot). This example illustrates this filtering of singularities/discontinuous points by graphing the function

\[ f(x, y) = \text{real}(\text{atan}(x + iy)) \]

over the default domain \(-2\pi < x < 2\pi, -2\pi < y < 2\pi:\)

```matlab
ezsurf('real(atan(x+i*y))')
```

![Graph of real(atan(x+i*y))](image)
Using `surf` to plot the same data produces a graph without filtering of discontinuities (as well as requiring more steps):

\[
[x,y] = \text{meshgrid(linspace(-2*pi,2*pi,60))};
\]
\[
z = \text{real(atan(x+i.*y))};
\]
\[
surf(x,y,z)
\]

Note also that `ezsurf` creates graphs that have axis labels, a title, and extend to the axis limits.

**See Also**

`ezmesh`, `ezsurf`, `function_handle`, `surf`

“Function Plots” on page 1-96 for related functions
**ezsurfc**

**Purpose**
Easy-to-use combination surface/contour plotter

**Syntax**

```
ezsurfc(fun)
ezsurfc(fun,domain)
ezsurfc(funx,funy,funz)
ezsurfc(funx,funy,funz,[smin,smax,tmin,tmax])
ezsurfc(funx,funy,funz,[min,max])
ezsurfc(...,n)
ezsurfc(...,'circ')
ezsurfc(axes_handle,...)
h = ezsurfc(...)
```

**Description**
ezsurfc(fun) creates a graph of fun(x,y) using the surf function. The function fun is plotted over the default domain: -2π < x < 2π, -2π < y < 2π.

fun 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(fun,domain) plots fun 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).

ezsurfc(funx,funy,funz) plots the parametric surface funx(s,t), funy(s,t), and funz(s,t) over the square: -2π < s < 2π, -2π < t < 2π.

ezsurfc(funx,funy,funz,[smin,smax,tmin,tmax]) or ezsurfc(funx,funy,funz,[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(...,'circ') 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 the graphics objects in h.

Remarks

ezsurf and ezsurfc do not accept complex inputs.

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.

If the function to be plotted 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, ezsurfc('u^2 - v^3',[0,1],[3,6]) plots \( u^2 - v^3 \) over \( 0 < u < 1, 3 < v < 6. \)

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 ezsurfc.

\[
\text{fh} = @(x,y) \sqrt{x.^2 + y.^2}; \\
\text{ezsurf(fh)}
\]

Note that when using function handles, you must use the array power, array multiplication, and array division operators (\( .^, .*, ./ \)) since ezsurfc does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example \( k \) in myfun:
function z = myfun(x,y,k1,k2,k3)
    z = x.*(y.^k1)./(x.^k2 + y.^k3);

then you can use an anonymous function to specify that parameter:

    ezsurfc(@(x,y)myfun(x,y,2,2,4))

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:

    ezsurfc('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, function_handle, surfc

“Function Plots” on page 1-96 for related functions
Symbols and Numerics

\& 2-52 2-59
\' 2-40
\* 2-40
\+ 2-40
\- 2-40
\/ 2-40
: 2-66
< 2-50
> 2-50
@ 2-1443
\ 2-40
^ 2-40
| 2-52 2-59
~ 2-52 2-59
&& 2-59
== 2-50
}} 2-65
|| 2-59
~= 2-50
1-norm 2-2570 2-3023
2-norm (estimate of) 2-2572

A

abs 2-69
absolute accuracy
    BVP 2-478
    DDE 2-896
    ODE 2-2620
absolute value 2-69
Accelerator
    Uimenu property 2-3898
accumarray 2-70
accuracy
    of linear equation solution 2-688
    of matrix inversion 2-688
acos 2-77
acosd 2-79
acosh 2-80
acot 2-82
acotd 2-84
acoth 2-85
acsc 2-87
acscd 2-89
acsch 2-90
activelegend 2-2810
actxcontrol 2-92
actxcontrollist 2-100
actxcontrolselect 2-101
actxserver 2-106
Adams-Bashforth-Moulton ODE solver 2-2609
addCause, MException method 2-110
addevent 2-114
addframe
    AVI files 2-116
addition (arithmetic operator) 2-40
addlistener 2-118
addOptional method
    of inputParser object 2-120
addParamValue method
    of inputParser object 2-123
addpath 2-126
addpref function 2-128
addprop dynamicprops method 2-129
addproperty 2-130
addRequired method
    of inputParser object 2-132
addressing selected array elements 2-66
addsample 2-135
addsampletocollection 2-137
adddate 2-139
addts 2-141
align function 2-145
aligning scattered data
Airy functions
    relationship to modified Bessel functions 2-143
multi-dimensional 2-2492
two-dimensional 2-1585
ALim, Axes property 2-296
all 2-153
allchild function 2-155
allocation of storage (automatic) 2-4231
AlphaData
  image property 2-1768
  surface property 2-3568
  surfaceplot property 2-3591
AlphaDataMapping
  image property 2-1769
  patch property 2-2713
  surface property 2-3569
  surfaceplot property 2-3591
AmbientLightColor, Axes property 2-297
AmbientStrength, Axes property
  Patch property 2-2714
  Surface property 2-3569
  surfaceplot property 2-3592
amd 2-161 2-2058
analytical partial derivatives (BVP) 2-479
analyzing M-file code 2-2413
and 2-166
and (M-file function equivalent for &) 2-56
AND, logical
  bit-wise 2-424
angle 2-168
annotating graphs
  deleting annotations 2-171
  in plot edit mode 2-2811
Annotation
  areaseries property 2-222
  contourgroup property 2-714
  errorbarseries property 2-1090
  hggroup property 2-1674
  hgtransform property 2-1703
  image property 2-1769
  line property 2-356 2-2117
lineseries property 2-2132
Patch property 2-2714
quivergroup property 2-2972
rectangle property 2-3042
scattergroup property 2-3201
stairseries property 2-3386
stemsseries property 2-3420
Surface property 2-3570
surfaceplot property 2-3592
text property 2-3675
annotation function 2-169
ans 2-212
anti-diagonal 2-1618
any 2-213
arccosecant 2-87
arccosine 2-77
arccotangent 2-82
arcsecant 2-245
arcsine 2-250
arctangent 2-259
  four-quadrant 2-261
arguments, M-file
  checking number of inputs 2-2483
  checking number of outputs 2-2487
  number of input 2-2485
  number of output 2-2485
  passing variable numbers of 2-4092
arithmetic operations, matrix and array
  distinguished 2-40
arithmetic operators
  reference 2-40
array
  addressing selected elements of 2-66
dimension
  rearrange 2-1354
displaying 2-990
  flip dimension of 2-1354
left division (arithmetic operator) 2-42
maximum elements of 2-2299
mean elements of 2-2305
median elements of 2-2308
minimum elements of 2-2385
multiplication (arithmetic operator) 2-41
of all ones 2-2641
of all zeros 2-4231
power (arithmetic operator) 2-42
product of elements 2-2885
rearrange
dimension 2-1354
removing first n singleton dimensions of 2-3274
removing singleton dimensions of 2-3373
reshaping 2-3114
reverse dimension of 2-1354
right division (arithmetic operator) 2-41
shift circularly 2-597
shifting dimensions of 2-3274
size of 2-3288
sorting elements of 2-3310
structure 2-1535 2-3140 2-3260
sum of elements 2-3548
swapping dimensions of 2-1924 2-2785
transpose (arithmetic operator) 2-42
arrayfun 2-238
arrays
detecting empty 2-1940
maximum size of 2-686
opening 2-2642
arrays, structure
field names of 2-1231
arrowhead matrix 2-673
ASCII
delimited files
writing 2-1008
ASCII data
converting sparse matrix after loading from 2-3323
reading 2-1004
reading from disk 2-2183
saving to disk 2-3176
asciifunction 2-244
asec 2-245
asecd 2-247
asech 2-248
asin 2-250
asind 2-252
asinh 2-253
aspect ratio of axes 2-816 2-2747
assert 2-255
assignin 2-257
atan 2-259
atan2 2-261
atand 2-263
atanh 2-264
.au files
reading 2-281
writing 2-282
audio
saving in AVI format 2-283
signal conversion 2-2110 2-2466
audiodevinfo 2-266
audioplayer 2-268
audiorecorder 2-274
aufinfo 2-280
auread 2-281
AutoScale
quivergroup property 2-2973
AutoScaleFactor
quivergroup property 2-2973
autoselection of OpenGL 2-1269
auwrite 2-282
average of array elements 2-2305
average, running 2-1320
avi 2-283
avifile 2-283
aviinfo 2-287
aviread 2-289
axes 2-290
editing 2-2811
setting and querying data aspect ratio 2-816
setting and querying limits 2-420
setting and querying plot box aspect ratio 2-2747
Axes
creating 2-290
defining default properties 2-294
fixed-width font 2-313
property descriptions 2-296
axis 2-334
axis crossing. See zero of a function
azimuth (spherical coordinates) 2-3339
azimuth of viewpoint 2-4112
B
BackFaceLighting
   Surface property 2-3571
   surfaceplot property 2-3594
BackFaceLightingpatch property 2-2716
BackgroundColor
   annotation textbox property 2-202
   Text property 2-3676
   Uitable property 2-3970
BackGroundColor
   Uicontrol property 2-3850
badly conditioned 2-3023
balance 2-340
BarLayout
   barseries property 2-357
BarWidth
   barseries property 2-357
base to decimal conversion 2-376
base two operations
   conversion from decimal to binary 2-912
   logarithm 2-2205
   next power of two 2-2566
base2dec 2-376
Baseline
   barseries property 2-357
   stem property 2-3421
BaseValue
   areaseries property 2-223
   barseries property 2-358
   stem property 2-3421
beep 2-377
BeingDeleted
   areaseries property 2-223
   barseries property 2-358
   contour property 2-715
   errorbar property 2-1091
   group property 2-1236 2-1770 2-3678
   hggroup property 2-1675
   hgtransform property 2-1704
   light property 2-2100
   line property 2-2118
   lineseries property 2-2133
   quivergroup property 2-2973
   rectangle property 2-3043
   scatter property 2-3202
   stairseries property 2-3387
   stem property 2-3421
   surface property 2-3571
   surfaceplot property 2-3594
   transform property 2-2716
   Uipushtool property 2-3934
   Uitable property 2-3971
   Uitoggletool property 2-4000
   Uitoolbar property 2-4013
bench 2-378
benchmark 2-378
Bessel functions
   first kind 2-387
   modified, first kind 2-384
   modified, second kind 2-390
   second kind 2-393
Bessel functions, modified
   relationship to Airy functions 2-143
Bessel's equation
   (defined) 2-387
   modified (defined) 2-384
besseli 2-384
besselj 2-387
besselk 2-390
bessely 2-393
beta 2-397
beta function
  (defined) 2-397
  incomplete (defined) 2-399
  natural logarithm 2-402
betainc 2-399
betaln 2-402
bicg 2-403
bicgstab 2-412
bicgstabl 2-418
BiConjugate Gradients method 2-403
BiConjugate Gradients Stabilized method 2-412
  2-418
big endian formats 2-1373
bin2dec 2-421
binary
  data
    writing to file 2-1455
  files
    reading 2-1403
    mode for opened files 2-1371
binary data
  reading from disk 2-2183
  saving to disk 2-3176
binary function 2-422
binary to decimal conversion 2-421
bisection search 2-1465
bit depth
  querying 2-1791
bit-wise operations
  AND 2-424
  get 2-427
  OR 2-431
  set bit 2-432
  shift 2-433
  XOR 2-435
bitand 2-424
bitcmp 2-425
bitget 2-427
bitmaps
  writing 2-1818
bitmax 2-429
bitor 2-431
bitset 2-432
bitshift 2-433
bitxor 2-435
blanks 2-436
  removing trailing 2-908
blkdiag 2-437
BMP files
  writing 2-1818
bold font
  TeX characters 2-3701
boundary value problems 2-485
box 2-438
Box, Axes property 2-298
braces, curly (special characters) 2-62
brackets (special characters) 2-62
break 2-439
breakpoints
  listing 2-866
  removing 2-853
  resuming execution from 2-856
  setting in M-files 2-870
brighten 2-440
browser
  for help 2-1658
brush 2-442
bsxfun 2-452
bubble plot (scatter function) 2-3196
Buckminster Fuller 2-3645
builtin 2-455
BusyAction
  areaseries property 2-223
  Axes property 2-298
  barseries property 2-358
contour property 2-715
errorbar property 2-1092
Figure property 2-1237
hggroupproperty 2-1676
hgtransform property 2-1705
Image property 2-1771
Light property 2-2100
line property 2-2119
Line property 2-2133
patch property 2-2716
quivergroupproperty 2-2974
rectangle property 2-3044
Root property 2-3144
scatter property 2-3203
stairseries property 2-3388
stem property 2-3422
Surface property 2-3571
surfaceplot property 2-3594
Text property 2-3678
Uicontextmenu property 2-3835
Uicontrol property 2-3851
Uimenuproperty 2-3899
Uitoggletool property 2-4001
Uitoolbar property 2-4013

ButtonDownFcn

area series property 2-224
Axes property 2-299
barseries property 2-359
contour property 2-716
errorbar property 2-1092
Figure property 2-1237
hggroupproperty 2-1676
hgtransform property 2-1705
Image property 2-1771
Light property 2-2101
Line property 2-2119
lineseries property 2-2134
patch property 2-2716
quivergroupproperty 2-2974
rectangle property 2-3044
Root property 2-3144
scatter property 2-3203
stairseries property 2-3388
stem property 2-3422
Surface property 2-3571
surfaceplot property 2-3595
Text property 2-3572
Uicontrol property 2-3679
Uitable property 2-3972

BVP solver properties
analytical partial derivatives 2-479
error tolerance 2-477
Jacobian matrix 2-479
mesh 2-482
singular BVPs 2-482
solution statistics 2-483
vectorization 2-478

bvp4c 2-456
bvp5c 2-467
bvpget 2-472
bvpinit 2-473
bvpset 2-476
bvpxtend 2-485

C

calendar 2-486
call history 2-2892
Callback

Uicontextmenu property 2-3836
Uicontrol property 2-3852
Uimenuproperty 2-3900
CallbackObject, Root property 2-3144
calllib 2-487
callSoapService 2-489
camdolly 2-491
camera
dollying position 2-491
moving camera and target positions 2-491
placing a light at 2-495
positioning to view objects 2-497
rotating around camera target 2-499 2-501
rotating around viewing axis 2-505
setting and querying position 2-502
setting and querying projection type 2-504
setting and querying target 2-506
setting and querying up vector 2-508
setting and querying view angle 2-510
CameraPosition, Axes property 2-300
CameraPositionMode, Axes property 2-300
CameraTarget, Axes property 2-301
CameraTargetMode, Axes property 2-301
CameraUpVector, Axes property 2-301
CameraUpVectorMode, Axes property 2-301
CameraViewAngle, Axes property 2-302
CameraViewAngleMode, Axes property 2-302
camlight 2-495
camlookat 2-497
camorbit 2-499
campan 2-501
campos 2-502
camproj 2-504
camroll 2-505
camtarget 2-506
camup 2-508
camva 2-510
camzoom 2-512
cart2pol 2-516
cart2sph 2-518
Cartesian coordinates 2-516 2-518 2-2823 2-3339
case 2-519
  in switch statement (defined) 2-3632
  lower to upper 2-4054
  upper to lower 2-2217
cast 2-521
cat 2-522
catch 2-524
caxis 2-527
Cayley-Hamilton theorem 2-2843
cd 2-532
cd (ftp) function 2-537
CData
  Image property 2-1772
  scatter property 2-3204
  Surface property 2-3573
  surfaceplot property 2-3596
  Uicontrol property 2-3853
  Uipushtool property 2-3935
  Uitoggletool property 2-4001
CDataMapping
  Image property 2-1774
  patch property 2-2719
  Surface property 2-3574
  surfaceplot property 2-3596
CDataMode
  surfaceplot property 2-3597
CDatapatch property 2-2717
CDataSource
  scatter property 2-3204
  surfaceplot property 2-3597
cdf2rdf 2-538
cdfepoch 2-540
cdfinfo 2-541
cdfread 2-545
cdfwrite 2-549
ceil 2-552
cell 2-553
cell array
  conversion to from numeric array 2-2580
  creating 2-553
  structure of, displaying 2-566
cell2mat 2-555
cell2struct 2-557
celldisp 2-559
CellEditCallback
  Uitable property 2-3973
cellfun 2-560
cellplot 2-566
CellSelectionCallback
   Uitable property 2-3975

cgs 2-569
char 2-574
characters
   conversion, in format specification string 2-1392 2-3362
   escape, in format specification string 2-1393 2-3362
check boxes 2-3843
Checked, Uimenu property 2-3900
checkerboard pattern (example) 2-3103
checkin 2-575
   examples 2-576
   options 2-575
checkout 2-578
   examples 2-579
   options 2-578
child functions 2-2887
Children
   areaseries property 2-225
   Axes property 2-303
   barseries property 2-360
   contour property 2-716
   errorbar property 2-1093
   Figure property 2-1238
   hggroup property 2-1677
   hgtransform property 2-1706
   Image property 2-1774
   Light property 2-2101
   Line property 2-2120
   lineseries property 2-2134
   patch property 2-2720
   quivergroup property 2-2975
   rectangle property 2-3045
   Root property 2-3144
   scatter property 2-3205
   stairseries property 2-3389
   stem property 2-3423
   Surface property 2-3574
   surfaceplot property 2-3598
   Text property 2-3680
   Uicontextmenu property 2-3836
   Uicontrol property 2-3854
   Uimenu property 2-3901
   Uitable property 2-3975
   Uitoolbar property 2-4014
chol 2-581
Cholesky factorization 2-581
   (as algorithm for solving linear equations) 2-2409
   lower triangular factor 2-2701
   preordering for 2-673
cholinc 2-586
cholupdate 2-594
circle
   rectangle function 2-3037
circshift 2-597
cla 2-601
clabel 2-602
class 2-608
class, object. See object classes
classes
   field names 2-1231
   loaded 2-1850
clc 2-613 2-623 2-3273
clear 2-614
   serial port I/O 2-622
clearing
   Command Window 2-613
   items from workspace 2-614
   Java import list 2-616
clf 2-623
ClickedCallback
   Uipushtool property 2-3936
   Uitoggletool property 2-4002
CLim, Axes property 2-304
CLimMode, Axes property 2-304
clipboard 2-624
Clipping
areaseries property 2-225
Axes property 2-305
barseries property 2-360
contour property 2-717
errobar property 2-1093
Figure property 2-1239
hggroup property 2-1677
hgtransform property 2-1706
Image property 2-1775
Light property 2-2101
Line property 2-2120
lineseries property 2-2135
quivergroupproperty 2-2975
rectangle property 2-3045
Root property 2-3145
scatter property 2-3205
stairseries property 2-3389
stem property 2-3423
Surface property 2-3575
surfaceplot property 2-3598
Text property 2-3680
Uicontrol property 2-3854
uitable property 2-3975
Clippingpatch property 2-2720
clock 2-625
close 2-626
   AVI files 2-628
close (ftp) function 2-629
CloseRequestFcn, Figure property 2-1239
closest point search 2-1032
closest triangle search 2-3795
closing
   files 2-1188
   MATLAB 2-2962
cmapeditor 2-653
cmopts 2-631
cmpermute 2-633
cmunique 2-634
code
   analyzer 2-2413
colamd 2-637
colon operator 2-66
color
   quantization performed by rgb2ind 2-3128
Color
   annotation arrow property 2-173
   annotation doublearrow property 2-177
   annotation line property 2-185
   annotation textbox property 2-202
   Axes property 2-305
   errorbar property 2-1093
   Figure property 2-1241
   Light property 2-2101
   Line property 2-2121
   lineseries property 2-2135
   quivergroupproperty 2-2976
   stairseries property 2-3389
   stem property 2-3424
   Text property 2-3680
textarrow property 2-191
   color approximation
      performed by rgb2ind 2-3128
   color of fonts, see also FontColor property 2-3701
colorbar 2-641
colormap 2-648
editor 2-653
Colormap, Figure property 2-1242
colormaps
   converting from RGB to HSV 2-3126
   plotting RGB components 2-3130
   rearranging colors in 2-633
   removing duplicate entries in 2-634
ColorOrder, Axes property 2-305
ColorSpec 2-671
colperm 2-673
ColumnEditable
   Uitable property 2-3976
ColumnFormat
   Uitable property 2-3976
ColumnName
Commands

- actxcontrol 2-92
- actxcontrollist 2-100
- actxcontrolselect 2-101
- actxserver 2-106
- addproperty 2-130
- delete 2-944
- deleteproperty 2-951
- eventlisteners 2-1126
- events 2-1130
- get 2-1511
- inspect 2-1866
- invoke 2-1921
- iscom 2-1936
- isevent 2-1949
- isinterface 2-1961
- ismethod 2-1972
- load 2-2188
- move 2-2444
- propedit 2-2896
- registerevent 2-3088
- release 2-3093
- save 2-3184
- set 2-3242
- unregisterallevents 2-4038
- unregistererevent 2-4041

Object properties

- isprop 2-1995

Server methods

- Execute 2-1132
- Feval 2-1201

Combinations of n elements 2-2491

Combs 2-2491

Comet 2-675

Comet3 2-677

Comma (special characters) 2-64

Command syntax 2-1655 2-3650

Command Window

- clearing 2-613
- cursor position 2-1726
- get width 2-680
- commandhistory 2-679

Commands

- help for 2-1654 2-1664
- system 2-3653
- UNIX 2-4029
- commandwindow 2-680

Comments

- block of 2-64

Common elements. See set operations, intersection

Compan 2-681

Companion matrix 2-681

Compass 2-682

CompilerConfiguration 2-2369

CompilerConfigurationDetails 2-2369

Complementary error function

- (defined) 2-1079
- scaled (defined) 2-1079

Complete elliptic integral

- (defined) 2-1062
- modulus of 2-1060 2-1062

Complex 2-684 2-1759

- exponential (defined) 2-1140
- logarithm 2-2202 to 2-2203
- numbers 2-1735
- numbers, sorting 2-3310 2-3314
- phase angle 2-168
- sine 2-3282
- unitary matrix 2-2925

See also imaginary

Complex conjugate 2-698

- sorting pairs of 2-775

Complex data

- creating 2-684

Complex numbers, magnitude 2-69
complex Schur form 2-3219
compression
  lossy 2-1822
computer 2-686
computer MATLAB is running on 2-686
concatenation
  of arrays 2-522
cond 2-688
condeig 2-689
condest 2-690
condition number of matrix 2-688 2-3023
  improving 2-340
coneplot 2-692
conj 2-698
conjugate, complex 2-698
  sorting pairs of 2-775
connecting to FTP server 2-1435
containers
  Map 2-1963 2-2042 2-2081 2-2254 2-3098
    2-3291 2-4085
contents.m file 2-1655
context menu 2-3831
continuation (..., special characters) 2-64
continue 2-699
continued fraction expansion 2-3017
contour
  and mesh plot 2-1160
  filled plot 2-1152
  functions 2-1148
    of mathematical expression 2-1149
    with surface plot 2-1178
contour3 2-705
contourc 2-708
contourf 2-710
ContourMatrix
  contour property 2-717
contours
  in slice planes 2-735
contourslice 2-735
contrast 2-739
conv 2-740
conv2 2-742
conversion
  base to decimal 2-376
  binary to decimal 2-421
  Cartesian to cylindrical 2-516
  Cartesian to polar 2-516
  complex diagonal to real block diagonal 2-538
  cylindrical to Cartesian 2-2823
  decimal number to base 2-905 2-911
  decimal to binary 2-912
  decimal to hexadecimal 2-913
  full to sparse 2-3320
  hexadecimal to decimal 2-1668
  integer to string 2-1880
  lowercase to uppercase 2-4054
  matrix to string 2-2264
  numeric array to cell array 2-2580
  numeric array to logical array 2-2206
  numeric array to string 2-2584
  partial fraction expansion to
    pole-residue 2-3116
  polar to Cartesian 2-2823
  pole-residue to partial fraction
    expansion 2-3116
  real to complex Schur form 2-3173
  spherical to Cartesian 2-3339
  string matrix to cell array 2-568
  string to numeric array 2-3447
  uppercase to lowercase 2-2217
  vector to character string 2-574
conversion characters in format specification
  string 2-1392 2-3362
convex hulls
  multidimensional visualization 2-751
  two-dimensional visualization 2-748
convhull 2-748
convhulln 2-751
convn 2-754
convolution 2-740
inverse. See deconvolution

two-dimensional 2-742

cordinate system and viewpoint 2-4112

cordinates

  Cartesian 2-516 2-518 2-2823 2-3339
  cylindrical 2-516 2-518 2-2823
  polar 2-516 2-518 2-2823
  spherical 2-3339

cordinates. 2-516

  See also conversion

  copyfile 2-755
  copyobj 2-758
  corrcoef 2-760
  cos 2-763
  cosd 2-765
  cosecant

    hyperbolic 2-789
    inverse 2-87
    inverse hyperbolic 2-90
  cosh 2-766
  cosine 2-763

    hyperbolic 2-766
    inverse 2-77
    inverse hyperbolic 2-80
  cot 2-768
  cotangent 2-768

    hyperbolic 2-771
    inverse 2-82
    inverse hyperbolic 2-85
  cotd 2-770
  coth 2-771
  cov 2-773
  cplxpair 2-775
  cputime 2-776
  create, RandStream method 2-777
  createClassFromWsdl 2-779
  createCopy method

    of inputParser object 2-781
  CreateFcn

    areaseries property 2-225

  Axes property 2-306
  barseries property 2-360
  contour property 2-718
  errorbar property 2-1094
  Figure property 2-1242
  group property 2-1706
  hggroup property 2-1677
  Image property 2-1775
  Light property 2-2102
  Line property 2-2121
  lineseries property 2-2135
  patch property 2-2720
  quivergroup property 2-2976
  rectangle property 2-3046
  Root property 2-3145
  scatter property 2-3205
  stairseries property 2-3390
  stemseries property 2-3424
  Surface property 2-3575
  surfaceplot property 2-3598
  Text property 2-3680
  Uicontextmenu property 2-3836
  Uicontrol property 2-3854
  Uimenu property 2-3901
  Uipushtool property 2-3936
  Uitable property 2-3983
  Uitoggletool property 2-4002
  Uitoolbar property 2-4014
  createSoapMessage 2-783

  creating your own MATLAB functions 2-1441
  cross 2-785
  cross product 2-785
  csc 2-786
  csd 2-788
  csch 2-789
  csvread 2-791
  csvwrite 2-794
  ctranspose (M-file function equivalent for

    \q) 2-46
  ctranspose (timeseries) 2-796
cubic interpolation 2-1897 2-1900 2-1903 2-2757
    piecewise Hermite 2-1887
cubic spline interpolation
    one-dimensional 2-1887 2-1897 2-1900 2-1903
cumprod 2-798
cumsum 2-800
cumtrapz 2-802
cumulative
    product 2-798
    sum 2-800
curl 2-804
curly braces (special characters) 2-62
current directory 2-2918
    changing 2-532
CurrentAxes 2-1243
CurrentAxes, Figure property 2-1243
CurrentCharacter, Figure property 2-1244
CurrentFigure, Root property 2-3145
CurrentObject, Figure property 2-1244
CurrentPoint
    Axes property 2-306
    Figure property 2-1244
cursor images
    reading 2-1805
cursor position 2-1726
Curvature, rectangle property 2-3047
curve fitting (polynomial) 2-2835
customverctrl 2-807
Cuthill-McKee ordering, reverse 2-3635 2-3645
cylinder 2-808
cylindrical coordinates 2-516 2-518 2-2823

D
daqread 2-811
daspect 2-816
data
    ASCII
        reading from disk 2-2183
    binary
        writing to file 2-1455
    binary, saving to disk 2-3176
computing 2-D stream lines 2-3457
computing 3-D stream lines 2-3459
formatted
    reading from files 2-1421
    writing to file 2-1391
formatting 2-1391 2-3360
isosurface from volume data 2-1987
reading binary from disk 2-2183
reading from files 2-3706
reducing number of elements in 2-3062
smoothing 3-D 2-3303
writing to strings 2-3360
Data
    Uitable property 2-3984
data aspect ratio of axes 2-816
data brushing
    different plot types 2-443
    gestures for 2-448
    restrictions on 2-445
data types
    complex 2-684
data, aligning scattered
        multi-dimensional 2-2492
        two-dimensional 2-1585
data, ASCII
    converting sparse matrix after loading
        from 2-3323
DataAspectRatio, Axes property 2-308
DataAspectRatioMode, Axes property 2-311
datatipinfo 2-827
date 2-828
date and time functions 2-1073
date string
    format of 2-833
date vector 2-850
datenum 2-829
Index

datestr 2-833
datevec 2-849
dbclear 2-853
dbcont 2-856
dbdown 2-857
dblquad 2-858
dbmx 2-860
dbquit 2-861
dbstack 2-863
dbstatus 2-866
dbstep 2-868
dbstop 2-870
dbtype 2-881
dbup 2-882
DDE solver properties
  error tolerance 2-895
  event location 2-901
  solver output 2-897
  step size 2-899
dde23 2-883
ddeget 2-888
ddephas2 output function 2-898
ddephas3 output function 2-898
ddeplot output function 2-898
ddeprint output function 2-898
ddesd 2-889
ddeset 2-894
del 2-905
deblank 2-908
debugging
  changing workspace context 2-857
  changing workspace to calling M-file 2-882
  displaying function call stack 2-863
  M-files 2-2041 2-2887
  MEX-files on UNIX 2-860
  removing breakpoints 2-853
  resuming execution from breakpoint 2-868
  setting breakpoints in 2-870
  stepping through lines 2-868
dec2base 2-905 2-911
dec2bin 2-912
dec2hex 2-913
decic function 2-914
decimal number to base conversion 2-905 2-911
decimal point (.)
  (special characters) 2-63
  to distinguish matrix and array operations 2-40
de decomposition
  Dulmage-Mendelsohn 2-1012
  "economy-size" 2-2925 2-3624
  orthogonal-triangular (QR) 2-2925
  Schur 2-3219
  singular value 2-3016 2-3624
deconv 2-916
deconvolution 2-916
definite integral 2-2937
del operator 2-917
del2 2-917
delaunay 2-926
Delaunay tessellation
  3-dimensional visualization 2-933
  multidimensional visualization 2-937
Delaunay triangulation
  visualization 2-933
delete 2-942 2-944
  serial port I/O 2-948
  timer object 2-950
delete (ftp) function 2-946
delete handle method 2-947
DeleteFcn
  areaseries property 2-226
  Axes property 2-312
  barseries property 2-361
  contour property 2-718
  errorbar property 2-1094
  Figure property 2-1246
  hggroupproperty 2-1678
hgtransform property 2-1707
Image property 2-1775
Light property 2-2103
lineseries property 2-2136
quivergroup property 2-2976
Root property 2-3145
scatter property 2-3206
stairseries property 2-3390
stem property 2-3425
Surface property 2-3575
surfaceplot property 2-3599
Text property 2-3681 2-3684
Uicontextmenu property 2-3837 2-3855
Uimenu property 2-3902
Uipushtool property 2-3937
Uitable property 2-3985
Uitoggletool property 2-4003
Uitoolbar property 2-4016
DeleteFcn, line property 2-2122
DeleteFcn, rectangle property 2-3047
DeleteFcnpatch property 2-2721
deleteproperty 2-2951
deleting
files 2-942
items from workspace 2-614
delevent 2-953
delimaters in ASCII files 2-1004 2-1008
delsample 2-954
delsamplefromcollection 2-955
demo 2-956
demos
in Command Window 2-1036
density
of sparse matrix 2-2567
depdir 2-962
dependence, linear 2-3540
dependent functions 2-2887
depfun 2-963
derivative
approximate 2-979
determinant of a matrix 2-967
detrend 2-968
detrend (timeseries) 2-970
deval 2-971
diag 2-973
diagonal 2-973
anti- 2-1618
k-th (illustration) 2-3766
main 2-973
sparse 2-3325
dialog 2-975
dialog box
error 2-1108
help 2-1662
input 2-1855
list 2-2178
message 2-2459
print 2-2875
question 2-2958
warning 2-4144
diary 2-977
Diary, Root property 2-3146
DiaryFile, Root property 2-3146
diff 2-979
differences
between adjacent array elements 2-979
between sets 2-3256
differential equation solvers
  defining an ODE problem 2-2611
  ODE boundary value problems 2-456 2-467
    adjusting parameters 2-476
    extracting properties 2-472
    extracting properties of 2-1112 to 2-1113
      2-3763 to 2-3764
    forming initial guess 2-473
  ODE initial value problems 2-2598
    adjusting parameters of 2-2618
    extracting properties of 2-2617
parabolic-elliptic PDE problems 2-2766
diffuse 2-981
DiffuseStrength
  Surface property 2-3576
  surfaceplot property 2-3599
DiffuseStrengthpatch property 2-2721
digamma function 2-2900
dimension statement (lack of in MATLAB) 2-4231
dimensions
  size of 2-3288
Diophantine equations 2-1496
dir 2-982
dir (ftp) function 2-987
directive term of a partial fraction expansion 2-3116
directive
  %#eml 2-2416
  %#ok 2-2416
directories 2-532
  adding to search path 2-126
  checking existence of 2-1135
  copying 2-755
  creating 2-2396
  listing 2-2218
  listing contents of 2-982
  listing MATLAB files in 2-4167
  removing 2-3136
removing from search path 2-3141
See also directory, search path
directory 2-982
  changing on FTP server 2-537
  listing for FTP server 2-987
  making on FTP server 2-2399
  MATLAB location 2-2274
  root 2-2274
  temporary system 2-3663
  See also directories
directory, changing 2-532
directory, current 2-2918
disconnect 2-629
discontinuities, eliminating (in arrays of phase angles) 2-4050
discontinuities, plotting functions with 2-1176
discontinuous problems 2-1369
disp 2-990
  memmapfile object 2-992
  serial port I/O 2-995
  timer object 2-996
disp, MException method 2-993
display 2-998
display format 2-1379
displaying output in Command Window 2-2442
DisplayName
  areaseries property 2-226
  barseries property 2-361
  contourgroup property 2-719
  errorbarsseries property 2-1094
  hggroup property 2-1678
  hgtransform property 2-1708
  image property 2-1776
  Line property 2-2123
  lineseries property 2-2136
  Patch property 2-2721
  quivergroup property 2-2977
  rectangle property 2-3048
  scattergroup property 2-3206
  stairseries property 2-3391
stemseries property 2-3425
surface property 2-3577
surfaceplot property 2-3600
text property 2-3682
distribution
  Gaussian 2-1079
dither 2-1000
division
  array, left (arithmetic operator) 2-42
  array, right (arithmetic operator) 2-41
  by zero 2-1842
  matrix, left (arithmetic operator) 2-41
  matrix, right (arithmetic operator) 2-41
  of polynomials 2-916
divisor
  greatest common 2-1496
dll libraries
  MATLAB functions
    calllib 2-487
    libfunctions 2-2085
    libfunctionsview 2-2086
    libisloaded 2-2087
    libpointer 2-2089
    libstruct 2-2091
    loadlibrary 2-2192
    unloadlibrary 2-4032
dlmread 2-1004
dlmwrite 2-1008
dmperm 2-1012
Dockable, Figure property 2-1247
docsearch 2-1020
documentation
  displaying online 2-1658
dolly camera 2-491
dos 2-1023
  UNC pathname error 2-1024
dot 2-1025
dot product 2-785 2-1025
dot-parentheses (special characters 2-64
  double 2-1026
double click, detecting 2-1271
double integral
  numerical evaluation 2-858
DoubleBuffer, Figure property 2-1247
downloading files from FTP server 2-2384
dragrect 2-1027
drawing shapes
  circles and rectangles 2-3037
DrawMode, Axes property 2-312
drawnow 2-1029
dsearch 2-1031
dsearchn 2-1032
Dulmage-Mendelsohn decomposition 2-1012
dynamic fields 2-64
dynamicprops class 2-1033
dynamicprops.addprop 2-129

E
  echo 2-1034
Echo, Root property 2-3146
echodemo 2-1036
edge finding, Sobel technique 2-744
EdgeAlpha
  patch property 2-2722
  surface property 2-3577
  surfaceplot property 2-3600
EdgeColor
  annotation ellipse property 2-182
  annotation rectangle property 2-188
  annotation textbox property 2-202
  areaseries property 2-227
  barseries property 2-362
  patch property 2-2723
  Surface property 2-3578
  surfaceplot property 2-3601
  Text property 2-3683
EdgeColor, rectangle property 2-3049
EdgeLighting
  patch property 2-2723
Surface property 2-3579
surfaceplot property 2-3602
editable text 2-3843
editing
   M-files 2-1041
eig 2-1044
eigensystem
      transforming 2-538
eigenvalue
      accuracy of 2-1044
      complex 2-538
      matrix logarithm and 2-2211
      modern approach to computation of 2-2828
      of companion matrix 2-681
      problem 2-1045 2-2833
      problem, generalized 2-1045 2-2833
      problem, polynomial 2-2833
      repeated 2-1046
      Wilkinson test matrix and 2-4188
eigenvalues
      effect of roundoff error 2-340
      improving accuracy 2-340
eigenvector
      left 2-1045
      matrix, generalized 2-2993
      right 2-1045
eigs 2-1050
elevation (spherical coordinates) 2-3339
elevation of viewpoint 2-4112
ellipj 2-1060
ellipke 2-1062
ellipsoid 2-1064
elliptic functions, Jacobian
      (defined) 2-1060
elliptic integral
      complete (defined) 2-1062
      modulus of 2-1060 2-1062
else 2-1066
elseif 2-1067
   %#eml 2-2416
Enable
   Uicontrol property 2-3856
   Uimenu property 2-3903
   Uipushtool property 2-3938
   Uitable property 2-3985
   Uitogglehtool property 2-4004
end 2-1071
end caps for isosurfaces 2-1977
end of line, indicating 2-64
end-of-file indicator 2-1196
eomday 2-1073
eps 2-1074
eq 2-1076
eq, MException method 2-1078
equal arrays
   detecting 2-1943 2-1947
equal sign (special characters) 2-63
equations, linear
   accuracy of solution 2-688
EraseMode
   areaseries property 2-227
   barseries property 2-362
   contour property 2-719
   errorbar property 2-1095
   hggroup property 2-1679
   hgtransform property 2-1708
   Image property 2-1777
   Line property 2-2124
   lineseries property 2-2137
   quivergroup property 2-2978
   rectangle property 2-3049
   scatter property 2-3207
   stairseries property 2-3392
   stem property 2-3426
   Surface property 2-3579
   surfaceplot property 2-3602
   Text property 2-3684
EraseMode patch property 2-2724
error 2-1081
   roundoff. See roundoff error
error function
  complementary 2-1079
  (defined) 2-1079
  scaled complementary 2-1079
error message
  displaying 2-1081
  Index into matrix is negative or zero 2-2207
  retrieving last generated 2-2047 2-2055
error messages
  Out of memory 2-2678
error tolerance
  BVP problems 2-477
  DDE problems 2-895
  ODE problems 2-2619
errorbars, confidence interval 2-1086
errordlg 2-1108
ErrorMessage, Root property 2-3146
errors
  in file input/output 2-1198
MException class 2-1078
  addCause 2-110
  constructor 2-2375
  disp 2-993
  eq 2-1078
  getReport 2-1549
  isequal 2-1946
  last 2-2045
  ne 2-2499
  rethrow 2-3123
  throw 2-3730
  throwAsCaller 2-3733
ErrorType, Root property 2-3147
escape characters in format specification
  string 2-1393 2-3362
etime 2-1111
etree 2-1112
etreeplot 2-1113
eval 2-1114
evalc 2-1117
evalin 2-1118
event location (DDE) 2-901
event location (ODE) 2-2626
event.EventData 2-1120
event.listener 2-1122
event.PropertyEvent 2-1121
event.proplistener 2-1124
eventlisteners 2-1126
events 2-1129 to 2-1130
examples
  calculating isosurface normals 2-1984
  contouring mathematical expressions 2-1149
  isosurface end caps 2-1977
  isosurfaces 2-1988
  mesh plot of mathematical function 2-1158
  mesh/contour plot 2-1162
  plotting filled contours 2-1153
  plotting function of two variables 2-1166
  plotting parametric curves 2-1169
  polar plot of function 2-1172
  reducing number of patch faces 2-3059
  reducing volume data 2-3062
  subsampling volume data 2-3545
  surface plot of mathematical function 2-1176
  surface/contour plot 2-1180
Excel spreadsheets
  loading 2-4207
exclamation point (special characters) 2-65
Execute 2-1132
executing statements repeatedly 2-1377 2-4175
executing statements repeatedly in parallel 2-2694
execution
  improving speed of by setting aside storage 2-4231
  pausing M-file 2-2745
  resuming from breakpoint 2-856
  time for M-files 2-2887
exifread 2-1134
exist 2-1135
exit 2-1139
exp 2-1140
expint 2-1141
expm 2-1142
expm1 2-1144
exponential 2-1140
    complex (defined) 2-1140
    integral 2-1141
    matrix 2-1142
exponentiation
    array (arithmetic operator) 2-42
    matrix (arithmetic operator) 2-42
export2wsdlg 2-1145
extension, filename
    .m 2-1441
    .mat 2-3176
Extent
    Text property 2-3686
    Uicontrol property 2-3856
    Uitable property 2-3986
eye 2-1147
ezcontour 2-1148
ezcontourf 2-1152
ezmesh 2-1156
ezmeshc 2-1160
ezplot 2-1164
ezplot3 2-1168
ezpolar 2-1171
ezsurf 2-1174
ezsurfc 2-1178

F
F-norm 2-2570
FaceAlpha
    annotation textbox property 2-203
FaceAlphapatch property 2-2725
FaceAlphasurface property 2-3580
FaceAlphasurfaceplot property 2-3603
FaceColor
    annotation ellipse property 2-182
annotation rectangle property 2-188
areaeries property 2-229
barseries property 2-364
Surface property 2-3581
surfaceplot property 2-3604
FaceColor, rectangle property 2-3050
FaceColorpatch property 2-2726
FaceLighting
    Surface property 2-3581
    surfaceplot property 2-3605
FaceLightingpatch property 2-2726
faces, reducing number in patches 2-3058
Faces,patch property 2-2727
FaceVertexAlphaData, patch property 2-2728
FaceVertexCData,patch property 2-2728
factor 2-1185
factorial 2-1186
factorization 2-2925
    LU 2-2234
    QZ 2-2834 2-2993
    See also decomposition
factorization, Cholesky 2-581
    (as algorithm for solving linear equations) 2-2409
    preordering for 2-673
factors, prime 2-1185
false 2-1187
fclose 2-1188
serial port I/O 2-1189
feather 2-1191
feof 2-1196
ferror 2-1198
feval 2-1199
Feval 2-1201
fft 2-1206
FFT. See Fourier transform
fft2 2-1211
fftn 2-1212
fftshift 2-1214
fftw 2-1217
FFT 2-1209
fgetl 2-1222
  serial port I/O 2-1224
fgets 2-1227
  serial port I/O 2-1228
field names of a structure, obtaining 2-1231
fieldnames 2-1231
fields, noncontiguous, inserting data into 2-1455
fields, of structures
  dynamic 2-64
figure 2-1233
Figure
  creating 2-1233
  defining default properties 2-1235
  properties 2-1236
  redrawing 2-3065
figure windows
  moving in front of MATLAB® desktop 2-3273
figure windows, displaying 2-1333
figurepalette 2-1291
figures
  annotating 2-2811
  opening 2-2642
  saving 2-3188
Figures
  updating from M-file 2-1029
file
  extension, getting 2-1308
  modification date 2-982
  position indicator
    finding 2-1310 2-1434
    setting 2-1432
    setting to start of file 2-1420
file formats
  getting list of supported formats 2-1794
  reading 2-811 2-1803
  writing 2-1816
file size
  querying 2-1791
fileattrib 2-1293
  filebrowser 2-1300
  filemarker 2-1306
  filename
    building from parts 2-1438
    parts 2-1308
    temporary 2-3664
  filename extension
    .m 2-1441
    .mat 2-3176
  fileparts 2-1308
  fileread 2-1310
  files 2-1188
    ASCII delimited
      reading 2-1004
      writing 2-1008
    beginning of, rewinding to 2-1420 2-1800
    checking existence of 2-1135
    closing 2-1188
    contents, listing 2-3803
    copying 2-755
    deleting 2-942
    deleting on FTP server 2-946
    end of, testing for 2-1196
    errors in input or output 2-1198
    Excel spreadsheets
      loading 2-4207
      fig 2-3188
    figure, saving 2-3188
    finding position within 2-1310 2-1434
    getting next line 2-1222
    getting next line (with line terminator) 2-1227
    listing
      in directory 2-4167
      names in a directory 2-982
    listing contents of 2-3803
    locating 2-4172
    mdl 2-3188
    mode when opened 2-1371
    model, saving 2-3188
opening 2-1370 2-2642
opening in Windows applications 2-4189
path, getting 2-1308
pathname for 2-4172
reading
  binary 2-1403
  data from 2-3706
  formatted 2-1421
reading data from 2-811
reading image data from 2-1803
rewinding to beginning of 2-1420 2-1800
setting position within 2-1432
size, determining 2-985
sound
  reading 2-281 2-4158
  writing 2-282 to 2-283 2-4163
startup 2-2273
version, getting 2-1308
.wav
  reading 2-4158
  writing 2-4163
WK1
  loading 2-4193
  writing to 2-4195
writing binary data to 2-1455
writing formatted data to 2-1391
writing image data to 2-1816
See also file
filesep 2-1311
fill 2-1313
Fill
  contour property 2-721
fill3 2-1316
filter 2-1319
  digital 2-1319
  finite impulse response (FIR) 2-1319
  infinite impulse response (IIR) 2-1319
  two-dimensional 2-742
filter (timeseries) 2-1322
filter2 2-1325
find 2-1327
findall function 2-1332
findfigs 2-1333
finding 2-1327
  sign of array elements 2-3281
  zero of a function 2-1461
  See also detecting
findobj 2-1334
findobj handle method 2-1337
findprop handle method 2-1338
findstr 2-1339
finish 2-1340
finish.m 2-2962
FIR filter 2-1319
FitBoxToText, annotation textbox
  property 2-203
FitHeightToText
  annotation textbox property 2-203
fitsinfo 2-1342
defitsread 2-1351
fix 2-1353
fixed-width font
  axes 2-313
  text 2-3687
  uicontrols 2-3858
  uitables 2-3987
FixedColors, Figure property 2-1248
FixedWidthFontName, Root property 2-3146
flints 2-2466
flip
  array dimension 2-1354
flip array
  along dimension 2-1354
flip matrix
  on horizontal axis 2-1356
  on vertical axis 2-1355
flipdim 2-1354
fliplr 2-1355
flipud 2-1356
floating-point
integer, maximum 2-429
floating-point arithmetic, IEEE
smallest positive number 2-3032
floor 2-1358
flow control
  break 2-439
  case 2-519
  end 2-1071
  error 2-1082
  for 2-1377
  keyboard 2-2041
  otherwise 2-2677
  parfor 2-2694
  return 2-3125
  switch 2-3632
  while 2-4175
fminbnd 2-1360
fminsearch 2-1365
font
  fixed-width, axes 2-313
  fixed-width, text 2-3687
  fixed-width, uicontrols 2-3858
  fixed-width, uitables 2-3987
FontAngle
  annotation textbox property 2-205
  Axes property 2-313
  Text property 2-3686
  Uicontrol property 2-3857
  Uitable property 2-3987
FontName
  annotation textbox property 2-205
  Axes property 2-313
  Text property 2-3686
  Uicontrol property 2-3857
  Uitable property 2-3987
fonts
  bold 2-192 2-206 2-3687
  italic 2-192 2-205 2-3686
  specifying size 2-3687
  TeX characters
    bold 2-3701
    italics 2-3701
    specifying family 2-3701
    specifying size 2-3701
    units 2-192 2-206 2-3687
FontSize
  annotation textbox property 2-206
  Axes property 2-314
  Text property 2-3687
  textarrow property 2-192
  Uicontrol property 2-3858
  Uitable property 2-3988
FontUnits
  Axes property 2-314
  Text property 2-3687
  Uicontrol property 2-3859
  Uitable property 2-3988
FontWeight
  annotation textbox property 2-206
  Axes property 2-314
  Text property 2-3687
  textarrow property 2-192
  Uicontrol property 2-3859
  Uitable property 2-3988
fopen 2-1370
  serial port I/O 2-1374
for 2-1376
ForegroundColor
  Uicontrol property 2-3859
  Uimenu property 2-3903
  Uitable property 2-3988
format 2-1379
  precision when writing 2-1403
  reading files 2-1422
  specification string, matching file data to 2-3377
Format 2-3147
formats
  big endian 2-1373
little endian 2-1373
FormatSpacing, Root property 2-3148
formatted data
  reading from file 2-1421
  writing to file 2-1391
formatting data 2-3360
Fourier transform
  algorithm, optimal performance of 2-1209
    2-1745 2-1747 2-2566
  as method of interpolation 2-1902
  convolution theorem and 2-741
  discrete, n-dimensional 2-1212
  discrete, one-dimensional 2-1206
  discrete, two-dimensional 2-1211
  fast 2-1206
  inverse, n-dimensional 2-1749
  inverse, one-dimensional 2-1745
  inverse, two-dimensional 2-1747
  shifting the zero-frequency component of 2-1215
fplot 2-1386 2-1402
fprintf 2-1391
  displaying hyperlinks with 2-1396
  serial port I/O 2-1398
fraction, continued 2-3017
fragmented memory 2-2678
frame2im 2-1402
frames 2-3843
fread 2-1403
  serial port I/O 2-1413
freqspace 2-1419
frequency response
  desired response matrix
    frequency spacing 2-1419
  frequency vector 2-2214
frewind 2-1420
fromName meta.class method 2-2339
fromName meta.package method 2-2350
fscanf 2-1421
  serial port I/O 2-1428
fseek 2-1432
ftell 2-1434
FTP
  connecting to server 2-1435
ftp function 2-1435
full 2-1437
fullfile 2-1438
func2str 2-1439
function 2-1441
  function handle 2-1443
  function handles
    overview of 2-1443
  function syntax 2-1655 2-3650
functions 2-1446
  call history 2-2892
  call stack for 2-863
  checking existence of 2-1135
  clearing from workspace 2-614
  finding using keywords 2-2215
  help for 2-1654 2-1664
  in memory 2-1850
  locating 2-4172
  pathname for 2-4172
  that work down the first non-singleton dimension 2-3274
funm 2-1450
fwrite 2-1455
  serial port I/O 2-1457
fzero 2-1461

G

gallery 2-1467
gamma function
  (defined) 2-1490
  incomplete 2-1490
  logarithm of 2-1490
  logarithmic derivative 2-2900
Gauss-Kronrod quadrature 2-2951
Gaussian distribution function 2-1079
Gaussian elimination
  (as algorithm for solving linear equations) 2-1917 2-2410
Gauss Jordan elimination with partial pivoting 2-3171
LU factorization 2-2234
gca 2-1493
gcbf function 2-1494
gcbo function 2-1495
gcd 2-1496
gcf 2-1498
gco 2-1499
ge 2-1500
generalized eigenvalue problem 2-1045 2-2833
generating a sequence of matrix names (M1 through M12) 2-1115
genpath 2-1502
genvarname 2-1504
geodesic dome 2-3645
get 2-1508 2-1511
  memmapfile object 2-1514
  serial port I/O 2-1518
  timer object 2-1520
get (timeseries) 2-1522
get (tscollection) 2-1523
get hgsetget class method 2-1513
get, RandStream method 2-1517
getabstime (timeseries) 2-1524
getabstime (tscollection) 2-1526
getAllPackages meta.package method 2-2351
getappdata function 2-1528
getCompilerConfigurations 2-2369
getdatasamplesize 2-1531
getDefaultStream, RandStream method 2-1532
getdisp hgsetget class method 2-1533
getenv 2-1534
getField 2-1535
getframe 2-1537
  image resolution and 2-1538
getinterpmethod 2-1543
getpixelposition 2-1544
getpref function 2-1546
getqualitydesc 2-1548
getReport, MException method 2-1549
getsampleusingtime (timeseries) 2-1552
getsampleusingtime (tscollection) 2-1553
gettimeseriesnames 2-1554
gettsafteratevent 2-1555
gettsafterevent 2-1556
getssatevent 2-1557
getssbeforeatevent 2-1558
getssbeforeevent 2-1559
getsstbweenevents 2-1560
GIF files
  writing 2-1818
ginput function 2-1565
global 2-1567
global variable
  defining 2-1567
global variables, clearing from workspace 2-614
gmres 2-1569
golden section search 2-1363
Goup
  defining default properties 2-1702
gplot 2-1575
grabcode function 2-1577
gradient 2-1579
gradient, numerical 2-1579
graph
  adjacency 2-1013
graph theory 2-4033
graphics objects
  Axes 2-290
  Figure 2-1233
  getting properties 2-1508
  Image 2-1760
  Light 2-2098
  Line 2-2111
  Patch 2-2702
  resetting properties 2-3111
Index

Root 2-3143
  setting properties 2-3238
Surface 2-3563
Text 2-3670
  uicontextmenu 2-3831
Uicontrol 2-3842
Uimenu 2-3895
graphics objects, deleting 2-942
graphs
  editing 2-2811
graymon 2-1582
greatest common divisor 2-1496
Greek letters and mathematical symbols 2-196
  2-208 2-3699
grid 2-1583
  aligning data to a 2-1585
grid arrays
  for volumetric plots 2-2333
  multi-dimensional 2-2492
griddata 2-1585
griddata3 2-1589
griddatan 2-1592
GridLineStyle, Axes property 2-315
  hggroup function 2-1671
group
  areaserseries property 2-229
  Axes property 2-315
  barseries property 2-364
  contour property 2-721
  errorbar property 2-1096
  Figure property 2-1248
  hggroup property 2-1681
  hgtransform property 2-1710
  Image property 2-1778
  Light property 2-2103
  Line property 2-2125
  lineseries property 2-2138
  patch property 2-2730
  quivergroup property 2-2979
  rectangle property 2-3050
  Root property 2-3148
  stairseries property 2-3393
  stem property 2-3427
  Surface property 2-3582
  surfaceplot property 2-3605
  Text property 2-3688
  Uicontextmenu property 2-3838
  Uicontrol property 2-3859
  Uimenu property 2-3903
  Uipushtool property 2-3938

hadamard 2-1613
  Hadamard matrix 2-1613
    subspaces of 2-3540
handle class 2-1614
handle graphics
  hgtransform 2-1691
  handle graphicshggroup 2-1671
  handle relational operators 2-3095
  handle.addlistener 2-118
  handle.delete 2-947
  handle.findobj 2-1337
  handle.findprop 2-1338
  handle.isvalid 2-2013
  handle.notify 2-2575
HandleVisibility
  areaserseries property 2-229
  Axes property 2-315
  barseries property 2-364
  contour property 2-721
  errorbar property 2-1096
  Figure property 2-1248
  hggroup property 2-1681
  hgtransform property 2-1710
  Image property 2-1778
  Light property 2-2103
  Line property 2-2125
  lineseries property 2-2138
  patch property 2-2730
  quivergroup property 2-2979
  rectangle property 2-3050
  Root property 2-3148
  stairseries property 2-3393
  stem property 2-3427
  Surface property 2-3582
  surfaceplot property 2-3605
  Text property 2-3688
  Uicontextmenu property 2-3838
  Uicontrol property 2-3859
  Uimenu property 2-3903
  Uipushtool property 2-3938

H
  H1 line 2-1656
<table>
<thead>
<tr>
<th>Index</th>
<th>Property/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uitable property</td>
<td>2-3989</td>
<td></td>
</tr>
<tr>
<td>Uitoggletool property</td>
<td>2-4005</td>
<td></td>
</tr>
<tr>
<td>Uitoolbar property</td>
<td>2-4016</td>
<td></td>
</tr>
<tr>
<td><strong>hankel</strong></td>
<td>2-1618</td>
<td></td>
</tr>
<tr>
<td><strong>Hankel matrix</strong></td>
<td>2-1618</td>
<td></td>
</tr>
<tr>
<td><strong>HDF</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>appending to when saving</td>
<td>(WriteMode) 2-1821</td>
<td></td>
</tr>
<tr>
<td>compression</td>
<td>2-1821</td>
<td></td>
</tr>
<tr>
<td>setting JPEG quality when writing</td>
<td>2-1821</td>
<td></td>
</tr>
<tr>
<td><strong>HDF files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>writing images</td>
<td>2-1818</td>
<td></td>
</tr>
<tr>
<td><strong>HDF4</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>summary of capabilities</td>
<td>2-1619</td>
<td></td>
</tr>
<tr>
<td><strong>HDF5</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high-level access</td>
<td>2-1621</td>
<td></td>
</tr>
<tr>
<td>summary of capabilities</td>
<td>2-1621</td>
<td></td>
</tr>
<tr>
<td><strong>HDF5 class</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>low-level access</td>
<td>2-1621</td>
<td></td>
</tr>
<tr>
<td><strong>hdf5info</strong></td>
<td>2-1624</td>
<td></td>
</tr>
<tr>
<td><strong>hdf5read</strong></td>
<td>2-1626</td>
<td></td>
</tr>
<tr>
<td><strong>hdf5write</strong></td>
<td>2-1628</td>
<td></td>
</tr>
<tr>
<td><strong>hdfinfo</strong></td>
<td>2-1632</td>
<td></td>
</tr>
<tr>
<td><strong>hdfread</strong></td>
<td>2-1640</td>
<td></td>
</tr>
<tr>
<td><strong>hdftool</strong></td>
<td>2-1653</td>
<td></td>
</tr>
<tr>
<td><strong>Head1Length</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>annotation doublearrow property</td>
<td>2-177</td>
<td></td>
</tr>
<tr>
<td><strong>Head1Style</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>annotation doublearrow property</td>
<td>2-178</td>
<td></td>
</tr>
<tr>
<td><strong>Head1Width</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>annotation doublearrow property</td>
<td>2-179</td>
<td></td>
</tr>
<tr>
<td><strong>Head2Length</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>annotation doublearrow property</td>
<td>2-177</td>
<td></td>
</tr>
<tr>
<td><strong>Head2Style</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>annotation doublearrow property</td>
<td>2-178</td>
<td></td>
</tr>
<tr>
<td><strong>Head2Width</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>annotation doublearrow property</td>
<td>2-179</td>
<td></td>
</tr>
<tr>
<td><strong>HeadLength</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>annotation arrow property</td>
<td>2-173</td>
<td></td>
</tr>
<tr>
<td>textarrow property</td>
<td>2-193</td>
<td></td>
</tr>
<tr>
<td><strong>HeadStyle</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>annotation arrow property</td>
<td>2-174</td>
<td></td>
</tr>
<tr>
<td>textarrow property</td>
<td>2-194</td>
<td></td>
</tr>
<tr>
<td><strong>HeadWidth</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>annotation arrow property</td>
<td>2-174</td>
<td></td>
</tr>
<tr>
<td>textarrow property</td>
<td>2-194</td>
<td></td>
</tr>
<tr>
<td><strong>Height</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>annotation ellipse property</td>
<td>2-183</td>
<td></td>
</tr>
<tr>
<td><strong>help</strong></td>
<td>2-1654</td>
<td></td>
</tr>
<tr>
<td>contents file</td>
<td>2-1655</td>
<td></td>
</tr>
<tr>
<td>keyword search in functions</td>
<td>2-2215</td>
<td></td>
</tr>
<tr>
<td>online</td>
<td>2-1654</td>
<td></td>
</tr>
<tr>
<td><strong>Help browser</strong></td>
<td>2-1658</td>
<td></td>
</tr>
<tr>
<td>accessing from doc</td>
<td>2-1015</td>
<td></td>
</tr>
<tr>
<td><strong>Help Window</strong></td>
<td>2-1664</td>
<td></td>
</tr>
<tr>
<td><strong>helpbrowser</strong></td>
<td>2-1658</td>
<td></td>
</tr>
<tr>
<td><strong>helpdesk</strong></td>
<td>2-1661</td>
<td></td>
</tr>
<tr>
<td><strong>helpdlg</strong></td>
<td>2-1662</td>
<td></td>
</tr>
<tr>
<td><strong>helpwin</strong></td>
<td>2-1664</td>
<td></td>
</tr>
<tr>
<td>Hermite transformations, elementary</td>
<td>2-1496</td>
<td></td>
</tr>
<tr>
<td><strong>hess</strong></td>
<td>2-1665</td>
<td></td>
</tr>
<tr>
<td>Hessenberg form of a matrix</td>
<td>2-1665</td>
<td></td>
</tr>
<tr>
<td><strong>hex2dec</strong></td>
<td>2-1668</td>
<td></td>
</tr>
<tr>
<td><strong>hex2num</strong></td>
<td>2-1669</td>
<td></td>
</tr>
<tr>
<td><strong>hgsetget class</strong></td>
<td>2-1690</td>
<td></td>
</tr>
<tr>
<td><strong>hgsetget.get</strong></td>
<td>2-1513</td>
<td></td>
</tr>
<tr>
<td><strong>hgsetget.getdisp</strong></td>
<td>2-1533</td>
<td></td>
</tr>
<tr>
<td><strong>hgsetget.set</strong></td>
<td>2-3243</td>
<td></td>
</tr>
<tr>
<td>hidden</td>
<td>2-1715</td>
<td></td>
</tr>
<tr>
<td>Hierarchical Data Format (HDF) files</td>
<td></td>
<td></td>
</tr>
<tr>
<td>writing images</td>
<td>2-1818</td>
<td></td>
</tr>
<tr>
<td><strong>hilb</strong></td>
<td>2-1716</td>
<td></td>
</tr>
<tr>
<td>Hilbert matrix</td>
<td>2-1716</td>
<td></td>
</tr>
<tr>
<td>inverse</td>
<td>2-1920</td>
<td></td>
</tr>
<tr>
<td><strong>hist</strong></td>
<td>2-1717</td>
<td></td>
</tr>
<tr>
<td><strong>histc</strong></td>
<td>2-1721</td>
<td></td>
</tr>
<tr>
<td>HitTest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>areaseries property</td>
<td>2-231</td>
<td></td>
</tr>
<tr>
<td>Axes property</td>
<td>2-316</td>
<td></td>
</tr>
<tr>
<td>barseries property</td>
<td>2-366</td>
<td></td>
</tr>
</tbody>
</table>
contour property 2-723
errorbar property 2-1098
Figure property 2-1250
hggroup property 2-1682
hgtransform property 2-1711
Image property 2-1780
Light property 2-2105
Line property 2-2125
lineseries property 2-2140
Patch property 2-2731
quivergroup property 2-2981
rectangle property 2-3051
Root property 2-3148
scatter property 2-3210
stairseries property 2-3395
stem property 2-3429
Surface property 2-3583
surfaceplot property 2-3607
Text property 2-3689
Uicontrol property 2-3860
Uipushtool property 2-3939
Uitable property 2-3989
Uitoggletool property 2-4005
Uitoolbarl property 2-4017
HitTestArea
    areaserices property 2-231
    barseries property 2-366
    contour property 2-723
    errorbar property 2-1098
    quivergroup property 2-2981
    scatter property 2-3210
    stairseries property 2-3395
    stem property 2-3429
hold 2-1724
home 2-1726
HorizontalAlignment
    Text property 2-3690
    textarrow property 2-194
    textbox property 2-206
    Uicontrol property 2-3861
horzcat 2-1727
horzcat (M-file function equivalent for [,]) 2-65
horzcat (tscollection) 2-1729
hostid 2-1730
Householder reflections (as algorithm for solving linear equations) 2-2411
hsv2rgb 2-1731
HTML
    in Command Window 2-2268
    save M-file as 2-2903
HTML browser
    in MATLAB 2-1658
hyperbolic
    cosecant 2-789
    cosecant, inverse 2-90
    cosine 2-766
    cosine, inverse 2-80
    cotangent 2-771
    cotangent, inverse 2-85
    secant 2-3226
    secant, inverse 2-248
    sine 2-3286
    sine, inverse 2-253
    tangent 2-3659
    tangent, inverse 2-264
hyperlink
    displaying in Command Window 2-990
hyperlinks
    in Command Window 2-2268
hyperplanes, angle between 2-3540
hypot 2-1732

i 2-1735
icon images
    reading 2-1805
idealfilter (timeseries) 2-1736
identity matrix 2-1147
    sparse 2-3336
idivide 2-1739
IEEE floating-point arithmetic
   smallest positive number 2-3032
if 2-1741
ifft 2-1745
ifft2 2-1747
ifftn 2-1749
ifftshift 2-1751
IIR filter 2-1319
ilu 2-1752
im2java 2-1757
imag 2-1759
image 2-1760
Image
   creating 2-1760
      properties 2-1768
image types
   querying 2-1791
images
   file formats 2-1803 2-1816
   reading data from files 2-1803
   returning information about 2-1790
   writing to files 2-1816
Images
   converting MATLAB image to Java
      Image 2-1757
imagesc 2-1784
imaginary 2-1759
   part of complex number 2-1759
   unit (sqrt(\xd0 1)) 2-1735 2-2018
   See also complex
imapprox 2-1788
iminfo
   returning file information 2-1790
imformats 2-1794
import 2-1797
importdata 2-1800
importing
   Java class and package names 2-1797
imread 2-1803
imwrite 2-1816
incomplete beta function
   (defined) 2-399
incomplete gamma function
   (defined) 2-1490
ind2sub 2-1838
Index into matrix is negative or zero (error message) 2-2207
indexed images
   converting from RGB 2-3127
indexing
   logical 2-2206
   indicator of file position 2-1420
indices, array
   of sorted elements 2-3311
Inf 2-1842
inferiorto 2-1844
infinity 2-1842
   norm 2-2570
info 2-1846
information
   returning file information 2-1790
inheritance, of objects 2-609
inline 2-1847
inmem 2-1850
inpolygon 2-1852
input 2-1854
   checking number of M-file arguments 2-2483
      name of array passed as 2-1859
         number of M-file arguments 2-2485
         prompting users for 2-1854
inputdlg 2-1855
inputname 2-1859
inputParser 2-1860
inspect 2-1866
installation, root directory of 2-2274
instance properties 2-129
instrcallback 2-1874
instrfind 2-1875
instrfindall 2-1877
example of 2-1878
int2str 2-1880
integer
    floating-point, maximum 2-429
IntegerHandle
    Figure property 2-1250
integration
    polynomial 2-2839
    quadrature 2-2937 2-2946
interfaces 2-1884
interp1 2-1886
interp1q 2-1894
interp2 2-1896
interp3 2-1900
interpft 2-1902
interpn 2-1903
interpolated shading and printing 2-2871
interpolation
    cubic method 2-1585 2-1886 2-1896 2-1900
    2-1903
    cubic spline method 2-1886 2-1896 2-1900
    2-1903
    FFT method 2-1902
    linear method 2-1886 2-1896 2-1900 2-1903
    multidimensional 2-1903
    nearest neighbor method 2-1585 2-1886
    2-1896 2-1900 2-1903
    one-dimensional 2-1886
    three-dimensional 2-1900
    trilinear method 2-1585
    two-dimensional 2-1896
Interpreter
    Text property 2-3691
    textarrow property 2-194
    textbox property 2-207
interpstreamspeed 2-1906
Interruptible
    areaseries property 2-231
    Axes property 2-316
    barseries property 2-366
    contour property 2-723
    errorbar property 2-1099
    Figure property 2-1250
    hggroup property 2-1682
    hgtransform property 2-1711
    Image property 2-1780
    Light property 2-2105
    Line property 2-2126
    lineseries property 2-2140
    patch property 2-2731
    quivergroup property 2-2981
    rectangle property 2-3052
    Root property 2-3148
    scatter property 2-3211
    stairseries property 2-3395
    stem property 2-3430
    Surface property 2-3583 2-3607
    Text property 2-3692
    Uicontextmenu property 2-3839
    Uicontrol property 2-3861
    Uimenu property 2-3904
    Uipushtool property 2-3939
    Uitable property 2-3990
    Uitoggletool property 2-4006
    Uitoolbar property 2-4017
intersect 2-1910
intmax 2-1911
intmin 2-1912
intwarning 2-1913
inv 2-1917
inverse
    cosecant 2-87
    cosine 2-77
    cotangent 2-82
    Fourier transform 2-1745 2-1747 2-1749
    Hilbert matrix 2-1920
    hyperbolic cosecant 2-90
    hyperbolic cosine 2-80
    hyperbolic cotangent 2-85
    hyperbolic secant 2-248
Index

hyperbolic sine 2-253
hyperbolic tangent 2-264
of a matrix 2-1917
secant 2-245
sine 2-250
tangent 2-259
tangent, four-quadrant 2-261
inversion, matrix
  accuracy of 2-688
InvertHardCopy, Figure property 2-1251
invhilb 2-1920
invoke 2-1921
involuntary matrix 2-2701
ipermute 2-1924
iqr (timeseries) 2-1925
is* 2-1927
isa 2-1930
isappdata function 2-1932
iscell 2-1933
iscellstr 2-1934
ischar 2-1935
iscom 2-1936
isdir 2-1937
isempty 2-1940
isempty (timeseries) 2-1941
isempty (tscollection) 2-1942
isequal 2-1943
isequal, MException method 2-1946
isequalwithnan 2-1947
isevent 2-1949
isfield 2-1951
isfinite 2-1953
isfloat 2-1954
isglobal 2-1955
ishandle 2-1957
isinf 2-1959
isinteger 2-1960
isinterface 2-1961
isjava 2-1962
iskeyword 2-1965
isletter 2-1967
islogical 2-1968
ismac 2-1969
ismember 2-1970
ismethod 2-1972
isnan 2-1973
isnumeric 2-1974
isobject 2-1975
isocap 2-1977
isnormals 2-1984
isosurface 2-1987
  calculate data from volume 2-1987
  end caps 2-1977
  vertex normals 2-1984
ispc 2-1992
isPlatformSupported 2-2436
ispref function 2-1993
isprime 2-1994
isprop 2-1995
isreal 2-1996
isscalar 2-1999
issorted 2-2000
isspace 2-2003 2-2006
issparse 2-2004
istr 2-2005
isstruct 2-2009
isstudent 2-2010
isunix 2-2011 to 2-2012
isValid 2-2014
  timer object 2-2015
isValid handle method 2-2013
isvarname 2-2016
isvector 2-2017
italics font
  TeX characters 2-3701

J

j 2-2018
Jacobi rotations 2-3358
Index

Jacobian elliptic functions
  (defined) 2-1060
Jacobian matrix (BVP) 2-479
Jacobian matrix (ODE) 2-2628
  generating sparse numerically 2-2629
  2-2631
  specifying 2-2628 2-2631
  vectorizing ODE function 2-2629 to 2-2631
Java
  class names 2-616 2-1797
  object methods
    inspect 2-1866
  objects 2-1962
Java Image class
  creating instance of 2-1757
Java import list
  adding to 2-1797
  clearing 2-616
Java version used by MATLAB 2-4102
java_method 2-2023 2-2031
java_object 2-2034
javaaddath 2-2019
javachk 2-2024
javaclasspath 2-2026
javaMethod 2-2031
javaMethodEDT 2-2033
javaObject 2-2034
javaObjectEDT 2-2036
javaxmpath 2-2037
joining arrays. See concatenation
Joint Photographic Experts Group (JPEG)
  writing 2-1818
JPEG
  setting Bitdepth 2-1822
  specifying mode 2-1822
JPEG comment
  setting when writing a JPEG image 2-1822
JPEG files
  parameters that can be set when
  writing 2-1822
JPEG quality
  setting when writing a JPEG image 2-1822
  2-1826
  setting when writing an HDF image 2-1821
jvm
  version used by MATLAB 2-4102

K

K>> prompt
  keyboard function 2-2041
keep
  some variables when clearing 2-620
keyboard 2-2041
keyboard mode 2-2041
  terminating 2-3125
KeyPressFcn
  Uicontrol property 2-3862
  Uitable property 2-3991
KeyPressFcn, Figure property 2-1252
KeyReleaseFcn, Figure property 2-1253
keyword search in functions 2-2215
keywords
  iskeyword function 2-1965
kron 2-2043
Kronecker tensor product 2-2043
Krylov subspaces 2-3727

L

Label, Uimenu property 2-3905
labeling
  axes 2-4200
  matrix columns 2-990
  plots (with numeric values) 2-2584
LabelSpacing
  contour property 2-724
Laplacian 2-917
Laplacian matrix 2-4033
largest array elements 2-2299
last, MException method 2-2045
lasterr 2-2047
lasterror 2-2050
lastwarn 2-2055
LaTeX, see TeX 2-196 2-208 2-3699
Layer, Axes property 2-317
Layout Editor
  starting 2-1608
1cm 2-2057
LData
  errorbar property 2-1099
LDataSource
  errorbar property 2-1099
ldivide (M-file function equivalent for .\() 2-45
le 2-2065
least common multiple 2-2057
least squares
  polynomial curve fitting 2-2835
  problem, overdetermined 2-2794
legend 2-2067
  properties 2-2072
  setting text properties 2-2072
legendre 2-2076
Legendre functions
  (defined) 2-2076
  Schmidt semi-normalized 2-2076
length 2-2080
  serial port I/O 2-2082
length (timeseries) 2-2083
length (tscollection) 2-2084
Levellist
  contour property 2-724
LevellistMode
  contour property 2-724
LevelStep
  contour property 2-725
LevelStepMode
  contour property 2-725
libfunctions 2-2085
libfunctionsview 2-2086
libisloaded 2-2087
libpointer 2-2089
libstruct 2-2091
license 2-2094
light 2-2098
Light
  creating 2-2098
  defining default properties 2-1766 2-2099
  positioning in camera coordinates 2-495
  properties 2-2100
Light object
  positioning in spherical coordinates 2-2108
lightangle 2-2108
lighting 2-2109
limits of axes, setting and querying 2-4202
line 2-2111
  editing 2-2811
Line
  creating 2-2111
  defining default properties 2-2116
  properties 2-2117 2-2132
line numbers in M-files 2-881
linear audio signal 2-2110 2-2466
linear dependence (of data) 2-3540
linear equation systems
  accuracy of solution 2-688
  solving overdetermined 2-2927 to 2-2928
linear equation systems, methods for solving
  Cholesky factorization 2-2409
  Gaussian elimination 2-2410
  Householder reflections 2-2411
  matrix inversion (inaccuracy of) 2-1917
linear interpolation 2-1886 2-1896 2-1900 2-1903
linear regression 2-2835
linearly spaced vectors, creating 2-2174
LineColor
  contour property 2-725
lines
  computing 2-D stream 2-3457
computing 3-D stream 2-3459
drawing stream lines 2-3461
LineSpec 2-2149
LineStyle
  annotation arrow property 2-174
  annotation doublearrow property 2-179
  annotation ellipse property 2-183
  annotation line property 2-185
  annotation rectangle property 2-189
  annotation textbox property 2-207
  areaseries property 2-232
  barseries property 2-367
  contour property 2-726
  errorbar property 2-1100
  Line property 2-2127
  lineseries property 2-2141
  patch property 2-2732
  quivergroup property 2-2982
  rectangle property 2-3052
  scatter property 2-3211
  stairseries property 2-3396
  stem property 2-3431
  Surface property 2-3584
  surfaceplot property 2-3608
  text object 2-3694
  textarrow property 2-195
LineStyleOrder
  Axes property 2-317
LineWidth
  annotation arrow property 2-175
  annotation doublearrow property 2-180
  annotation ellipse property 2-183
  annotation line property 2-186
  annotation rectangle property 2-189
  annotation textbox property 2-207
  areaseries property 2-232
  Axes property 2-318
  barseries property 2-367
  contour property 2-726
  errorbar property 2-1100
  Line property 2-2127
  lineseries property 2-2141
  Patch property 2-2732
  quivergroup property 2-2982
  rectangle property 2-3052
  scatter property 2-3211
  stairseries property 2-3396
  stem property 2-3431
  Surface property 2-3584
  surfaceplot property 2-3608
  text object 2-3694
  textarrow property 2-195
 linkaxes 2-2155
 linkdata 2-2159
 linkprop 2-2167
 links
  in Command Window 2-2268
 linsolve 2-2171
 linspace 2-2174
 lint tool for checking problems 2-2413
 list boxes 2-3844
  defining items 2-3868
 list, RandStream method 2-2175
 ListboxTop, Uicontrol property 2-3863
 listdlg 2-2178
 listfonts 2-2181
 little endian formats 2-1373
 load 2-2183 2-2188
  serial port I/O 2-2190
 loadlibrary 2-2192
 loadobj 2-2200
 Lobatto IIIa ODE solver 2-465 2-471
 local variables 2-1441 2-1567
 locking M-files 2-2427
 log 2-2202
  saving session to file 2-977
 log10 [log010] 2-2203
 log1p 2-2204
 log2 2-2205
 logarithm
  base ten 2-2203
base two 2-2205
complex 2-2202 to 2-2203
natural 2-2202
of beta function (natural) 2-402
of gamma function (natural) 2-1491
of real numbers 2-3030
plotting 2-2208
logarithmic derivative
gamma function 2-2900
logarithmically spaced vectors, creating 2-2214
logical 2-2206
logical array
converting numeric array to 2-2206
detecting 2-1968
logical indexing 2-2206
logical operations
AND, bit-wise 2-424
OR, bit-wise 2-431
XOR 2-4228
XOR, bit-wise 2-435
logical operators 2-52 2-59
logical OR
bit-wise 2-431
logical tests 2-1930
all 2-153
any 2-213
See also detecting
logical XOR 2-4228
bit-wise 2-435
loglog 2-2208
logm 2-2211
logspace 2-2214
lookfor 2-2215
lossy compression
writing JPEG files with 2-1822
Lotus WK1 files
loading 2-4193
writing 2-4195
lower 2-2217
lower triangular matrix 2-3766
lowercase to uppercase 2-4054
ls 2-2218
lscov 2-2219
lsqnonneg 2-2224
lsqr 2-2227
lt 2-2232
lu 2-2234
LU factorization 2-2234
storage requirements of (sparse) 2-2588
luinc 2-2242

M
M-file
debbuging 2-2041
displaying during execution 2-1034
function 2-1441
function file, echoing 2-1034
naming conventions 2-1441
pausing execution of 2-2745
programming 2-1441
script 2-1441
script file, echoing 2-1034
M-file execution
resuming after suspending 2-3953
suspending from GUI 2-4020
M-files
checking existence of 2-1135
checking for problems 2-2413
clearing from workspace 2-614
cyclomatic complexity of 2-2413
debugging with profile 2-2887
deleting 2-942
editing 2-1041
line numbers, listing 2-881
lint tool 2-2413
listing names of in a directory 2-4167
locking (preventing clearing) 2-2427
McCabe complexity of 2-2413
opening 2-2642
optimizing 2-2887
problems, checking for 2-2413
save to HTML 2-2903
setting breakpoints 2-870
unlocking (allowing clearing) 2-2478
M-Lint
  function 2-2413
  function for entire directory 2-2423
  HTML report 2-2423
machine epsilon 2-4177
magic 2-2249
magic squares 2-2249
Map containers
  constructor 2-2254 2-3291
  methods 2-2081 2-3098 2-4085
Map methods
  constructor 2-1963 2-2042
Margin
  annotation textbox property 2-208
  text object 2-3696
Marker
  Line property 2-2127
  lineseries property 2-2141
  marker property 2-1101
  Patch property 2-2732
  quivergroup property 2-2982
  scatter property 2-3212
  stairseries property 2-3396
  stem property 2-3431
  Surface property 2-3584
  surfaceplot property 2-3608
MarkerEdgeColor
  errorbar property 2-1101
  Line property 2-2128
  lineseries property 2-2142
  Patch property 2-2733
  quivergroup property 2-2983
  scatter property 2-3212
  stairseries property 2-3397
  stem property 2-3432
  Surface property 2-3585
  surfaceplot property 2-3609
MarkerFaceColor
  errorbar property 2-1102
  Line property 2-2128
  lineseries property 2-2142
  Patch property 2-2734
  quivergroup property 2-2983
  scatter property 2-3213
  stairseries property 2-3397
  stem property 2-3432
  Surface property 2-3586
  surfaceplot property 2-3609
MarkerSize
  errorbar property 2-1102
  Line property 2-2129
  lineseries property 2-2143
  Patch property 2-2734
  quivergroup property 2-2984
  stairseries property 2-3398
  stem property 2-3432
  Surface property 2-3586
  surfaceplot property 2-3610
mass matrix (ODE) 2-2632
  initial slope 2-2633 to 2-2634
  singular 2-2633
  sparsity pattern 2-2633
  specifying 2-2633
  state dependence 2-2633
MAT-file 2-3176
  converting sparse matrix after loading from 2-3323
MAT-files 2-2183
  listing for directory 2-4167
mat2cell 2-2261
mat2str 2-2264
material 2-2266
MATLAB
  directory location 2-2274
  installation directory 2-2274
quitting 2-2962
startup 2-2273
version number, comparing 2-4100
version number, displaying 2-4094
\texttt{matlab}: function 2-2268
\texttt{matlab} (UNIX command) 2-2277
\texttt{matlab} (Windows command) 2-2292
\texttt{matlab} function for UNIX 2-2277
\texttt{matlab} function for Windows 2-2292
MATLAB startup file 2-3406
MATLAB\textsuperscript{®} desktop
  moving figure windows in front of 2-3273
\texttt{matlab}.mat 2-2183 2-3176
\texttt{matlabcolon} function 2-2268
\texttt{matlabrc} 2-2273
\texttt{matlabroot} 2-2274
$\texttt{matlabroot}$ 2-2274
matrices
  preallocation 2-4231
matrix 2-40
  addressing selected rows and columns
    of 2-66
  arrowhead 2-673
columns
  rearrange 2-1355
companion 2-681
complex unitary 2-2925
condition number of 2-688 2-3023
condition number, improving 2-340
converting to formatted data file 2-1391
converting to from string 2-3376
converting to vector 2-66
decomposition 2-2925
defective (defined) 2-1046
detecting sparse 2-2004
determinant of 2-967
diagonal of 2-973
Dulmage-Mendelsohn decomposition 2-1012
evaluating functions of 2-1450
exponential 2-1142
Hadamard 2-1613 2-3540
Hankel 2-1618
Hermitian Toeplitz 2-3756
Hessenberg form of 2-1665
Hilbert 2-1716
identity 2-1147
inverse 2-1917
inverse Hilbert 2-1920
inversion, accuracy of 2-688
involuntary 2-2701
left division (arithmetic operator) 2-41
lower triangular 2-3766
magic squares 2-2249 2-3548
maximum size of 2-686
modal 2-1044
multiplication (defined) 2-41
orthonormal 2-2925
Pascal 2-2701 2-2842
permutation 2-2234 2-2925
poorly conditioned 2-1716
power (arithmetic operator) 2-42
pseudoinverse 2-2794
reading files into 2-1004
rearrange
  columns 2-1355
  rows 2-1356
reduced row echelon form of 2-3171
replicating 2-3103
right division (arithmetic operator) 2-41
rotating $90^\circ$ 2-3160
rows
  rearrange 2-1356
Schur form of 2-3173 2-3219
singularity, test for 2-967
sorting rows of 2-3314
sparse. See sparse matrix
specialized 2-1467
square root of 2-3370
subspaces of 2-3540
test 2-1467
Toeplitz 2-3756
trace of 2-973 2-3758
transpose (arithmetic operator) 2-42
transposing 2-63
unimodular 2-1496
unitary 2-3624
upper triangular 2-3785
Vandermonde 2-2837
Wilkinson 2-3329 2-4188
writing as binary data 2-1455
writing formatted data to 2-1421
writing to ASCII delimited file 2-1008
writing to spreadsheet 2-4195
See also array
Matrix
   hgtransform property 2-1712
matrix functions
   evaluating 2-1450
matrix names, (M1 through M12) generating a
   sequence of 2-1115
matrix power. See matrix, exponential
max 2-2299
max (timeseries) 2-2300
Max, Uicontrol property 2-1256
MaxHeadSize
   quivergroupproperty 2-2984
maximum matching 2-1012
MDL-files
   checking existence of 2-1135
mean 2-2305
mean (timeseries) 2-2306
median 2-2308
median (timeseries) 2-2309
median value of array elements 2-2308
memmapfile 2-2311
memory 2-2317
   clearing 2-614
   minimizing use of 2-2678
   variables in 2-4181
menu (of user input choices) 2-2326
menu function 2-2326
MenuBar, Figure property 2-1256
Mersenne twister 2-3010 2-3014
mesh plot
   tetrahedron 2-3665
   mesh size (BVP) 2-482
meshc 2-2328
meshgrid 2-2333
MeshStyle, Surface property 2-3586
MeshStyle, surfaceplot property 2-3610
meshz 2-2328
message
   error See error message 2-4147
   warning See warning message 2-4147
meta.class 2-2335
meta.DynamicProperty 2-2340
meta.event 2-2344
meta.method 2-2346
meta.package class 2-2349
meta.property 2-2352
methods 2-2357
   inheritance of 2-609
   locating 2-4172
methodview 2-2359
mex 2-2361
mex build script
   switches 2-2362
      -ada sfcn.ads 2-2363
      -arch 2-2363
      -argcheck 2-2363
      -c 2-2363
      -compatibleArrayDims 2-2363
      -cxx 2-2364
      -Dname 2-2364
      -Dname=value 2-2364
      -f optionsfile 2-2364
      -fortran 2-2364
      -g 2-2364
      -h[elp] 2-2364
      -inline 2-2365
-Ipathname 2-2365
-largeArrayDims 2-2365
-Ldirectory 2-2365
-lname 2-2365
-n 2-2365
name=value 2-2366
-O 2-2366
-outdir dirname 2-2366
-output resultname 2-2366
@rsp_file 2-2362
-setup 2-2366
-Uname 2-2366
-v 2-2366
mex.CompilerConfiguration 2-2369
mex.CompilerConfigurationDetails 2-2369
MEX-files
  clearing from workspace 2-614
  debugging on UNIX 2-860
  listing for directory 2-4167
mex.getCompilerConfigurations 2-2369
MException
  constructor 2-1078 2-2375
  methods
    addCause 2-110
    disp 2-993
    eq 2-1078
    getReport 2-1549
    isequal 2-1946
    last 2-2045
    ne 2-2499
    rethrow 2-3123
    throw 2-3730
    throwAsCaller 2-3733
mexext 2-2382
mfilename 2-2383
mget function 2-2384
Microsoft Excel files
  loading 2-4207
min 2-2385
min (timeseries) 2-2386
Min, Uicontrol property 2-3864
MinColormap, Figure property 2-1256
MinorGridLineStyle, Axes property 2-319
minres 2-2390
minus (M-file function equivalent for -) 2-45
mislocked 2-2395
mkdir 2-2396
mkdir (ftp) 2-2399
mkpp 2-2400
mldivide (M-file function equivalent for \) 2-45
mlint 2-2413
mlintrpt 2-2423
  suppressing messages 2-2426
mlock 2-2427
mmfileinfo 2-2428
mmreader 2-2431
mmreader.isPlatformSupported 2-2436
mod 2-2437
modal matrix 2-1044
mode 2-2439
mode objects
  pan, using 2-2683
  rotate3d, using 2-3164
  zoom, using 2-4237
models
  opening 2-2642
  saving 2-3188
modification date
  of a file 2-982
modified Bessel functions
  relationship to Airy functions 2-143
modulo arithmetic 2-2437
MonitorPosition
  Root property 2-3148
Moore-Penrose pseudoinverse 2-2794
more 2-2442 2-2466
move 2-2444
movefile 2-2446
movegui function 2-2449
movie 2-2451
movie2avi 2-2455
movies
  exporting in AVI format 2-283
mpower (M-file function equivalent for ^) 2-46
mput function 2-2458
mrdivide (M-file function equivalent for /) 2-45
msgbox 2-2459
mtimes 2-2462
mtimes (M-file function equivalent for *) 2-45
mu-law encoded audio signals 2-2110 2-2466
multibandread 2-2467
multibandwrite 2-2472
multidimensional arrays 2-2080
  concatenating 2-522
  interpolation of 2-1903
  longest dimension of 2-2080
  number of dimensions of 2-2494
  rearranging dimensions of 2-1924 2-2785
  removing singleton dimensions of 2-3373
  reshaping 2-3114
  size of 2-3288
  sorting elements of 2-3310
  See also array
multiple
  least common 2-2057
multiplication
  array (arithmetic operator) 2-41
  matrix (defined) 2-41
  of polynomials 2-740
multistep ODE solver 2-2609
munlock 2-2478

N
Name, Figure property 2-1257
namelengthmax 2-2480
naming conventions
  M-file 2-1441
NaN 2-2481
NaN (Not-a-Number) 2-2481

NET

.NET summary of functions 2-2502
.NET summary of functions 2-2502
netcdf
  summary of capabilities 2-2513 2-2545
  netcdf.abort
    revert recent netCDF file definitions 2-2516
  netcdf.close
    close netCDF file 2-2518
  netcdf.copyAtt
    copy attribute to new location 2-2519
  netcdf.create
    create netCDF file 2-2521
  netcdf.defDim
    create dimension in netCDF file 2-2523
  netcdf.defVar
    define variable in netCDF dataset 2-2524
  netcdf.delAtt
    delete netCDF attribute 2-2525
  netcdf.endDef
    takes a netCDF file out of define mode 2-2527
  netcdf.getAtt
    return data from netCDF attribute 2-2529
  netcdf.getConstant
    get numeric value of netCDF constant 2-2531
  netcdf.getConstantNames
    get list of netCDF constants 2-2532
  netcdf.getVar
    return data from netCDF variable 2-2533

Index-40
netcdf.inq
    return information about netCDF file 2-2535
netcdf.inqAtt
    return information about a netCDF attribute 2-2537
netcdf.inqAttID
    return identifier of netCDF attribute 2-2539
netcdf.inqAttName
    return name of netCDF attribute 2-2540
netcdf.inqDim
    return information about netCDF dimension 2-2542
netcdf.inqDimID
    return dimension ID for netCDF file 2-2543
netcdf.inqLibVers
    return version of netCDF library 2-2544
netcdf.inqVarID
    return netCDF variable identifier 2-2547
netcdf.open
    open an existing netCDF file 2-2548
netcdf.putAtt
    write a netCDF attribute 2-2549
netcdf.putVar
    write data to netCDF variable 2-2551
netcdf.reDef
    put netCDF file into define mode 2-2553
netcdf.renameAtt
    netCDF function to change the name of an attribute 2-2554
netcdf.renameDim
    netCDF function to change the name of a dimension 2-2556
netcdf.renameVar
    change the name of a netCDF variable 2-2558
netcdf.setDefaultFormat
    change the default netCDF file format 2-2560
netcdf.setFill
    set netCDF fill behavior 2-2561
netcdf.sync
    synchronize netCDF dataset to disk 2-2562
newplot 2-2563
NextPlot
    Axes property 2-319
    Figure property 2-1257
nextpow2 2-2566
nnz 2-2567
no derivative method 2-1369
nodesetup startup option 2-2281
noncontiguous fields, inserting data into 2-1455
nonzero entries
    specifying maximum number of in sparse matrix 2-3320
nonzero entries (in sparse matrix)
    allocated storage for 2-2588
    number of 2-2567
    replacing with ones 2-3350
    vector of 2-2569
nonzeros 2-2569
norm 2-2570
    1-norm 2-2570 2-3023
    2-norm (estimate of) 2-2572
    F-norm 2-2570
    infinity 2-2570
    matrix 2-2570
    pseudoinverse and 2-2794 2-2796
    vector 2-2570
normal vectors, computing for volumes 2-1984
NormalMode
    Patch property 2-2734
    Surface property 2-3587
    surfaceplot property 2-3610
normest 2-2572
not 2-2573
not (M-file function equivalent for ~) 2-56
notebook 2-2574
notify 2-2575
now 2-2576
nthroot 2-2577
null 2-2578
null space 2-2578
num2cell 2-2580
num2hex 2-2583
num2str 2-2584
number
  of array dimensions 2-2494
numbers
  imaginary 2-1759
  NaN 2-2481
  plus infinity 2-1842
  prime 2-2853
  real 2-3029
  smallest positive 2-3032
NumberTitle, Figure property 2-1258
numel 2-2586
numeric format 2-1379
numeric precision
  format reading binary data 2-1403
numerical differentiation formula ODE
solvers 2-2609
numerical evaluation
  double integral 2-858
  triple integral 2-3768
nzmax 2-2588

O

object
  determining class of 2-1930
  inheritance 2-609
object classes, list of predefined 2-608 2-1930
objects
  Java 2-1962
ODE file template 2-2612
ODE solver properties
  error tolerance 2-2619
  event location 2-2626
  Jacobian matrix 2-2628
  mass matrix 2-2632
  ode15s 2-2634
  solver output 2-2621
  step size 2-2625
ODE solvers
  backward differentiation formulas 2-2634
  numerical differentiation formulas 2-2634
  obtaining solutions at specific times 2-2596
  variable order solver 2-2634
ode15i function 2-2589
odefile 2-2611
odeget 2-2617
odephas2 output function 2-2623
odephas3 output function 2-2623
odeplot output function 2-2623
odeprint output function 2-2623
odeset 2-2618
odextend 2-2636
off-screen figures, displaying 2-1333
OffCallback
  Uitoggletool property 2-4007
%#ok 2-2416
OnCallback
  Uitoggletool property 2-4007
one-step ODE solver 2-2608
ones 2-2641
online documentation, displaying 2-1658
online help 2-1654
open 2-2642
openfig 2-2646
OpenGL 2-1265
  autoslection criteria 2-1269
opening
  files in Windows applications 2-4189
opening files 2-1370
openvar 2-2653
operating system
  MATLAB is running on 2-686
  operating system command 2-3653
  operating system command, issuing 2-65
operators
  arithmetic 2-40
logical 2-52 2-59
overloading arithmetic 2-46
overloading relational 2-50
relational 2-50 2-2206
symbols 2-1654
optimget 2-2657
optimization parameters structure 2-2657 to 2-2658
optimizing M-file execution 2-2887
optimset 2-2658
or 2-2662
or (M-file function equivalent for |) 2-56
ordeig 2-2664
orderfields 2-2667
ordering
   reverse Cuthill-McKee 2-3635 2-3645
ordqz 2-2670
ordschur 2-2672
orient 2-2674
orth 2-2676
orthogonal-triangular decomposition 2-2925
orthographic projection, setting and
   querying 2-504
orthonormal matrix 2-2925
otherwise 2-2677
Out of memory (error message) 2-2678
OuterPosition
   Axes property 2-319
   Figure property 2-1258
output
   checking number of M-file arguments 2-2487
   controlling display format 2-1379
   in Command Window 2-2442
   number of M-file arguments 2-2485
output points (ODE)
   increasing number of 2-2621
output properties (DDE) 2-897
output properties (ODE) 2-2621
   increasing number of output points 2-2621
overdetermined equation systems,
solving 2-2927 to 2-2928
overflow 2-1842
overloading
   arithmetic operators 2-46
   relational operators 2-50
   special characters 2-65
P
P-files
   checking existence of 2-1135
pack 2-2678
padecoef 2-2680
pagesetupdlg 2-2681
paging
   of screen 2-1656
paging in the Command Window 2-2442
pan mode objects 2-2683
PaperOrientation, Figure property 2-1259
PaperPosition, Figure property 2-1259
PaperPositionMode, Figure property 2-1260
PaperSize, Figure property 2-1260
PaperType, Figure property 2-1260
PaperUnits, Figure property 2-1261
parametric curve, plotting 2-1168
Parent
   areaseries property 2-233
   Axes property 2-321
   barseries property 2-368
   contour property 2-726
   errorbar property 2-1102
   Figure property 2-1262
   hggroup property 2-1683
   hgtransform property 2-1712
   Image property 2-1780
   Light property 2-2105
   Line property 2-2129
   lineseries property 2-2143
   Patch property 2-2734
quivergroup property 2-2984
rectangle property 2-3052
Root property 2-3149
scatter property 2-3213
stairseries property 2-3398
stem property 2-3432
Surface property 2-3587
surfaceplot property 2-3611
Text property 2-3697
Uicontextmenu property 2-3840
Uicontrol property 2-3865
Uimenu property 2-3906
Uipushtool property 2-3941
Uitable property 2-3992
Uitoggletool property 2-4008
Uitoolbar property 2-4018
parentheses (special characters) 2-63
parfor 2-2693
parse method
  of inputParser object 2-2695
parseSoapResponse 2-2697
partial fraction expansion 2-3116
partialpath 2-2699
pascal 2-2701
Pascal matrix 2-2701 2-2842
patch 2-2702
Patch
  converting a surface to 2-3561
  creating 2-2702
  defining default properties 2-2712
  properties 2-2713
  reducing number of faces 2-3058
  reducing size of face 2-3277
path 2-2739
  adding directories to 2-126
  building from parts 2-1438
  current 2-2739
  removing directories from 2-3141
  toolbox directory 2-3757
  viewing 2-2743
path2rc 2-2741
pathname
  partial 2-2699
pathnames
  of functions or files 2-4172
  relative 2-2699
pathsep 2-2742
pathtool 2-2743
pause 2-2745
pauses, removing 2-853
pausing M-file execution 2-2745
pbaspect 2-2747
PBM
  parameters that can be set when
  writing 2-1822
PBM files
  writing 2-1818
pcg 2-2753
pchop 2-2757
pcode 2-2760
pcolor 2-2762
PCX files
  writing 2-1818
PDE. See Partial Differential Equations
pdepe 2-2766
pdeval 2-2778
percent sign (special characters) 2-64
percent-brace (special characters) 2-64
perfect matching 2-1012
performance 2-378
period (.), to distinguish matrix and array
  operations 2-40
period (special characters) 2-63
perl 2-2781
perl function 2-2781
Perl scripts in MATLAB 2-2781
perms 2-2784
permutation
  matrix 2-2234 2-2925
  of array dimensions 2-2785
random 2-3008
permutations of n elements 2-2784
permute 2-2785
persistent 2-2786
persistent variable 2-2786
perspective projection, setting and
querying 2-504
PGM
parameters that can be set when
writing 2-1822
PGM files
writing 2-1819
phase angle, complex 2-168
phase, complex
correcting angles 2-4047
pi 2-2789
pie 2-2790
pie3 2-2792
pinv 2-2794
planerot 2-2797
platform MATLAB is running on 2-686
playshow function 2-2798
plot 2-2799
editing 2-2811
plot (timeseries) 2-2806
plot box aspect ratio of axes 2-2747
plot editing mode
overview 2-2812
Plot Editor
interface 2-2812 2-2895
plot, volumetric
generating grid arrays for 2-2333
slice plot 2-3297
PlotBoxAspectRatio, Axes property 2-321
PlotBoxAspectRatioMode, Axes property 2-321
plotedit 2-2811
plotting
2-D plot 2-2799
3-D plot 2-2807
contours (a 2-1148
contours (ez function) 2-1148
ez-function mesh plot 2-1156
feather plots 2-1191
filled contours 2-1152
function plots 2-1386
functions with discontinuities 2-1176
histogram plots 2-1717
in polar coordinates 2-1171
isosurfaces 2-1987
loglog plot 2-2208
mathematical function 2-1164
mesh contour plot 2-1160
mesh plot 2-2328
parametric curve 2-1168
plot with two y-axes 2-2818
ribbon plot 2-3132
rose plot 2-3156
scatter plot 2-2814
scatter plot, 3-D 2-3198
stem plot, 3-D 2-3417
surface plot 2-3555
surfaces 2-1174
velocity vectors 2-692
volumetric slice plot 2-3297
. See visualizing
plus (M-file function equivalent for +) 2-45
PNG
writing options for 2-1823
alpha 2-1823
background color 2-1823
chromaticities 2-1824
gamma 2-1824
interlace type 2-1824
resolution 2-1825
significant bits 2-1824
transparency 2-1825
PNG files
writing 2-1819
PNM files
writing 2-1819
Pointer, Figure property 2-1262
PointerLocation, Root property 2-3149
PointerShapeCData, Figure property 2-1262
PointerShapeHotSpot, Figure property 2-1263
PointerWindow, Root property 2-3150
pol2cart 2-2823
polar 2-2825
polar coordinates 2-2823
computing the angle 2-168
converting from Cartesian 2-516
converting to cylindrical or Cartesian 2-2823
plotting in 2-1171
poles of transfer function 2-3116
poly 2-2827
polyarea 2-2830
polyder 2-2832
polyeig 2-2833
polyfit 2-2835
polygamma function 2-2900
polygon
area of 2-2830
creating with patch 2-2702
detecting points inside 2-1852
polyint 2-2839
polynomial
analytic integration 2-2839
characteristic 2-2827 to 2-2828 2-3154
coefficients (transfer function) 2-3116
curve fitting with 2-2835
derivative of 2-2832
division 2-916
eigenvalue problem 2-2833
evaluation 2-2840
evaluation (matrix sense) 2-2842
make piecewise 2-2400
multiplication 2-740
polyval 2-2840
polyvalm 2-2842
poorly conditioned
power 2-2845
  matrix. See matrix exponential
  of real numbers  2-3033
  of two, next  2-2566
power (M-file function equivalent for .^)  2-46
PPM
  parameters that can be set when
    writing 2-1822
PPM files
  writing 2-1819
ppval 2-2846
preallocation
  matrix 2-4231
precision 2-1379
  reading binary data writing 2-1403
prefdir 2-2848
preferences 2-2852
  opening the dialog box 2-2852
prime factors 2-1185
  dependence of Fourier transform on 2-1209
    2-1211 to 2-1212
prime numbers 2-2853
primes 2-2853
printdlg 2-2875
printdlg function 2-2875
printer
  default for linux and unix 2-2860
printer drivers
  GhostScript drivers 2-2856
  interpolated shading 2-2871
  MATLAB printer drivers 2-2856
printing
  GUIs 2-2869
  interpolated shading 2-2871
  on MS-Windows 2-2869
  with a variable file name 2-2872
  with nodisplay 2-2863
  with noFigureWindows 2-2863
  with non-normal EraseMode 2-2125 2-2725
    2-3050 2-3580 2-3685
  printing figures
    preview 2-2876
  printing tips 2-2869
  printing, suppressing 2-64
  printpreview 2-2876
  prod 2-2885
product
  cumulative 2-798
    Kronecker tensor 2-2043
    of array elements 2-2885
    of vectors (cross) 2-785
    scalar (dot) 2-785
profile 2-2887
profsave 2-2894
projection type, setting and querying 2-504
  ProjectionType, Axes property 2-322
prompting users for input 2-1854
prompting users to choose an item 2-2326
  propedit 2-2895 to 2-2896
properties 2-2897
proppanel 2-2899
pseudoinverse 2-2794
psi 2-2900
publish function 2-2902
push buttons 2-3845
PutFullMatrix 2-2911
pwd 2-2918

Q
qmr 2-2919
qr 2-2925
QR decomposition 2-2925
  deleting column from 2-2930
qrdelete 2-2930
qrinsert 2-2932
qrupdate 2-2934
quad 2-2937
quadgk 2-2946
quadl 2-2952
quadrature 2-2937 2-2946
quadv 2-2955
quantization
  performed by rgb2ind 2-3128
questdlg 2-2958
questdlg function 2-2958
quit 2-2962
quitting MATLAB 2-2962
quiver 2-2965
quiver3 2-2969
quotation mark
  inserting in a string 2-1396
qz 2-2993
QZ factorization 2-2834 2-2993

R
radio buttons 2-3845
rand, RandStream method 2-2997
randi, RandStream method 2-3002
randn, RandStream method 2-3007
random
  permutation 2-3008
  sparse matrix 2-3356 to 2-3357
  symmetric sparse matrix 2-3358
random number generators 2-2175 2-2997
  2-3002 2-3007 2-3010 2-3014
randperm 2-3008
randStream
  constructor 2-3014
RandStream 2-3010 2-3014
  constructor 2-3010
methods
  create 2-777
  get 2-1517
  getDefaultStream 2-1532
list 2-2175
rand 2-2997
randi 2-3002
randn 2-3007
setDefaultStream 2-3255
range space 2-2676
rank 2-3016
rank of a matrix 2-3016
RAS files
  parameters that can be set when
    writing 2-1826
    writing 2-1819
RAS image format
  specifying color order 2-1826
  writing alpha data 2-1826
Raster image files
  writing 2-1819
rational fraction approximation 2-3017
rbbox 2-3021 2-3065
rcond 2-3023
rdivide (M-file function equivalent for ./) 2-45
read 2-3024
readasync 2-3026
reading
  binary files 2-1403
  data from files 2-3706
  formatted data from file 2-1421
  formatted data from strings 2-3376
readme files, displaying 2-1937 2-4171
real 2-3029
real numbers 2-3029
reallog 2-3030
realmax 2-3031
realmin 2-3032
realpow 2-3033
realsqrt 2-3034
rearrange array
  flip along dimension 2-1354
  reverse along dimension 2-1354
rearrange matrix
  flip left-right 2-1355
  flip up-down 2-1356
  reverse column order 2-1355
  reverse row order 2-1356
RearrangeableColumn
  Uitable property 2-3993
rearranging arrays
  converting to vector 2-66
  removing first n singleton dimensions 2-3274
  removing singleton dimensions 2-3373
  reshaping 2-3114
  shifting dimensions 2-3274
  swapping dimensions 2-1924 2-2785
rearranging matrices
  converting to vector 2-66
  rotating 90\(\times fb\) 2-3160
  transposing 2-63
record 2-3035
rectangle
  properties 2-3042
  rectangle function 2-3037
rectint 2-3055
RecursionLimit
  Root property 2-3150
recycle 2-3056
reduced row echelon form 2-3171
reducepacth 2-3058
reducevolume 2-3062
reference page
  accessing from doc 2-1015
refresh 2-3065
regexprep 2-3081
regextranslate 2-3085
registerevent 2-3088
regression
  linear 2-2835
regularly spaced vectors, creating 2-66 2-2174
rehash 2-3091
relational operators 2-50 2-2206
relational operators for handle objects 2-3095
relative accuracy
  BVP 2-478
  DDE 2-896
  norm of DDE solution 2-896
  norm of ODE solution 2-2620
  ODE 2-2620
release 2-3093
rem 2-3097
removets 2-3100
rename function 2-3102
renderer
  OpenGL 2-1265
  painters 2-1264
  zbuffer 2-1265
Renderer, Figure property 2-1264
RendererMode, Figure property 2-1268
repeatedly executing statements 2-1377 2-4175
repeatedly executing statements in
  parallel 2-2694
replicating a matrix 2-3103
repmat 2-3103
resample (timeseries) 2-3105
resample (tscollection) 2-3108
reset 2-3111
reshape 2-3114
residue 2-3116
residues of transfer function 2-3116
Resize, Figure property 2-1270
ResizeFcn, Figure property 2-1270
restoredefaultpath 2-3120
rethrow 2-3121
rethrow, MException method 2-3123
return 2-3125
reverse
  array along dimension 2-1354
  array dimension 2-1354
  matrix column order 2-1355
  matrix row order 2-1356
reverse Cuthill-McKee ordering 2-3635 2-3645
rewinding files to beginning of 2-1420 2-1800
RGB images
  converting to indexed 2-3127
RGB, converting to HSV 2-3126
rgb2hsv 2-3126
rgb2ind 2-3127
rgbplot 2-3130
ribbon 2-3132
right-click and context menus 2-3831
rmappdata function 2-3135
rmdir 2-3136
rmdir (ftp) function 2-3139
rmfield 2-3140
rmpath 2-3141
rmpref function 2-3142
RMS. See root-mean-square
rolling camera 2-505
root directory 2-2274
root directory for MATLAB 2-2274
Root graphics object 2-3143
root object 2-3143
root, see rootobject 2-3143
root-mean-square
  of vector 2-2570
roots 2-3154
roots of a polynomial 2-2827 to 2-2828 2-3154
rose 2-3156
Rosenbrock
  banana function 2-1367
  ODE solver 2-2609
rosser 2-3159
rot90 2-3160
rotate 2-3161
rotate3d 2-3164
rotate3d mode objects 2-3164
rotating camera 2-499
rotating camera target 2-501
Rotation, Text property 2-3698
rotations
  Jacobi 2-3358
round 2-3170
  to nearest integer 2-3170
  towards infinity 2-552
  towards minus infinity 2-1358
  towards zero 2-1353
roundoff error
  characteristic polynomial and 2-2828
  convolution theorem and 2-741
  effect on eigenvalues 2-340
  evaluating matrix functions 2-1453
  in inverse Hilbert matrix 2-1920
  partial fraction expansion and 2-3117
  polynomial roots and 2-3154
  sparse matrix conversion and 2-3324
RowName
  Uitable property 2-3993
RowStriping
  Uitable property 2-3993
rref 2-3171
rrefmovie 2-3171
rsf2csf 2-3173
rubberband box 2-3021
run 2-3175
Runge-Kutta ODE solvers 2-2608
running average 2-1320
S
save 2-3176 2-3184
  serial port I/O 2-2609
saveas 2-3188
saveobj 2-3192
savepath 2-3194
saving
  ASCII data 2-3176
  session to a file 2-977
  workspace variables 2-3176
scalar product (of vectors) 2-785
scaled complementary error function
  (defined) 2-1079
scatter 2-3195
scatter3 2-3198
scattered data, aligning
  multi-dimensional 2-2492
  two-dimensional 2-1585
scattergroup
  properties 2-3201
Schmidt semi-normalized Legendre functions 2-2076
schur 2-3219
Schur decomposition 2-3219
Schur form of matrix 2-3173 2-3219
screen, paging 2-1656
ScreenWidth, Root property 2-3150
ScreenPixelsPerInch, Root property 2-3151
ScreenSize, Root property 2-3151
script 2-3222
scrolling screen 2-1656
search path 2-3141
  adding directories to 2-126
  MATLAB 2-2739
  modifying 2-2743
  user directory 2-4061
  viewing 2-2743
search, string 2-1339
sec 2-3223
secant 2-3223
  hyperbolic 2-3226
  inverse 2-245
  inverse hyperbolic 2-248
secd 2-3225
sech 2-3226
Selected
  areaseries property 2-233
  Axes property 2-323
  barseries property 2-368
  contour property 2-726
  errorbar property 2-1102
  Figure property 2-1271
  hggroup property 2-1683
  hgtransform property 2-1712
  Image property 2-1781
  Light property 2-2106
  Line property 2-2129
  lineseries property 2-2143
Patch property 2-2735
quivergroup property 2-2984
rectangle property 2-3053
Root property 2-3151
scatter property 2-3213
stairseries property 2-3398
stem property 2-3433
Surface property 2-3587
surfaceplot property 2-3611
Text property 2-3698
Uicontrol property 2-3866
Uitable property 2-3994
selecting areas 2-3021
SelectionHighlight
  areaseries property 2-233
  Axes property 2-323
  barseries property 2-368
  contour property 2-727
  errorbar property 2-1103
  Figure property 2-1271
  hggroup property 2-1683
  hgtransform property 2-1712
  Image property 2-1781
  Light property 2-2106
  Line property 2-2129
  lineseries property 2-2143
  Patch property 2-2735
  quivergroup property 2-2985
  rectangle property 2-3053
  scatter property 2-3213
  stairseries property 2-3398
  stem property 2-3433
  Surface property 2-3587
  surfaceplot property 2-3611
  Text property 2-3698
  Uicontrol property 2-3866
  Uitable property 2-3994
SelectionType, Figure property 2-1271
selectmoveresize 2-3228
semicolon (special characters) 2-64
sendmail 2-3232
Separator
  Upushtool property 2-3941
  Uitoggletool property 2-4008
Separator, Uimenu property 2-3907
sequence of matrix names (M1 through M12)
  generating 2-1115
serial 2-3234
serialbreak 2-3237
server (FTP)
  connecting to 2-1435
server variable 2-1201
session
  saving 2-977
set 2-3238 2-3242
  serial port I/O 2-3245
  timer object 2-3247
set (timeseries) 2-3250
set (tscollection) 2-3251
set hgsetget class method 2-3243
set operations
  difference 2-3256
  exclusive or 2-3270
  intersection 2-1910
  membership 2-1970
  union 2-4024
  unique 2-4026
setabstime (timeseries) 2-3252
setabstime (tscollection) 2-3253
setappdata 2-3254
setDefaultStream, RandStream method 2-3255
setdiff 2-3256
setdisp hgsetget class method 2-3257
setenv 2-3258
setfield 2-3260
setinterpmethod 2-3262
setpixelposition 2-3264
setpref function 2-3267
setstr 2-3268
settimeseriesnames 2-3269
setxor 2-3270
shading 2-3271
shading colors in surface plots 2-3271
shared libraries
  MATLAB functions
    calllib 2-487
    libfunctions 2-2085
    libfunctionsview 2-2086
    libisloaded 2-2087
    libpointer 2-2089
    libstruct 2-2091
    loadlibrary 2-2192
    unloadlibrary 2-4032
shell script 2-3653 2-4029
shiftdim 2-3274
shifting array
  circular 2-597
ShowArrowHead
  quivergroup property 2-2985
ShowBaseLine
  barseries property 2-368
ShowHiddenHandles, Root property 2-3152
showplottool 2-3275
ShowText
  contour property 2-727
shrinkfaces 2-3277
shutdown 2-2962
sign 2-3281
signum function 2-3281
simplex search 2-1369
Simpson’s rule, adaptive recursive 2-2939
Simulink
  version number, comparing 2-4100
  version number, displaying 2-4094
sin 2-3282
sind 2-3284
sine 2-3282
  hyperbolic 2-3286
  inverse 2-250
  inverse hyperbolic 2-253
single 2-3285
single quote (special characters) 2-63
singular value
decomposition 2-3016 2-3624
largest 2-2570
rank and 2-3016
sinh 2-3286
size
array dimensions 2-3288
serial port I/O 2-3293
size (timeseries) 2-3294
size (tscollection) 2-3296
size of array dimensions 2-3288
size of fonts, see also FontSize property 2-3701
size vector 2-3114
SizeData
scatter property 2-3214
skipping bytes (during file I/O) 2-1455
slice 2-3297
slice planes, contouring 2-735
sliders 2-3845
SliderStep, Uicontrol property 2-3866
smallest array elements 2-2385
smooth3 2-3303
smoothing 3-D data 2-3303
soccer ball (example) 2-3645
solution statistics (BVP) 2-483
sort 2-3310
sorting
array elements 2-3310
complex conjugate pairs 2-775
matrix rows 2-3314
sortrows 2-3314
sound 2-3317 to 2-3318
converting vector into 2-3317 to 2-3318
files
reading 2-281 2-4158
writing 2-282 2-4163
playing 2-4156
recording 2-4161
resampling 2-4156
sampling 2-4161
source control on UNIX platforms
checking out files
function 2-578
source control system
viewing current system 2-631
source control systems
checking in files 2-575
undo checkout 2-4022
spalloc 2-3319
sparse 2-3320
sparse matrix
allocating space for 2-3319
applying function only to nonzero elements
of 2-3337
density of 2-2567
detecting 2-2004
diagonal 2-3325
finding indices of nonzero elements of 2-1327
identity 2-3336
number of nonzero elements in 2-2567
permuting columns of 2-673
random 2-3356 to 2-3357
random symmetric 2-3358
replacing nonzero elements of with
ones 2-3350
results of mixed operations on 2-3321
solving least squares linear system 2-2926
specifying maximum number of nonzero
elements 2-3320
vector of nonzero elements 2-2569
visualizing sparsity pattern of 2-3367
sparse storage
criterion for using 2-1437
spaugment 2-3322
spconvert 2-3323
spdiags 2-3325
special characters
descriptions 2-1654
overloading 2-65
specular 2-3335
SpecularColorReflectance
  Patch property 2-2735
  Surface property 2-3587
  surfaceplot property 2-3611
SpecularExponent
  Patch property 2-2736
  Surface property 2-3588
  surfaceplot property 2-3612
SpecularStrength
  Patch property 2-2736
  Surface property 2-3588
  surfaceplot property 2-3612
speye 2-3336
spfun 2-3337
sph2cart 2-3339
sphere 2-3340
spherical coordinates
  defining a Light position in 2-2108
spherical coordinates 2-3339
spinmap 2-3342
spline 2-3343
spline interpolation (cubic)
  one-dimensional 2-1887 2-1897 2-1900
  2-1903
Spline Toolbox 2-1892
spones 2-3350
spparms 2-3351
sprand 2-3356
sprandn 2-3357
sprandsym 2-3358
sprank 2-3359
spreadsheets
  loading WK1 files 2-4193
  loading XLS files 2-4207
  reading into a matrix 2-1004
  writing from matrix 2-4195
  writing matrices into 2-1008
sprintf 2-3360
sqrt 2-3369
sqrtm 2-3370
square root
  of a matrix 2-3370
  of array elements 2-3369
  of real numbers 2-3034
squeeze 2-3373
sscanf 2-3376
stack, displaying 2-863
standard deviation 2-3407
start
  timer object 2-3403
startat
  timer object 2-3404
startup 2-3406
  directory and path 2-4061
startup file 2-3406
startup files 2-2273
State
  Uitoggletool property 2-4008
static text 2-3845
std 2-3407
std (timeseries) 2-3409
stem 2-3411
stem3 2-3417
step size (DDE)
  initial step size 2-900
  upper bound 2-901
step size (ODE) 2-899 2-2625
  initial step size 2-2625
  upper bound 2-2625
stop
  timer object 2-3439
stopping 2-3440
stopwatch timer 2-3735
storage
  allocated for nonzero entries (sparse) 2-2588
  sparse 2-3320
storage allocation 2-4231
str2cell 2-568
Index

str2double 2-3441
str2func 2-3442
str2mat 2-3446
str2num 2-3447
strcat 2-3451
stream lines
  computing 2-D 2-3457
  computing 3-D 2-3459
  drawing 2-3461
stream2 2-3457
stream3 2-3459
stretch-to-fill 2-291
strfind 2-3489
string
  comparing one to another 2-3453 2-3495
  converting from vector to 2-574
  converting matrix into 2-2264 2-2584
  converting to lowercase 2-2217
  converting to numeric array 2-3447
  converting to uppercase 2-4054
  dictionary sort of 2-3314
  finding first token in 2-3507
  searching and replacing 2-3506
  searching for 2-1339
String
  Text property 2-3698
  textarrow property 2-196
  textbox property 2-208
  Uicontrol property 2-3867
string matrix to cell array conversion 2-568
strings 2-3491
  converting to matrix (formatted) 2-3376
  inserting a quotation mark in 2-1396
  writing data to 2-3360
strjust 2-3493
strmatch 2-3494
strread 2-3498
strrep 2-3506
strtok 2-3507
strtrim 2-3510
struct 2-3511
struct2cell 2-3516
structfun 2-3517
structure array
  getting contents of field of 2-1535
  remove field from 2-3140
  setting contents of a field of 2-3260
structure arrays
  field names of 2-1231
structures
  dynamic fields 2-64
strvcat 2-3520
Style
  Light property 2-2106
    Uicontrol property 2-3870
sub2ind 2-3522
subfunction 2-1441
subplot 2-3524
subplots
  assymetrical 2-3529
  suppressing ticks in 2-3532
subsasgn 1-77 2-3537
subscripts
  in axis title 2-3753
  in text strings 2-3702
subsindex 2-3539
subspace 2-3540
subsref 2-3541
subsref (M-file function equivalent for
  A(i,j,k...) 2-65
substruct 2-3543
subtraction (arithmetic operator) 2-40
subvolume 2-3545
sum 2-3548
  cumulative 2-800
    of array elements 2-3548
  sum (timeseries) 2-3551
superiorsto 2-3553
superscripts
  in axis title 2-3753
in text strings 2-3702
support 2-3554
surf2patch 2-3561
surface 2-3563
Surface
    and contour plotter 2-1178
    converting to a patch 2-3561
    creating 2-3563
    defining default properties 2-3040 2-3567
    plotting mathematical functions 2-1174
    properties 2-3568 2-3591
surface normals, computing for volumes 2-1984
surf1 2-3618
surfnorm 2-3622
svd 2-3624
svds 2-3627
swapbytes 2-3630
switch 2-3632
symamd 2-3634
symbfact 2-3638
symbols
    operators 2-1654
symbols in text 2-196 2-208 2-3699
symmlq 2-3640
symrcm 2-3645
synchronize 2-3648
syntax 2-1655
syntax, command 2-3650
syntax, function 2-3650
syntaxes
    of M-file functions, defining 2-1441
system 2-3653
    UNC pathname error 2-3654
system directory, temporary 2-3663

T

table lookup. See interpolation
Tag
    areareaseries property 2-233

Axes property 2-323
barseries property 2-369
contour property 2-727
errorbar property 2-1103
Figure property 2-1273
hggroup property 2-1683
hgtransform property 2-1713
Image property 2-1781
Light property 2-2106
Line property 2-2130
lineseries property 2-2144
Patch property 2-2736
quivergroup property 2-2985
rectangle property 2-3053
Root property 2-3152
scatter property 2-3214
stairseries property 2-3399
stem property 2-3433
Surface property 2-3588
surfaceplot property 2-3612
Text property 2-3703
Uicontextmenu property 2-3840
Uicontrol property 2-3870
Uimenu property 2-3907
Uipushtool property 2-3941
Uitable property 2-3994
Uitoggletool property 2-4008
Uitoolbar property 2-4018
Tagged Image File Format (TIFF)
    writing 2-1819
tan 2-3656
tand 2-3658
tangent 2-3656
    four-quadrant, inverse 2-261
    hyperbolic 2-3659
    inverse 2-259
    inverse hyperbolic 2-264	anh 2-3659
tar 2-3661
target, of camera 2-506
tcpip 2-4056
tempdir 2-3663
tempname 2-3664
temporary
  files 2-3664
  system directory 2-3663
tensor, Kronecker product 2-2043
terminating MATLAB 2-2962
test matrices 2-1467
test, logical.  See logical tests and detecting
tetrahedron
  mesh plot 2-3665
tetramesh 2-3665
TeX commands in text 2-196 2-208 2-3699
text 2-3670
  editing 2-2811
  subscripts 2-3702
  superscripts 2-3702
Text
  creating 2-3670
  defining default properties 2-3673
  fixed-width font 2-3687
  properties 2-3675
text mode for opened files 2-1371
TextBackgroundColor
textarrow property 2-198
TextColor
  textarrow property 2-198
TextEdgeColor
textarrow property 2-198
TextLineWidth
  textarrow property 2-199
TextList
  contour property 2-728
TextListMode
  contour property 2-728
TextMargin
  textarrow property 2-199
textrread 2-3706
TextRotation, textarrow property 2-199
textscan 2-3712
TextStep
  contour property 2-729
TextStepMode
  contour property 2-729
textwrap 2-3726
Tfqmr 2-3727
throw, MException method 2-3730
throwAsCaller, MException method 2-3733
TickDir, Axes property 2-324
TickDirMode, Axes property 2-324
TickLength, Axes property 2-324
TIFF
  compression 2-1827
  encoding 2-1822
  ImageDescription field 2-1827
  maxvalue 2-1822
  parameters that can be set when
    writing 2-1826
  resolution 2-1827
  writemode 2-1827
  writing 2-1819
TIFF image format
  specifying color space 2-1826
tiling (copies of a matrix) 2-3103
time
  CPU 2-776
  elapsed (stopwatch timer) 2-3735
  required to execute commands 2-1111
time and date functions 2-1073
timer
  properties 2-3738
  timer object 2-3738
timerfind
timer object 2-3745
timerfindall
timer object 2-3747
times (M-file function equivalent for .*) 2-45
timeseries 2-3749
timestamp 2-982
title 2-3752
    with superscript 2-3753
Title, Axes property 2-325
todatenum 2-3755
toeplitz 2-3756
Toeplitz matrix 2-3756
toggle buttons 2-3845
token 2-3507
    See also string
Toolbar
    Figure property 2-1273
Toolbox
    Spline 2-1892
toolbox directory, path 2-3757
toolboxdir 2-3757
TooltipString
    Uicontrol property 2-3870
    Uipushtool property 2-3941
    Uitable property 2-3994
    Uitoggletool property 2-4008
trace 2-3758
trace of a matrix 2-973 2-3758
trailing blanks
    removing 2-908
transform
    hgtransform function 2-1691
transform, Fourier
    discrete, n-dimensional 2-1212
    discrete, one-dimensional 2-1206
    discrete, two-dimensional 2-1211
    inverse, n-dimensional 2-1749
    inverse, one-dimensional 2-1745
    inverse, two-dimensional 2-1747
    shifting the zero-frequency component of 2-1215
transformation
    See also conversion 2-538
transformations
    elementary Hermite 2-1496
transmitting file to FTP server 2-2458
transpose
    array (arithmetic operator) 2-42
    matrix (arithmetic operator) 2-42
transpose (M-file function equivalent for \q 2-46
transpose (timeseries) 2-3759
trapz 2-3761
treelayout 2-3763
treeplot 2-3764
triangulation
    2-D plot 2-3770
tricubic interpolation 2-1585
tril 2-3766
trilinear interpolation 2-1585
trimesh 2-3767
triple integral
    numerical evaluation 2-3768
triplequad 2-3768
triplot 2-3770
trisurf 2-3784
triu 2-3785
true 2-3786
truth tables (for logical operations) 2-52
try 2-3787
tscollection 2-3790
tsdta.evnet 2-3793
tsearch 2-3794
tsearchn 2-3795
tsprops 2-3796
tstool 2-3802
type 2-3803
Type
    areaseries property 2-234
    Axes property 2-325
    barseries property 2-369
    contour property 2-729
    errorbar property 2-1103
    Figure property 2-1274
    hggroup property 2-1684
    hgtransform property 2-1713
Index

Image property 2-1782
Light property 2-2106
Line property 2-2130
lineseries property 2-2144
Patch property 2-2737
quivergroupproperty 2-2986
rectangle property 2-3054
Root property 2-3152
scatter property 2-3214
stairseries property 2-3399
stem property 2-3434
Surface property 2-3588
surfaceplot property 2-3613
Text property 2-3703
Uicontextmenu property 2-3841
Uicontrol property 2-3870
Uimenu property 2-3907
Uipushtool property 2-3941
Uitable property 2-3995
Uitoggletool property 2-4009
Uitoolbar property 2-4019

typecast 2-3804

U

UData
  errorbar property 2-1104
  quivergroupproperty 2-2987
UDataSource
  errorbar property 2-1104
  quivergroupproperty 2-2987
Uibuttongroup
  defining default properties 2-3813
uibuttongroup function 2-3808
Uibuttongroup Properties 2-3813
uicontextmenu 2-3831
UIContextMenu
  Uicontrol property 2-3871
  Uipushtool property 2-3942
  Uitoggletool property 2-4009
  Uitoolbar property 2-4019
  UIContextMenu
    areaseries property 2-234
    Axes property 2-326
    barseries property 2-369
    contour property 2-730
    errorbar property 2-1104
    Figure property 2-1274
    hggroup property 2-1684
    hgtransform property 2-1713
    Image property 2-1782
    Light property 2-2107
    Line property 2-2130
    lineseries property 2-2144
    Patch property 2-2737
    quivergroupproperty 2-2986
    rectangle property 2-3054
    scatter property 2-3215
    stairseries property 2-3399
    stem property 2-3434
    Surface property 2-3589
    surfaceplot property 2-3613
    Text property 2-3704
    Uitable property 2-3995
Uicontextmenu Properties 2-3834
uicontrol 2-3842
Uicontrol
  defining default properties 2-3848
  fixed-width font 2-3858
  types of 2-3842
Uicontrol Properties 2-3848
uicontrols
  printing 2-2869
uigetdir 2-3874
uigetfile 2-3879
uigetpref function 2-3890
uiimport 2-3894
uiuimenu 2-3895
Uimenu
  creating 2-3895
defining default properties 2-3897
Properties 2-3897
Uimenu Properties 2-3897
uint16 2-3908
uint32 2-3908
uint64 2-3908
uint8 2-1881 2-3908
uiopen 2-3910
Uipanel
  defining default properties 2-3915
uipanel function 2-3912
Uipanel Properties 2-3915
uipushtool 2-3931
Uipushtool
  defining default properties 2-3933
Uipushtool Properties 2-3933
uiputfile 2-3943
uiresume 2-3953
uisave 2-3955
uisetcolor function 2-3957
uisetfont 2-3958
uisetpref function 2-3960
uistack 2-3961
Uitable
  defining default properties 2-3968
  fixed-width font 2-3987
uitable function 2-3962
Uitable Properties 2-3968
uicontroltool 2-3997
Uicontroltool
  defining default properties 2-3999
Uicontroltool Properties 2-3999
uicontrolbar 2-4010
Uicontrolbar
  defining default properties 2-4012
Uicontrolbar Properties 2-4012
uiwait 2-4020
uminus (M-file function equivalent for unary \xd0 ) 2-45
UNCPathname error and dos 2-1024
UNCPathname error and system 2-3654
unconstrained minimization 2-1365
undefined numerical results 2-2481
undocheckout 2-4022
unicode2native 2-4023
unimodular matrix 2-1496
union 2-4024
unique 2-4026
unitary matrix (complex) 2-2925
Units
  annotation ellipse property 2-183
  annotation rectangle property 2-190
  arrow property 2-175
  Axes property 2-326
  doublearrow property 2-180
  Figure property 2-1274
  line property 2-186
  Root property 2-3152
  Text property 2-3703
  textarrow property 2-199
  textbox property 2-210
  Uicontrol property 2-3871
  Uitable property 2-3895
unix 2-4029
UNIX
  Web browser 2-1018
unloadlibrary 2-4032
unlocking M-files 2-2478
unmkpp 2-4037
unregisterallevents 2-4038
unregisterevent 2-4041
untar 2-4045
unwrap 2-4047
unzip 2-4052
up vector, of camera 2-508
updating figure during M-file execution 2-1029
uplus (M-file function equivalent for unary +) 2-45
upper 2-4054
upper triangular matrix 2-3785
uppercase to lowercase 2-2217
urlread 2-4055
urlwrite 2-4057
usejava 2-4059
user input
  from a button menu 2-2326
UserData
  areaseries property 2-234
  Axes property 2-327
  barseries property 2-370
  contour property 2-730
  errorbar property 2-1105
  Figure property 2-1275
  hggroupproperty 2-1684
  hgtransform property 2-1714
  Image property 2-1782
  Light property 2-2107
  Line property 2-2130
  lineseries property 2-2145
  Patch property 2-2737
  quivergroupproperty 2-2986
  rectangle property 2-3054
  Root property 2-3153
  scatter property 2-3215
  stairseries property 2-3400
  stem property 2-3434
  Surface property 2-3589
  surfaceplot property 2-3613
  Text property 2-3704
  Uicontextmenu property 2-3841
  Uicontrol property 2-3871
  Uimenu property 2-3907
  Uipushtool property 2-3942
  Uitable property 2-3996
  Uitoggletool property 2-4009
  Uitoolbar property 2-4019
userpath 2-4061

V
validateattributes 2-4071
validatestring 2-4079
Value, Uicontrol property 2-3872
vander 2-4086
Vandermonde matrix 2-2837
var 2-4087
var (timeseries) 2-4088
varargin 2-4090
varargout 2-4092
variable numbers of M-file arguments 2-4092
variable-order solver (ODE) 2-2634
variables
  checking existence of 2-1135
  clearing from workspace 2-614
  global 2-1567
  in workspace 2-4197
  keeping some when clearing 2-620
  linking to graphs with linkdata 2-2159
  listing 2-4181
  local 2-1441 2-1567
  name of passed 2-1859
  opening 2-2642 2-2653
  persistent 2-2786
  saving 2-3176
  sizes of 2-4181
VData
  quivergroupproperty 2-2987
VDataSource
  quivergroupproperty 2-2988
vector
  dot product 2-1025
  frequency 2-2214
  length of 2-2080
  product (cross) 2-785
vector field, plotting 2-692
vectorize 2-4093
vectorizing ODE function (BVP) 2-479
vectors, creating
  logarithmically spaced 2-2214
regularly spaced 2-66 2-2174
velocity vectors, plotting 2-692
ver 2-4094
verctrl function (Windows) 2-4096
verLessThan 2-4100
version 2-4102
version numbers
   comparing 2-4100
   displaying 2-4094
vertcat 2-4104
vertcat (M-file function equivalent for [ 2-65
vertcat (timeseries) 2-4106
vertcat (tscollection) 2-4107
VertexNormals
   Patch property 2-2737
   Surface property 2-3589
   surfaceplot property 2-3613
VerticalAlignment, Text property 2-3704
VerticalAlignment, textarrow property 2-200
VerticalAlignment, textbox property 2-210
Vertices, Patch property 2-2737
video
   saving in AVI format 2-283
view 2-4111
   azimuth of viewpoint 2-4112
   coordinate system defining 2-4112
   elevation of viewpoint 2-4112
view angle, of camera 2-510
View, Axes property (obsolete) 2-327
viewing
   a group of object 2-497
   a specific object in a scene 2-497
viewmtx 2-4114
Visible
   areaserseries property 2-235
   Axes property 2-327
   barseries property 2-370
   contour property 2-730
   errorbar property 2-1105
   Figure property 2-1275
   hgtestparent property 2-1685
   hgtransform property 2-1714
   Image property 2-1782
   Light property 2-2107
   Line property 2-2130
   lineseries property 2-2145
   Patch property 2-2737
   quivergroup property 2-2986
   rectangle property 2-3054
   Root property 2-3153
   scatter property 2-3215
   stairseries property 2-3400
   stem property 2-3434
   Surface property 2-3589
   surfaceplot property 2-3614
   Text property 2-3705
   Uicontextmenu property 2-3841
   Uicontrol property 2-3872
   Uimenu property 2-3907
   Uipushtool property 2-3942
   Uitable property 2-3996
   Uitoggletool property 2-4009
   Uitoolbar property 2-4019
visualizing
   cell array structure 2-566
   sparse matrices 2-3367
volumes
   calculating isosurface data 2-1987
   computing 2-D stream lines 2-3457
   computing 3-D stream lines 2-3459
   computing isosurface normals 2-1984
   contouring slice planes 2-735
drawing stream lines 2-3461
end caps 2-1977
reducing face size in isosurfaces 2-3277
reducing number of elements in 2-3062
voronoi 2-4126
Voronoi diagrams
   multidimensional visualization 2-4133
two-dimensional visualization 2-4126
W

wait
    timer object 2-4137
waitbar 2-4138
waitfor 2-4142
waitforbuttonpress 2-4143
warndlg 2-4144
warning 2-4147
warning message (enabling, suppressing, and displaying) 2-4147
waterfall 2-4151
.wav files
    reading 2-4158
    writing 2-4163
waverecord 2-4161
wavinfo 2-4155
wavplay 2-4156
wavread 2-4155 2-4158
wavrecord 2-4161
wavwrite 2-4163
WData
    quivergroupproperty 2-2988
WDataSource
    quivergroupproperty 2-2989
Web browser
    displaying help in 2-1658
    specifying for UNIX 2-1018
weekday 2-4165
well conditioned 2-3023
what 2-4167
whatsnew 2-4171
which 2-4172
while 2-4175
white space characters, ASCII 2-2003 2-3507
whitebg 2-4179
who, whos
    who 2-4181

wilkinson 2-4188
Wilkinson matrix 2-3329 2-4188
WindowButtonDownFcn, Figure property 2-1276
WindowButtonMotionFcn, Figure property 2-1276
WindowButtonUpFcn, Figure property 2-1277
WindowKeyPressFcn, Figure property 2-1278
WindowKeyReleaseFcn, Figure property 2-1279
Windows Paintbrush files
    writing 2-1818
WindowScrollWheelFcn, Figure property 2-1280
WindowStyle, Figure property 2-1282
winopen 2-4189
winqueryreg 2-4190
WK1 files
    loading 2-4193
    writing from matrix 2-4195
wk1finfo 2-4192
wk1read 2-4193
wk1write 2-4195
workspace 2-4197
    changing context while debugging 2-857 2-882
    clearing items from 2-614
    consolidating memory 2-2678
    predefining variables 2-3406
    saving 2-3176
    variables in 2-4181
    viewing contents of 2-4197
workspace variables
    reading from disk 2-2183
writing
    binary data to file 2-1455
    formatted data to file 2-1391
WVisual, Figure property 2-1284
WVisualMode, Figure property 2-1287

X

X
Index

annotation arrow property 2-176 2-180
annotation line property 2-186
text arrow property 2-201

X Windows Dump files
writing 2-1819

x-axis limits, setting and querying 2-4202
XAxisLocation, Axes property 2-327
XColor, Axes property 2-328
XData
areaseries property 2-235
barseries property 2-370
contour property 2-730
errorbar property 2-1105
Image property 2-1782
Line property 2-2131
lineseries property 2-2145
Patch property 2-2738
quivergroup property 2-2989
scatter property 2-3215
stairseries property 2-3400
stem property 2-3435
Surface property 2-3589
surfaceplot property 2-3614
XDataMode
areaseries property 2-235
barseries property 2-370
contour property 2-731
errorbar property 2-1105
lineseries property 2-2145
quivergroup property 2-2990
stairseries property 2-3401
stem property 2-3435
surfaceplot property 2-3614
XDataSource
areaseries property 2-236
barseries property 2-371
contour property 2-731
errorbar property 2-1106
lineseries property 2-2146
quivergroup property 2-2990

scatter property 2-3216
stairseries property 2-3401
stem property 2-3435
surfaceplot property 2-3614

XDir, Axes property 2-328
XDisplay, Figure property 2-1287
XGrid, Axes property 2-329
xlabel 2-4200
XLabel, Axes property 2-329
xlim 2-4202
XLim, Axes property 2-330
XLimMode, Axes property 2-330
XLS files
loading 2-4207
xlsfinfo 2-4205
xlsread 2-4207
xlswrite 2-4217
xmlread 2-4222
xmlwrite 2-4227
xor 2-4228
XOR, printing 2-228 2-229 2-720 2-1096 2-1709
2-1778 2-2125 2-2138 2-2725 2-2979 2-3050
2-3208 2-3393 2-3427 2-3580 2-3603 2-3685
XScale, Axes property 2-331
xslt 2-4229
XTick, Axes property 2-331
XTickLabel, Axes property 2-332
XTickLabelMode, Axes property 2-333
XTickMode, Axes property 2-332
XVisual, Figure property 2-1288
XVisualMode, Figure property 2-1289
XWD files
writing 2-1819
xyz coordinates. See Cartesian coordinates

Y

Y

annotation arrow property 2-176 2-180 2-187
textarrow property 2-201
y-axis limits, setting and querying 2-4202
YAxisLocation, Axes property 2-327
YColor, Axes property 2-328
YData
  areaseries property 2-236
  barseries property 2-371
  contour property 2-732
  errorbar property 2-1106
  Image property 2-1783
  Line property 2-2131
  lineseries property 2-2146
  Patch property 2-2738
  quivergroupproperty 2-2991
  scatter property 2-3216
  stairseries property 2-3402
  stem property 2-3436
  Surface property 2-3590
  surfaceplot property 2-3615
YDataMode
  contour property 2-732
  quivergroupproperty 2-2991
  surfaceplot property 2-3615
YDataSource
  areaseries property 2-237
  barseries property 2-372
  contour property 2-732
  errorbar property 2-1107
  lineseries property 2-2147
  quivergroupproperty 2-2991
  scatter property 2-3217
  stairseries property 2-3402
  stem property 2-3436
  surfaceplot property 2-3615
YDir, Axes property 2-328
YGrid, Axes property 2-329
ylabel 2-4200
YLabel, Axes property 2-329
ylim 2-4202
YLim, Axes property 2-330
YLimMode, Axes property 2-330
YMinorGrid, Axes property 2-331
YScale, Axes property 2-331
YTick, Axes property 2-331
YTickLabel, Axes property 2-332
YTickLabelMode, Axes property 2-333
YTickMode, Axes property 2-332
Z
z-axis limits, setting and querying 2-4202
ZColor, Axes property 2-328
ZData
  contour property 2-733
  lineseries property 2-2147
  Patch property 2-2738
  quivergroupproperty 2-2992
  scatter property 2-3217
  stemseries property 2-3437
  Surface property 2-3590
  surfaceplot property 2-3616
ZDataSource
  contour property 2-733
  lineseries property 2-2147
  scatter property 2-3217
  surfaceplot property 2-3616
ZDir, Axes property 2-328
zero of a function, finding 2-1461
zeros 2-4231
ZGrid, Axes property 2-329
Ziggurat 2-3010 2-3014
zip 2-4233
zlabel 2-4200
zlim 2-4202
ZLim, Axes property 2-330
ZLimMode, Axes property 2-330
ZMinorGrid, Axes property 2-331
zoom 2-4236
zoom mode objects 2-4237
ZScale, Axes property 2-331
ZTick, Axes property 2-331
ZTickLabel, Axes property 2-332
ZTickLabelMode, Axes property 2-333
ZTickMode, Axes property 2-332