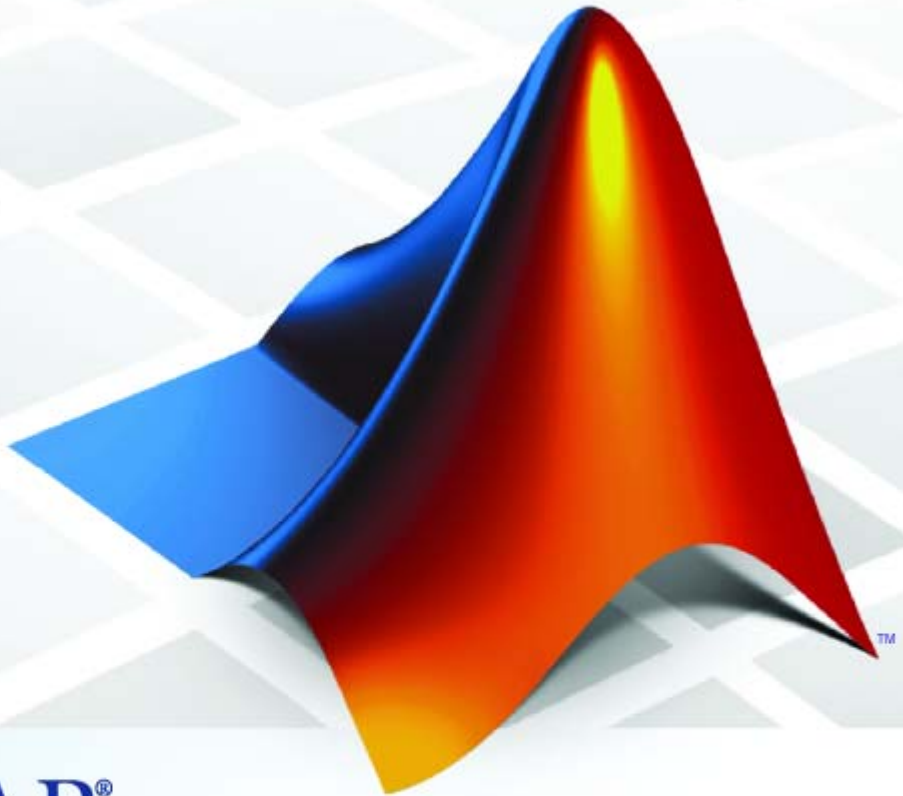


**MATLAB® 7**

Function Reference: F - O



**MATLAB®**

## How to Contact The MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto: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*

© COPYRIGHT 1984–2008 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or 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](http://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](http://www.mathworks.com/patents) for more information.

### **Revision History**

December 1996	First printing	For MATLAB 5.0 (Release 8)
June 1997	Online only	Revised for MATLAB 5.1 (Release 9)
October 1997	Online only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online only	Revised for MATLAB 5.3 (Release 11)
June 1999	Second printing	For MATLAB 5.3 (Release 11)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for 6.5 (Release 13)
June 2004	Online only	Revised for 7.0 (Release 14)
September 2006	Online only	Revised for 7.3 (Release 2006b)
March 2007	Online only	Revised for 7.4 (Release 2007a)
September 2007	Online only	Revised for Version 7.5 (Release 2007b)
March 2008	Online only	Revised for Version 7.6 (Release 2008a)
October 2008	Online only	Revised for Version 7.7 (Release 2008b)



## Function Reference

**1**

<b>Desktop Tools and Development Environment</b> . . . . .	<b>1-3</b>
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
<b>Mathematics</b> . . . . .	<b>1-13</b>
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 . . . . .	1-31
Nonlinear Numerical Methods . . . . .	1-32
Specialized Math . . . . .	1-35
Sparse Matrices . . . . .	1-36
Math Constants . . . . .	1-40
<b>Data Analysis</b> . . . . .	<b>1-41</b>
Basic Operations . . . . .	1-41
Descriptive Statistics . . . . .	1-41
Filtering and Convolution . . . . .	1-42
Interpolation and Regression . . . . .	1-42
Fourier Transforms . . . . .	1-43
Derivatives and Integrals . . . . .	1-43
Time Series Objects . . . . .	1-44
Time Series Collections . . . . .	1-47
<b>Programming and Data Types</b> . . . . .	<b>1-49</b>
Data Types . . . . .	1-49
Data Type Conversion . . . . .	1-57
Operators and Special Characters . . . . .	1-59
Strings . . . . .	1-61

Bit-Wise Operations .....	1-64
Logical Operations .....	1-65
Relational Operations .....	1-66
Set Operations .....	1-66
Date and Time Operations .....	1-66
Programming in MATLAB .....	1-67
<b>Object-Oriented Programming .....</b>	<b>1-75</b>
Classes and Objects .....	1-75
Handle Classes .....	1-76
Events and Listeners .....	1-77
Meta-Classes .....	1-77
<b>File I/O .....</b>	<b>1-79</b>
File Name Construction .....	1-79
File Opening, Loading, and Saving .....	1-80
Memory Mapping .....	1-80
Low-Level File I/O .....	1-80
Text Files .....	1-81
XML Documents .....	1-82
Spreadsheets .....	1-82
Scientific Data .....	1-83
Audio and Audio/Video .....	1-86
Images .....	1-88
Internet Exchange .....	1-88
<b>Graphics .....</b>	<b>1-90</b>
Basic Plots and Graphs .....	1-90
Plotting Tools .....	1-91
Annotating Plots .....	1-91
Specialized Plotting .....	1-92
Bit-Mapped Images .....	1-96
Printing .....	1-96
Handle Graphics .....	1-97
<b>3-D Visualization .....</b>	<b>1-101</b>
Surface and Mesh Plots .....	1-101
View Control .....	1-103
Lighting .....	1-105
Transparency .....	1-105
Volume Visualization .....	1-106

<b>GUI Development</b> .....	<b>1-108</b>
Predefined Dialog Boxes .....	<b>1-108</b>
User Interface Deployment .....	<b>1-109</b>
User Interface Development .....	<b>1-109</b>
User Interface Objects .....	<b>1-110</b>
Objects from Callbacks .....	<b>1-111</b>
GUI Utilities .....	<b>1-111</b>
Program Execution .....	<b>1-112</b>
<b>External Interfaces</b> .....	<b>1-113</b>
Dynamic Link Libraries .....	<b>1-113</b>
Java .....	<b>1-114</b>
Component Object Model and ActiveX .....	<b>1-115</b>
Web Services .....	<b>1-117</b>
Serial Port Devices .....	<b>1-118</b>

## Functions — Alphabetical List

**2**

**Index**





# Function Reference

---

Desktop Tools and Development Environment (p. 1-3)

Startup, Command Window, help, editing and debugging, tuning, other general functions

Mathematics (p. 1-13)

Arrays and matrices, linear algebra, other areas of mathematics

Data Analysis (p. 1-41)

Basic data operations, descriptive statistics, covariance and correlation, filtering and convolution, numerical derivatives and integrals, Fourier transforms, time series analysis

Programming and Data Types (p. 1-49)

Function/expression evaluation, program control, function handles, object oriented programming, error handling, operators, data types, dates and times, timers

Object-Oriented Programming (p. 1-75)

Functions for working with classes and objects

File I/O (p. 1-79)

General and low-level file I/O, plus specific file formats, like audio, spreadsheet, HDF, images

Graphics (p. 1-90)

Line plots, annotating graphs, specialized plots, images, printing, Handle Graphics

3-D Visualization (p. 1-101)

Surface and mesh plots, view control, lighting and transparency, volume visualization

GUI Development (p. 1-108)

GUIDE, programming graphical user interfaces

External Interfaces (p. 1-113)

Interfaces to DLLs, Java, 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

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

<code>startup</code>	Startup M-file for user-defined options
<code>userpath</code>	View or change user portion of search path

## **Command Window and History**

<code>clc</code>	Clear Command Window
<code>commandhistory</code>	Open Command History window, or select it if already open
<code>commandwindow</code>	Open Command Window, or select it if already open
<code>diary</code>	Save session to file
<code>dos</code>	Execute DOS command and return result
<code>format</code>	Set display format for output
<code>home</code>	Move cursor to upper-left corner of Command Window
<code>matlabcolon (matlab:)</code>	Run specified function via hyperlink
<code>more</code>	Control paged output for Command Window
<code>perl</code>	Call Perl script using appropriate operating system executable
<code>system</code>	Execute operating system command and return result
<code>unix</code>	Execute UNIX command and return result

## Help for Using MATLAB

<code>builddocsearchdb</code>	Build searchable documentation database
<code>demo</code>	Access product demos via Help browser
<code>doc</code>	Reference page in Help browser
<code>docopt</code>	Web browser for UNIX platforms
<code>docsearch</code>	Open Help browser and search for specified term
<code>echodemo</code>	Run M-file demo step-by-step in Command Window
<code>help</code>	Help for functions in Command Window
<code>helpbrowser</code>	Open Help browser to access all online documentation and demos
<code>helpwin</code>	Provide access to M-file help for all functions
<code>info</code>	Information about contacting The MathWorks
<code>lookfor</code>	Search for keyword in all help entries
<code>playshow</code>	Run M-file demo (deprecated; use <code>echodemo</code> instead)
<code>support</code>	Open MathWorks Technical Support Web page
<code>web</code>	Open Web site or file in Web browser or Help browser
<code>whatsnew</code>	Release Notes for MathWorks™ products

## **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 tool for graphical editing

`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

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

## File Operations

See also “File I/O” on page 1-79 functions.

<code>cd</code>	Change working directory
<code>copyfile</code>	Copy file or directory
<code>delete</code>	Remove files or graphics objects
<code>dir</code>	Directory listing
<code>exist</code>	Check existence of variable, function, directory, or Java programming language class
<code>fileattrib</code>	Set or get attributes of file or directory
<code>filebrowser</code>	Current Directory browser
<code>isdir</code>	Determine whether input is a directory
<code>lookfor</code>	Search for keyword in all help entries

<code>ls</code>	Directory contents
<code>matlabroot</code>	Root directory
<code>mkdir</code>	Make new directory
<code>movefile</code>	Move file or directory
<code>pwd</code>	Identify current directory
<code>recycle</code>	Set option to move deleted files to recycle folder
<code>rehash</code>	Refresh function and file system path caches
<code>rmdir</code>	Remove directory
<code>toolboxdir</code>	Root directory for specified toolbox
<code>type</code>	Display contents of file
<code>visdiff</code>	Compare two text files, MAT-Files, or binary files
<code>what</code>	List MATLAB files in current directory
<code>which</code>	Locate functions and files

## **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	Change local workspace context when in debug mode
dbquit	Quit debug mode
dbstack	Function call stack
dbstatus	List all breakpoints
dbstep	Execute one or more lines from current breakpoint
dbstop	Set breakpoints
dbtype	List M-file with line numbers
dbup	Change local workspace context
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 <code>mlint</code> for file or directory, reporting results in browser
pack	Consolidate workspace memory
profile	Profile execution time for function

<code>profsave</code>	Save profile report in HTML format
<code>rehash</code>	Refresh function and file system path caches
<code>sparse</code>	Create sparse matrix
<code>zeros</code>	Create array of all zeros

### **Source Control**

<code>checkin</code>	Check files into a source control system (UNIX platforms)
<code>checkout</code>	Check files out of a source control system (UNIX platforms)
<code>cmopts</code>	Name of source control system
<code>customverctrl</code>	Allow custom source control system (UNIX platforms)
<code>undocheckout</code>	Undo previous checkout from source control system (UNIX platforms)
<code>verctrl</code>	Source control actions (Windows platforms)

### **Publishing**

<code>grabcode</code>	MATLAB code from M-files published to HTML
<code>notebook</code>	Open M-book in Microsoft® Word software (on Microsoft Windows platforms)
<code>publish</code>	Publish M-file containing cells, saving output to a file of specified type
<code>snapnow</code>	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	Controls 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**

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

# Mathematics

Arrays and Matrices (p. 1-14)	Basic array operators and operations, creation of elementary and specialized arrays and matrices
Linear Algebra (p. 1-20)	Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization
Elementary Math (p. 1-24)	Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math
Polynomials (p. 1-28)	Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion
Interpolation and Computational Geometry (p. 1-29)	Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation
Cartesian Coordinate System Conversion (p. 1-31)	Conversions between Cartesian and polar or spherical coordinates
Nonlinear Numerical Methods (p. 1-32)	Differential equations, optimization, integration
Specialized Math (p. 1-35)	Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions
Sparse Matrices (p. 1-36)	Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations
Math Constants (p. 1-40)	Pi, imaginary unit, infinity, Not-a-Number, largest and smallest positive floating point numbers, floating point relative accuracy

## Arrays and Matrices

Basic Information (p. 1-14)

Display array contents, get array information, determine array type

Operators (p. 1-15)

Arithmetic operators

Elementary Matrices and Arrays (p. 1-16)

Create elementary arrays of different types, generate arrays for plotting, array indexing, etc.

Array Operations (p. 1-17)

Operate on array content, apply function to each array element, find cumulative product or sum, etc.

Array Manipulation (p. 1-18)

Create, sort, rotate, permute, reshape, and shift array contents

Specialized Matrices (p. 1-19)

Create Hadamard, Companion, Hankel, Vandermonde, Pascal matrices, etc.

## Basic Information

`disp`

Display text or array

`display`

Display text or array (overloaded method)

`isempty`

Determine whether array is empty

`isequal`

Test arrays for equality

`isequalwithequalnans`

Test arrays for equality, treating NaNs as equal

`isfinite`

Array elements that are finite

`isfloat`

Determine whether input is floating-point array

`isinf`

Array elements that are infinite

`isinteger`

Determine whether input is integer array

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

## Operators

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

<code>.^</code>	Array power (element-wise)
<code>.\</code>	Left array divide (element-wise)
<code>./</code>	Right array divide (element-wise)

## **Elementary Matrices and Arrays**

<code>blkdiag</code>	Construct block diagonal matrix from input arguments
<code>create (RandStream)</code>	Create random number streams
<code>diag</code>	Diagonal matrices and diagonals of matrix
<code>eye</code>	Identity matrix
<code>freqspace</code>	Frequency spacing for frequency response
<code>get (RandStream)</code>	Random stream properties
<code>getDefaultStream (RandStream)</code>	Default random number stream
<code>ind2sub</code>	Subscripts from linear index
<code>linspace</code>	Generate linearly spaced vectors
<code>list (RandStream)</code>	Random number generator algorithms
<code>logspace</code>	Generate logarithmically spaced vectors
<code>meshgrid</code>	Generate X and Y arrays for 3-D plots
<code>ndgrid</code>	Generate arrays for N-D functions and interpolation
<code>ones</code>	Create array of all ones
<code>rand</code>	Uniformly distributed pseudorandom numbers
<code>rand (RandStream)</code>	Uniformly distributed random numbers



<code>randi</code>	Uniformly distributed pseudorandom integers
<code>randi (RandStream)</code>	Uniformly distributed pseudorandom integers
<code>randn</code>	Normally distributed pseudorandom numbers
<code>randn (RandStream)</code>	Normally distributed pseudorandom numbers
<code>randperm (RandStream)</code>	
<code>RandStream</code>	Random number stream
<code>RandStream (RandStream)</code>	Random number stream
<code>set (RandStream)</code>	Set a random stream property
<code>setDefaultStream (RandStream)</code>	Set the default random number stream
<code>sub2ind</code>	Single index from subscripts
<code>zeros</code>	Create array of all zeros

## Array Operations

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

<code>accumarray</code>	Construct array with accumulation
<code>arrayfun</code>	Apply function to each element of array
<code>bsxfun</code>	Apply element-by-element binary operation to two arrays with singleton expansion enabled
<code>cast</code>	Cast variable to different data type
<code>cross</code>	Vector cross product
<code>cumprod</code>	Cumulative product

<code>cumsum</code>	Cumulative sum
<code>dot</code>	Vector dot product
<code>idivide</code>	Integer division with rounding option
<code>kron</code>	Kronecker tensor product
<code>prod</code>	Product of array elements
<code>sum</code>	Sum of array elements
<code>tril</code>	Lower triangular part of matrix
<code>triu</code>	Upper triangular part of matrix

### **Array Manipulation**

<code>blkdiag</code>	Construct block diagonal matrix from input arguments
<code>cat</code>	Concatenate arrays along specified dimension
<code>circshift</code>	Shift array circularly
<code>diag</code>	Diagonal matrices and diagonals of matrix
<code>end</code>	Terminate block of code, or indicate last array index
<code>flipdim</code>	Flip array along specified dimension
<code>fliplr</code>	Flip matrix left to right
<code>flipud</code>	Flip matrix up to down
<code>horzcat</code>	Concatenate arrays horizontally
<code>inline</code>	Construct inline object
<code>ipermute</code>	Inverse permute dimensions of N-D array
<code>permute</code>	Rearrange dimensions of N-D array
<code>repmat</code>	Replicate and tile array

reshape	Reshape array
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
vectorize	Vectorize expression
vertcat	Concatenate arrays vertically

### **Specialized Matrices**

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

## Linear Algebra

Matrix Analysis (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

<code>cond</code>	Condition number with respect to inversion
<code>condeig</code>	Condition number with respect to eigenvalues
<code>det</code>	Matrix determinant
<code>norm</code>	Vector and matrix norms
<code>normest</code>	2-norm estimate
<code>null</code>	Null space
<code>orth</code>	Range space of matrix
<code>rank</code>	Rank of matrix
<code>rcond</code>	Matrix reciprocal condition number estimate
<code>rref</code>	Reduced row echelon form

---

subspace	Angle between two subspaces
trace	Sum of diagonal elements

## Linear Equations

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

## Eigenvalues and Singular Values

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

## Matrix Logarithms and Exponentials

<code>expm</code>	Matrix exponential
<code>logm</code>	Matrix logarithm
<code>sqrtm</code>	Matrix square root

## Factorization

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

qz	QZ factorization for generalized eigenvalues
rsf2csf	Convert real Schur form to complex Schur form
svd	Singular value decomposition

## Elementary Math

Trigonometric (p. 1-24)	Trigonometric functions with results in radians or degrees
Exponential (p. 1-26)	Exponential, logarithm, power, and root functions
Complex (p. 1-26)	Numbers with real and imaginary components, phase angles
Rounding and Remainder (p. 1-27)	Rounding, modulus, and remainder
Discrete Math (p. 1-27)	Prime factors, factorials, permutations, rational fractions, least common multiple, greatest common divisor

## Trigonometric

acos	Inverse cosine; result in radians
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees
acsch	Inverse hyperbolic cosecant



asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine; result in radians
asind	Inverse sine; result in degrees
asinh	Inverse hyperbolic sine
atan	Inverse tangent; result in radians
atan2	Four-quadrant inverse tangent
atand	Inverse tangent; result in degrees
atanh	Inverse hyperbolic tangent
cos	Cosine of argument in radians
cosd	Cosine of argument in degrees
cosh	Hyperbolic cosine
cot	Cotangent of argument in radians
cotd	Cotangent of argument in degrees
coth	Hyperbolic cotangent
csc	Cosecant of argument in radians
cscd	Cosecant of argument in degrees
csch	Hyperbolic cosecant
hypot	Square root of sum of squares
sec	Secant of argument in radians
secd	Secant of argument in degrees
sech	Hyperbolic secant
sin	Sine of argument in radians
sind	Sine of argument in degrees
sinh	Hyperbolic sine of argument in radians
tan	Tangent of argument in radians

<code>tand</code>	Tangent of argument in degrees
<code>tanh</code>	Hyperbolic tangent

## **Exponential**

<code>exp</code>	Exponential
<code>expm1</code>	Compute $\exp(x) - 1$ accurately for small values of $x$
<code>log</code>	Natural logarithm
<code>log10</code>	Common (base 10) logarithm
<code>log1p</code>	Compute $\log(1+x)$ accurately for small values of $x$
<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
<code>nextpow2</code>	Next higher power of 2
<code>nthroot</code>	Real $n$ th root of real numbers
<code>pow2</code>	Base 2 power and scale floating-point numbers
<code>reallog</code>	Natural logarithm for nonnegative real arrays
<code>realpow</code>	Array power for real-only output
<code>realsqrt</code>	Square root for nonnegative real arrays
<code>sqrt</code>	Square root

## **Complex**

<code>abs</code>	Absolute value and complex magnitude
<code>angle</code>	Phase angle

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

## **Rounding and Remainder**

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

## **Discrete Math**

<code>factor</code>	Prime factors
<code>factorial</code>	Factorial function
<code>gcd</code>	Greatest common divisor

<code>isprime</code>	Array elements that are prime numbers
<code>lcm</code>	Least common multiple
<code>nchoosek</code>	Binomial coefficient or all combinations
<code>perms</code>	All possible permutations
<code>primes</code>	Generate list of prime numbers
<code>rat, rats</code>	Rational fraction approximation

## **Polynomials**

<code>conv</code>	Convolution and polynomial multiplication
<code>deconv</code>	Deconvolution and polynomial division
<code>poly</code>	Polynomial with specified roots
<code>polyder</code>	Polynomial derivative
<code>polyeig</code>	Polynomial eigenvalue problem
<code>polyfit</code>	Polynomial curve fitting
<code>polyint</code>	Integrate polynomial analytically
<code>polyval</code>	Polynomial evaluation
<code>polyvalm</code>	Matrix polynomial evaluation
<code>residue</code>	Convert between partial fraction expansion and polynomial coefficients
<code>roots</code>	Polynomial roots

## Interpolation and Computational Geometry

Interpolation (p. 1-29)	Data interpolation, data gridding, polynomial evaluation, nearest point search
Delaunay Triangulation and Tessellation (p. 1-30)	Delaunay triangulation and tessellation, triangular surface and mesh plots
Convex Hull (p. 1-31)	Plot convex hull, plotting functions
Voronoi Diagrams (p. 1-31)	Plot Voronoi diagram, patch graphics object, plotting functions
Domain Generation (p. 1-31)	Generate arrays for 3-D plots, or for N-D functions and interpolation

### Interpolation

dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
griddata	Data gridding
griddata3	Data gridding and hypersurface fitting for 3-D data
griddatan	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

mkpp	Make piecewise polynomial
ndgrid	Generate arrays for N-D functions and interpolation
padecoef	Padé approximation of time delays
pchip	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)
ppval	Evaluate piecewise polynomial
spline	Cubic spline data interpolation
tsearchn	N-D closest simplex search
unmkpp	Piecewise polynomial details

### **Delaunay Triangulation and Tessellation**

delaunay	Delaunay triangulation
delaunay3	3-D Delaunay tessellation
delaunayn	N-D Delaunay tessellation
dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
tetramesh	Tetrahedron mesh plot
trimesh	Triangular mesh plot
triplot	2-D triangular plot
trisurf	Triangular surface plot
tsearch	Search for enclosing Delaunay triangle
tsearchn	N-D closest simplex search

## Convex Hull

<code>convhull</code>	Convex hull
<code>convhulln</code>	N-D convex hull
<code>patch</code>	Create patch graphics object
<code>plot</code>	2-D line plot
<code>trisurf</code>	Triangular surface plot

## Voronoi Diagrams

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

## Domain Generation

<code>meshgrid</code>	Generate X and Y arrays for 3-D plots
<code>ndgrid</code>	Generate arrays for N-D functions and interpolation

## Cartesian Coordinate System Conversion

<code>cart2pol</code>	Transform Cartesian coordinates to polar or cylindrical
<code>cart2sph</code>	Transform Cartesian coordinates to spherical

<code>pol2cart</code>	Transform polar or cylindrical coordinates to Cartesian
<code>sph2cart</code>	Transform spherical coordinates to Cartesian

## **Nonlinear Numerical Methods**

Ordinary Differential Equations (p. 1-32)	Solve stiff and nonstiff differential equations, define the problem, set solver options, evaluate solution
Delay Differential Equations (p. 1-33)	Solve delay differential equations with constant and general delays, set solver options, evaluate solution
Boundary Value Problems (p. 1-34)	Solve boundary value problems for ordinary differential equations, set solver options, evaluate solution
Partial Differential Equations (p. 1-34)	Solve initial-boundary value problems for parabolic-elliptic PDEs, evaluate solution
Optimization (p. 1-34)	Find minimum of single and multivariable functions, solve nonnegative least-squares constraint problem
Numerical Integration (Quadrature) (p. 1-35)	Evaluate Simpson, Lobatto, and vectorized quadratures, evaluate double and triple integrals

## **Ordinary Differential Equations**

<code>decic</code>	Compute consistent initial conditions for <code>ode15i</code>
<code>deval</code>	Evaluate solution of differential equation problem



<code>ode15i</code>	Solve fully implicit differential equations, variable order method
<code>ode23</code> , <code>ode45</code> , <code>ode113</code> , <code>ode15s</code> , <code>ode23s</code> , <code>ode23t</code> , <code>ode23tb</code>	Solve initial value problems for ordinary differential equations
<code>odefile</code>	Define differential equation problem for ordinary differential equation solvers
<code>odeget</code>	Ordinary differential equation options parameters
<code>odeset</code>	Create or alter options structure for ordinary differential equation solvers
<code>odextend</code>	Extend solution of initial value problem for ordinary differential equation

## **Delay Differential Equations**

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

## **Boundary Value Problems**

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

## **Partial Differential Equations**

<code>pdepe</code>	Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D
<code>pdeval</code>	Evaluate numerical solution of PDE using output of <code>pdepe</code>

## **Optimization**

<code>fminbnd</code>	Find minimum of single-variable function on fixed interval
<code>fminsearch</code>	Find minimum of unconstrained multivariable function using derivative-free method
<code>fzero</code>	Find root of continuous function of one variable

lsqnonneg	Solve nonnegative least-squares constraints problem
optimget	Optimization options values
optimset	Create or edit optimization options structure

## Numerical Integration (Quadrature)

dblquad	Numerically evaluate double integral
quad	Numerically evaluate integral, adaptive Simpson quadrature
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
beta	Beta function
betainc	Incomplete beta function

betaln	Logarithm of beta function
ellipj	Jacobi elliptic functions
ellipke	Complete elliptic integrals of first and second kind
erf, erfc, erfcx, erfinv, erfcinv	Error functions
expint	Exponential integral
gamma, gammainc, gammaln	Gamma functions
legendre	Associated Legendre functions
psi	Psi (polygamma) function

## **Sparse Matrices**

Elementary Sparse Matrices (p. 1-37)	Create random and nonrandom sparse matrices
Full to Sparse Conversion (p. 1-37)	Convert full matrix to sparse, sparse matrix to full
Sparse Matrix Manipulation (p. 1-37)	Test matrix for sparseness, get information on sparse matrix, allocate sparse matrix, apply function to nonzero elements, visualize sparsity pattern
Reordering Algorithms (p. 1-38)	Random, column, minimum degree, Dulmage-Mendelsohn, and reverse Cuthill-McKee permutations
Linear Algebra (p. 1-39)	Compute norms, eigenvalues, factorizations, least squares, structural rank

Linear Equations (Iterative  
Methods) (p. 1-39)

Methods for conjugate and  
biconjugate gradients, residuals,  
lower quartile

Tree Operations (p. 1-40)

Elimination trees, tree plotting,  
factorization analysis

## Elementary Sparse Matrices

spdiags

Extract and create sparse band and  
diagonal matrices

speye

Sparse identity matrix

sprand

Sparse uniformly distributed  
random matrix

sprandn

Sparse normally distributed random  
matrix

sprandsym

Sparse symmetric random matrix

## Full to Sparse Conversion

find

Find indices and values of nonzero  
elements

full

Convert sparse matrix to full matrix

sparse

Create sparse matrix

sconvert

Import matrix from sparse matrix  
external format

## Sparse Matrix Manipulation

issparse

Determine whether input is sparse

nnz

Number of nonzero matrix elements

nonzeros

Nonzero matrix elements

<code>nzmax</code>	Amount of storage allocated for nonzero matrix elements
<code>spalloc</code>	Allocate space for sparse matrix
<code>spfun</code>	Apply function to nonzero sparse matrix elements
<code>spones</code>	Replace nonzero sparse matrix elements with ones
<code>spparms</code>	Set parameters for sparse matrix routines
<code>spy</code>	Visualize sparsity pattern

### **Reordering Algorithms**

<code>amd</code>	Approximate minimum degree permutation
<code>colamd</code>	Column approximate minimum degree permutation
<code>colperm</code>	Sparse column permutation based on nonzero count
<code>dmperm</code>	Dulmage-Mendelsohn decomposition
<code>ldl</code>	Block LDL' factorization for Hermitian indefinite matrices
<code>randperm</code>	Random permutation
<code>symamd</code>	Symmetric approximate minimum degree permutation
<code>symrcm</code>	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 a matrix
ilu	Sparse incomplete LU factorization
luinc	Sparse incomplete LU factorization
normest	2-norm estimate
spaugment	Form least squares augmented system
sprank	Structural rank
svds	Find singular values and vectors

## Linear Equations (Iterative Methods)

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

## Tree Operations

<code>etree</code>	Elimination tree
<code>etreeplot</code>	Plot elimination tree
<code>gplot</code>	Plot nodes and links representing adjacency matrix
<code>sybmfact</code>	Symbolic factorization analysis
<code>treelayout</code>	Lay out tree or forest
<code>treeplot</code>	Plot picture of tree

## Math Constants

<code>eps</code>	Floating-point relative accuracy
<code>i</code>	Imaginary unit
<code>Inf</code>	Infinity
<code>intmax</code>	Largest value of specified integer type
<code>intmin</code>	Smallest value of specified integer type
<code>j</code>	Imaginary unit
<code>NaN</code>	Not-a-Number
<code>pi</code>	Ratio of circle's circumference to its diameter, $\pi$
<code>realmax</code>	Largest positive floating-point number
<code>realmin</code>	Smallest positive normalized floating-point number



## Data Analysis

Basic Operations (p. 1-41)	Sums, products, sorting
Descriptive Statistics (p. 1-41)	Statistical summaries of data
Filtering and Convolution (p. 1-42)	Data preprocessing
Interpolation and Regression (p. 1-42)	Data fitting
Fourier Transforms (p. 1-43)	Frequency content of data
Derivatives and Integrals (p. 1-43)	Data rates and accumulations
Time Series Objects (p. 1-44)	Methods for <code>timeseries</code> objects
Time Series Collections (p. 1-47)	Methods for <code>tscollection</code> objects

### Basic Operations

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

### Descriptive Statistics

<code>corrcoef</code>	Correlation coefficients
<code>cov</code>	Covariance matrix

<code>max</code>	Largest elements in array
<code>mean</code>	Average or mean value of array
<code>median</code>	Median value of array
<code>min</code>	Smallest elements in array
<code>mode</code>	Most frequent values in array
<code>std</code>	Standard deviation
<code>var</code>	Variance

## **Filtering and Convolution**

<code>conv</code>	Convolution and polynomial multiplication
<code>conv2</code>	2-D convolution
<code>convn</code>	N-D convolution
<code>deconv</code>	Deconvolution and polynomial division
<code>detrend</code>	Remove linear trends
<code>filter</code>	1-D digital filter
<code>filter2</code>	2-D digital filter

## **Interpolation and Regression**

<code>interp1</code>	1-D data interpolation (table lookup)
<code>interp2</code>	2-D data interpolation (table lookup)
<code>interp3</code>	3-D data interpolation (table lookup)
<code>interpn</code>	N-D data interpolation (table lookup)
<code>mldivide \, mrdivide /</code>	Left or right matrix division
<code>polyfit</code>	Polynomial curve fitting
<code>polyval</code>	Polynomial evaluation

## Fourier Transforms

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

## Derivatives and Integrals

<code>cumtrapz</code>	Cumulative trapezoidal numerical integration
<code>del2</code>	Discrete Laplacian
<code>diff</code>	Differences and approximate derivatives

<code>gradient</code>	Numerical gradient
<code>polyder</code>	Polynomial derivative
<code>polyint</code>	Integrate polynomial analytically
<code>trapez</code>	Trapezoidal numerical integration

## **Time Series Objects**

Utilities (p. 1-44)	Combine <code>timeseries</code> objects, query and set <code>timeseries</code> object properties, plot <code>timeseries</code> objects
Data Manipulation (p. 1-45)	Add or delete data, manipulate <code>timeseries</code> objects
Event Data (p. 1-46)	Add or delete events, create new <code>timeseries</code> objects based on event data
Descriptive Statistics (p. 1-46)	Descriptive statistics for <code>timeseries</code> objects

## **Utilities**

<code>get (timeseries)</code>	Query <code>timeseries</code> object property values
<code>getdatasamplesize</code>	Size of data sample in <code>timeseries</code> object
<code>getqualitydesc</code>	Data quality descriptions
<code>isempty (timeseries)</code>	Determine whether <code>timeseries</code> object is empty
<code>length (timeseries)</code>	Length of time vector
<code>plot (timeseries)</code>	Plot time series
<code>set (timeseries)</code>	Set properties of <code>timeseries</code> object
<code>size (timeseries)</code>	Size of <code>timeseries</code> object

---

<code>timeseries</code>	Create <code>timeseries</code> object
<code>tsdata.event</code>	Construct event object for <code>timeseries</code> object
<code>tsprops</code>	Help on <code>timeseries</code> object properties
<code>tstool</code>	Open Time Series Tools GUI

## Data Manipulation

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

<code>synchronize</code>	Synchronize and resample two <code>timeseries</code> objects using common time vector
<code>transpose (timeseries)</code>	Transpose <code>timeseries</code> object
<code>vertcat (timeseries)</code>	Vertical concatenation of <code>timeseries</code> objects

### **Event Data**

<code>addevent</code>	Add event to <code>timeseries</code> object
<code>delevent</code>	Remove <code>tsdata.event</code> objects from <code>timeseries</code> object
<code>gettsafteratevent</code>	New <code>timeseries</code> object with samples occurring at or after event
<code>gettsafterevent</code>	New <code>timeseries</code> object with samples occurring after event
<code>gettsatevent</code>	New <code>timeseries</code> object with samples occurring at event
<code>gettsbeforeatevent</code>	New <code>timeseries</code> object with samples occurring before or at event
<code>gettsbeforeevent</code>	New <code>timeseries</code> object with samples occurring before event
<code>gettsbetweenevents</code>	New <code>timeseries</code> object with samples occurring between events

### **Descriptive Statistics**

<code>iqr (timeseries)</code>	Interquartile range of <code>timeseries</code> data
<code>max (timeseries)</code>	Maximum value of <code>timeseries</code> data
<code>mean (timeseries)</code>	Mean value of <code>timeseries</code> data
<code>median (timeseries)</code>	Median value of <code>timeseries</code> data

<code>min (timeseries)</code>	Minimum value of timeseries data
<code>std (timeseries)</code>	Standard deviation of timeseries data
<code>sum (timeseries)</code>	Sum of timeseries data
<code>var (timeseries)</code>	Variance of timeseries data

## Time Series Collections

Utilities (p. 1-47)	Query and set <code>tscollection</code> object properties, plot <code>tscollection</code> objects
Data Manipulation (p. 1-48)	Add or delete data, manipulate <code>tscollection</code> objects

## Utilities

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

## Data Manipulation

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



## Programming and Data Types

Data Types (p. 1-49)	Numeric, character, structures, cell arrays, and data type conversion
Data Type Conversion (p. 1-57)	Convert one numeric type to another, numeric to string, string to numeric, structure to cell array, etc.
Operators and Special Characters (p. 1-59)	Arithmetic, relational, and logical operators, and special characters
Strings (p. 1-61)	Create, identify, manipulate, parse, evaluate, and compare strings
Bit-Wise Operations (p. 1-64)	Perform set, shift, and, or, compare, etc. on specific bit fields
Logical Operations (p. 1-65)	Evaluate conditions, testing for true or false
Relational Operations (p. 1-66)	Compare values for equality, greater than, less than, etc.
Set Operations (p. 1-66)	Find set members, unions, intersections, etc.
Date and Time Operations (p. 1-66)	Obtain information about dates and times
Programming in MATLAB (p. 1-67)	M-files, function/expression evaluation, program control, function handles, object oriented programming, error handling

### Data Types

Numeric Types (p. 1-50)	Integer and floating-point data
Characters and Strings (p. 1-51)	Characters and arrays of characters
Structures (p. 1-52)	Data of varying types and sizes stored in fields of a structure

Cell Arrays (p. 1-53)	Data of varying types and sizes stored in cells of array
Function Handles (p. 1-54)	Invoke a function indirectly via handle
Java Classes and Objects (p. 1-54)	Access Java classes through MATLAB interface
Data Type Identification (p. 1-56)	Determine data type of a variable

## **Numeric Types**

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

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

## Characters and Strings

See “Strings” on page 1-61 for all string-related functions.

<code>cellstr</code>	Create cell array of strings from character array
<code>char</code>	Convert to character array (string)
<code>eval</code>	Execute string containing MATLAB expression
<code>findstr</code>	Find string within another, longer string
<code>isstr</code>	Determine whether input is character array
<code>regexp, regexpi</code>	Match regular expression
<code>sprintf</code>	Write formatted data to string

<code>sscanf</code>	Read formatted data from string
<code>strcat</code>	Concatenate strings horizontally
<code>strcmp, strcmpi</code>	Compare strings
<code>strings</code>	String handling
<code>strjust</code>	Justify character array
<code>strmatch</code>	Find possible matches for string
<code>strread</code>	Read formatted data from string
<code>strrep</code>	Find and replace substring
<code>strtrim</code>	Remove leading and trailing white space from string
<code>strvcat</code>	Concatenate strings vertically

## **Structures**

<code>arrayfun</code>	Apply function to each element of array
<code>cell2struct</code>	Convert cell array to structure array
<code>class</code>	Create object or return class of object
<code>deal</code>	Distribute inputs to outputs
<code>fieldnames</code>	Field names of structure, or public fields of object
<code>getfield</code>	Field of structure array
<code>isa</code>	Determine whether input is object of given class
<code>isequal</code>	Test arrays for equality
<code>isfield</code>	Determine whether input is structure array field
<code>isscalar</code>	Determine whether input is scalar
<code>isstruct</code>	Determine whether input is structure array

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

## Cell Arrays

<code>cell</code>	Construct cell array
<code>cell2mat</code>	Convert cell array of matrices to single matrix
<code>cell2struct</code>	Convert cell array to structure array
<code>celldisp</code>	Cell array contents
<code>cellfun</code>	Apply function to each cell in cell array
<code>cellplot</code>	Graphically display structure of cell array
<code>cellstr</code>	Create cell array of strings from character array
<code>class</code>	Create object or return class of object
<code>deal</code>	Distribute inputs to outputs
<code>isa</code>	Determine whether input is object of given class
<code>iscell</code>	Determine whether input is cell array
<code>iscellstr</code>	Determine whether input is cell array of strings

<code>isequal</code>	Test arrays for equality
<code>isscalar</code>	Determine whether input is scalar
<code>isvector</code>	Determine whether input is vector
<code>mat2cell</code>	Divide matrix into cell array of matrices
<code>num2cell</code>	Convert numeric array to cell array
<code>struct2cell</code>	Convert structure to cell array

### **Function Handles**

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

### **Java Classes and Objects**

<code>cell</code>	Construct cell array
<code>class</code>	Create object or return class of object
<code>clear</code>	Remove items from workspace, freeing up system memory
<code>depfun</code>	List dependencies of M-file or P-file

<code>exist</code>	Check existence of variable, function, directory, or Java programming language class
<code>fieldnames</code>	Field names of structure, or public fields of object
<code>im2java</code>	Convert image to Java image
<code>import</code>	Add package or class to current import list
<code>inmem</code>	Names of M-files, MEX-files, Sun Java classes in memory
<code>isa</code>	Determine whether input is object of given class
<code>isjava</code>	Determine whether input is Sun Java object
<code>javaaddpath</code>	Add entries to dynamic Sun Java class path
<code>javaArray</code>	Construct Sun Java array
<code>javachk</code>	Generate error message based on Sun Java feature support
<code>javaclasspath</code>	Set and get dynamic Sun Java class path
<code>javaMethod</code>	Invoke Sun Java method
<code>javaObject</code>	Construct Sun Java object
<code>javarmpath</code>	Remove entries from dynamic Sun Java class path
<code>methods</code>	Information on class methods
<code>methodsview</code>	Information on class methods in separate window
<code>usejava</code>	Determine whether Sun Java feature is supported in MATLAB software
<code>which</code>	Locate functions and files

## Data Type Identification

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



## Data Type Conversion

Numeric (p. 1-57)	Convert data of one numeric type to another numeric type
String to Numeric (p. 1-57)	Convert characters to numeric equivalent
Numeric to String (p. 1-58)	Convert numeric to character equivalent
Other Conversions (p. 1-58)	Convert to structure, cell array, function handle, etc.

## Numeric

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

## String to Numeric

<code>base2dec</code>	Convert base N number string to decimal number
<code>bin2dec</code>	Convert binary number string to decimal number
<code>cast</code>	Cast variable to different data type
<code>hex2dec</code>	Convert hexadecimal number string to decimal number
<code>hex2num</code>	Convert hexadecimal number string to double-precision number

<code>str2double</code>	Convert string to double-precision value
<code>str2num</code>	Convert string to number
<code>unicode2native</code>	Convert Unicode® characters to numeric bytes

### **Numeric to String**

<code>cast</code>	Cast variable to different data type
<code>char</code>	Convert to character array (string)
<code>dec2base</code>	Convert decimal to base N number in string
<code>dec2bin</code>	Convert decimal to binary number in string
<code>dec2hex</code>	Convert decimal to hexadecimal number in string
<code>int2str</code>	Convert integer to string
<code>mat2str</code>	Convert matrix to string
<code>native2unicode</code>	Convert numeric bytes to Unicode characters
<code>num2str</code>	Convert number to string

### **Other Conversions**

<code>cell2mat</code>	Convert cell array of matrices to single matrix
<code>cell2struct</code>	Convert cell array to structure array
<code>datestr</code>	Convert date and time to string format
<code>func2str</code>	Construct function name string from function handle

<code>logical</code>	Convert numeric values to logical
<code>mat2cell</code>	Divide matrix into cell array of matrices
<code>num2cell</code>	Convert numeric array to cell array
<code>num2hex</code>	Convert singles and doubles to IEEE <sup>®</sup> hexadecimal strings
<code>str2func</code>	Construct function handle from function name string
<code>str2mat</code>	Form blank-padded character matrix from strings
<code>struct2cell</code>	Convert structure to cell array

## Operators and Special Characters

Arithmetic Operators (p. 1-59)	Plus, minus, power, left and right divide, transpose, etc.
Relational Operators (p. 1-60)	Equal to, greater than, less than or equal to, etc.
Logical Operators (p. 1-60)	Element-wise and short circuit and, or, not
Special Characters (p. 1-61)	Array constructors, line continuation, comments, etc.

### Arithmetic Operators

<code>+</code>	Plus
<code>-</code>	Minus
<code>.</code>	Decimal point
<code>=</code>	Assignment
<code>*</code>	Matrix multiplication
<code>/</code>	Matrix right division

<code>\</code>	Matrix left division
<code>^</code>	Matrix power
<code>'</code>	Matrix transpose
<code>.*</code>	Array multiplication (element-wise)
<code>./</code>	Array right division (element-wise)
<code>.\</code>	Array left division (element-wise)
<code>.^</code>	Array power (element-wise)
<code>.'</code>	Array transpose

### **Relational Operators**

<code>&lt;</code>	Less than
<code>&lt;=</code>	Less than or equal to
<code>&gt;</code>	Greater than
<code>&gt;=</code>	Greater than or equal to
<code>==</code>	Equal to
<code>~=</code>	Not equal to

### **Logical Operators**

See also for functions like `xor`, `all`, `any`, etc.

<code>&amp;&amp;</code>	Logical AND
<code>  </code>	Logical OR
<code>&amp;</code>	Logical AND for arrays
<code> </code>	Logical OR for arrays
<code>~</code>	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. 1-62)	Basics of string handling in MATLAB
String Creation (p. 1-62)	Create strings, cell arrays of strings, concatenate strings together
String Identification (p. 1-62)	Identify characteristics of strings

String Manipulation (p. 1-63)	Convert case, strip blanks, replace characters
String Parsing (p. 1-63)	Formatted read, regular expressions, locate substrings
String Evaluation (p. 1-64)	Evaluate stated expression in string
String Comparison (p. 1-64)	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**

class	Create object or return class of object
isa	Determine whether input is object of given class
iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array

<code>isletter</code>	Array elements that are alphabetic letters
<code>isscalar</code>	Determine whether input is scalar
<code>isspace</code>	Array elements that are space characters
<code>isstrprop</code>	Determine whether string is of specified category
<code>isvector</code>	Determine whether input is vector
<code>validatestring</code>	Check validity of text string

### **String Manipulation**

<code>deblank</code>	Strip trailing blanks from end of string
<code>lower</code>	Convert string to lowercase
<code>strjust</code>	Justify character array
<code>strrep</code>	Find and replace substring
<code>strtrim</code>	Remove leading and trailing white space from string
<code>upper</code>	Convert string to uppercase

### **String Parsing**

<code>findstr</code>	Find string within another, longer string
<code>regexp, regexpi</code>	Match regular expression
<code>regexprep</code>	Replace string using regular expression
<code>regexptranslate</code>	Translate string into regular expression
<code>sscanf</code>	Read formatted data from string

<code>strfind</code>	Find one string within another
<code>strread</code>	Read formatted data from string
<code>strtok</code>	Selected parts of string

### **String Evaluation**

<code>eval</code>	Execute string containing MATLAB expression
<code>evalc</code>	Evaluate MATLAB expression with capture
<code>evalin</code>	Execute MATLAB expression in specified workspace

### **String Comparison**

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

### **Bit-Wise Operations**

<code>bitand</code>	Bitwise AND
<code>bitcmp</code>	Bitwise complement
<code>bitget</code>	Bit at specified position
<code>bitmax</code>	Maximum double-precision floating-point integer
<code>bitor</code>	Bitwise OR
<code>bitset</code>	Set bit at specified position
<code>bitshift</code>	Shift bits specified number of places



<code>bitxor</code>	Bitwise XOR
<code>swapbytes</code>	Swap byte ordering

## Logical Operations

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

See “Operators and Special Characters” on page 1-59 for logical operators.

## Relational Operations

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

See “Operators and Special Characters” on page 1-59 for relational operators.

## Set Operations

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

## Date and Time Operations

<code>addtodate</code>	Modify date number by field
<code>calendar</code>	Calendar for specified month
<code>clock</code>	Current time as date vector
<code>cputime</code>	Elapsed CPU time
<code>date</code>	Current date string

<code>datenum</code>	Convert date and time to serial date number
<code>datestr</code>	Convert date and time to string format
<code>datevec</code>	Convert date and time to vector of components
<code>eomday</code>	Last day of month
<code>etime</code>	Time elapsed between date vectors
<code>now</code>	Current date and time
<code>weekday</code>	Day of week

## Programming in MATLAB

M-Files and Scripts (p. 1-68)	Declare functions, handle arguments, identify dependencies, etc.
Evaluation (p. 1-69)	Evaluate expression in string, apply function to array, run script file, etc.
Timer (p. 1-70)	Schedule execution of MATLAB commands
Variables and Functions in Memory (p. 1-70)	List files in memory, clear M-files in memory, assign to variable in nondefault workspace, refresh caches
Control Flow (p. 1-71)	if-then-else, for loops, switch-case, try-catch
Error Handling (p. 1-72)	Generate warnings and errors, test for and catch errors, retrieve most recent error message
MEX Programming (p. 1-73)	Compile MEX function from C or Fortran code, list MEX-files in memory, debug MEX-files

## M-Files and Scripts

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

---

<code>varargin</code>	Variable length input argument list
<code>varargout</code>	Variable length output argument list

## Evaluation

<code>ans</code>	Most recent answer
<code>arrayfun</code>	Apply function to each element of array
<code>assert</code>	Generate error when condition is violated
<code>builtin</code>	Execute built-in function from overloaded method
<code>cellfun</code>	Apply function to each cell in cell array
<code>echo</code>	Echo M-files during execution
<code>eval</code>	Execute string containing MATLAB expression
<code>evalc</code>	Evaluate MATLAB expression with capture
<code>evalin</code>	Execute MATLAB expression in specified workspace
<code>feval</code>	Evaluate function
<code>iskeyword</code>	Determine whether input is MATLAB keyword
<code>isvarname</code>	Determine whether input is valid variable name
<code>pause</code>	Halt execution temporarily
<code>run</code>	Run script that is not on current path
<code>script</code>	Script M-file description

<code>structfun</code>	Apply function to each field of scalar structure
<code>symvar</code>	Determine symbolic variables in expression
<code>tic, toc</code>	Measure performance using stopwatch timer

## **Timer**

<code>delete (timer)</code>	Remove timer object from memory
<code>disp (timer)</code>	Information about timer object
<code>get (timer)</code>	Timer object properties
<code>isvalid (timer)</code>	Determine whether timer object is valid
<code>set (timer)</code>	Configure or display timer object properties
<code>start</code>	Start timer(s) running
<code>startat</code>	Start timer(s) running at specified time
<code>stop</code>	Stop timer(s)
<code>timer</code>	Construct timer object
<code>timerfind</code>	Find timer objects
<code>timerfindall</code>	Find timer objects, including invisible objects
<code>wait</code>	Wait until timer stops running

## **Variables and Functions in Memory**

<code>ans</code>	Most recent answer
<code>assignin</code>	Assign value to variable in specified workspace

<code>datatipinfo</code>	Produce short description of input variable
<code>genvarname</code>	Construct valid variable name from string
<code>global</code>	Declare global variables
<code>inmem</code>	Names of M-files, MEX-files, Sun Java classes in memory
<code>isglobal</code>	Determine whether input is global variable
<code>memory</code>	Display memory information
<code>mislocked</code>	Determine whether M-file or MEX-file cannot be cleared from memory
<code>mlock</code>	Prevent clearing M-file or MEX-file from memory
<code>munlock</code>	Allow clearing M-file or MEX-file from memory
<code>namelengthmax</code>	Maximum identifier length
<code>pack</code>	Consolidate workspace memory
<code>persistent</code>	Define persistent variable
<code>rehash</code>	Refresh function and file system path caches

### **Control Flow**

<code>break</code>	Terminate execution of <code>for</code> or <code>while</code> loop
<code>case</code>	Execute block of code if condition is <code>true</code>
<code>catch</code>	Specify how to respond to error in <code>try</code> statement

<code>continue</code>	Pass control to next iteration of <code>for</code> or <code>while</code> loop
<code>else</code>	Execute statements if condition is false
<code>elseif</code>	Execute statements if additional condition is true
<code>end</code>	Terminate block of code, or indicate last array index
<code>error</code>	Display message and abort function
<code>for</code>	Execute block of code specified number of times
<code>if</code>	Execute statements if condition is true
<code>otherwise</code>	Default part of switch statement
<code>parfor</code>	Parallel for-loop
<code>return</code>	Return to invoking function
<code>switch</code>	Switch among several cases, based on expression
<code>try</code>	Attempt to execute block of code, and catch errors
<code>while</code>	Repeatedly execute statements while condition is true

### **Error Handling**

<code>addCause (MException)</code>	Append MException objects
<code>assert</code>	Generate error when condition is violated
<code>catch</code>	Specify how to respond to error in <code>try</code> statement
<code>disp (MException)</code>	Display MException object



<code>eq (MException)</code>	Compare MException objects for equality
<code>error</code>	Display message and abort function
<code>ferror</code>	Query the MATLAB software about errors in file input or output
<code>getReport (MException)</code>	Get error message for exception
<code>intwarning</code>	Control state of integer warnings
<code>isequal (MException)</code>	Compare MException objects for equality
<code>last (MException)</code>	Last uncaught exception
<code>lasterr</code>	Last error message
<code>lasterror</code>	Last error message and related information
<code>lastwarn</code>	Last warning message
<code>MException</code>	Construct MException object
<code>ne (MException)</code>	Compare MException objects for inequality
<code>rethrow</code>	Reissue error
<code>rethrow (MException)</code>	Reissue existing exception
<code>throw (MException)</code>	Terminate function and issue exception
<code>try</code>	Attempt to execute block of code, and catch errors
<code>warning</code>	Warning message

## **MEX Programming**

<code>dbmex</code>	Enable MEX-file debugging
<code>inmem</code>	Names of M-files, MEX-files, Sun Java classes in memory

<code>mex</code>	Compile MEX-function from C/ C++ or Fortran source code
<code>mex.getCompilerConfigurations</code>	Get compiler configuration information for building MEX-files
<code>mexext</code>	Binary MEX-file name extension

# Object-Oriented Programming

Classes and Objects (p. 1-75)	Get information about classes and objects
Handle Classes (p. 1-76)	Define and use handle classes
Events and Listeners (p. 1-77)	Define and use events and listeners
Meta-Classes (p. 1-77)	Access information about classes without requiring instances

## Classes and Objects

<code>class</code>	Create object or return class of object
<code>classdef</code>	Class definition key words
<code>fieldnames</code>	Field names of structure, or public fields of object
<code>inferiorto</code>	Specify inferior class relationship
<code>isa</code>	Determine whether input is object of given class
<code>isobject</code>	Determine if input is MATLAB object
<code>loadobj</code>	User-defined class method called by load function
<code>methods</code>	Information on class methods
<code>methodsview</code>	Information on class methods in separate window
<code>properties</code>	Display class property names
<code>saveobj</code>	Method called by save function for user-defined objects
<code>subsasgn</code>	Subscripted assignment for objects
<code>subsindex</code>	Subscripted indexing for objects
<code>subsref</code>	Subscripted reference for objects

<code>substruct</code>	Create structure argument for <code>subsasgn</code> or <code>subsref</code>
<code>superiorto</code>	Establish superior class relationship

## Handle Classes

<code>addlistener (handle)</code>	Create event listener
<code>addprop (dynamicprops)</code>	Add dynamic property
<code>delete (handle)</code>	Handle object destructor function
<code>dynamicprops</code>	Abstract class used to derive handle class with dynamic properties
<code>findobj (handle)</code>	Finds objects matching specified conditions
<code>findprop (handle)</code>	Find <code>meta.property</code> object associated with property name
<code>get (hgsetget)</code>	Query property values of handle objects derived from <code>hgsetget</code> class
<code>getdisp (hgsetget)</code>	Override to change command window display
<code>handle</code>	Abstract class for deriving handle classes
<code>hgsetget</code>	Abstract class used to derive handle class with <code>set</code> and <code>get</code> methods
<code>isvalid (handle)</code>	Is object valid handle object
<code>notify (handle)</code>	notify listeners that event is occurring
<code>relationaloperators (handle)</code>	Equality and sorting of handle objects

<code>set (hgsetget)</code>	Assign property values to handle objects derived from <code>hgsetget</code> class
<code>setdisp (hgsetget)</code>	Override to change command window display

## Events and Listeners

<code>addlistener (handle)</code>	Create event listener
<code>event.EventData</code>	Base class for all data objects passed to event listeners
<code>event.listener</code>	Class defining listener objects
<code>event.PropertyEvent</code>	Listener for property events
<code>event.proplistener</code>	Define listener object for property events
<code>events</code>	Display class event names
<code>notify (handle)</code>	notify listeners that event is occurring

## Meta-Classes

<code>meta.class</code>	<code>meta.class</code> class describes MATLAB classes
<code>meta.class.fromName</code>	Return <code>meta.class</code> object associated with named class
<code>meta.DynamicProperty</code>	<code>meta.DynamicProperty</code> class describes dynamic property of MATLAB object
<code>meta.event</code>	<code>meta.event</code> class describes MATLAB class events
<code>meta.method</code>	<code>meta.method</code> class describes MATLAB class methods

<code>meta.package</code>	<code>meta.package</code> class describes MATLAB packages
<code>meta.package.fromName</code>	Return <code>meta.package</code> object for specified package
<code>meta.package.getAllPackages</code>	Get all top-level packages
<code>meta.property</code>	<code>meta.property</code> class describes MATLAB class properties
<code>metaclass</code>	Return <code>meta.class</code> object

## File I/O

File Name Construction (p. 1-79)	Get path, directory, filename information; construct filenames
File Opening, Loading, and Saving (p. 1-80)	Open files; transfer data between files and MATLAB workspace
Memory Mapping (p. 1-80)	Access file data via memory map using MATLAB array indexing
Low-Level File I/O (p. 1-80)	Low-level operations that use a file identifier
Text Files (p. 1-81)	Delimited or formatted I/O to text files
XML Documents (p. 1-82)	Documents written in Extensible Markup Language
Spreadsheets (p. 1-82)	Excel and Lotus 1-2-3 files
Scientific Data (p. 1-83)	CDF, FITS, HDF formats
Audio and Audio/Video (p. 1-86)	General audio functions; SparcStation, WAVE, AVI files
Images (p. 1-88)	Graphics files
Internet Exchange (p. 1-88)	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

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

<code>tempdir</code>	Name of system's temporary directory
<code>tempname</code>	Unique name for temporary file

## **File Opening, Loading, and Saving**

<code>daqread</code>	Read Data Acquisition Toolbox™ (.daq) file
<code>filehandle</code>	Construct file handle object
<code>importdata</code>	Load data from disk file
<code>load</code>	Load workspace variables from disk
<code>open</code>	Open files based on extension
<code>save</code>	Save workspace variables to disk
<code>uiimport</code>	Open Import Wizard to import data
<code>winopen</code>	Open file in appropriate application (Windows)

## **Memory Mapping**

<code>disp (memmapfile)</code>	Information about memmapfile object
<code>get (memmapfile)</code>	Memmapfile object properties
<code>memmapfile</code>	Construct memmapfile object

## **Low-Level File I/O**

<code>fclose</code>	Close one or more open files
<code>feof</code>	Test for end-of-file
<code>ferror</code>	Query the MATLAB software about errors in file input or output



<code>fgetl</code>	Read line from file, discarding newline character
<code>fgets</code>	Read line from file, keeping newline character
<code>fopen</code>	Open file, or obtain information about open files
<code>fprintf</code>	Write formatted data to file
<code>fread</code>	Read binary data from file
<code>frewind</code>	Move file position indicator to beginning of open file
<code>fscanf</code>	Read formatted data from file
<code>fseek</code>	Set file position indicator
<code>ftell</code>	File position indicator
<code>fwrite</code>	Write binary data to file

## **Text Files**

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

## **XML Documents**

<code>xmlread</code>	Parse XML document and return Document Object Model node
<code>xmlwrite</code>	Serialize XML Document Object Model node
<code>xslt</code>	Transform XML document using XSLT engine

## **Spreadsheets**

Microsoft Excel (p. 1-82)	Read and write Microsoft Excel spreadsheet
Lotus 1-2-3 (p. 1-82)	Read and write Lotus WK1 spreadsheet

## **Microsoft Excel**

<code>xlsinfo</code>	Determine whether file contains Microsoft® Excel® (.xls) spreadsheet
<code>xlsread</code>	Read Microsoft Excel spreadsheet file (.xls)
<code>xlswrite</code>	Write Microsoft Excel spreadsheet file (.xls)

## **Lotus 1-2-3**

<code>wk1info</code>	Determine whether file contains 1-2-3 WK1 worksheet
<code>wk1read</code>	Read Lotus 1-2-3 WK1 spreadsheet file into matrix
<code>wk1write</code>	Write matrix to Lotus 1-2-3 WK1 spreadsheet file

## Scientific Data

Common Data Format (p. 1-83)	Work with CDF files
Network Common Data Form (p. 1-83)	Work with netCDF files
Flexible Image Transport System (p. 1-85)	Work with FITS files
Hierarchical Data Format (p. 1-85)	Work with HDF files
Band-Interleaved Data (p. 1-86)	Work with band-interleaved files

## Common Data Format

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

## Network Common Data Form

### File Operations

<code>netcdf</code>	Summary of MATLAB Network Common Data Form (netCDF) capabilities
<code>netcdf.abort</code>	Revert recent netCDF file definitions
<code>netcdf.close</code>	Close netCDF file
<code>netcdf.create</code>	Create new netCDF dataset

<code>netcdf.endDef</code>	End netCDF file define mode
<code>netcdf.getConstant</code>	Return numeric value of named constant
<code>netcdf.getConstantNames</code>	Return list of constants known to netCDF library
<code>netcdf.inq</code>	Return information about netCDF file
<code>netcdf.inqLibVers</code>	Return netCDF library version information
<code>netcdf.open</code>	Open netCDF file
<code>netcdf.reDef</code>	Put open netCDF file into define mode
<code>netcdf.setDefaultFormat</code>	Change default netCDF file format
<code>netcdf.setFill</code>	Set netCDF fill mode
<code>netcdf.sync</code>	Synchronize netCDF file to disk

### **Dimensions**

<code>netcdf.defDim</code>	Create netCDF dimension
<code>netcdf.inqDim</code>	Return netCDF dimension name and length
<code>netcdf.inqDimID</code>	Return dimension ID
<code>netcdf.renameDim</code>	Change name of netCDF dimension

### **Variables**

<code>netcdf.defVar</code>	Create netCDF variable
<code>netcdf.getVar</code>	Return data from netCDF variable
<code>netcdf.inqVar</code>	Return information about variable
<code>netcdf.inqVarID</code>	Return ID associated with variable name

<code>netcdf.putVar</code>	Write data to netCDF variable
<code>netcdf.renameVar</code>	Change name of netCDF variable

### Attributes

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

### Flexible Image Transport System

<code>fitsinfo</code>	Information about FITS file
<code>fitsread</code>	Read data from FITS file

### Hierarchical Data Format

<code>hdf</code>	Summary of MATLAB HDF4 capabilities
<code>hdf5</code>	Summary of MATLAB HDF5 capabilities
<code>hdf5info</code>	Information about HDF5 file
<code>hdf5read</code>	Read HDF5 file
<code>hdf5write</code>	Write data to file in HDF5 format

<code>hdfinfo</code>	Information about HDF4 or HDF-EOS file
<code>hdfread</code>	Read data from HDF4 or HDF-EOS file
<code>hdftool</code>	Browse and import data from HDF4 or HDF-EOS files

### **Band-Interleaved Data**

<code>multibandread</code>	Read band-interleaved data from binary file
<code>multibandwrite</code>	Write band-interleaved data to file

### **Audio and Audio/Video**

Utilities (p. 1-86)	Create audio player object, obtain information about multimedia files, convert to/from audio signal
SPARCstation-Specific Sound (p. 1-87)	Access NeXT/SUN (.au) sound files
Microsoft WAVE Sound (p. 1-87)	Access Microsoft WAVE (.wav) sound files
Audio/Video Interleaved (p. 1-88)	Access Audio/Video interleaved (.avi) sound files

### **Utilities**

<code>audioplayer</code>	Create audio player object
<code>audiorecorder</code>	Create audio recorder object
<code>beep</code>	Produce beep sound
<code>lin2mu</code>	Convert linear audio signal to mu-law

---

<code>mmfileinfo</code>	Information about multimedia file
<code>mmreader</code>	Create multimedia reader object for reading video files
<code>mu2lin</code>	Convert mu-law audio signal to linear
<code>read</code>	Read video frame data from multimedia reader object
<code>sound</code>	Convert vector into sound
<code>soundsc</code>	Scale data and play as sound

### **SPARCstation-Specific Sound**

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

### **Microsoft WAVE Sound**

<code>wavfinfo</code>	Information about Microsoft WAVE (.wav) sound file
<code>wavplay</code>	Play recorded sound on PC-based audio output device
<code>wavread</code>	Read Microsoft WAVE (.wav) sound file
<code>wavrecord</code>	Record sound using PC-based audio input device
<code>wavwrite</code>	Write Microsoft WAVE (.wav) sound file

## Audio/Video Interleaved

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

## Images

<code>exifread</code>	Read EXIF information from JPEG and TIFF image files
<code>im2java</code>	Convert image to Java image
<code>imfinfo</code>	Information about graphics file
<code>imread</code>	Read image from graphics file
<code>imwrite</code>	Write image to graphics file

## Internet Exchange

URL, Zip, Tar, E-Mail (p. 1-89)	Send e-mail, read from given URL, extract from tar or zip file, compress and decompress files
FTP (p. 1-89)	Connect to FTP server, download from server, manage FTP files, close server connection



**URL, Zip, Tar, E-Mail**

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

**FTP**

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

## Graphics

Basic Plots and Graphs (p. 1-90)	Linear line plots, log and semilog plots
Plotting Tools (p. 1-91)	GUIs for interacting with plots
Annotating Plots (p. 1-91)	Functions for and properties of titles, axes labels, legends, mathematical symbols
Specialized Plotting (p. 1-92)	Bar graphs, histograms, pie charts, contour plots, function plotters
Bit-Mapped Images (p. 1-96)	Display image object, read and write graphics file, convert to movie frames
Printing (p. 1-96)	Printing and exporting figures to standard formats
Handle Graphics (p. 1-97)	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
LineStyle (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

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

## Annotating Plots

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

## Specialized Plotting

Area, Bar, and Pie Plots (p. 1-92)	1-D, 2-D, and 3-D graphs and charts
Contour Plots (p. 1-93)	Unfilled and filled contours in 2-D and 3-D
Direction and Velocity Plots (p. 1-93)	Comet, compass, feather and quiver plots
Discrete Data Plots (p. 1-93)	Stair, step, and stem plots
Function Plots (p. 1-93)	Easy-to-use plotting utilities for graphing functions
Histograms (p. 1-94)	Plots for showing distributions of data
Polygons and Surfaces (p. 1-94)	Functions to generate and plot surface patches in two or more dimensions
Scatter/Bubble Plots (p. 1-95)	Plots of point distributions
Animation (p. 1-95)	Functions to create and play movies of plots

## Area, Bar, and Pie Plots

<code>area</code>	Filled area 2-D plot
<code>bar</code> , <code>barh</code>	Plot bar graph (vertical and horizontal)
<code>bar3</code> , <code>bar3h</code>	Plot 3-D bar chart
<code>pareto</code>	Pareto chart
<code>pie</code>	Pie chart
<code>pie3</code>	3-D pie chart

## Contour Plots

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

## Direction and Velocity Plots

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

## Discrete Data Plots

<code>stairs</code>	Stairstep graph
<code>stem</code>	Plot discrete sequence data
<code>stem3</code>	Plot 3-D discrete sequence data

## Function Plots

<code>ezcontour</code>	Easy-to-use contour plotter
<code>ezcontourf</code>	Easy-to-use filled contour plotter
<code>ezmesh</code>	Easy-to-use 3-D mesh plotter

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

## **Histograms**

hist	Histogram plot
histc	Histogram count
rose	Angle histogram plot

## **Polygons and Surfaces**

convhull	Convex hull
cylinder	Generate cylinder
delaunay	Delaunay triangulation
delaunay3	3-D Delaunay tessellation
delaunayn	N-D Delaunay tessellation
dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
ellipsoid	Generate ellipsoid

<code>fill</code>	Filled 2-D polygons
<code>fill3</code>	Filled 3-D polygons
<code>inpolygon</code>	Points inside polygonal region
<code>pcolor</code>	Pseudocolor (checkerboard) plot
<code>polyarea</code>	Area of polygon
<code>rectint</code>	Rectangle intersection area
<code>ribbon</code>	Ribbon plot
<code>slice</code>	Volumetric slice plot
<code>sphere</code>	Generate sphere
<code>tsearch</code>	Search for enclosing Delaunay triangle
<code>tsearchn</code>	N-D closest simplex search
<code>voronoi</code>	Voronoi diagram
<code>waterfall</code>	Waterfall plot

### **Scatter/Bubble Plots**

<code>plotmatrix</code>	Scatter plot matrix
<code>scatter</code>	Scatter plot
<code>scatter3</code>	3-D scatter plot

### **Animation**

<code>frame2im</code>	Return image data associated with movie frame
<code>getframe</code>	Capture movie frame
<code>im2frame</code>	Convert image to movie frame

<code>movie</code>	Play recorded movie frames
<code>noanimate</code>	Change EraseMode of all objects to normal

## **Bit-Mapped Images**

<code>frame2im</code>	Return image data associated with movie frame
<code>im2frame</code>	Convert image to movie frame
<code>im2java</code>	Convert image to Java image
<code>image</code>	Display image object
<code>imagesc</code>	Scale data and display image object
<code>imfinfo</code>	Information about graphics file
<code>imformats</code>	Manage image file format registry
<code>imread</code>	Read image from graphics file
<code>imwrite</code>	Write image to graphics file
<code>ind2rgb</code>	Convert indexed image to RGB image

## **Printing**

<code>hgexport</code>	Export figure
<code>orient</code>	Hardcopy paper orientation
<code>print, printopt</code>	Print figure or save to file and configure printer defaults
<code>printdlg</code>	Print dialog box
<code>printpreview</code>	Preview figure to print
<code>saveas</code>	Save figure or Simulink block diagram using specified format



## Handle Graphics

Graphics Object Identification (p. 1-97)	Find and manipulate graphics objects via their handles
Object Creation (p. 1-98)	Constructors for core graphics objects
Plot Objects (p. 1-98)	Property descriptions for plot objects
Figure Windows (p. 1-99)	Control and save figures
Axes Operations (p. 1-100)	Operate on axes objects
Object Property Operations (p. 1-100)	Query, set, and link object properties

## Graphics Object Identification

<code>allchild</code>	Find all children of specified objects
<code>ancestor</code>	Ancestor of graphics object
<code>copyobj</code>	Copy graphics objects and their descendants
<code>delete</code>	Remove files or graphics objects
<code>findall</code>	Find all graphics objects
<code>findfigs</code>	Find visible offscreen figures
<code>findobj</code>	Locate graphics objects with specific properties
<code>gca</code>	Current axes handle
<code>gcbf</code>	Handle of figure containing object whose callback is executing
<code>gcbo</code>	Handle of object whose callback is executing
<code>gco</code>	Handle of current object
<code>get</code>	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 object properties
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

<code>newplot</code>	Determine where to draw graphics objects
<code>opengl</code>	Control OpenGL® rendering
<code>refresh</code>	Redraw current figure
<code>saveas</code>	Save figure or Simulink block diagram using specified format

### **Axes Operations**

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

### **Object Property Operations**

<code>get</code>	Query Handle Graphics object properties
<code>linkaxes</code>	Synchronize limits of specified 2-D axes
<code>linkprop</code>	Keep same value for corresponding properties
<code>refreshdata</code>	Refresh data in graph when data source is specified
<code>set</code>	Set Handle Graphics object properties

## 3-D Visualization

Surface and Mesh Plots (p. 1-101)	Plot matrices, visualize functions of two variables, specify colormap
View Control (p. 1-103)	Control the camera viewpoint, zooming, rotation, aspect ratio, set axis limits
Lighting (p. 1-105)	Add and control scene lighting
Transparency (p. 1-105)	Specify and control object transparency
Volume Visualization (p. 1-106)	Visualize gridded volume data

### Surface and Mesh Plots

Surface and Mesh Creation (p. 1-101)	Visualizing gridded and triangulated data as lines and surfaces
Domain Generation (p. 1-102)	Gridding data and creating arrays
Color Operations (p. 1-102)	Specifying, converting, and manipulating color spaces, colormaps, colorbars, and backgrounds
Colormaps (p. 1-103)	Built-in colormaps you can use

### Surface and Mesh Creation

<code>hidden</code>	Remove hidden lines from mesh plot
<code>mesh</code> , <code>meshc</code> , <code>meshz</code>	Mesh plots
<code>peaks</code>	Example function of two variables
<code>surf</code> , <code>surfc</code>	3-D shaded surface plot
<code>surface</code>	Create surface object
<code>surf1</code>	Surface plot with colormap-based lighting

tetramesh	Tetrahedron mesh plot
trimesh	Triangular mesh plot
triplot	2-D triangular plot
trisurf	Triangular surface plot

### **Domain Generation**

griddata	Data gridding
meshgrid	Generate X and Y arrays for 3-D plots

### **Color Operations**

brighten	Brighten or darken colormap
caxis	Color axis scaling
colorbar	Colorbar showing color scale
colordef	Set default property values to display different color schemes
colormap	Set and get current colormap
colormapeditor	Start colormap editor
ColorSpec (Color Specification)	Color specification
graymon	Set default figure properties for grayscale monitors
hsv2rgb	Convert HSV colormap to RGB colormap
rgb2hsv	Convert RGB colormap to HSV colormap
rgbplot	Plot colormap
shading	Set color shading properties
spinmap	Spin colormap

<code>surfnorm</code>	Compute and display 3-D surface normals
<code>whitebg</code>	Change axes background color

## Colormaps

<code>contrast</code>	Grayscale colormap for contrast enhancement
-----------------------	---

## View Control

Camera Viewpoint (p. 1-103)	Orbiting, dollying, pointing, rotating camera positions and setting fields of view
Aspect Ratio and Axis Limits (p. 1-104)	Specifying what portions of axes to view and how to scale them
Object Manipulation (p. 1-104)	Panning, rotating, and zooming views
Region of Interest (p. 1-105)	Interactively identifying rectangular regions

## Camera Viewpoint

<code>camdolly</code>	Move camera position and target
<code>cameratoolbar</code>	Control camera toolbar programmatically
<code>camlookat</code>	Position camera to view object or group of objects
<code>camorbit</code>	Rotate camera position around camera target
<code>campan</code>	Rotate camera target around camera position

<code>campos</code>	Set or query camera position
<code>camproj</code>	Set or query projection type
<code>camroll</code>	Rotate camera about view axis
<code>camtarget</code>	Set or query location of camera target
<code>camup</code>	Set or query camera up vector
<code>camva</code>	Set or query camera view angle
<code>camzoom</code>	Zoom in and out on scene
<code>makehgtform</code>	Create 4-by-4 transform matrix
<code>view</code>	Viewpoint specification
<code>viewmtx</code>	View transformation matrices

### **Aspect Ratio and Axis Limits**

<code>daspect</code>	Set or query axes data aspect ratio
<code>pbaspect</code>	Set or query plot box aspect ratio
<code>xlim, ylim, zlim</code>	Set or query axis limits

### **Object Manipulation**

<code>pan</code>	Pan view of graph interactively
<code>reset</code>	Reset graphics object properties to their defaults
<code>rotate</code>	Rotate object in specified direction
<code>rotate3d</code>	Rotate 3-D view using mouse
<code>selectmoveresize</code>	Select, move, resize, or copy axes and uicontrol graphics objects
<code>zoom</code>	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

<code>coneplot</code>	Plot velocity vectors as cones in 3-D vector field
<code>contourslice</code>	Draw contours in volume slice planes
<code>curl</code>	Compute curl and angular velocity of vector field
<code>divergence</code>	Compute divergence of vector field
<code>flow</code>	Simple function of three variables
<code>interpstreamspeed</code>	Interpolate stream-line vertices from flow speed
<code>isocaps</code>	Compute isosurface end-cap geometry
<code>isocolors</code>	Calculate isosurface and patch colors
<code>isonormals</code>	Compute normals of isosurface vertices
<code>isosurface</code>	Extract isosurface data from volume data
<code>reducepatch</code>	Reduce number of patch faces
<code>reducevolume</code>	Reduce number of elements in volume data set
<code>shrinkfaces</code>	Reduce size of patch faces
<code>slice</code>	Volumetric slice plot
<code>smooth3</code>	Smooth 3-D data
<code>stream2</code>	Compute 2-D streamline data
<code>stream3</code>	Compute 3-D streamline data
<code>streamline</code>	Plot streamlines from 2-D or 3-D vector data
<code>streamparticles</code>	Plot stream particles
<code>streamribbon</code>	3-D stream ribbon plot from vector volume data

---

<code>streamslice</code>	Plot streamlines in slice planes
<code>streamtube</code>	Create 3-D stream tube plot
<code>subvolume</code>	Extract subset of volume data set
<code>surf2patch</code>	Convert surface data to patch data
<code>volumebounds</code>	Coordinate and color limits for volume data

## GUI Development

Predefined Dialog Boxes (p. 1-108)	Dialog boxes for error, user input, waiting, etc.
User Interface Deployment (p. 1-109)	Launch GUIs, create the handles structure
User Interface Development (p. 1-109)	Start GUIDE, manage application data, get user input
User Interface Objects (p. 1-110)	Create GUI components
Objects from Callbacks (p. 1-111)	Find object handles from within callbacks functions
GUI Utilities (p. 1-111)	Move objects, wrap text
Program Execution (p. 1-112)	Wait and resume based on user input

### Predefined Dialog Boxes

<code>dialog</code>	Create and display dialog box
<code>errordlg</code>	Create and open error dialog box
<code>export2wsdlg</code>	Export variables to workspace
<code>helpdlg</code>	Create and open help dialog box
<code>inputdlg</code>	Create and open input dialog box
<code>listdlg</code>	Create and open list-selection dialog box
<code>msgbox</code>	Create and open message box
<code>printdlg</code>	Print dialog box
<code>printpreview</code>	Preview figure to print
<code>questdlg</code>	Create and open question dialog box
<code>uigetdir</code>	Open standard dialog box for selecting a directory

<code>uigetfile</code>	Open standard dialog box for retrieving files
<code>uigetpref</code>	Open dialog box for retrieving preferences
<code>uiopen</code>	Open file selection dialog box with appropriate file filters
<code>uiputfile</code>	Open standard dialog box for saving files
<code>uisave</code>	Open standard dialog box for saving workspace variables
<code>uisetcolor</code>	Open standard dialog box for setting object's <code>ColorSpec</code>
<code>uisetfont</code>	Open standard dialog box for setting object's font characteristics
<code>waitbar</code>	Open waitbar
<code>warndlg</code>	Open warning dialog box

## User Interface Deployment

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

## User Interface Development

<code>addpref</code>	Add preference
<code>getappdata</code>	Value of application-defined data
<code>getpref</code>	Preference

<code>ginput</code>	Graphical input from mouse or cursor
<code>guidata</code>	Store or retrieve GUI data
<code>guide</code>	Open GUI Layout Editor
<code>inspect</code>	Open Property Inspector
<code>isappdata</code>	True if application-defined data exists
<code>ispref</code>	Test for existence of preference
<code>rmappdata</code>	Remove application-defined data
<code>rmpref</code>	Remove preference
<code>setappdata</code>	Specify application-defined data
<code>setpref</code>	Set preference
<code>uigetpref</code>	Open dialog box for retrieving preferences
<code>uisetpref</code>	Manage preferences used in <code>uigetpref</code>
<code>waitfor</code>	Wait for condition before resuming execution
<code>waitforbuttonpress</code>	Wait for key press or mouse-button click

## **User Interface Objects**

<code>menu</code>	Generate menu of choices for user input
<code>uibuttongroup</code>	Create container object to exclusively manage radio buttons and toggle buttons
<code>uicontextmenu</code>	Create context menu
<code>uicontrol</code>	Create user interface control object

<code>uimenu</code>	Create menus on figure windows
<code>uipanel</code>	Create panel container object
<code>uipushtool</code>	Create push button on toolbar
<code>uitable</code>	Create 2-D graphic table GUI component
<code>uitoggletool</code>	Create toggle button on toolbar
<code>uitoolbar</code>	Create toolbar on figure

## Objects from Callbacks

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

## GUI Utilities

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

## **Program Execution**

`uiresume`

Resume execution of blocked M-file

`uiwait`

Block execution and wait for resume



## External Interfaces

Dynamic Link Libraries (p. 1-113)	Access functions stored in external shared library (.dll) files
Java (p. 1-114)	Work with objects constructed from Java API and third-party class packages
Component Object Model and ActiveX (p. 1-115)	Integrate COM components into your application
Web Services (p. 1-117)	Communicate between applications over a network using SOAP and WSDL
Serial Port Devices (p. 1-118)	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.

## Dynamic Link Libraries

<code>calllib</code>	Call function in shared library
<code>libfunctions</code>	Return information on functions in shared library
<code>libfunctionsview</code>	View functions in a shared library
<code>libisloaded</code>	Determine if shared library is loaded
<code>libpointer</code>	Create pointer object for use with shared libraries
<code>libstruct</code>	Create structure pointer for use with shared libraries

<code>loadlibrary</code>	Load shared library into MATLAB software
<code>unloadlibrary</code>	Unload shared library from memory

## **Java**

<code>class</code>	Create object or return class of object
<code>fieldnames</code>	Field names of structure, or public fields of object
<code>import</code>	Add package or class to current import list
<code>inspect</code>	Open Property Inspector
<code>isa</code>	Determine whether input is object of given class
<code>isjava</code>	Determine whether input is Sun Java object
<code>javaaddpath</code>	Add entries to dynamic Sun Java class path
<code>javaArray</code>	Construct Sun Java array
<code>javachk</code>	Generate error message based on Sun Java feature support
<code>javaclasspath</code>	Set and get dynamic Sun Java class path
<code>javaMethod</code>	Invoke Sun Java method
<code>javaObject</code>	Construct Sun Java object
<code>javarmpath</code>	Remove entries from dynamic Sun Java class path
<code>methods</code>	Information on class methods

<code>methodsview</code>	Information on class methods in separate window
<code>usejava</code>	Determine whether Sun Java feature is supported in MATLAB software

## **Component Object Model and ActiveX**

<code>actxcontrol</code>	Create Microsoft® ActiveX® control in figure window
<code>actxcontrollist</code>	List all currently installed Microsoft ActiveX controls
<code>actxcontrolselect</code>	Open GUI to create Microsoft ActiveX control
<code>actxGetRunningServer</code>	Get handle to running instance of Automation server
<code>actxserver</code>	Create COM server
<code>addproperty</code>	Add custom property to COM object
<code>class</code>	Create object or return class of object
<code>delete (COM)</code>	Remove COM control or server
<code>deleteproperty</code>	Remove custom property from COM object
<code>enableservice</code>	Enable, disable, or report status of Automation server
<code>eventlisteners</code>	List all event handler functions registered for COM object
<code>events (COM)</code>	List of events COM object can trigger
<code>Execute</code>	Execute MATLAB command in Automation server
<code>Feval (COM)</code>	Evaluate MATLAB function in Automation server

fieldnames	Field names of structure, or public fields of object
get (COM)	Get property value from interface, or display properties
GetCharArray	Get character array from server
GetFullMatrix	Get matrix from server
GetVariable	Get data from variable in server workspace
GetWorkspaceData	Get data from server workspace
inspect	Open Property Inspector
interfaces	List custom interfaces to COM server
invoke	Invoke method on COM object or interface, or display methods
isa	Determine whether input is object of given class
iscom	Is input COM object
isevent	True if COM object event
isinterface	Is input COM interface
ismethod	Determine whether input is COM object method
isprop	Determine whether input is COM object property
load (COM)	Initialize control object from file
MaximizeCommandWindow	Open server window on Microsoft Windows desktop
methods	Information on class methods
methodsview	Information on class methods in separate window
MinimizeCommandWindow	Minimize size of server window

<code>move</code>	Move or resize control in parent window
<code>propedit (COM)</code>	Open built-in property page for control
<code>PutCharArray</code>	Store character array in server
<code>PutFullMatrix</code>	Store matrix in server
<code>PutWorkspaceData</code>	Store data in server workspace
<code>Quit (COM)</code>	Terminate MATLAB server
<code>registerevent</code>	Register event handler for COM object event at run-time
<code>release</code>	Release COM interface
<code>save (COM)</code>	Serialize control object to file
<code>set (COM)</code>	Set object or interface property to specified value
<code>unregisterallevents</code>	Unregister all event handlers for COM object event at run-time
<code>unregisterevent</code>	Unregister event handler for COM object event at run-time

## Web Services

<code>callSoapService</code>	Send SOAP message off to endpoint
<code>createClassFromWsd1</code>	Create MATLAB object based on WSDL file
<code>createSoapMessage</code>	Create SOAP message to send to server
<code>parseSoapResponse</code>	Convert response string from SOAP server into MATLAB types

## Serial Port Devices

<code>clear (serial)</code>	Remove serial port object from MATLAB workspace
<code>delete (serial)</code>	Remove serial port object from memory
<code>fgetl (serial)</code>	Read line of text from device and discard terminator
<code>fgets (serial)</code>	Read line of text from device and include terminator
<code>fopen (serial)</code>	Connect serial port object to device
<code>fprintf (serial)</code>	Write text to device
<code>fread (serial)</code>	Read binary data from device
<code>fscanf (serial)</code>	Read data from device, and format as text
<code>fwrite (serial)</code>	Write binary data to device
<code>get (serial)</code>	Serial port object properties
<code>instrcallback</code>	Event information when event occurs
<code>instrfind</code>	Read serial port objects from memory to MATLAB workspace
<code>instrfindall</code>	Find visible and hidden serial port objects
<code>isvalid (serial)</code>	Determine whether serial port objects are valid
<code>length (serial)</code>	Length of serial port object array
<code>load (serial)</code>	Load serial port objects and variables into MATLAB workspace
<code>readasync</code>	Read data asynchronously from device
<code>record</code>	Record data and event information to file

---

<code>save (serial)</code>	Save serial port objects and variables to MAT-file
<code>serial</code>	Create serial port object
<code>serialbreak</code>	Send break to device connected to serial port
<code>set (serial)</code>	Configure or display serial port object properties
<code>size (serial)</code>	Size of serial port object array
<code>stopasync</code>	Stop asynchronous read and write operations





# Functions — Alphabetical List

---

Arithmetic Operators + - \* / \ ^ '   
Relational Operators < > <= >= == ~=   
Logical Operators: Elementwise & | ~   
Logical Operators: Short-circuit && ||   
Special Characters [ ] ( ) { } = ' . ... , ; : % ! @   
colon (:)  
abs  
accumarray  
acos  
acosd  
acosh  
acot  
acotd  
acoth  
acsc  
acscd  
acsch  
actxcontrol  
actxcontrollist  
actxcontrolselect  
actxGetRunningServer  
actxserver  
addCause (MException)  
addevent  
addframe  
addlistener (handle)

addOptional (inputParser)  
addParamValue (inputParser)  
addpath  
addpref  
addprop (dynamicprops)  
addproperty  
addRequired (inputParser)  
addsample  
addsampletocollection  
addtodate  
addts  
airy  
align  
alim  
all  
allchild  
alpha  
alphamap  
amd  
ancestor  
and  
angle  
annotation  
Annotation Arrow Properties  
Annotation Doublearrow Properties  
Annotation Ellipse Properties  
Annotation Line Properties  
Annotation Rectangle Properties  
Annotation Textarrow Properties  
Annotation Textbox Properties  
ans  
any  
area  
Areaseries Properties  
arrayfun  
ascii  
asec

---

asecd  
asech  
asin  
asind  
asinh  
assert  
assignin  
atan  
atan2  
atand  
atanh  
audioplayer  
audiorecorder  
aufinfo  
auread  
auwrite  
avifile  
aviinfo  
aviread  
axes  
Axes Properties  
axis  
balance  
bar, barh  
bar3, bar3h  
Barseries Properties  
base2dec  
beep  
bench  
besselh  
besseli  
besselj  
besselk  
bessely  
beta  
betainc  
betaln

big  
bigstab  
bin2dec  
binary  
bitand  
bitemp  
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  
cart2pol  
cart2sph  
case  
cast  
cat  
catch  
caxis  
cd  
cd (ftp)  
cdf2rdf  
cdfepoch  
cdfinfo  
cdfread  
cdfwrite  
ceil  
cell  
cell2mat  
cell2struct  
celldisp  
cellfun  
cellplot  
cellstr  
cgs  
char  
checkin  
checkout  
chol  
cholinc  
cholupdate  
circshift  
cla

clabel  
class  
classdef  
clc  
clear  
clearvars  
clear (serial)  
clf  
clipboard  
clock  
close  
close (avifile)  
close (ftp)  
closereq  
cmopts  
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  
corrcoef  
cos  
cosd  
cosh  
cot  
cotd  
coth  
cov  
cplxpair  
cputime  
create (RandStream)  
createClassFromWsd  
createCopy (inputParser)  
createSoapMessage  
cross  
csc  
cscd  
csch  
csvread  
csvwrite  
ctranspose (timeseries)  
cumprod  
cumsum  
cumtrapz

curl  
customverctrl  
cylinder  
daqread  
daspect  
datacursormode  
datatipinfo  
date  
datenum  
datestr  
datetick  
datevec  
dbclear  
dbcont  
dbdown  
dblquad  
dbmex  
dbquit  
dbstack  
dbstatus  
dbstep  
dbstop  
dbtype  
dbup  
dde23  
ddeget  
ddesd  
ddeset  
deal  
deblank  
debug  
dec2base  
dec2bin  
dec2hex  
decic  
deconv  
del2



---

delaunay  
delaunay3  
delaunayn  
delete  
delete (COM)  
delete (ftp)  
delete (handle)  
delete (serial)  
delete (timer)  
deleteproperty  
delevent  
delsample  
delsamplefromcollection  
demo  
depdir  
depfun  
det  
detrend  
detrend (timeseries)  
deval  
diag  
dialog  
diary  
diff  
diffuse  
dir  
dir (ftp)  
disp  
disp (memmapfile)  
disp (MException)  
disp (serial)  
disp (timer)  
display  
divergence  
dlmread  
dlmwrite  
dmperm

doc  
docopt  
docsearch  
dos  
dot  
double  
dragrect  
drawnow  
dsearch  
dsearchn  
dynamicprops  
echo  
echodemo  
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  
factor  
factorial  
false  
fclose  
fclose (serial)  
feather  
feof  
ferror  
feval  
Feval (COM)

fft  
fft2  
fftn  
fftshift  
fftw  
fgetl  
fgetl (serial)  
fgets  
fgets (serial)  
fieldnames  
figure  
Figure Properties  
figurepalette  
fileattrib  
filebrowser  
File Formats  
filehandle  
filemarker  
fileparts  
fileread  
filesep  
fill  
fill3  
filter  
filter (timeseries)  
filter2  
find  
findall  
findfigs  
findobj  
findobj (handle)  
findprop (handle)  
findstr  
finish  
fitsinfo  
fitsread  
fix

---

flipdim  
fliplr  
flipud  
floor  
flow  
fminbnd  
fminsearch  
fopen  
fopen (serial)  
for  
format  
fplot  
fprintf  
fprintf (serial)  
frame2im  
fread  
fread (serial)  
freqspace  
frewind  
fscanf  
fscanf (serial)  
fseek  
ftell  
ftp  
full  
fullfile  
func2str  
function  
function\_handle (@)  
functions  
funm  
fwrite  
fwrite (serial)  
fzero  
gallery  
gamma, gammainc, gammaln  
gca

gcbf  
gcbo  
gcd  
gcf  
gco  
ge  
genpath  
genvarname  
get  
get (COM)  
get (hgsetget)  
get (memmapfile)  
get (RandStream)  
get (serial)  
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)  
getsamplesusingtime (timeseries)  
getsamplesusingtime (tscollection)  
gettimeseriesnames  
gettsafteratevent

---

gettsafterevent  
gettsatevent  
gettsbeforeatevent  
gettsbeforeevent  
gettsbetweenevents  
GetVariable  
GetWorkspaceData  
ginput  
global  
gmres  
gplot  
grabcode  
gradient  
graymon  
grid  
griddata  
griddata3  
griddatan  
gsvd  
gt  
gtext  
guidata  
guide  
guihandles  
gunzip  
gzip  
hadamard  
handle  
hankel  
hdf  
hdf5  
hdf5info  
hdf5read  
hdf5write  
hdfinfo  
hdfread  
hdftool

help  
helpbrowser  
helpdesk  
helpdlg  
helpwin  
hess  
hex2dec  
hex2num  
hgexport  
hggroup  
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  
imfinfo  
imformats  
import  
importdata  
imread  
imwrite  
ind2rgb  
ind2sub  
Inf  
inferiorto  
info  
inline  
inmem  
inpolygon  
input  
inputdlg  
inputname  
inputParser  
inspect  
instrcallback  
instrfind  
instrfindall  
int2str  
int8, int16, int32, int64  
interfaces  
interp1  
interp1q  
interp2  
interp3  
interpft  
interp  
interpstreamspeed

intersect  
intmax  
intmin  
intwarning  
inv  
invhilb  
invoke  
ipermute  
iqr (timeseries)  
is\*  
isa  
isappdata  
iscell  
iscellstr  
ischar  
iscom  
isdir  
isempty  
isempty (timeseries)  
isempty (tscollection)  
isequal  
isequal (MException)  
isequalwithequalnans  
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)  
isvalid (serial)  
isvalid (timer)  
isvarname  
isvector  
j  
javaaddpath  
javaArray  
javachk  
javaclasspath  
javaMethod  
javaObject

javarmpath  
keyboard  
keys (Map)  
kron  
last (MException)  
lasterr  
lasterror  
lastwarn  
lcm  
ldl  
ldivide, rdivide  
le  
legend  
legendre  
length  
length (Map)  
length (serial)  
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)  
load (serial)  
loadlibrary  
loadobj  
log  
log10  
log1p  
log2  
logical  
loglog  
logm  
logspace  
lookfor  
lower  
ls  
lscov  
lsqnonneg  
lsqr  
lt  
lu  
luinc  
magic  
makehgtform  
containers.Map  
mat2cell  
mat2str  
material  
matlabcolon (matlab:)  
matlabrc  
matlabroot  
matlab (UNIX)

matlab (Windows)  
max  
max (timeseries)  
MaximizeCommandWindow  
maxNumCompThreads  
mean  
mean (timeseries)  
median  
median (timeseries)  
memmapfile  
memory  
menu  
mesh, meshc, meshz  
meshgrid  
meta.class  
meta.class.fromName  
meta.DynamicProperty  
meta.event  
meta.method  
meta.package  
meta.package.fromName  
meta.package.getAllPackages  
meta.property  
metaclass  
methods  
methodsview  
mex  
mex.getCompilerConfigurations  
MException  
mexext  
mfilename  
mget  
min  
min (timeseries)  
MinimizeCommandWindow  
minres  
mislocked

---

mkdir  
mkdir (ftp)  
mkpp  
mldivide \, mrdivide /  
mlint  
mlintrpt  
mlock  
mmfileinfo  
mmreader  
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  
ne (MException)  
netcdf  
netcdf.abort

netcdf.close  
netcdf.copyAtt  
netcdf.create  
netcdf.defDim  
netcdf.defVar  
netcdf.delAtt  
netcdf.endDef  
netcdf.getAtt  
netcdf.getConstant  
netcdf.getConstantNames  
netcdf.getVar  
netcdf.inq  
netcdf.inqAtt  
netcdf.inqAttID  
netcdf.inqAttName  
netcdf.inqDim  
netcdf.inqDimID  
netcdf.inqLibVers  
netcdf.inqVar  
netcdf.inqVarID  
netcdf.open  
netcdf.putAtt  
netcdf.putVar  
netcdf.reDef  
netcdf.renameAtt  
netcdf.renameDim  
netcdf.renameVar  
netcdf.setDefaultFormat  
netcdf.setFill  
netcdf.sync  
newplot  
nextpow2  
nnz  
noanimate  
nonzeros  
norm  
normest



---

not  
notebook  
notify (handle)  
now  
nthroot  
null  
num2cell  
num2hex  
num2str  
numel  
nzmax  
ode15i  
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb  
odefile  
odeget  
odeset  
odextend  
onCleanup  
ones  
open  
openfig  
opengl  
openvar  
optimget  
optimset  
or  
ordeig  
orderfields  
ordqz  
ordschur  
orient  
orth  
otherwise  
pack  
padecoef  
pagesetupdlg  
pan

pareto  
parfor  
parse (inputParser)  
parseSoapResponse  
partialpath  
pascal  
patch  
Patch Properties  
path  
path2rc  
pathsep  
pathtool  
pause  
pbaspect  
pcg  
pchip  
pcode  
pcolor  
pdepe  
pdeval  
peaks  
perl  
perms  
permute  
persistent  
pi  
pie  
pie3  
pinv  
planerot  
playshow  
plot  
plot (timeseries)  
plot3  
plotbrowser  
plottedit  
plotmatrix

---

plottools  
plotyy  
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  
quadgk  
quadl  
quadv  
questdlg  
quit  
Quit (COM)  
quiver  
quiver3  
Quivergroup Properties  
qz  
rand  
rand (RandStream)  
randi  
randi (RandStream)  
randn  
randn (RandStream)  
randperm  
randperm (RandStream)  
RandStream  
RandStream (RandStream)  
rank  
rat, rats  
rbbox  
rcond  
read  
readasync  
real  
realloc  
realmax  
realmin  
realpow  
realsqrt  
record  
rectangle

---

Rectangle Properties  
rectint  
recycle  
reducepatch  
reducevolume  
refresh  
refreshdata  
regexp, regexpi  
regexprep  
regxptranslate  
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  
rgbplot  
ribbon  
rmapdata  
rmdir  
rmdir (ftp)  
rmfield  
rmpath

rmpref  
root object  
Root Properties  
roots  
rose  
rosser  
rot90  
rotate  
rotate3d  
round  
rref  
rsf2csf  
run  
save  
save (COM)  
save (serial)  
saveas  
saveobj  
savepath  
scatter  
scatter3  
Scattergroup Properties  
schur  
script  
sec  
secd  
sech  
selectmoveresize  
semilogx, semilogy  
sendmail  
serial  
serialbreak  
set  
set (COM)  
set (hgsetget)  
set (RandStream)  
set (serial)

---

set (timer)  
set (timeseries)  
set (tscollection)  
setabstime (timeseries)  
setabstime (tscollection)  
setappdata  
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)  
size (serial)  
size (timeseries)  
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  
streamslic  
streamtube  
strfind  
strings  
strjust  
strmatch  
strncmp, strncmpi  
stread  
strrep  
strtok  
strtrim  
struct  
struct2cell  
structfun  
strvcat  
sub2ind  
subplot  
subsasgn  
subsindex  
subspace  
subsref  
substruct  
subvolume  
sum

sum (timeseries)  
superiorto  
support  
surf, surfc  
surf2patch  
surface  
Surface Properties  
Surfaceplot Properties  
surfl  
surfnorm  
svd  
svds  
swapbytes  
switch  
symamd  
sybifact  
symmlq  
symrcm  
symvar  
synchronize  
syntax  
system  
tan  
tand  
tanh  
tar  
tempdir  
tempname  
tetramesh  
texlabel  
text  
Text Properties  
textread  
textscan  
textwrap  
throw (MException)  
throwAsCaller (MException)

---

tic, toc  
timer  
timerfind  
timerfindall  
timeseries  
title  
todatenum  
toeplitz  
toolboxdir  
trace  
transpose (timeseries)  
trapz  
treelayout  
treeplot  
tril  
trimesh  
triplequad  
triplet  
trisurf  
triu  
true  
try  
tscollection  
tsdata.event  
tsearch  
tsearchn  
tsprops  
tstool  
type  
typecast  
uibbuttongroup  
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  
unmkpp  
unregisterallevents  
unregisterevent  
untar  
unwrap  
unzip

---

upper  
urlread  
urlwrite  
usejava  
userpath  
validateattributes  
validatestring  
values (Map)  
vander  
var  
var (timeseries)  
varargin  
varargout  
vectorize  
ver  
verctrl  
verLessThan  
version  
vertcat  
vertcat (timeseries)  
vertcat (tscollection)  
view  
viewmtx  
visdiff  
volumebounds  
voronoi  
voronoin  
wait  
waitbar  
waitfor  
waitforbuttonpress  
warndlg  
warning  
waterfall  
wavinfo  
wavplay  
wavread

wavrecord  
wavwrite  
web  
weekday  
what  
whatsnew  
which  
while  
whitebg  
who, whos  
wilkinson  
winopen  
winqueryreg  
wk1finfo  
wk1read  
wk1write  
workspace  
xlabel, ylabel, zlabel  
xlim, ylim, zlim  
xlsfinfo  
xlsread  
xlswrite  
xmlread  
xmlwrite  
xor  
xslt  
zeros  
zip  
zoom

**Purpose** Prime factors

**Syntax** `f = factor(n)`

**Description** `f = factor(n)` returns a row vector containing the prime factors of `n`.

**Examples**

```
f = factor(123)
f =
     3     41
```

**See Also** `isprime`, `primes`

# factorial

---

**Purpose** Factorial function

**Syntax** `factorial(N)`

**Description** `factorial(N)`, for scalar `N`, is the product of all the integers from 1 to `N`, i.e. `prod(1:n)`. When `N` is an `N`-dimensional array, `factorial(N)` is the factorial for each element of `N`.

Since double precision numbers only have about 15 digits, the answer is only accurate for `n <= 21`. For larger `n`, the answer will have the right magnitude, and is accurate for the first 15 digits.

**See Also** `prod`



---

<b>Purpose</b>	Logical 0 (false)
<b>Syntax</b>	<code>false</code> <code>false(n)</code> <code>false(m, n)</code> <code>false(m, n, p, ...)</code> <code>false(size(A))</code>
<b>Description</b>	<p><code>false</code> is shorthand for <code>logical(0)</code>.</p> <p><code>false(n)</code> is an n-by-n matrix of logical zeros.</p> <p><code>false(m, n)</code> or <code>false([m, n])</code> is an m-by-n matrix of logical zeros.</p> <p><code>false(m, n, p, ...)</code> or <code>false([m n p ...])</code> is an m-by-n-by-p-by-... array of logical zeros.</p> <hr/> <p><b>Note</b> The size inputs <code>m, n, p, ...</code> should be nonnegative integers. Negative integers are treated as 0.</p> <hr/> <p><code>false(size(A))</code> is an array of logical zeros that is the same size as array <code>A</code>.</p>
<b>Remarks</b>	<code>false(n)</code> is much faster and more memory efficient than <code>logical(zeros(n))</code> .
<b>See Also</b>	<code>true</code> , <code>logical</code>

# fclose

---

**Purpose** Close one or more open files

**Syntax** `status = fclose(fid)`  
`status = fclose('all')`

**Description** `status = fclose(fid)` closes the specified file if it is open, returning 0 if successful and -1 if unsuccessful. Argument `fid` is a file identifier associated with an open file. (See `fopen` for a complete description of `fid`).

If `fid` does not represent an open file, or if it is equal to 0, 1, or 2, then `fclose` throws an error.

`status = fclose('all')` closes all open files (except standard input, output, and error), returning 0 if successful and -1 if unsuccessful.

**See Also** `ferror`, `fopen`, `fprintf`, `fread`, `frewind`, `fscanf`, `fseek`, `ftell`, `fwrite`

**Purpose** Disconnect serial port object from device

**Syntax** `fclose(obj)`

**Description** `fclose(obj)` disconnects `obj` from the device, where `obj` is a serial port object or an array of serial port objects.

**Remarks** If `obj` was successfully disconnected, then the `Status` property is configured to `closed` and the `RecordStatus` property is configured to `off`. You can reconnect `obj` to the device using the `fopen` function.

An error is returned if you issue `fclose` while data is being written asynchronously. In this case, you should abort the write operation with the `stopasync` function, or wait for the write operation to complete.

If you use the `help` command to display help for `fclose`, then you need to supply the pathname shown below.

```
help serial/fclose
```

**Example** This example creates the serial port object `s`, connects `s` to the device, writes and reads text data, and then disconnects `s` from the device using `fclose`.

```
s = serial('COM1');
fopen(s)
fprintf(s, '*IDN?')
idn = fscanf(s);
fclose(s)
```

At this point, the device is available to be connected to a serial port object. If you no longer need `s`, you should remove from memory with the `delete` function, and remove it from the workspace with the `clear` command.

**See Also** **Functions**

`clear`, `delete`, `fopen`, `stopasync`

# fclose (serial)

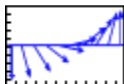
---


## Properties

RecordStatus, Status

**Purpose**

Plot velocity vectors

**GUI Alternatives**

Use the Plot Selector  to graph selected variables in the Workspace Browser and the Plot Catalog, accessed from the Figure Palette. Directly manipulate graphs in *plot edit* mode, and modify them using the Property Editor. For details, see “Working in Plot Edit Mode”, and “The Figure Palette” in the MATLAB Graphics documentation, and also Creating Graphics from the Workspace Browser in the MATLAB Desktop documentation.

**Syntax**

```
feather(U,V)
feather(Z)
feather(...,LineStyle)
feather(axes_handle,...)
h = feather(...)
```

**Description**

A feather plot displays vectors emanating from equally spaced points along a horizontal axis. You express the vector components relative to the origin of the respective vector.

`feather(U,V)` displays the vectors specified by `U` and `V`, where `U` contains the  $x$  components as relative coordinates, and `V` contains the  $y$  components as relative coordinates.

`feather(Z)` displays the vectors specified by the complex numbers in `Z`. This is equivalent to `feather(real(Z), imag(Z))`.

`feather(...,LineStyle)` draws a feather plot using the line type, marker symbol, and color specified by `LineStyle`.

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

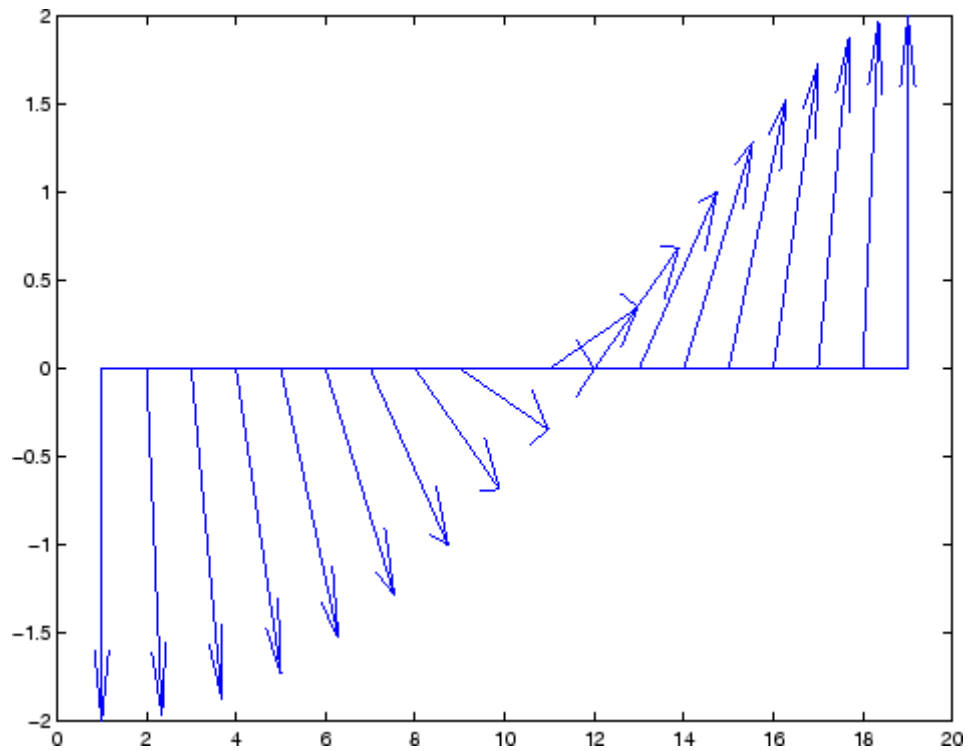
`h = feather(...)` returns the handles to line objects in `h`.

# feather

## Examples

Create a feather plot showing the direction of theta.

```
theta = (-90:10:90)*pi/180;  
r = 2*ones(size(theta));  
[u,v] = pol2cart(theta,r);  
feather(u,v);
```



## See Also

compass, LineSpec, rose

“Direction and Velocity Plots” on page 1-93 for related functions

**Purpose** Test for end-of-file

**Syntax** eofstat = feof(fid)

**Description** eofstat = feof(fid) returns 1 if the end-of-file indicator for the file fid has been set and 0 otherwise. (See fopen for a complete description of fid.)

The end-of-file indicator is set when there is no more input from the file.

**See Also** fopen

# ferror

---

**Purpose** Query the MATLAB software about errors in file input or output

**Syntax**

```
message = ferror(fid)
message = ferror(fid, 'clear')
[message,errno] = ferror(...)
```

**Description**

`message = ferror(fid)` returns the error string `message`. Argument `fid` is a file identifier associated with an open file (see `fopen` for a complete description of `fid`).

`message = ferror(fid, 'clear')` clears the error indicator for the specified file.

`[message,errno] = ferror(...)` returns the error status number `errno` of the most recent file I/O operation associated with the specified file.

If the most recent I/O operation performed on the specified file was successful, the value of `message` is empty and `ferror` returns an `errno` value of 0.

A nonzero `errno` indicates that an error occurred in the most recent file I/O operation. The value of `message` is a string that can contain information about the nature of the error. If the message is not helpful, consult the C run-time library manual for your host operating system for further details.

**See Also** `fclose`, `fopen`, `fprintf`, `fread`, `fscanf`, `fseek`, `ftell`, `fwrite`



**Purpose** Evaluate function

**Syntax** `[y1, y2, ...] = feval(fhandle, x1, ..., xn)`  
`[y1, y2, ...] = feval(function, x1, ..., xn)`

**Description** `[y1, y2, ...] = feval(fhandle, x1, ..., xn)` evaluates the function handle, `fhandle`, using arguments `x1` through `xn`. If the function handle is bound to more than one built-in or M-file, (that is, it represents a set of overloaded functions), then the data type of the arguments `x1` through `xn` determines which function is dispatched to.

---

**Note** It is not necessary to use `feval` to call a function by means of a function handle. This is explained in “Calling a Function Using Its Handle” in the MATLAB Programming Fundamentals documentation.

---

`[y1, y2, ...] = feval(function, x1, ..., xn)`. If `function` is a quoted string containing the name of a function (usually defined by an M-file), then `feval(function, x1, ..., xn)` evaluates that function at the given arguments. The `function` parameter must be a simple function name; it cannot contain path information.

**Remarks** The following two statements are equivalent.

```
[V,D] = eig(A)
[V,D] = feval(@eig, A)
```

Nested functions are not accessible to `feval`. To call a nested function, you must either call it directly by name, or construct a function handle for it using the `@` operator.

**Examples** The following example passes a function handle, `fhandle`, in a call to `fminbnd`. The `fhandle` argument is a handle to the `humps` function.

```
fhandle = @humps;
x = fminbnd(fhandle, 0.3, 1);
```

# feval

---

The `fminbnd` function uses `feval` to evaluate the function handle that was passed in.

```
function [xf, fval, exitflag, output] = ...  
    fminbnd(funfcn, ax, bx, options, varargin)  
    .  
    .  
    .  
    fx = feval(funfcn, x, varargin{:});
```

## See Also

`assignin`, `function_handle`, `functions`, `builtin`, `eval`, `evalin`

**Purpose** Evaluate MATLAB function in Automation server

**Syntax**

**MATLAB Client**

```
result = h.Feval('functionname', numout, arg1, arg2, ...)
result = Feval(h, 'functionname', numout, arg1, arg2, ...)
result = invoke(h, 'Feval', 'functionname', numout, ...
arg1, arg2, ...)
```

**Method Signatures**

```
HRESULT Feval([in] BSTR functionname, [in] long nargout,
[out] VARIANT* result, [in, optional] VARIANT arg1, arg2, ...)
```

**Microsoft Visual Basic Client**

```
Feval(String functionname, long numout,
arg1, arg2, ...) As Object
```

**Description**

Feval executes the MATLAB function specified by the string functionname in the Automation server attached to handle h.

Indicate the number of outputs to be returned by the function in a 1-by-1 double array, numout. The server returns output from the function in the cell array, result.

You can specify as many as 32 input arguments to be passed to the function. These arguments follow numout in the Feval argument list. There are four ways to pass an argument to the function being evaluated.

Passing Mechanism	Description
Pass the value itself	To pass any numeric or string value, specify the value in the Feval argument list: <pre>a = h.Feval('sin', 1, -pi:0.01:pi);</pre>

# Feval (COM)

Passing Mechanism	Description
Pass a client variable	To pass an argument that is assigned to a variable in the client, specify the variable name alone:  <pre>x = -pi:0.01:pi; a = h.Feval('sin', 1, x);</pre>
Reference a server variable	To reference a variable that is defined in the server, specify the variable name followed by an equals (=) sign:  <pre>h.PutWorkspaceData('x', 'base', -pi:0.01:pi); a = h.Feval('sin', 1, 'x=');</pre> Note that the server variable is not reassigned.

## Remarks

If you want output from `Feval` to be displayed at the client window, you must assign a returned value.

Server function names, like `Feval`, are case sensitive when using the first two syntaxes shown in the Syntax section.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

COM functions are available on Microsoft Windows systems only.

## Examples

### Passing Arguments – MATLAB Client

This section contains a number of examples showing how to use `Feval` to execute MATLAB commands on a MATLAB Automation server.

- Concatenate two strings in the server by passing the input strings in a call to `strcat` through `Feval` (`strcat` deletes trailing spaces; use leading spaces):

```
h = actxserver('matlab.application');  
a = h.Feval('strcat', 1, 'hello', ' world')
```

MATLAB displays:

```
a =  
    'hello world'
```

- Perform the same concatenation, passing a string and a local variable `clistr` that contains the second string:

```
clistr = ' world';  
a = h.Feval('strcat', 1, 'hello', clistr)
```

MATLAB displays:

```
a =  
    'hello world'
```

- In this example, the variable `srvstr` is defined in the server, not the client. Putting an equals sign after a variable name (e.g., `srvstr=`) indicates that it is a server variable, and that MATLAB software should not expect the variable to be defined on the client:

```
% Define the variable srvstr on the server.  
h.PutCharArray('srvstr', 'base', ' world')  
  
% Pass the name of the server variable using 'name=' syntax  
a = h.Feval('strcat', 1, 'hello', 'srvstr=')
```

MATLAB displays:

```
a =  
    'hello world'
```

## Visual Basic .NET Client

Here are the same examples shown above, but written for a Visual Basic .NET client. These examples return the same strings as shown above.

- Pass the two strings to the MATLAB function `strcat` on the server:

```
Dim Matlab As Object  
Dim out As Object
```

# Feval (COM)

---

```
Matlab = CreateObject("matlab.application")
out = Matlab.Feval("strcat", 1, "hello", " world")
```

- Define `clistr` locally and pass this variable:

```
Dim clistr As String
clistr = " world"
out = Matlab.Feval("strcat", 1, "hello", clistr)
```

- Pass the name of a variable defined on the server:

```
Matlab.PutCharArray("srvstr", "base", " world")
out = Matlab.Feval("strcat", 1, "hello", "srvstr=")
```

**Feval Return Values – MATLAB Client.** `Feval` returns data from the evaluated function in a cell array. The cell array has one row for every return value. You can control how many values are returned using the second input argument to `Feval`, as shown in this example.

The second argument in the following example specifies that `Feval` return three outputs from the `fileparts` function. As is the case here, you can request fewer than the maximum number of return values for a function (`fileparts` can return up to four):

```
a = h.Feval('fileparts', 3, 'd:\work\ConsoleApp.cpp')
```

MATLAB displays:

```
a =
    'd:\work'
    'ConsoleApp'
    '.cpp'
```

Convert the returned values from the cell array `a` to char arrays:

```
a{:}
```

MATLAB displays:

```
ans =  
d:\work
```

```
ans =  
ConsoleApp
```

```
ans =  
.cpp
```

## Feval Return Values – Visual Basic .NET Client

Here is the same example, but coded in Visual Basic. Define the argument returned by Feval as an Object.

```
Dim Matlab As Object  
Dim out As Object  
Matlab = CreateObject("matlab.application")  
out = Matlab.Feval("fileparts", 3, "d:\work\ConsoleApp.cpp")
```

## See Also

Execute, PutFullMatrix, GetFullMatrix, PutCharArray, GetCharArray

**Purpose** Discrete Fourier transform

**Syntax**  
`Y = fft(X)`  
`Y = fft(X,n)`  
`Y = fft(X,[],dim)`  
`Y = fft(X,n,dim)`

**Definition** The functions `Y=fft(x)` and `y=ifft(X)` implement the transform and inverse transform pair given for vectors of length  $N$  by:

$$X(k) = \sum_{j=1}^N x(j)\omega_N^{(j-1)(k-1)}$$

$$x(j) = (1/N) \sum_{k=1}^N X(k)\omega_N^{-(j-1)(k-1)}$$

where

$$\omega_N = e^{(-2\pi i)/N}$$

is an  $N$ th root of unity.

**Description** `Y = fft(X)` returns the discrete Fourier transform (DFT) of vector  $X$ , computed with a fast Fourier transform (FFT) algorithm.

If  $X$  is a matrix, `fft` returns the Fourier transform of each column of the matrix.

If  $X$  is a multidimensional array, `fft` operates on the first nonsingleton dimension.

`Y = fft(X,n)` returns the  $n$ -point DFT. If the length of  $X$  is less than  $n$ ,  $X$  is padded with trailing zeros to length  $n$ . If the length of  $X$  is greater than  $n$ , the sequence  $X$  is truncated. When  $X$  is a matrix, the length of the columns are adjusted in the same manner.

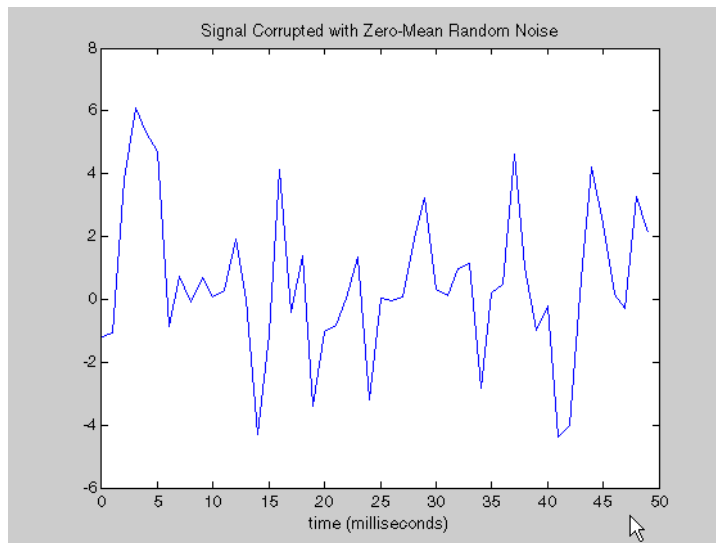


$Y = \text{fft}(X, [], \text{dim})$  and  $Y = \text{fft}(X, n, \text{dim})$  applies the FFT operation across the dimension  $\text{dim}$ .

## Examples

A common use of Fourier transforms is to find the frequency components of a signal buried in a noisy time domain signal. Consider data sampled at 1000 Hz. Form a signal containing a 50 Hz sinusoid of amplitude 0.7 and 120 Hz sinusoid of amplitude 1 and corrupt it with some zero-mean random noise:

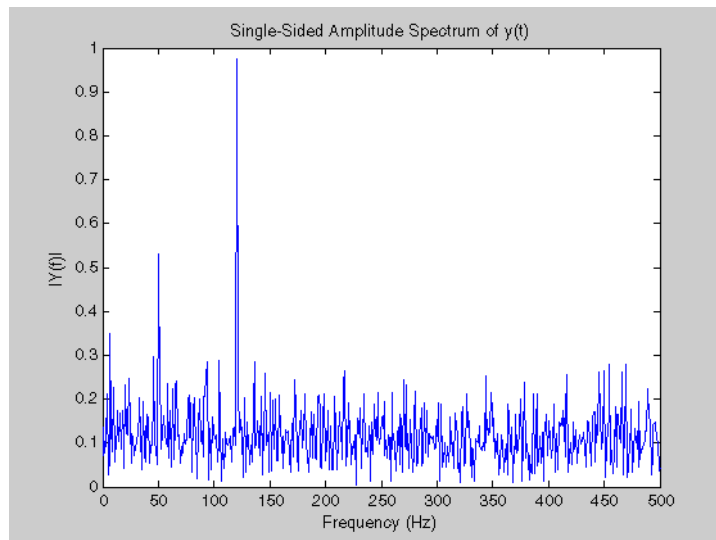
```
Fs = 1000; % Sampling frequency
T = 1/Fs; % Sample time
L = 1000; % Length of signal
t = (0:L-1)*T; % Time vector
% Sum of a 50 Hz sinusoid and a 120 Hz sinusoid
x = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
y = x + 2*randn(size(t)); % Sinusoids plus noise
plot(Fs*t(1:50),y(1:50))
title('Signal Corrupted with Zero-Mean Random Noise')
xlabel('time (milliseconds)')
```



It is difficult to identify the frequency components by looking at the original signal. Converting to the frequency domain, the discrete Fourier transform of the noisy signal  $y$  is found by taking the fast Fourier transform (FFT):

```
NFFT = 2^nextpow2(L); % Next power of 2 from length of y
Y = fft(y,NFFT)/L;
f = Fs/2*linspace(0,1,NFFT/2+1);

% Plot single-sided amplitude spectrum.
plot(f,2*abs(Y(1:NFFT/2+1)))
title('Single-Sided Amplitude Spectrum of y(t)')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
```



The main reason the amplitudes are not exactly at 0.7 and 1 is because of the noise. Several executions of this code (including recomputation of  $y$ ) will produce different approximations to 0.7 and 1. The other reason is that you have a finite length signal. Increasing  $L$  from 1000 to

10000 in the example above will produce much better approximations on average.

## Algorithm

The FFT functions (`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, `ifftn`) are based on a library called FFTW [3],[4]. To compute an  $N$ -point DFT when  $N$  is composite (that is, when  $N = N_1 N_2$ ), the FFTW library decomposes the problem using the Cooley-Tukey algorithm [1], which first computes  $N_1$  transforms of size  $N_2$ , and then computes  $N_2$  transforms of size  $N_1$ . The decomposition is applied recursively to both the  $N_1$ - and  $N_2$ -point DFTs until the problem can be solved using one of several machine-generated fixed-size "codelets." The codelets in turn use several algorithms in combination, including a variation of Cooley-Tukey [5], a prime factor algorithm [6], and a split-radix algorithm [2]. The particular factorization of  $N$  is chosen heuristically.

When  $N$  is a prime number, the FFTW library first decomposes an  $N$ -point problem into three  $(N - 1)$ -point problems using Rader's algorithm [7]. It then uses the Cooley-Tukey decomposition described above to compute the  $(N - 1)$ -point DFTs.

For most  $N$ , real-input DFTs require roughly half the computation time of complex-input DFTs. However, when  $N$  has large prime factors, there is little or no speed difference.

The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

---

**Note** You might be able to increase the speed of `fft` using the utility function `fftw`, which controls the optimization of the algorithm used to compute an FFT of a particular size and dimension.

---

**Data Type Support**

fft supports inputs of data types `double` and `single`. If you call `fft` with the syntax `y = fft(X, ...)`, the output `y` has the same data type as the input `X`.

**See Also**

`fft2`, `fftn`, `fftw`, `fftshift`, `ifft`  
`dftmtx`, `filter`, and `freqz` in the Signal Processing Toolbox

**References**

- [1] Cooley, J. W. and J. W. Tukey, "An Algorithm for the Machine Computation of the Complex Fourier Series," *Mathematics of Computation*, Vol. 19, April 1965, pp. 297-301.
- [2] Duhamel, P. and M. Vetterli, "Fast Fourier Transforms: A Tutorial Review and a State of the Art," *Signal Processing*, Vol. 19, April 1990, pp. 259-299.
- [3] FFTW (<http://www.fftw.org>)
- [4] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.
- [5] Oppenheim, A. V. and R. W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 611.
- [6] Oppenheim, A. V. and R. W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 619.
- [7] Rader, C. M., "Discrete Fourier Transforms when the Number of Data Samples Is Prime," *Proceedings of the IEEE*, Vol. 56, June 1968, pp. 1107-1108.

---

<b>Purpose</b>	2-D discrete Fourier transform
<b>Syntax</b>	$Y = \text{fft2}(X)$ $Y = \text{fft2}(X,m,n)$
<b>Description</b>	<p><math>Y = \text{fft2}(X)</math> returns the two-dimensional discrete Fourier transform (DFT) of <math>X</math>, computed with a fast Fourier transform (FFT) algorithm. The result <math>Y</math> is the same size as <math>X</math>.</p> <p><math>Y = \text{fft2}(X,m,n)</math> truncates <math>X</math>, or pads <math>X</math> with zeros to create an <math>m</math>-by-<math>n</math> array before doing the transform. The result is <math>m</math>-by-<math>n</math>.</p>
<b>Algorithm</b>	<p><math>\text{fft2}(X)</math> can be simply computed as</p> <pre>fft(fft(X).')'</pre> <p>This computes the one-dimensional DFT of each column <math>X</math>, then of each row of the result. The execution time for <code>fft</code> depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.</p> <hr/> <p><b>Note</b> You might be able to increase the speed of <code>fft2</code> using the utility function <code>fftw</code>, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.</p> <hr/>
<b>Data Type Support</b>	<code>fft2</code> supports inputs of data types <code>double</code> and <code>single</code> . If you call <code>fft2</code> with the syntax $y = \text{fft2}(X, \dots)$ , the output $y$ has the same data type as the input $X$ .
<b>See Also</b>	<code>fft</code> , <code>fftn</code> , <code>fftw</code> , <code>fftshift</code> , <code>ifft2</code>

**Purpose** N-D discrete Fourier transform

**Syntax** `Y = fftn(X)`  
`Y = fftn(X,siz)`

**Description** `Y = fftn(X)` returns the discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`Y = fftn(X,siz)` pads `X` with zeros, or truncates `X`, to create a multidimensional array of size `siz` before performing the transform. The size of the result `Y` is `siz`.

**Algorithm** `fftn(X)` is equivalent to

```
Y = X;
for p = 1:length(size(X))
    Y = fft(Y,[],p);
end
```

This computes in-place the one-dimensional fast Fourier transform along each dimension of `X`. The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

---

**Note** You might be able to increase the speed of `fftn` using the utility function `fftw`, which controls the optimization of the algorithm used to compute an FFT of a particular size and dimension.

---

**Data Type Support** `fftn` supports inputs of data types `double` and `single`. If you call `fftn` with the syntax `y = fftn(X, ...)`, the output `y` has the same data type as the input `X`.

**See Also**      `fft`, `fft2`, `fftn`, `fftw`, `ifftn`

# fftshift

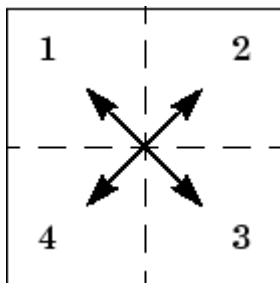
---

**Purpose** Shift zero-frequency component to center of spectrum

**Syntax**  
 $Y = \text{fftshift}(X)$   
 $Y = \text{fftshift}(X, \text{dim})$

**Description**  $Y = \text{fftshift}(X)$  rearranges the outputs of `fft`, `fft2`, and `fftn` by moving the zero-frequency component to the center of the array. It is useful for visualizing a Fourier transform with the zero-frequency component in the middle of the spectrum.

For vectors, `fftshift(X)` swaps the left and right halves of  $X$ . For matrices, `fftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth.

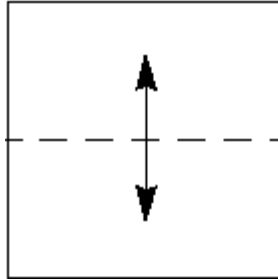


For higher-dimensional arrays, `fftshift(X)` swaps “half-spaces” of  $X$  along each dimension.

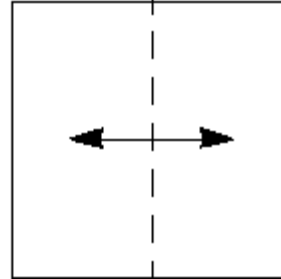
$Y = \text{fftshift}(X, \text{dim})$  applies the `fftshift` operation along the dimension `dim`.



For dim = 1:



For dim = 2:




---

**Note** `ifftshift` will undo the results of `fftshift`. If the matrix  $X$  contains an odd number of elements, `ifftshift(fftshift(X))` must be done to obtain the original  $X$ . Simply performing `fftshift(X)` twice will not produce  $X$ .

---

## Examples

For any matrix  $X$

```
Y = fft2(X)
```

has  $Y(1,1) = \text{sum}(\text{sum}(X))$ ; the zero-frequency component of the signal is in the upper-left corner of the two-dimensional FFT. For

```
Z = fftshift(Y)
```

this zero-frequency component is near the center of the matrix.

The difference between `fftshift` and `ifftshift` is important for input sequences of odd-length.

```
N = 5;
X = 0:N-1;
Y = fftshift(fftshift(X));
Z = ifftshift(fftshift(X));
```

Notice that  $Z$  is a correct replica of  $X$ , but  $Y$  is not.

# fftshift

---

```
isequal(X,Y),isequal(X,Z)
```

```
ans =
```

```
0
```

```
ans =
```

```
1
```

## See Also

`circshift`, `fft`, `fft2`, `fftn`, `ifftshift`

**Purpose**

Interface to FFTW library run-time algorithm tuning control

**Syntax**

```
fftw('planner', method)
method = fftw('planner')
str = fftw('dwisdom')
str = fftw('swisdom')
fftw('dwisdom', str)
fftw('swisdom', str)
```

**Description**

fftw enables you to optimize the speed of the MATLAB FFT functions `fft`, `ifft`, `fft2`, `ifft2`, `fftn`, and `ifftn`. You can use `fftw` to set options for a tuning algorithm that experimentally determines the fastest algorithm for computing an FFT of a particular size and dimension at run time. MATLAB software records the optimal algorithm in an internal data base and uses it to compute FFTs of the same size throughout the current session. The tuning algorithm is part of the FFTW library that MATLAB software uses to compute FFTs.

`fftw('planner', method)` sets the method by which the tuning algorithm searches for a good FFT algorithm when the dimension of the FFT is not a power of 2. You can specify `method` to be one of the following. The default method is `estimate`:

- 'estimate'
- 'measure'
- 'patient'
- 'exhaustive'
- 'hybrid'

When you call `fftw('planner', method)`, the next time you call one of the FFT functions, such as `fft`, the tuning algorithm uses the specified method to optimize the FFT computation. Because the tuning involves trying different algorithms, the first time you call an FFT function, it might run more slowly than if you did not call `fftw`. However,

subsequent calls to any of the FFT functions, for a problem of the same size, often run more quickly than they would without using `fftw`.

---

**Note** The FFT functions only use the optimal FFT algorithm during the current MATLAB session. “Reusing Optimal FFT Algorithms” on page 2-1168 explains how to reuse the optimal algorithm in a future MATLAB session.

---

If you set the method to `'estimate'`, the FFTW library does not use run-time tuning to select the algorithms. The resulting algorithms might not be optimal.

If you set the method to `'measure'`, the FFTW library experiments with many different algorithms to compute an FFT of a given size and chooses the fastest. Setting the method to `'patient'` or `'exhaustive'` has a similar result, but the library experiments with even more algorithms so that the tuning takes longer the first time you call an FFT function. However, subsequent calls to FFT functions are faster than with `'measure'`.

If you set `'planner'` to `'hybrid'`, MATLAB software

- Sets method to `'measure'` method for FFT dimensions 8192 or smaller.
- Sets method to `'estimate'` for FFT dimensions greater than 8192.

`method = fftw('planner')` returns the current planner method.

`str = fftw('dwisdom')` returns the information in the FFTW library's internal double-precision database as a string. The string can be saved and then later reused in a subsequent MATLAB session using the next syntax.

`str = fftw('swisdom')` returns the information in the FFTW library's internal single-precision database as a string.

`fftw('dwisdom', str)` loads fftw wisdom represented by the string `str` into the FFTW library's internal double-precision wisdom database. `fftw('dwisdom', '')` or `fftw('dwisdom', [])` clears the internal wisdom database.

`fftw('swisdom', str)` loads fftw wisdom represented by the string `str` into the FFTW library's internal single-precision wisdom database. `fftw('swisdom', '')` or `fftw('swisdom', [])` clears the internal wisdom database.

---

**Note on large powers of 2** For FFT dimensions that are powers of 2, between  $2^{14}$  and  $2^{22}$ , MATLAB software uses special preloaded information in its internal database to optimize the FFT computation. No tuning is performed when the dimension of the FFT is a power of 2, unless you clear the database using the command `fftw('wisdom', [])`.

---

For more information about the FFTW library, see <http://www.fftw.org>.

## Example

### Comparison of Speed for Different Planner Methods

The following example illustrates the run times for different settings of `planner`. The example first creates some data and applies `fft` to it using the default method, `estimate`.

```
t=0:.001:5;
x = sin(2*pi*50*t)+sin(2*pi*120*t);
y = x + 2*randn(size(t));

tic; Y = fft(y,1458); toc
Elapsed time is 0.000521 seconds.
```

If you execute the commands

```
tic; Y = fft(y,1458); toc
Elapsed time is 0.000151 seconds.
```

a second time, MATLAB software reports the elapsed time as essentially 0. To measure the elapsed time more accurately, you can execute the command `Y = fft(y,1458)` 1000 times in a loop.

```
tic; for k=1:1000
Y = fft(y,1458);
end; toc
Elapsed time is 0.056532 seconds.
```

This tells you that it takes on order of 1/10000 of a second to execute `fft(y, 1458)` a single time.

For comparison, set planner to `patient`. Since this planner explores possible algorithms more thoroughly than `hybrid`, the first time you run `fft`, it takes longer to compute the results.

```
fftw('planner','patient')
tic;Y = fft(y,1458);toc
Elapsed time is 0.100637 seconds.
```

However, the next time you call `fft`, it runs at approximately the same speed as before you ran the method `patient`.

```
tic;for k=1:1000
Y=fft(y,1458);
end;toc
Elapsed time is 0.057209 seconds.
```

## Reusing Optimal FFT Algorithms

In order to use the optimized FFT algorithm in a future MATLAB session, first save the “wisdom” using the command

```
str = fftw('wisdom')
```

You can save `str` for a future session using the command

```
save str
```

The next time you open a MATLAB session, load `str` using the command

```
load str
```

and then reload the “wisdom” into the FFTW database using the command

```
fftw('wisdom', str)
```

**See Also**

`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, `ifftn`, `fftshift`.

# fgetl

---

**Purpose** Read line from file, discarding newline character

**Syntax** `tline = fgetl(fid)`

**Description** `tline = fgetl(fid)` returns the next line of the file associated with the file identifier `fid`. If `fgetl` encounters the end-of-file indicator, it returns `-1`. (See `fopen` for a complete description of `fid`.) `fgetl` is intended for use with files that contain newline characters.

The MATLAB software reads characters using the encoding scheme associated with the file. See `fopen` for more information.

The returned string `tline` does not include the line terminator(s) with the text line. To obtain the line terminators, use `fgets`.

**Examples** The example reads every line of the M-file `fgetl.m`.

```
fid=fopen('fgetl.m');
while 1
    tline = fgetl(fid);
    if ~ischar(tline), break, end
    disp(tline)
end
fclose(fid);
```

**See Also** `fgets`, `fread`



**Purpose** Read line of text from device and discard terminator

**Syntax**

```
tline = fgetl(obj)
[tline,count] = fgetl(obj)
[tline,count,msg] = fgetl(obj)
```

**Description** `tline = fgetl(obj)` reads one line of text from the device connected to the serial port object, `obj`, and returns the data to `tline`. This returned data does not include the terminator with the text line. To include the terminator, use `fgets`.

`[tline,count] = fgetl(obj)` returns the number of values read to `count`, including the terminator.

`[tline,count,msg] = fgetl(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

**Remarks** Before you can read text from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgetl` is issued.

If you use the `help` command to display help for `fgetl`, then you need to supply the pathname shown below.

```
help serial/fgetl
```

## Rules for Completing a Read Operation with fgetl

A read operation with `fgetl` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.

# fgetl (serial)

---

- The time specified by the Timeout property passes.
- The input buffer is filled.

## Example

Create the serial port object `s`, connect `s` to a Tektronix® TDS 210 oscilloscope, and write the RS232? command with the `fprintf` function. RS232? instructs the scope to return serial port communications settings.

```
s = serial('COM1');  
fopen(s)  
fprintf(s, 'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable  
ans =  
    17
```

Use `fgetl` to read the data returned from the previous write operation, and discard the terminator.

```
settings = fgetl(s)  
settings =  
9600;0;0;NONE;LF  
length(settings)  
ans =  
    16
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)  
delete(s)  
clear s
```

## See Also

## Functions

fgets, fopen

## Properties

BytesAvailable, InputBufferSize, ReadAsyncMode, Status, Terminator, Timeout, ValuesReceived

# fgets

---

**Purpose** Read line from file, keeping newline character

**Syntax**  
`tline = fgets(fid)`  
`tline = fgets(fid, nchar)`

**Description** `tline = fgets(fid)` returns the next line of the file associated with file identifier `fid`. If `fgets` encounters the end-of-file indicator, it returns `-1`. (See `fopen` for a complete description of `fid`.) `fgets` is intended for use with files that contain newline characters.

The MATLAB software reads characters using the encoding scheme associated with the file. See `fopen` for more information.

The returned string `tline` includes the line terminators associated with the text line. To obtain the string without the line terminators, use `fgetl`.

`tline = fgets(fid, nchar)` returns at most `nchar` characters of the next line. No additional characters are read after the line terminators or an end-of-file.

**See Also** `fgetl`, `fread`

**Purpose** Read line of text from device and include terminator

**Syntax**

```
tline = fgets(obj)
[tline,count] = fgets(obj)
[tline,count,msg] = fgets(obj)
```

**Description** `tline = fgets(obj)` reads one line of text from the device connected to the serial port object, `obj`, and returns the data to `tline`. This returned data includes the terminator with the text line. To exclude the terminator, use `fgetl`.

`[tline,count] = fgets(obj)` returns the number of values read to `count`, including the terminator.

`[tline,count,msg] = fgets(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

**Remarks** Before you can read text from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgets` is issued.

If you use the `help` command to display help for `fgets`, then you need to supply the pathname shown below.

```
help serial/fgets
```

## Rules for Completing a Read Operation with fgets

A read operation with `fgets` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.

# fgets (serial)

---

- The time specified by the Timeout property passes.
- The input buffer is filled.

## Example

Create the serial port object `s`, connect `s` to a Tektronix TDS 210 oscilloscope, and write the RS232? command with the `fprintf` function. RS232? instructs the scope to return serial port communications settings.

```
s = serial('COM1');  
fopen(s)  
fprintf(s, 'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable  
ans =  
    17
```

Use `fgets` to read the data returned from the previous write operation, and include the terminator.

```
settings = fgets(s)  
settings =  
9600;0;0;NONE;LF  
length(settings)  
ans =  
    17
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)  
delete(s)  
clear s
```

## See Also

## Functions

fgetc1, fopen

## Properties

BytesAvailable, BytesAvailableFcn, InputBufferSize, Status, Terminator, Timeout, ValuesReceived

# fieldnames

---

**Purpose** Field names of structure, or public fields of object

**Syntax**

```
names = fieldnames(s)
names = fieldnames(obj)
names = fieldnames(obj, '-full')
```

**Description**

`names = fieldnames(s)` returns a cell array of strings containing the structure field names associated with the structure `s`.

`names = fieldnames(obj)` returns a cell array of strings containing field names for `obj`. If `obj` is a MATLAB object, then return value `names` contains the names of the fields in that object. If `obj` is an object of the Java programming language, then `names` contains the names of the public fields. MATLAB objects may override `fieldnames` and define their own behavior.

`names = fieldnames(obj, '-full')` returns a cell array of strings containing the name, type, attributes, and inheritance of each field associated with `obj`, which is a COM or Java object. Note that `fieldnames` does not support the `full` option for MATLAB objects.

**Examples** Given the structure

```
mystr(1,1).name = 'alice';
mystr(1,1).ID = 0;
mystr(2,1).name = 'gertrude';
mystr(2,1).ID = 1
```

the command `n = fieldnames(mystr)` yields

```
n =
    'name'
    'ID'
```

In another example, if `i` is an object of Java class `java.awt.Integer`, the command `fieldnames(i)` lists the properties of `i`.

```
i = java.lang.Integer(0);
```



```
fieldnames(i)
```

MATLAB displays:

```
ans =  
    'MIN_VALUE'  
    'MAX_VALUE'  
    'TYPE'  
    'SIZE'
```

## See Also

setfield, getfield, isfield, orderfields, rmfield, “Using Dynamic Field Names”

# figure

---

**Purpose** Create figure graphics object

**Syntax**

```
figure
figure('PropertyName',propertyvalue,...)
figure(h)
h = figure(...)
```

**Description** `figure` creates figure graphics objects. Figure objects are the individual windows on the screen in which the MATLAB software displays graphical output.

`figure` creates a new figure object using default property values.

`figure('PropertyName',propertyvalue,...)` creates a new figure object using the values of the properties specified. MATLAB uses default values for any properties that you do not explicitly define as arguments.

`figure(h)` does one of two things, depending on whether or not a figure with handle `h` exists. If `h` is the handle to an existing figure, `figure(h)` makes the figure identified by `h` the current figure, makes it visible, and raises it above all other figures on the screen. The current figure is the target for graphics output. If `h` is not the handle to an existing figure, but is an integer, `figure(h)` creates a figure and assigns it the handle `h`. `figure(h)` where `h` is not the handle to a figure, and is not an integer, is an error.

`h = figure(...)` returns the handle to the figure object.

**Remarks** To create a figure object, MATLAB creates a new window whose characteristics are controlled by default figure properties (both factory installed and user defined) and properties specified as arguments. See the [Figure Properties](#) section for a description of these properties.

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

Use `set` to modify the properties of an existing figure or `get` to query the current values of figure properties.

The `gcf` command returns the handle to the current figure and is useful as an argument to the `set` and `get` commands.

Figures can be docked in the desktop. The `Dockable` property determines whether you can dock the figure.

### **Making a Figure Current**

The current figure is the target for graphics output. There are two ways to make a figure `h` the current figure.

- Make the figure `h` current, visible, and displayed on top of other figures:

```
figure(h)
```

- Make the figure `h` current, but do not change its visibility or stacking with respect to other figures:

```
set(0, 'CurrentFigure', h)
```

## **Examples**

### **Specifying Figure Size and Screen Location**

To create a figure window that is one quarter the size of your screen and is positioned in the upper left corner, use the root object's `ScreenSize` property to determine the size. `ScreenSize` is a four-element vector: `[left, bottom, width, height]`:

```
scrsz = get(0, 'ScreenSize');  
figure('Position', [1 scrsz(4)/2 scrsz(3)/2 scrsz(4)/2])
```

To position the full figure window including the menu bar, title bar, tool bars, and outer edges, use the `OuterPosition` property in the same manner.

### **Specifying the Figure Window Title**

You can add your own title to a figure by setting the `Name` property and you can turn off the figure number with the `NumberTitle` property:

```
figure('Name', 'Simulation Plot Window', 'NumberTitle', 'off')
```

# figure

---

See the **Figure Properties** section for a description of all figure properties.

## Setting Default Properties

You can set default figure properties only on the root object level.

```
set(0, 'DefaultFigureProperty', PropertyValue...)
```

where *Property* is the name of the figure property and *PropertyValue* is the value you are specifying. Use `set` and `get` to access figure properties.

## See Also

`axes`, `uicontrol`, `uimenu`, `close`, `clf`, `gcf`, `rootobject`

“Object Creation” on page 1-98 for related functions

Figure Properties descriptions of all figure properties

See “Figure Properties” in the MATLAB Graphics User Guide for more information on figures.

## Purpose

Define figure properties

## Modifying Properties

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

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

To change the default values of properties, see “Setting Default Property Values” in the Handle Graphics Objects documentation.

## Figure Property Descriptions

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

### Alphamap

m-by-1 matrix of alpha values

*Figure alphamap.* This property is an m-by-1 array of non-NaN alpha values. MATLAB accesses alpha values by their row number. For example, an index of 1 specifies the first alpha value, an index of 2 specifies the second alpha value, and so on. Alphamaps can be any length. The default alphamap contains 64 values that progress linearly from 0 to 1.

Alphamaps affect the rendering of surface, image, and patch objects, but do not affect other graphics objects.

### 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.

## Figure Properties

---

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

See the `close` and `delete` function reference pages for related information.

### `BusyAction`

`cancel` | `{queue}`

*Callback function interruption.* The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback functions. If there is a callback function executing, callback functions 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 function.
- `queue` — Queue the event that attempted to execute a second callback function until the current callback finishes.

### `ButtonDownFcn`

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

*Button press callback function.* A callback function that executes whenever you press a mouse button while the pointer is in the figure window, but not over a child object (i.e., `uicontrol`, `uipanel`, `axes`, or `axes child`). Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments

(handle of figure associated with the mouse button press and an empty event structure)

See the figure's `SelectionType` property to determine whether modifier keys were also pressed.

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

## Using the `ButtonDownFcn`

This example creates a figure and defines a function handle callback for the `ButtonDownFcn` property. When the user **Ctrl**-clicks the figure, the callback creates a new figure having the same callback.

Click to view in editor — This link opens the MATLAB Editor with the following example.

Click to run example — **Ctrl**-click the figure to create a new figure.

```
fh_cb = @newfig; % Create function handle for newfig function
figure('ButtonDownFcn',fh_cb);

function newfig(src,evt)
    if strcmp(get(src,'SelectionType'),'alt')
        figure('ButtonDownFcn',fh_cb)
    else
        disp('Use control-click to create a new figure')
    end
end
```

## Children

vector of handles

*Children of the figure.* A vector containing the handles of all axes, user-interface objects displayed within the figure. You can change

# Figure Properties

---

the order of the handles and thereby change the stacking of the objects on the display.

When an object's `HandleVisibility` property is set to `off`, it is not listed in its parent's `Children` property. See `HandleVisibility` for more information.

## Clipping

`{on} | off`

This property has no effect on figures.

## CloseRequestFcn

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

*Function executed on figure close.* This property defines a function that MATLAB executes whenever you issue the `close` command (either a `close(handle)` or a `close all`), when you close a figure window from the computer's window manager menu, or when you quit MATLAB.

The `CloseRequestFcn` provides a mechanism to intervene in the closing of a figure. It allows you to, for example, display a dialog box to ask a user to confirm or cancel the close operation or to prevent users from closing a figure that contains a GUI.

The basic mechanism is

- A user issues the `close` command from the command line, by closing the window from the computer's window manager menu, or by quitting MATLAB.
- The close operation executes the function defined by the figure `CloseRequestFcn`. The default function is named `closereq` and is predefined as

```
if isempty(gcbf)
    if length(dbstack) == 1
```



```
        warning('MATLAB:closereq', ...
            'Calling closereq from the command line ...
            is now obsolete, use close instead');
    end
    close force
else
    delete(gcf);
end
```

These statements unconditionally delete the current figure, destroying the window. `closereq` takes advantage of the fact that the `close` command makes all figures specified as arguments the current figure before calling the respective close request function.

Note that `closereq` honors the user's `ShowHiddenHandles` setting during figure deletion and will not delete hidden figures.

## Redefining the CloseRequestFcn

Define the `CloseRequestFcn` as a function handle. For example,

```
set(gcf, 'CloseRequestFcn', @my_closefcn)
```

Where `@my_closefcn` is a function handle referencing function `my_closefcn`.

Unless the close request function calls `delete` or `close`, MATLAB never closes the figure. (Note that you can always call `delete(figure_handle)` from the command line if you have created a window with a nondestructive close request function.)

A useful application of the close request function is to display a question dialog box asking the user to confirm the close operation. The following function illustrates how to do this.

[Click to view in editor](#) — This link opens the MATLAB editor with the following example.

# Figure Properties

---

Click to run example — **Ctrl**-click the figure to create a new figure.

```
function my_closereq(src,evnt)
% User-defined close request function
% to display a question dialog box
    selection = questdlg('Close This Figure?',...
        'Close Request Function',...
        'Yes','No','Yes');
    switch selection,
        case 'Yes',
            delete(gcf)
        case 'No'
            return
    end
end
```

Now create a figure using the `CloseRequestFcn`:

```
figure('CloseRequestFcn',@my_closereq)
```

To make this function your default close request function, set a default value on the root level.

```
set(0,'DefaultFigureCloseRequestFcn',@my_closereq)
```

MATLAB then uses this setting for the `CloseRequestFcn` of all subsequently created figures.

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

## Color

`ColorSpec`

*Background color.* This property controls the figure window background color. You can specify a color using a three-element vector of RGB values or one of the MATLAB predefined names. See `ColorSpec` for more information.

## Colormap

m-by-3 matrix of RGB values

*Figure colormap.* This property is an m-by-3 array of red, green, and blue (RGB) intensity values that define m individual colors. MATLAB accesses colors by their row number. For example, an index of 1 specifies the first RGB triplet, an index of 2 specifies the second RGB triplet, and so on.

### Number of Colors Allowed

Colormaps can be any length (up to 256 only on Microsoft Windows), but must be three columns wide. The default figure colormap contains 64 predefined colors.

### Objects That Use Colormaps

Colormaps affect the rendering of surface, image, and patch objects, but generally do not affect other graphics objects. See `colormap` and `ColorSpec` for more information.

## CreateFcn

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

*Callback function executed during figure creation.* This property defines a callback function that executes when MATLAB creates a figure object. You must define this property as a default value on the root level. For example, the statement

```
set(0, 'DefaultFigureCreateFcn', @fig_create)
```

defines a default value on the root level that causes all figures created to execute the setup function `fig_create`, which is defined below:

```
function fig_create(src, evnt)
set(src, 'Color', [.2 .1 .5], ...
```

# Figure Properties

---

```
        'IntegerHandle', 'off', ...  
        'MenuBar', 'none', ...  
        'ToolBar', 'none')  
    end
```

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

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

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

## CurrentAxes

handle of current axes

*Target axes in this figure.* MATLAB sets this property to the handle of the figure’s current axes (i.e., the handle returned by the `gca` command when this figure is the current figure). In all figures for which axes children exist, there is always a current axes. The current axes does not have to be the topmost axes, and setting an axes to be the `CurrentAxes` does not restack it above all other axes.

You can make an axes current using the `axes` and `set` commands. For example, `axes(axes_handle)` and `set(gcf, 'CurrentAxes', axes_handle)` both make the axes identified by the handle `axes_handle` the current axes. In addition, `axes(axes_handle)` restacks the axes above all other axes in the figure.

If a figure contains no axes, `get(gcf, 'CurrentAxes')` returns the empty matrix. Note that the `gca` function actually creates an axes if one does not exist.

**CurrentCharacter**  
single character

*Last key pressed.* MATLAB sets this property to the last key pressed in the figure window. `CurrentCharacter` is useful for obtaining user input.

**CurrentObject**  
object handle

*Handle of current object.* MATLAB sets this property to the handle of the last object clicked on by the mouse. This object is the foremost object in the view. You can use this property to determine which object a user has selected. The function `gco` provides a convenient way to retrieve the `CurrentObject` of the `CurrentFigure`.

Note that the `HitTest` property controls whether an object can become the `CurrentObject`.

## Hidden Handle Objects

Clicking an object whose `HandleVisibility` property is set to `off` (such as axis labels and title) causes the `CurrentObject` property to be set to empty `[]`. To avoid returning an empty value when users click hidden objects, set the hidden object's `HitTest` property to `off`.

## Mouse Over

Note that cursor motion over objects does not update the `CurrentObject`; you must click objects to update this property. See the `CurrentPoint` property for related information.

**CurrentPoint**  
two-element vector: [*x*-coordinate, *y*-coordinate]

# Figure Properties

---

*Location of last button click in this figure.* MATLAB sets this property to the location of the pointer at the time of the most recent mouse button press. MATLAB updates this property whenever you press the mouse button while the pointer is in the figure window.

Note that if you select a point in the figure and then use the values returned by the `CurrentPoint` property to plot that point, there can be differences in the position due to round-off errors.

## CurrentPoint and Cursor Motion

In addition to the behavior described above, MATLAB updates `CurrentPoint` before executing callback routines defined for the figure `WindowButtonMotionFcn` and `WindowButtonUpFcn` properties. This enables you to query `CurrentPoint` from these callback routines. It behaves like this:

- If there is no callback routine defined for the `WindowButtonMotionFcn` or the `WindowButtonUpFcn`, then MATLAB updates the `CurrentPoint` only when the mouse button is pressed down within the figure window.
- If there is a callback routine defined for the `WindowButtonMotionFcn`, then MATLAB updates the `CurrentPoint` just before executing the callback. Note that the `WindowButtonMotionFcn` executes only within the figure window unless the mouse button is pressed down within the window and then held down while the pointer is moved around the screen. In this case, the routine executes (and the `CurrentPoint` is updated) anywhere on the screen until the mouse button is released.
- If there is a callback routine defined for the `WindowButtonUpFcn`, MATLAB updates the `CurrentPoint` just before executing the callback. Note that the `WindowButtonUpFcn` executes only while the pointer is within the figure window unless the mouse button is pressed down initially within the window. In this case,

releasing the button anywhere on the screen triggers callback execution, which is preceded by an update of the `CurrentPoint`.

The figure `CurrentPoint` is updated only when certain events occur, as previously described. In some situations (such as when the `WindowButtonMotionFcn` takes a long time to execute and the pointer is moved very rapidly), the `CurrentPoint` may not reflect the actual location of the pointer, but rather the location at the time when the `WindowButtonMotionFcn` began execution.

The `CurrentPoint` is measured from the lower-left corner of the figure window, in units determined by the `Units` property.

The root `PointerLocation` property contains the location of the pointer updated synchronously with pointer movement. However, the location is measured with respect to the screen, not a figure window.

See `uicontrol` for information on how this property is set when you click a `uicontrol` object.

## DeleteFcn

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

*Delete figure callback function.* A callback function that executes when the figure object is deleted (e.g., when you issue a `delete` or a `close` command). MATLAB executes the function before destroying the object's properties so these values are available to the callback routine.

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

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

# Figure Properties

---

See also the figure `CloseRequestFcn` property

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

`DockControls`  
{on} | off

*Displays controls used to dock figure.* This property determines whether the figure enables the **Desktop** menu item and the dock figure button in the title bar that allow you to dock the figure into the MATLAB desktop.

By default, the figure docking controls are visible. If you set this property to `off`, the **Desktop** menu item that enables you to dock the figure is disabled and the figure dock button is not displayed.

See also the `WindowState` property for more information on docking figure.

`DoubleBuffer`  
{on} | off

*Flash-free rendering for simple animations.* Double buffering is the process of drawing to an off-screen pixel buffer and then printing the buffer contents to the screen once the drawing is complete. Double buffering generally produces flash-free rendering for simple animations (such as those involving lines, as opposed to objects containing large numbers of polygons). Use double buffering with the animated objects’ `EraseMode` property set to `normal`. Use the `set` command to disable double buffering.

```
set(figure_handle, 'DoubleBuffer', 'off')
```

Double buffering works only when the figure `Renderer` property is set to `painters`.

`FileName`  
String



*GUI FIG-filename.* GUIDE stores the name of the FIG-file used to save the GUI layout in this property.

## FixedColors

m-by-3 matrix of RGB values (read only)

*Noncolormap colors.* Fixed colors define all colors appearing in a figure window that are not from the figure colormap. These colors include axis lines and labels, the colors of `line`, `text`, `uicontrol`, and `uimenu` objects, and any colors explicitly defined, for example, with a statement like

```
set(gcf, 'Color', [0.3,0.7,0.9])
```

Fixed color definitions reside in the system color table and do not appear in the figure colormap. For this reason, fixed colors can limit the number of simultaneously displayed colors if the number of fixed colors plus the number of entries in the figure colormap exceed your system's maximum number of colors.

(See the root `ScreenDepth` property for information on determining the total number of colors supported on your system. See the `MinColorMap` property for information on how MATLAB shares colors between applications.)

## 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.

## Callback Visibility

# Figure Properties

---

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.

## Visibility Off

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.

## Visibility and Handles Returned by Other Functions

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

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

## Making All Handles Visible

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).

## Validity of Hidden Handles

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

`HitTest`  
{on} | off

*Selectable by mouse click.* `HitTest` determines if the figure can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the figure. If `HitTest` is off, clicking the figure sets the `CurrentObject` to the empty matrix.

`IntegerHandle`  
{on} | off

*Figure handle mode.* Figure object handles are integers by default. When creating a new figure, MATLAB uses the lowest integer that is not used by an existing figure. If you delete a figure, its integer handle can be reused.

If you set this property to off, MATLAB assigns nonreusable real-number handles (e.g., 67.0001221) instead of integers. This feature is designed for dialog boxes where removing the handle from integer values reduces the likelihood of inadvertently drawing into the dialog box.

`Interruptible`  
{on} | off

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

## How Callbacks Are Interrupted

MATLAB checks for queued events that can interrupt a callback function only when it encounters a call to `drawnow`, `figure`,

# Figure Properties

---

getframe, or pause in the executing callback function. When executing one of these functions, MATLAB processes all pending events, including executing all waiting callback functions. The interrupted callback then resumes execution.

## What Property Callbacks Are Interruptible

The Interruptible property only affects callback functions defined for the ButtonDownFcn, KeyPressFcn, KeyReleaseFcn, WindowButtonDownFcn, WindowButtonMotionFcn, WindowButtonUpFcn, WindowKeyPressFcn, WindowKeyReleaseFcn, and WindowScrollWheelFcn.

See the BusyAction property for related information.

**InvertHardcopy**  
{on} | off

*Change hardcopy to black objects on white background.* This property affects only printed output. Printing a figure having a background color (Color property) that is not white results in poor contrast between graphics objects and the figure background and also consumes a lot of printer toner.

When InvertHardCopy is on, MATLAB eliminates this effect by changing the color of the figure and axes to white and the axis lines, tick marks, axis labels, etc., to black. Lines, text, and the edges of patches and surfaces might be changed, depending on the print command options specified.

If you set InvertHardCopy to off, the printed output matches the colors displayed on the screen.

See print for more information on printing MATLAB figures.

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

*Key press callback function.* This is a callback function invoked by a key press that occurs while the figure window has focus. Define the `KeyPressFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

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

When there is no callback specified for this property (which is the default state), MATLAB passes any key presses to the Command Window. However, when you define a callback for this property, the figure retains focus with each key press and executes the specified callback with each key press.

## KeyPressFcn Event Structure

When the callback is a function handle, MATLAB passes a structure to the callback function that contains the following fields.

Field	Contents
Character	The character displayed as a result of the key(s) pressed.
Modifier	This field is a cell array that contains the names of one or more modifier keys that the user pressed (i.e., <b>control</b> , <b>alt</b> , <b>shift</b> ). On Macintosh computers, MATLAB can also return 'command' if the user pressed the <b>command</b> modifier key.
Key	The key pressed (lowercase label on key).

Some key combinations do not define a value for the Character field.

## Using the KeyPressFcn

# Figure Properties

---

This example, creates a figure and defines a function handle callback for the `KeyPressFcn` property. When the `e` key is pressed, the callback exports the figure as an EPS file. When `Ctrl-t` is pressed, the callback exports the figure as a TIFF file.

```
function figure_keypress
    figure('KeyPressFcn',@printfig);
    function printfig(src,evnt)
        if evnt.Character == 'e'
            print ('-deps', ['-f' num2str(src)])
        elseif length(evnt.Modifier) == 1 & strcmp(evnt.Modifier{:},'control') & ...
            evnt.Key == 't'
            print ('-dtiff', '-r200', ['-f' num2str(src)])
        end
    end
end
```

## KeyReleaseFcn

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

*Key release callback function.* This is a callback function invoked by a key release that occurs while the figure window has focus. Define the `KeyReleaseFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

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

## KeyReleaseFcn Event Structure

When the callback is a function handle, MATLAB passes a structure as the second argument to the callback function that contains the following fields.

Field	Contents
Character	The character displayed as a result of the key(s) released.
Modifier	This field is a cell array that contains the names of one or more modifier keys that the user releases (i.e., <b>control</b> , <b>alt</b> , <b>shift</b> , or empty if no modifier keys were released). On Macintosh computers, MATLAB can also return 'command' if the user released the <b>command</b> modifier key.
Key	The lowercase label on key that was released.

Some key combinations do not define a value for the Character field.

### Properties Affected by the KeyReleaseFcn

When a callback is defined for the KeyReleaseFcn property, MATLAB updates the CurrentCharacter, CurrentKey, and CurrentModifier figure properties just before executing the callback.

### Multiple-Key Press Events and a Single-Key Release Event

Consider a figure having callbacks defined for both the KeyPressFcn and KeyReleaseFcn. In the case where a user presses multiple keys, one after another, MATLAB generates repeated KeyPressFcn events only for the last key pressed.

For example, suppose you press and hold down the **a** key, then press and hold down the **s** key. MATLAB generates repeated KeyPressFcn events for the **a** key until you press the **s** key, at which point MATLAB generates repeated KeyPressFcn events for the **s** key. If you then release the **s** key, MATLAB generates a KeyReleaseFcn event for the **s** key, but no new KeyPressFcn

# Figure Properties

---

events for the **a** key. When you then release the **a** key, the `KeyReleaseFcn` again executes.

The `KeyReleaseFcn` behavior is such that it executes its callback every time you release a key while the figure is in focus, regardless of what `KeyPressFcns` MATLAB generates.

## Modifier Keys

When the user presses and releases a key and a modifier key, the modifier key is returned in the event structure `Modifier` field. If a modifier key is the only key pressed and released, it is not returned in the event structure of the `KeyReleaseFcn`, but is returned in the event structure of the `KeyPressFcn`.

## Explore the Results

[Click to view in editor](#) — This link opens the MATLAB editor with the following example.

[Click to run example](#) — Press and release various key combinations while the figure has focus to see the data returned in the event structure.

The following code creates a figure and defines a function handle callback for the `KeyReleaseFcn` property. The callback simply displays the values returned by the event structure and enables you to explore the `KeyReleaseFcn` behavior when you release various key combinations.

```
function key_releaseFcn
figure('KeyReleaseFcn',@cb)
function cb(src,evnt)
if ~isempty(evnt.Modifier)
for ii = 1:length(evnt.Modifier)
out = sprintf('Character: %c\nModifier: %s\nKey: %s\n',...
evnt.Character,evnt.Modifier{ii},evnt.Key);
```



```
        disp(out)
    end
    else
        out = sprintf('Character: %c\nModifier: %s\nKey: %s\n',...
            evnt.Character, 'No modifier key', evnt.Key);
        disp(out)
    end
end
end
```

## MenuBar

none | {figure}

*Enable-disable figure menu bar.* This property enables you to display or hide the menu bar that MATLAB places at the top of a figure window. The default (`figure`) is to display the menu bar.

This property affects only built-in menus. This property does not affect menus defined with the `uimenu` command.

If you start MATLAB with the `nojvm` option, figures do not display the menu bar because most items require Java figures.

## MinColormap

scalar (default = 64)

*Minimum number of color table entries used.* This property specifies the minimum number of system color table entries used by MATLAB to store the colormap defined for the figure (see the `Colormap` property). In certain situations, you may need to increase this value to ensure proper use of colors.

For example, suppose you are running color-intensive applications in addition to MATLAB and have defined a large figure colormap (e.g., 150 to 200 colors). MATLAB may select colors that are close but not exact from the existing colors in the system color table because there are not enough slots available to define all the colors you specified.

# Figure Properties

---

To ensure that MATLAB uses exactly the colors you define in the figure colormap, set `MinColorMap` equal to the length of the colormap.

```
set(gcf, 'MinColormap', length(get(gcf, 'ColorMap')))
```

Note that the larger the value of `MinColorMap`, the greater the likelihood that other windows (including other MATLAB figure windows) will be displayed in false colors.

Name

string

*Figure window title.* This property specifies the title displayed in the figure window. By default, Name is empty and the figure title is displayed as `Figure 1`, `Figure 2`, and so on. When you set this parameter to a string, the figure title becomes `Figure 1: <string>`. See the `NumberTitle` property.

NextPlot

new | {add} | replace | replacechildren

*How to add next plot.* `NextPlot` determines which figure MATLAB uses to display graphics output. If the value of the current figure is

- `new` — Create a new figure to display graphics (unless an existing parent is specified in the graphing function as a property/value pair).
- `add` — Use the current figure to display graphics (the default).
- `replace` — Reset all figure properties except `Position` to their defaults and delete all figure children before displaying graphics (equivalent to `clf reset`).
- `replacechildren` — Remove all child objects, but do not reset figure properties (equivalent to `clf`).

The `newplot` function provides an easy way to handle the `NextPlot` property. Also see the `NextPlot` axes property and “Controlling Graphics Output” for more information.

## NumberTitle

{on} | off (GUIDE default off)

*Figure window title number.* This property determines whether the string `Figure No. N` (where `N` is the figure number) is prefixed to the figure window title. See the `Name` property.

## OuterPosition

four-element vector

*Figure position including title bar, menu bar, tool bars, and outer edges.* This property specifies the size and location on the screen of the full figure window including the title bar, menu bar, tool bars, and outer edges. Specify the position rectangle with a four-element vector of the form:

```
rect = [left, bottom, width, height]
```

where `left` and `bottom` define the distance from the lower-left corner of the screen to the lower-left corner of the full figure window. `width` and `height` define the dimensions of the window. See the `Units` property for information on the units used in this specification. The `left` and `bottom` elements can be negative on systems that have more than one monitor.

## Position of Docked Figures

If the figure is docked in the MATLAB desktop, then the `OuterPosition` property is specified with respect to the figure group container instead of the screen.

## Moving and Resizing Figures

# Figure Properties

---

Use the `get` function to obtain this property and determine the position of the figure. Use the `set` function to resize and move the figure to a new location. You cannot set the figure `OuterPosition` when it is docked.

---

**Note** On Windows systems, figure windows cannot be less than 104 pixels wide, regardless of the value of the `OuterPosition` property.

---

`PaperOrientation`  
{portrait} | landscape

*Horizontal or vertical paper orientation.* This property determines how to orient printed figures on the page. `portrait` orients the longest page dimension vertically; `landscape` orients the longest page dimension horizontally. See the `orient` command for more detail.

`PaperPosition`  
four-element rect vector

*Location on printed page.* A rectangle that determines the location of the figure on the printed page. Specify this rectangle with a vector of the form

```
rect = [left, bottom, width, height]
```

where `left` specifies the distance from the left side of the paper to the left side of the rectangle and `bottom` specifies the distance from the bottom of the page to the bottom of the rectangle. Together these distances define the lower-left corner of the rectangle. `width` and `height` define the dimensions of the rectangle. The `PaperUnits` property specifies the units used to define this rectangle.

## PaperPositionMode

auto | {manual}

*WYSIWYG printing of figure.* In manual mode, MATLAB honors the value specified by the PaperPosition property. In auto mode, MATLAB prints the figure the same size as it appears on the computer screen, centered on the page.

See Pixels Per Inch Solution at Technical Solutions for information on specifying a pixels per inch resolution setting for MATLAB figures. Doing so might be necessary to obtain a printed figure that is the same size as it is on the screen.

## PaperSize

[width height]

*Paper size.* This property contains the size of the current PaperType, measured in PaperUnits. See PaperType to select standard paper sizes.

## PaperType

Select a value from the following table.

*Selection of standard paper size.* This property sets the PaperSize to one of the following standard sizes.

Property Value	Size (Width x Height)
usletter (default)	8.5-by-11 inches
uslegal	11-by-14 inches
tabloid	11-by-17 inches
A0	841-by-1189 mm
A1	594-by-841 mm
A2	420-by-594 mm
A3	297-by-420 mm

## Figure Properties

---

Property Value	Size (Width x Height)
A4	210-by-297 mm
A5	148-by-210 mm
B0	1029-by-1456 mm
B1	728-by-1028 mm
B2	514-by-728 mm
B3	364-by-514 mm
B4	257-by-364 mm
B5	182-by-257 mm
arch-A	9-by-12 inches
arch-B	12-by-18 inches
arch-C	18-by-24 inches
arch-D	24-by-36 inches
arch-E	36-by-48 inches
A	8.5-by-11 inches
B	11-by-17 inches
C	17-by-22 inches
D	22-by-34 inches
E	34-by-43 inches

Note that you may need to change the `PaperPosition` property in order to position the printed figure on the new paper size. One solution is to use `normalized PaperUnits`, which enables MATLAB to automatically size the figure to occupy the same relative amount of the printed page, regardless of the paper size.

`PaperUnits`

`normalized` | `{inches}` | `centimeters` | `points`

*Hardcopy measurement units.* This property specifies the units used to define the `PaperPosition` and `PaperSize` properties. MATLAB measures all units from the lower-left corner of the page. `normalized` units map the lower-left corner of the page 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).

If you change the value of `PaperUnits`, it is good practice to return the property to its default value after completing your computation so as not to affect other functions that assume `PaperUnits` is set to the default value.

## Parent

handle

*Handle of figure's parent.* The parent of a figure object is the root object. The handle to the root is always 0.

## Pointer

crosshair | {arrow} | watch | topl |  
topr | botl | botr | circle | cross |  
fleur | left | right | top | bottom |  
fullcrosshair | ibeam | custom

*Pointer symbol selection.* This property determines the symbol used to indicate the pointer (cursor) position in the figure window. Setting `Pointer` to `custom` allows you to define your own pointer symbol. See the `PointerShapeCData` property and “Specifying the Figure Pointer” for more information.

## PointerShapeCData

16-by-16 matrix

*User-defined pointer.* This property defines the pointer that is used when you set the `Pointer` property to `custom`. It is a 16-by-16 element matrix defining the 16-by-16 pixel pointer using the following values:

# Figure Properties

---

- 1 — Color pixel black.
- 2 — Color pixel white.
- NaN — Make pixel transparent (underlying screen shows through).

Element (1,1) of the `PointerShapeCData` matrix corresponds to the upper-left corner of the pointer. Setting the `Pointer` property to one of the predefined pointer symbols does not change the value of the `PointerShapeCData`. Computer systems supporting 32-by-32 pixel pointers fill only one quarter of the available pixmap.

`PointerShapeHotSpot`  
two-element vector

*Pointer active area.* A two-element vector specifying the row and column indices in the `PointerShapeCData` matrix defining the pixel indicating the pointer location. The location is contained in the `CurrentPoint` property and the root object's `PointerLocation` property. The default value is element (1,1), which is the upper-left corner.

`Position`  
four-element vector

*Figure position.* This property specifies the size and location on the screen of the figure window, not including title bar, menu bar, tool bars, and outer edges. Specify the position rectangle with a four-element vector of the form:

```
rect = [left, bottom, width, height]
```

where `left` and `bottom` define the distance from the lower-left corner of the screen to the lower-left corner of the figure window. `width` and `height` define the dimensions of the window. See the `Units` property for information on the units used in this



specification. The `left` and `bottom` elements can be negative on systems that have more than one monitor.

## Position of Docked Figures

If the figure is docked in the MATLAB desktop, then the `Position` property is specified with respect to the figure group container instead of the screen.

## Moving and Resizing Figures

You can use the `get` function to obtain this property and determine the position of the figure and you can use the `set` function to resize and move the figure to a new location. You cannot set the figure `Position` when it is docked.

---

**Note** On Windows systems, figure windows cannot be less than 104 pixels wide, regardless of the value of the `Position` property.

Also, the figure window includes the area to which MATLAB can draw; it does not include the title bar, menu bar, tool bars, and outer edges. To place the full window, use the `OuterPosition` property.

---

## Renderer

`painters` | `zbuffer` | `OpenGL`

*Rendering method used for screen and printing.* This property enables you to select the method used to render MATLAB graphics. The choices are

- `painters` — The original rendering method used by MATLAB is faster when the figure contains only simple or small graphics objects.

# Figure Properties

---

- **zbuffer** — MATLAB draws graphics objects faster and more accurately because it colors objects on a per-pixel basis and MATLAB renders only those pixels that are visible in the scene (thus eliminating front-to-back sorting errors). Note that this method can consume a lot of system memory if MATLAB is displaying a complex scene.
- **OpenGL** — OpenGL is a renderer that is available on many computer systems. This renderer is generally faster than painters or zbuffer and in some cases enables MATLAB to access graphics hardware that is available on some systems.

## Hardware vs. Software OpenGL Implementations

There are two kinds of OpenGL implementations — hardware and software.

The hardware implementation uses special graphics hardware to increase performance and is therefore significantly faster than the software version. Many computers have this special hardware available as an option or may come with this hardware right out of the box.

Software implementations of OpenGL are much like the ZBuffer renderer that is available on MATLAB Version 5.0 and later; however, OpenGL generally provides superior performance to ZBuffer.

## OpenGL Availability

OpenGL is available on all computers that run MATLAB. MATLAB automatically finds hardware-accelerated versions of OpenGL if such versions are available. If the hardware-accelerated version is not available, then MATLAB uses the software version (except on Macintosh systems, which do not support software OpenGL).

The following software versions are available:

- On UNIX systems, MATLAB uses the software version of OpenGL that is included in the MATLAB distribution.
- On Windows, OpenGL is available as part of the operating system. If you experience problems with OpenGL, contact your graphics driver vendor to obtain the latest qualified version of OpenGL.
- On Macintosh systems, software OpenGL is not available.

MATLAB issues a warning if it cannot find a usable OpenGL library.

## Selecting Hardware-Accelerated or Software OpenGL

MATLAB enables you to switch between hardware-accelerated and software OpenGL. However, Windows and UNIX systems behave differently:

- On Windows systems, you can toggle between software and hardware versions any time during the MATLAB session.
- On UNIX systems, you must set the OpenGL version before MATLAB initializes OpenGL. Therefore, you cannot issue the `opengl info` command or create graphs before you call `opengl software`. To reenable hardware accelerated OpenGL, you must restart MATLAB.
- On Macintosh systems, software OpenGL is not available.

If you do not want to use hardware OpenGL, but do want to use object transparency, you can issue the following command.

```
opengl software
```

This command forces MATLAB to use software OpenGL. Software OpenGL is useful if your hardware-accelerated version of OpenGL does not function correctly and you want to use image, patch, or

# Figure Properties

---

surface transparency, which requires the OpenGL renderer. To reenable hardware OpenGL, use the command:

```
opengl hardware
```

on Windows systems or restart MATLAB on UNIX systems.

By default, MATLAB uses hardware-accelerated OpenGL.

See the `opengl` reference page for additional information

## Determining What Version You Are Using

To determine the version and vendor of the OpenGL library that MATLAB is using on your system, type the following command at the MATLAB prompt:

```
opengl info
```

The returned information contains a line that indicates if MATLAB is using software (`Software = true`) or hardware-accelerated (`Software = false`) OpenGL.

This command also returns a string of extensions to the OpenGL specification that are available with the particular library MATLAB is using. This information is helpful to The MathWorks, so please include this information if you need to report bugs.

Note that issuing the `opengl info` command causes MATLAB to initialize OpenGL.

## OpenGL vs. Other MATLAB Renderers

There are some differences between drawings created with OpenGL and those created with other renderers. The OpenGL specific differences include

- OpenGL does not do colormap interpolation. If you create a surface or patch using indexed color and interpolated face or edge coloring, OpenGL interpolates the colors through the RGB color cube instead of through the colormap.
- OpenGL does not support the `phong` value for the `FaceLighting` and `EdgeLighting` properties of surfaces and patches.
- OpenGL does not support logarithmic-scale axes.
- OpenGL and Zbuffer renderers display objects sorted in front to back order, as seen on the monitor, and lines always draw in front of faces when at the same location on the plane of the monitor. Painters sorts by child order (order specified).

## If You Are Having Problems

Consult the OpenGL Technical Note if you are having problems using OpenGL. This technical note contains a wealth of information on MATLAB renderers.

`RendererMode`  
{auto} | manual

*Automatic or user selection of renderer.* This property enables you to specify whether MATLAB should choose the `Renderer` based on the contents of the figure window, or whether the `Renderer` should remain unchanged.

When the `RendererMode` property is set to `auto`, MATLAB selects the rendering method for printing as well as for screen display based on the size and complexity of the graphics objects in the figure.

For printing, MATLAB switches to `zbuffer` at a greater scene complexity than for screen rendering because printing from a z-buffered figure can be considerably slower than one using the `painters` rendering method, and can result in large PostScript® files. However, the output does always match what is on the

# Figure Properties

---

screen. The same holds true for OpenGL: the output is the same as that produced by the `zbuffer` renderer — a bitmap with a resolution determined by the `print` command's `-r` option.

## Criteria for Autoselection of the OpenGL Renderer

When the `RendererMode` property is set to `auto`, MATLAB uses the following criteria to determine whether to select the OpenGL renderer:

If the `opengl` autoselection mode is `autoselect`, MATLAB selects OpenGL if

- The host computer has OpenGL installed and is in True Color mode (OpenGL does not fully support 8-bit color mode).
- The figure contains no logarithmic axes (OpenGL does not support logarithmic axes).
- MATLAB would select `zbuffer` based on figure contents.
- Patch objects' faces have no more than three vertices (some OpenGL implementations of patch tessellation are unstable).
- The figure contains less than 10 `uicontrols` (OpenGL clipping around `uicontrols` is slow).
- No line objects use markers (drawing markers is slow).
- You do not specify Phong lighting (OpenGL does not support Phong lighting; if you specify Phong lighting, MATLAB uses the `ZBuffer` renderer).

Or

- Figure objects use transparency (OpenGL is the only MATLAB renderer that supports transparency).

When the `RendererMode` property is set to `manual`, MATLAB does not change the `Renderer`, regardless of changes to the figure contents.

## Resize

{on} | off

*Window resize mode.* This property determines if you can resize the figure window with the mouse. `on` means you can resize the window, `off` means you cannot. When `Resize` is `off`, the figure window does not display any resizing controls (such as boxes at the corners), to indicate that it cannot be resized.

## ResizeFcn

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

*Window resize callback function.* MATLAB executes the specified callback function whenever you resize the figure window and also when the figure is created. You can query the figure's `Position` property to determine the new size and position of the figure. During execution of the callback routine, the handle to the figure being resized is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

You can use `ResizeFcn` to maintain a GUI layout that is not directly supported by the MATLAB `Position/Units` paradigm.

For example, consider a GUI layout that maintains an object at a constant height in pixels and attached to the top of the figure, but always matches the width of the figure. The following `ResizeFcn` accomplishes this; it keeps the `uicontrol` whose `Tag` is `'StatusBar'` 20 pixels high, as wide as the figure, and attached to the top of the figure. Note the use of the `Tag` property to retrieve the `uicontrol` handle, and the `gcbo` function to retrieve the figure handle. Also note the defensive programming regarding figure `Units`, which the callback requires to be in pixels in order to work correctly, but which the callback also restores to their previous value afterwards.

```
u = findobj('Tag','StatusBar');  
fig = gcbo;
```

## Figure Properties

---

```
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

You can change the figure `Position` from within the `ResizeFcn` callback; however, the `ResizeFcn` is not called again as a result.

Note that the `print` command can cause the `ResizeFcn` to be called if the `PaperPositionMode` property is set to `manual` and you have defined a resize function. If you do not want your resize function called by `print`, set the `PaperPositionMode` to `auto`.

See “Introduction” for an example of how to implement a resize function for a GUI.

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

`Selected`  
on | off

*Is object selected?* This property indicates whether the figure is selected. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`  
{on} | off

Figures do not indicate selection.

`SelectionType`  
{normal} | extend | alt | open

*Mouse selection type.* MATLAB maintains this property to provide information about the last mouse button press that occurred within the figure window. This information indicates the type



of selection made. Selection types are actions that MATLAB generally associates with particular responses from the user interface software (e.g., single-clicking a graphics object places it in move or resize mode; double-clicking a file name opens it, etc.).

The physical action required to make these selections varies on different platforms. However, all selection types exist on all platforms.

<b>Selection Type</b>	<b>Microsoft Windows</b>	<b>X-Windows</b>
Normal	Click left mouse button.	Click left mouse button.
Extend	<b>Shift</b> - click left mouse button or click both left and right mouse buttons.	<b>Shift</b> -click left mouse button or click middle mouse button.
Alternate	<b>Control</b> - click left mouse button or click right mouse button.	<b>Control</b> -click left mouse button or click right mouse button.
Open	Double-click any mouse button.	Double-click any mouse button.

---

**Note** For uicontrols whose Enable property is on, a single left-click, **Ctrl**-left click, or **Shift**-left click sets the figure SelectionType property to normal. For a list box uicontrol whose Enable property is on, the second click of a double-click sets the figure SelectionType property to open. All clicks on uicontrols whose Enable property is inactive or off and all right-clicks on uicontrols whose Enable property is on set the figure SelectionType property as specified in the preceding table.

---

# Figure Properties

---

## Tag

string

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

For example, suppose you want to direct all graphics output from an M-file to a particular figure, regardless of user actions that may have changed the current figure. To do this, identify the figure with a `Tag`.

```
figure('Tag','Plotting Figure')
```

Then make that figure the current figure before drawing by searching for the `Tag` with `findobj`.

```
figure(findobj('Tag','Plotting Figure'))
```

## Toolbar

none | {auto} | figure

*Control display of figure toolbar.* The `Toolbar` property enables you to control whether MATLAB displays the default figure toolbar on figures. There are three possible values:

- `none` — Do not display the figure toolbar.
- `auto` — Display the figure toolbar, but remove it if a `uicontrol` is added to the figure.
- `figure` — Display the figure toolbar.

Note that this property affects only the figure toolbar; it does not affect other toolbars (e.g., the Camera Toolbar or Plot Edit

Toolbar). Selecting **Figure Toolbar** from the figure **View** menu sets this property to figure.

If you start MATLAB with the `nojvm` option, figures do not display the toolbar because most tools require Java figures.

## Type

string (read only)

*Object class.* This property identifies the kind of graphics object. For figures, Type is always the string 'figure'.

## UIContextMenu

handle of a uicontextmenu object

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

## Units

{pixels} | normalized | inches |  
centimeters | points | characters

*Units of measurement.* This property specifies the units MATLAB uses to interpret size and location data. All units are measured from the lower-left corner of the window.

- `normalized` units map the lower-left corner of the figure window to (0,0) and the upper-right corner to (1.0,1.0).
- `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch).
- The size of a pixel depends on screen resolution.
- `characters` units are defined by characters from the default system font; the width of one character is the width of the

# Figure Properties

---

letter x, the height of one character is the distance between the baselines of two lines of text.

This property affects the `CurrentPoint` and `Position` properties. If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

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.

`UserData`  
matrix

*User-specified data.* You can specify `UserData` as any matrix you want to associate with the figure object. The object does not use this data, but you can access it using the `set` and `get` commands.

`Visible`  
{on} | off

*Object visibility.* The `Visible` property determines whether an object is displayed on the screen. If the `Visible` property of a figure is `off`, the entire figure window is invisible.

## A Note About Using the Window Button Properties

Your window button callback functions might need to update the display by calling `drawnow` or `pause`, which causes MATLAB to process all events in the queue. Processing the event queue can cause your window button callback functions to be reentered. For example, a `drawnow` in the `WindowButtonDownFcn` might result in the `WindowButtonDownFcn` being called again before the first call has finished. You should design your code to handle reentrancy and you should not depend on global variables that might change state during reentrance.

You can use the `Interruptible` and `BusyAction` figure properties to control how events interact.

## `WindowButtonDownFcn`

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

*Button press callback function.* Use this property to define a callback that MATLAB executes whenever you press a mouse button while the pointer is in the figure window. See the `WindowButtonMotionFcn` property for an example.

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

## `WindowButtonMotionFcn`

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

*Mouse motion callback function.* Use this property to define a callback that MATLAB executes whenever you move the pointer within the figure window. Define the `WindowButtonMotionFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

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

## **Example Using All Window Button Properties**

Click to view in editor — This example enables you to use mouse motion to draw lines. It uses all three window button functions.

Click to run example — Click the left mouse button in the axes and move the cursor, left-click to define the line end point, right-click to end drawing mode.

# Figure Properties

---

---

**Note** On some computer systems, the `WindowButtonMotionFcn` is executed when a figure is created even though there has been no mouse motion within the figure.

---

## `WindowButtonUpFcn`

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

*Button release callback function.* Use this property to define a callback that MATLAB executes whenever you release a mouse button. Define the `WindowButtonUpFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

The button up event is associated with the figure window in which the preceding button down event occurred. Therefore, the pointer need not be in the figure window when you release the button to generate the button up event.

If the callback routines defined by `WindowButtonDownFcn` or `WindowButtonMotionFcn` contain `drawnow` commands or call other functions that contain `drawnow` commands and the `Interruptible` property is set to `off`, the `WindowButtonUpFcn` might not be called. You can prevent this problem by setting `Interruptible` to `on`.

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

## `WindowKeyPressFcn`

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

*Key press callback function for the figure window.* Use this property to define a callback that MATLAB executes whenever a key press occurs. This is a callback function invoked by a key press

that occurs while either the figure window or any of its children has focus. Define the `WindowKeyPressFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

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

When there is no callback specified for this property (which is the default state), MATLAB passes any key presses to the command window. However, when you define a callback for this property, the figure retains focus with each key press and executes the specified callback.

## WindowKeyPressFcn Event Structure

When the callback is a function handle, MATLAB passes a structure to the callback function that contains the following fields.

Field	Contents
Character	The character displayed as a result of the key(s) pressed.
Modifier	This field is a cell array that contains the names of one or more modifier keys that the user pressed (i.e., <b>control</b> , <b>alt</b> , <b>shift</b> ). On Macintosh computers, MATLAB can also return 'command' if the user pressed the <b>command</b> modifier key.
Key	The key pressed (lowercase label on key).

## WindowKeyReleaseFcn

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

# Figure Properties

---

*Key release callback function for the figure window.* Use this property to define a callback that MATLAB executes whenever a key release occurs. This is a callback function invoked by a key release that occurs while the figure window or any of its children has focus. Define the `WindowKeyReleaseFcn` as a function handle. The function must define at least two input arguments (handle of the figure associated with key release and an event structure).

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

## WindowKeyReleaseFcn Event Structure

When the callback is a function handle, MATLAB passes a structure to the callback function that contains the following fields.

Field	Contents
Character	The character corresponding to the key(s) released.
Modifier	This field is a cell array that contains the names of one or more modifier keys that the user released (i.e., <b>control</b> , <b>alt</b> , <b>shift</b> ). On Macintosh computers, MATLAB can also return 'command' if the user released the <b>command</b> modifier key.
Key	The key released (lower case label on key).

## WindowScrollWheelFcn

string, function handle, or cell array containing function handle and additional arguments

*Respond to mouse scroll wheel.* Use this property to define a callback that MATLAB executes when the mouse wheel is scrolled while the figure has focus. MATLAB executes the callback with each single mouse wheel click.



Note that it is possible for another object to capture the event from MATLAB. For example, if the figure contains Java or ActiveX control objects that are listening for mouse scroll wheel events, then these objects can consume the events and prevent the `WindowScrollWheelFcn` from executing.

There is no default callback defined for this property.

## WindowScrollWheelFcn Event Structure

When the callback is a function handle, MATLAB passes a structure to the callback function that contains the following fields.

Field	Contents
<code>VerticalScrollCount</code>	A positive or negative integer that indicates the number of scroll wheel clicks. Positive values indicate clicks of the wheel scrolled in the down direction. Negative values indicate clicks of the wheel scrolled in the up direction.
<code>VerticalScrollAmount</code>	The current system setting for the number of lines that are scrolled for each click of the scroll wheel. If the mouse property setting for scrolling is set to <code>One screen at a time</code> , <code>VerticalScrollAmount</code> returns a value of 1.

## Effects on Other Properties

- **CurrentObject** property — Mouse scrolling does not update this figure property.
- **CurrentPoint** property — If there is no callback defined for the **WindowScrollWheelFcn** property, then MATLAB does not update the **CurrentPoint** property as the scroll wheel is turned. However, if there is a callback defined for the **WindowScrollWheelFcn** property, then MATLAB updates the **CurrentPoint** property just before executing the callback. This enables you to determine the point at which the mouse scrolling occurred.
- **HitTest** property — The **WindowScrollWheelFcn** callback executes regardless of the setting of the figure **HitTest** property.
- **SelectionType** property — The **WindowScrollWheelFcn** callback has no effect on this property.

## Values Returned by VerticalScrollCount

When a user moves the mouse scroll wheel by one click, MATLAB increments the count by +/- 1, depending on the direction of the scroll (scroll down being positive). When MATLAB calls the **WindowScrollWheelFcn** callback, the counter is reset. In most cases, this means that the absolute value of the returned value is 1. However, if the **WindowScrollWheelFcn** callback takes a long enough time to return and/or the user spins the scroll wheel very fast, then the returned value can have an absolute value greater than one.

The actual value returned by **VerticalScrollCount** is the algebraic sum of all scroll wheel clicks that occurred since last processed. This enables your callback to respond correctly to the user's action.

## Example

Click to view in editor — This example creates a graph of a function and enables you to use the mouse scroll wheel to change the range over which a mathematical function is evaluated and update the graph to reflect the new limits as you turn the scroll wheel.

Click to run example — Mouse over the figure and scroll your mouse wheel.

## Related Information

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

### WindowState

`{normal} | modal | docked`

*Normal, modal, or dockable window behavior.* When `WindowState` is set to `modal`:

- The figure window traps all keyboard and mouse events over all MATLAB windows as long as they are visible.
- Windows belonging to applications other than MATLAB are unaffected.
- Modal figures remain stacked above all normal figures and the MATLAB Command Window.
- When multiple modal windows exist, the most recently created window keeps focus and stays above all other windows until it becomes invisible, or is returned to `WindowState normal`, or is deleted. At that time, focus reverts to the window that last had focus.

Use modal figures to create dialog boxes that force the user to respond without being able to interact with other windows. Typing **Ctrl+C** while the figure has focus causes all figures with

# Figure Properties

---

`WindowStyle modal` to revert to `WindowStyle normal`, allowing you to type at the command line.

## Invisible Modal Figures

Figures with `WindowStyle modal` and `Visible off` do not behave modally until they are made visible, so it is acceptable to hide a modal window for later reuse instead of destroying it.

## Stacking Order of Modal Figures

Creating a figure with `WindowStyle modal` stacks it on top of all existing figure windows, making them inaccessible as long as the top figure exists and remains modal. However, any new figures created after a modal figure is displayed (for example, plots created by a modal GUI) stack on top of it and are accessible; they can be modal as well.

## Changing Modes

You can change the `WindowStyle` of a figure at any time, including when the figure is visible and contains children. However, on some systems this may cause the figure to flash or disappear and reappear, depending on the windowing system's implementation of normal and modal windows. For best visual results, you should set `WindowStyle` at creation time or when the figure is invisible.

## Window Decorations on Modal Figures

Modal figures do not display `uimenu` children, built-in menus, or toolbars but it is not an error to create `uimenu`s in a modal figure or to change `WindowStyle` to modal on a figure with `uimenu` children. The `uimenu` objects exist and their handles are retained by the figure. If you reset the figure's `WindowStyle` to normal, the `uimenu`s are displayed.

## Docked WindowStyle

When `WindowState` is set to `docked`, the figure is docked in the desktop or a document window. When you issue the following command,

```
set(figure_handle, 'WindowState', 'docked')
```

MATLAB docks the figure identified by *figure\_handle* and sets the `DockControls` property to `on`, if it was `off`.

Note that if `WindowState` is `docked`, you cannot set the `DockControls` property to `off`.

The value of the `WindowState` property is not changed by calling `reset` on a figure.

## WVisual

identifier string (Windows only)

*Specify pixel format for figure.* MATLAB automatically selects a pixel format for figures based on your current display settings, the graphics hardware available on your system, and the graphical content of the figure.

Usually, MATLAB chooses the best pixel format to use in any given situation. However, in cases where graphics objects are not rendered correctly, you might be able to select a different pixel format and improve results. See for more information.

## Querying Available Pixel Formats on Window Systems

You can determine what pixel formats are available on your system for use with MATLAB using the following statement:

```
set(gcf, 'WVisual')
```

MATLAB returns a list of the currently available pixel formats for the current figure. For example, the following are the first three entries from a typical list:

# Figure Properties

---

01 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, OpenGL, GDI, Window)

02 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, OpenGL, Double Buffered, Window)

03 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, OpenGL, Double Buffered, Window)

Use the number at the beginning of the string to specify which pixel format to use. For example,

```
set(gcf, 'WVisual', '02')
```

specifies the second pixel format in the list above. Note that pixel formats might differ on your system.

## Understanding the WVisual String

The string returned by querying the WVisual property provides information on the pixel format. For example:

- **RGB 16 bits(05 06 05 00)** — Indicates true color with 16-bit resolution (5 bits for red, 6 bits for green, 5 bits for blue, and 0 for alpha (transparency). MATLAB requires true color.
- **zdepth 24** — Indicates 24-bit resolution for sorting object's front to back position on the screen. Selecting pixel formats with higher (24 or 32) zdepth might solve sorting problems.
- **Hardware Accelerated** — Some graphics functions may be performed by hardware for increased speed. If there are incompatibilities between your particular graphic hardware and MATLAB, select a pixel format in which the term **Generic** appears instead of **Hardware Accelerated**.
- **Opengl** — Supports OpenGL. See for more information.
- **GDI** — Supports for Windows 2-D graphics interface.

- **Double Buffered** — Support for double buffering with the OpenGL renderer. Note that the figure `DoubleBuffer` property applies only to the painters renderer.
- **Bitmap** — Support for rendering into a bitmap (as opposed to drawing in the window).
- **Window** — Support for rendering into a window.

## Pixel Formats and OpenGL

If you are experiencing problems using hardware OpenGL on your system, you can try using generic OpenGL, which is implemented in software. To do this, first instruct MATLAB to use the software version of OpenGL with the following statement:

```
opengl software
```

Then allow MATLAB to select best pixel format to use.

See the `Renderer` property for more information on how MATLAB uses OpenGL.

### `WVisualMode`

auto | manual (Windows only)

*Auto or manual selection of pixel format.* `WVisualMode` can take on two values — `auto` (the default) and `manual`. In `auto` mode, MATLAB selects the best pixel format to use based on your computer system and the graphical content of the figure. In `manual` mode, MATLAB does not change the visual from the one currently in use. Setting the `WVisual` property sets this property to `manual`.

### `XDisplay`

display identifier (UNIX only)

*Contains the display used for MATLAB.* You can query this property to determine the name of the display that MATLAB is

# Figure Properties

---

using. For example, if MATLAB is running on a system called mycomputer, querying XDisplay returns a string of the following form:

```
get(gcf, 'XDisplay')
ans
mycomputer:0.0
```

## Setting XDisplay on Motif

If your computer uses Motif-based figures, you can specify the display MATLAB uses for a figure by setting the value of the figure's XDisplay property. For example, to display the current figure on a system called fred, use the command

```
set(gcf, 'XDisplay', 'fred:0.0')
```

## XVisual

visual identifier (UNIX only)

*Select visual used by MATLAB.* You can select the visual used by MATLAB by setting the XVisual property to the desired visual ID. This can be useful if you want to test your application on an 8-bit or grayscale visual. To see what visuals are available on your system, use the UNIX xdpinfo command. From MATLAB, type

```
!xdpinfo
```

The information returned contains a line specifying the visual ID. For example:

```
visual id:    0x23
```

To use this visual with the current figure, set the XVisual property to the ID.

```
set(gcf, 'XVisual', '0x23')
```



To see which of the available visuals MATLAB can use, call `set` on the `XVisual` property:

```
set(gcf, 'XVisual')
```

The following typical output shows the visual being used (in curly braces) and other possible visuals. Note that MATLAB requires a `TrueColor` visual.

```
{ 0x23 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff) }
 0x24 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
 0x25 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
 0x26 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
 0x27 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
 0x28 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
 0x29 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
 0x2a (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
```

You can also use the `glxinfo` UNIX command to see what visuals are available for use with the OpenGL renderer. From MATLAB, type

```
!glxinfo
```

After providing information about the implementation of OpenGL on your system, `glxinfo` returns a table of visuals. The partial listing below shows typical output:

```
visual  x  bf lv rg d st colorbuffer ax dp st accumbuffer  ms  cav
id dep cl sp sz l  ci b ro  r  g  b  a bf th cl  r  g  b  a ns b eat
-----
-
0x23 24 tc  0 24  0 r y .  8  8  8  8  0  0  0  0  0  0  0  0  0  0  0  0  0  0 None
0x24 24 tc  0 24  0 r . .  8  8  8  8  0  0  0  0  0  0  0  0  0  0  0  0 None
0x25 24 tc  0 24  0 r y .  8  8  8  8  0 24  8  0  0  0  0  0  0  0  0 None
0x26 24 tc  0 24  0 r . .  8  8  8  8  0 24  8  0  0  0  0  0  0  0  0 None
0x27 24 tc  0 24  0 r y .  8  8  8  8  0  0  0  0 16 16 16  0  0  0 Slow
```

# Figure Properties

---

The third column is the class of visual. `tc` means a true color visual. Note that some visuals may be labeled `Slow` under the caveat column. Such visuals should be avoided.

To determine which visual MATLAB will use by default with the OpenGL renderer, use the MATLAB `opengl info` command. The returned entry for the visual might look like the following:

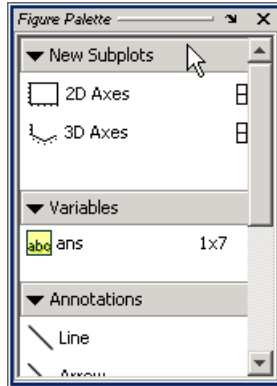
```
Visual = 0x23 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
```

Experimenting with a different TrueColor visual may improve certain rendering problems.



`XVisualMode`  
`auto | manual`

*Auto or manual selection of visual.* `XVisualMode` can take on two values — `auto` (the default) and `manual`. In `auto` mode, MATLAB selects the best visual to use based on the number of colors, availability of the OpenGL extension, etc. In `manual` mode, MATLAB does not change the visual from the one currently in use. Setting the `XVisual` property sets this property to `manual`.

**Purpose** Show or hide figure palette



## GUI Alternatives

Click the larger **Plotting Tools** icon  on the figure toolbar to collectively enable plotting tools, and the smaller icon  to collectively disable them. Open or close the **Figure Palette** tool from the figure's **View** menu. For details, see “The Figure Palette” in the MATLAB Graphics documentation.

## Syntax

```
figurepalette('show')
figurepalette('hide')
figurepalette('toggle')
figurepalette(figure_handle,...)
```

## Description

`figurepalette('show')` displays the palette on the current figure.

`figurepalette('hide')` hides the palette on the current figure.

`figurepalette('toggle')` or `figurepalette` toggles the visibility of the palette on the current figure.

`figurepalette(figure_handle,...)` shows or hides the palette on the figure specified by *figure\_handle*.

# figurepalette

---

## **See Also**

plottools, plotbrowser, propertyeditor

**Purpose**

Set or get attributes of file or directory

**Graphical Interface**

As an alternative to the `fileattrib` function, you can view attributes using the Current Directory browser.

**Syntax**

```
fileattrib
fileattrib('name')
fileattrib('name','attrib')
fileattrib('name','attrib','users')
fileattrib('name','attrib','users','s')
[status,message,messageid] = fileattrib('name','attrib',
    'users','s')
```

# fileattrib

---

## Description

The `fileattrib` function is like the DOS `attrib` command, or the `chmod` command used on UNIX<sup>8</sup> platforms.

`fileattrib` displays the attributes for the current directory. Values are as follows.

Value	Description
0	Attribute is off
1	Attribute is set (on)
NaN	Attribute does not apply

`fileattrib('name')` displays the attributes for `name`, where `name` is the absolute or relative path for a directory or file. Use the wildcard `*` at the end of `name` to view attributes for all matching files.

`fileattrib('name', 'attrib')` sets the attribute for `name`, where `name` is the absolute or relative path for a directory or file. Specify the `+` qualifier before the attribute to set it, and specify the `-` qualifier before the attribute to clear it. Use the wildcard `*` at the end of `name` to set attributes for all matching files. Values for `attrib` are as follows.

Value for attrib	Description
a	Archive (Microsoft Windows platform only)
h	Hidden file (Windows platform only)
s	System file (Windows platform only)
w	Write access (Windows and UNIX platforms)
x	Executable (UNIX platform only)

For example, `fileattrib('myfile.m', '+w')` makes `myfile.m` a writable file.

8. UNIX is a registered trademark of The Open Group in the United States and other countries.

`fileattrib('name','attrib','users')` sets the attribute for `name`, where `name` is the absolute or relative path for a directory or file, and defines which users are affected by `attrib`, where `users` is applicable only for UNIX platforms. For more information about these attributes, see reference information for `chmod` in UNIX operating system documentation. The default value for `users` is `u`. Values for `users` are listed here.

Value for users	Description
a	All users
g	Group of users
o	All other users
u	Current user

`fileattrib('name','attrib','users','s')` sets the attribute for `name`, where `name` is the absolute or relative path for a file or a directory and its contents, and defines which users are affected by `attrib`. Here the `s` specifies that `attrib` be applied to all contents of `name`, where `name` is a directory.

`[status,message,messageid] = fileattrib('name','attrib','users','s')` sets the attribute for `name`, returning the status, a message, and the MATLAB error message ID (see `error` and `lasterror`). Here, `status` is 1 for success and is 0 for error. If `attrib`, `users`, and `s` are not specified, and `status` is 1, `message` is a structure containing the file attributes and `messageid` is blank. If `status` is 0, `messageid` contains the error. If you use a wildcard `*` at the end of `name`, `mess` will be a structure.

## Remarks

Not all platforms or applications respond to file attributes that you have set in the same way. For example, even though you disable the “write” privilege for a directory with `fileattrib`, files in that directory could still be writable on some platforms or by certain applications.

## Examples

### Get Attributes of File

To view the attributes of `myfile.m`, type

```
fileattrib('myfile.m')
```

MATLAB returns

```
      Name: 'd:/work/myfile.m'  
      archive: 0  
      system: 0  
      hidden: 0  
      directory: 0  
      UserRead: 1  
      UserWrite: 0  
      UserExecute: 1  
      GroupRead: NaN  
      GroupWrite: NaN  
      GroupExecute: NaN  
      OtherRead: NaN  
      OtherWrite: NaN  
      OtherExecute: NaN
```

`UserWrite` is 0, meaning `myfile.m` is read only. The `Group` and `Other` values are `NaN` because they do not apply to the current operating system, Windows.

### Set File Attribute

To make `myfile.m` become writable, type

```
fileattrib('myfile.m','+w')
```

Running `fileattrib('myfile.m')` now shows `UserWrite` to be 1.

### Set Attributes for Specified Users

To make the directory `d:/work/results` be a read-only directory for all users, type

```
fileattrib('d:/work/results','-w','a')
```



The - preceding the write attribute, w, specifies that write status is removed.

### Set Multiple Attributes for Directory and Its Contents

To make the directory `d:/work/results` and all its contents be read only and be hidden, on Windows platforms, type

```
fileattrib('d:/work/results','+h-w','','s')
```

Because *users* is not applicable on Windows systems, its value is empty. Here, *s* applies the attribute to the contents of the specified directory.

### Return Status and Structure of Attributes

To return the attributes for the directory `results` to a structure, type

```
[stat,mess]=fileattrib('results')
```

MATLAB returns

```
stat =  
    1  
  
mess =  
    Name: 'd:\work\results'  
  archive: 0  
  system: 0  
  hidden: 0  
 directory: 1  
  UserRead: 1  
  UserWrite: 1  
  UserExecute: 1  
  GroupRead: NaN  
  GroupWrite: NaN  
  GroupExecute: NaN  
  OtherRead: NaN  
  OtherWrite: NaN  
  OtherExecute: NaN
```

# fileattrib

---

The operation was successful as indicated by the status, `stat`, being 1. The structure `mess` contains the file attributes. Access the attribute values in the structure. For example, typing

```
mess.Name
```

returns the path for results

```
ans =  
d:\work\results
```

## Return Attributes with Wildcard for Name

Return the attributes for all files in the current directory whose names begin with `new`.

```
[stat,mess]=fileattrib('new*')
```

MATLAB returns

```
stat =  
    1  
  
mess =  
1x3 struct array with fields:  
    Name  
    archive  
    system  
    hidden  
    directory  
    UserRead  
    UserWrite  
    UserExecute  
    GroupRead  
    GroupWrite  
    GroupExecute  
    OtherRead  
    OtherWrite  
    OtherExecute
```

The results indicate there are three matching files. To view the file names, type

```
mess.Name
```

MATLAB returns

```
ans =  
d:\work\results\newname.m
```

```
ans =  
d:\work\results\newone.m
```

```
ans =  
d:\work\results\newtest.m
```

To view just the first file name, type

```
mess(1).Name
```

```
ans =  
d:\work\results\newname.m
```

**See Also**

copyfile, cd, dir, filebrowser, fileparts, ls, mfilename, mkdir, movefile, rmdir

“Managing Files and Working with the Current Directory”

# filebrowser

**Purpose** Current Directory browser

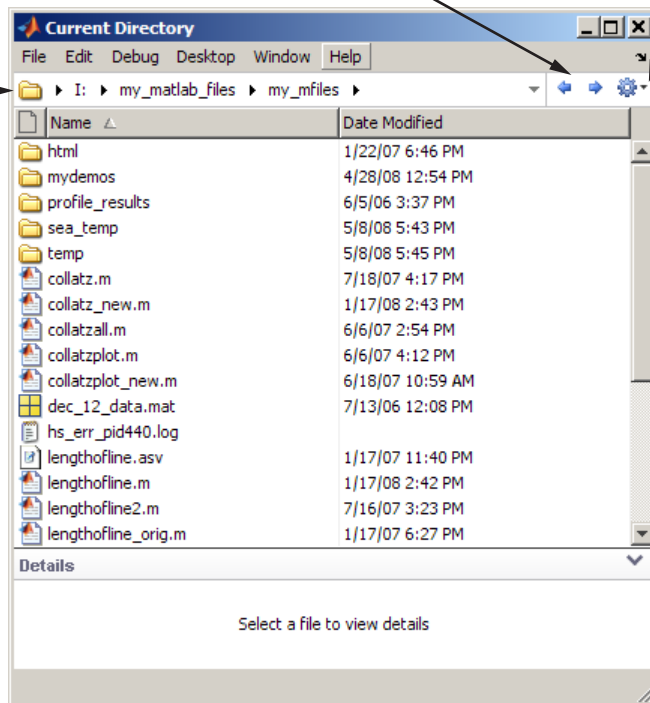
**Graphical Interface** As an alternative to the filebrowser function, select **Desktop > Current Directory** in the MATLAB desktop.

**Syntax** filebrowser

**Description** filebrowser displays the Current Directory Browser:

To add buttons, right-click the toolbar and select **Customize**. Select operations from the Actions button.

View subdirectories and change the current directory using the address bar.



**See Also**

`cd`, `copyfile`, `fileattrib`, `ls`, `mkdir`, `movefile`, `pwd`, `rmdir`  
“Managing Files and Working with the Current Directory”

# File Formats

---

**Purpose**            Readable file formats

**Description**        This table shows the file formats that the MATLAB software is capable of reading.

<b>File Format</b>	<b>Extension</b>	<b>File Content</b>	<b>Read Command</b>	<b>Returns</b>
Text	MAT	Saved MATLAB workspace	load	Variables in file
	CSV	Comma-separated numbers	csvread	Double array
	DAT	Formatted text	importdata	Double array
	DLM	Delimited text	dlmread	Double array
	TAB	Tab-separated text	dlmread	Double array
Scientific Data	CDF	Data in Common Data Format	cdfread	Cell array of CDF records
	FITS	Flexible Image Transport System data	fitsread	Primary or extension table data
	HDF4	Data in Hierarchical Data Format, version 4	hdfread	HDF 4 or HDF-EOS 2 data set
	HDF5	Data in Hierarchical Data Format, version 5	hdf5read	HDF5 or HDF-EOS 5 data set
	NC	Data in Network Common Data Form (netCDF)	See netcdf	Numeric array data

<b>File Format</b>	<b>Extension</b>	<b>File Content</b>	<b>Read Command</b>	<b>Returns</b>
Spreadsheet	XLS	Microsoft Excel worksheet	xlsread	Double or cell array
	WK1	Lotus 123 worksheet	wk1read	Double or cell array

# File Formats

File Format	Extension	File Content	Read Command	Returns
Image	BMP	BMP image	imread	True color or indexed image
	CUR	Cursor image	imread	Indexed image
	GIF	GIF image	imread	Indexed image
	HDF4	HDF4 image	imread	True color, grayscale, or indexed image(s)
	ICO	Icon image	imread	Indexed image
	JPEG	JPEG image	imread	True color or grayscale image
	JP2	JPEG2000 image	imread	True color or grayscale image
	PBM	PBM image	imread	Grayscale image
	PCX	PCX image	imread	Indexed image
	PGM	PGM image	imread	Grayscale image
	PNG	PNG image	imread	True color, grayscale, or indexed image
	PPM	PPM image	imread	True color image
	RAS	SUN raster image	imread	True color or indexed
	TIFF	TIFF image	imread	True color, grayscale, or indexed image(s)
XWD	XWD image	imread	Indexed image	
Video (all platforms)	AVI	AVI movie	aviread	MATLAB movie



<b>File Format</b>	<b>Extension</b>	<b>File Content</b>	<b>Read Command</b>	<b>Returns</b>
Video (Windows and Mac® platforms)	AVI	AVI movie	mmreader	True color image(s)
	MPG	MPEG 1	mmreader	True color image(s)
	MPEG	MPEG 1 and MPEG 2 videos	mmreader	True color image(s)
Video (Windows platform only)	ASF	Windows Media video	mmreader	True color image(s)
	ASX	Windows Media video	mmreader	True color image(s)
	WMV	Windows Media video	mmreader	True color image(s)
Video (Mac platform only)	3GP	3GPP mobile video	mmreader	True color image(s)
	3G2	3GPP2 mobile video	mmreader	True color image(s)
	DV	Digital video stream	mmreader	True color image(s)
	MP4	MPEG-4 video	mmreader	True color image(s)
	M4V	MPEG-4 video	mmreader	True color image(s)
	MOV	Quicktime movie	mmreader	True color image(s)

# File Formats

---

File Format	Extension	File Content	Read Command	Returns
Audio file	AU	NeXT/SUN sound	auread	Sound data and sample rate
	SND	NeXT/SUN sound	auread	Sound data and sample rate
	WAV	Microsoft WAVE sound	wavread	Sound data and sample rate

**See Also** `imformats`, `fscanf`, `fread`, `textread`, `importdata`, `hdf`, `hdf5`

<b>Purpose</b>	Construct file handle object
<b>Syntax</b>	<code>output = filehandle(arglist)</code>
<b>Description</b>	<code>output = filehandle(arglist)</code> this file is a place-holder for now.
<b>Example</b>	
<b>See Also</b>	<code>dialog</code> , <code>errordlg</code> , <code>helpdlg</code> , <code>listdlg</code> , <code>msgbox</code> , <code>questdlg</code> , <code>warndlg</code> <code>figure</code> , <code>uiwait</code> , <code>uiresume</code> “Predefined Dialog Boxes” on page 1-108 for related functions

# filemarker

---

- Purpose** Character to separate file name and internal function name
- Syntax** `M = filemarker`
- Description** `M = filemarker` returns the character that separates a file and a within-file function name.
- Examples** On the Microsoft Windows platform, for example, `filemarker` returns the `'>'` character:

```
filemarker
ans =
>
```

You can use the following command on any platform to get the help text for subfunction `pdeodes` defined in M-file `pdepe.m`:

```
helptext = help(['pdepe' filemarker 'pdeodes'])

helptext =
PDEODES Assemble the difference equations and
        evaluate the time derivative for the ODE system.
```

You can use the `filemarker` character to indicate a location within an M-file where you want to set a breakpoint, for example. On all platforms, if you need to distinguish between two nested functions with the same name, use the forward slash (`/`) character to indicate the path to a particular instance of a function.

For instance, suppose an M-file, `myfile.m`, contains the following code:

```
function x = A(p1, p2)
...
    function y = B(p3)
        ...
    end
    function m = C(p4)
        ...
```

```
        end
    end

    function z = C(p5)
    ...
        function y = D(p6)
        ...
        end
    end
end
```

To indicate that you want to set a breakpoint at function `y` nested within function `x`, use the following command on the Windows platform:

```
dbstop myfile>x/y
```

To indicate that you want to set a breakpoint at function `m` nested within function `x` use the following command on the Windows platform:

```
dbstop myfile>m
```

In the first case, you specify `x/y` because the M-file contains two nested functions named `y`. In the second case, there is no need to specify `x/m` because there is only one function `m` within `myfile.m`.

## See Also

`filesep`

# fileparts

---

**Purpose** Parts of file name and path

**Syntax** `[pathstr, name, ext, versn] = fileparts(filename)`

**Description** `[pathstr, name, ext, versn] = fileparts(filename)` returns the path, filename, extension, and version for the specified file. `filename` is a string enclosed in single quotes. The returned `ext` field contains a dot (.) before the file extension.

The `fileparts` function is platform dependent.

You can reconstruct the file from the parts using

```
fullfile(pathstr,[name ext versn])
```

## Examples

Return the pieces of a file specification string to the separate string outputs `pathstr`, `name`, `ext`, and `versn`. The full file specification is

```
file = '\home\user4\matlab\classpath.txt';
```

Note that the character used to separate the segments of a pathname is dependent on the operating system you are currently running on. In this example, it is the backslash (\) character which is used as a separator on Microsoft Windows platforms. You can use the `filesep` function as shown below to insert the correct separator character:

```
sep = filesep;  
file = [' ' sep 'home' sep 'user4' sep 'matlab' sep ...  
       'classpath.txt' ''];
```

Now use `fileparts` to return the path, filename, user name, and file version, if there is one:

```
[pathstr, name, ext, versn] = fileparts(file)  
  
pathstr =  
    \home\user4\matlab
```

```
name =  
    classpath  
  
ext =  
    .txt  
  
versn =  
    ''
```

**Remarks**

On Windows platforms, `C:\` and `C:` are two distinct entities, where `C:\` (with backslash) is the `C` drive in your computer, and `C:` (without backslash) represents your current working directory.

**See Also**

`fullfile`

# fileread

---

**Purpose** Return contents of file as string vector

**Syntax** `text = fileread('filename')`

**Description** `text = fileread('filename')` returns the contents of the file `filename` as a MATLAB string.

**Examples** Read the M-file help for this function. Search the resulting character array for the call to `fread`.

```
helptext = fileread('fileread.m');  
  
expr = '^[^n]*fread[^n]*';  
fread_line = regexp(helptext, expr, 'match')  
fread_line =  
    '    out = fread(fid, '*char')';'
```

**See Also** `fread`, `textscan`, `load`, `web`



**Purpose** Directory separator for current platform

**Syntax** `f = filesep`

**Description** `f = filesep` returns the platform-specific file separator character. The file separator is the character that separates individual directory names in a path string.

**Examples** Create a path to the `iofun` directory on a Microsoft Windows platform:

```
iofun_dir = ['toolbox' filesep 'matlab' filesep 'iofun']
```

```
iofun_dir =  
    toolbox\matlab\iofun
```

# filesep

---

Create a path to `iofun` on a UNIX<sup>9</sup> platform.

```
iodir = ['toolbox' filesep 'matlab' filesep 'iofun']
```

```
iodir =  
    toolbox/matlab/iofun
```

## See Also

`fullfile`, `fileparts`, `pathsep`

9. UNIX is a registered trademark of The Open Group in the United States and other countries.

**Purpose**

Filled 2-D polygons

**Syntax**

```
fill(X,Y,C)
fill(X,Y,ColorSpec)
fill(X1,Y1,C1,X2,Y2,C2,...)
fill(...,'PropertyName',PropertyValue)
h = fill(...)
```

**Description**

The `fill` function creates colored polygons.

`fill(X,Y,C)` creates filled polygons from the data in `X` and `Y` with vertex color specified by `C`. `C` is a vector or matrix used as an index into the colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`. If necessary, `fill` closes the polygon by connecting the last vertex to the first.

`fill(X,Y,ColorSpec)` fills two-dimensional polygons specified by `X` and `Y` with the color specified by `ColorSpec`.

`fill(X1,Y1,C1,X2,Y2,C2,...)` specifies multiple two-dimensional filled areas.

`fill(...,'PropertyName',PropertyValue)` allows you to specify property names and values for a patch graphics object.

`h = fill(...)` returns a vector of handles to patch graphics objects, one handle per patch object.

**Remarks**

If `X` or `Y` is a matrix, and the other is a column vector with the same number of elements as rows in the matrix, `fill` replicates the column vector argument to produce a matrix of the required size. `fill` forms a vertex from corresponding elements in `X` and `Y` and creates one polygon from the data in each column.

The type of color shading depends on how you specify color in the argument list. If you specify color using `ColorSpec`, `fill` generates flat-shaded polygons by setting the patch object's `FaceColor` property to the corresponding RGB triple.

If you specify color using `C`, `fill` scales the elements of `C` by the values specified by the axes property `CLim`. After scaling `C`, `C` indexes the current colormap.

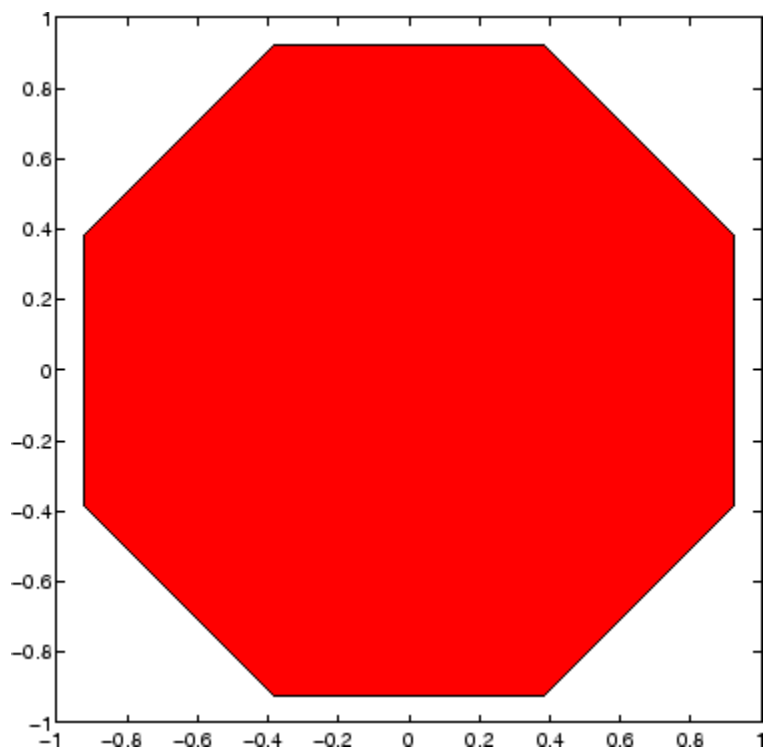
If `C` is a row vector, `fill` generates flat-shaded polygons where each element determines the color of the polygon defined by the respective column of the `X` and `Y` matrices. Each patch object's `FaceColor` property is set to `'flat'`. Each row element becomes the `CData` property value for the  $n$ th patch object, where  $n$  is the corresponding column in `X` or `Y`.

If `C` is a column vector or a matrix, `fill` uses a linear interpolation of the vertex colors to generate polygons with interpolated colors. It sets the patch graphics object `FaceColor` property to `'interp'` and the elements in one column become the `CData` property value for the respective patch object. If `C` is a column vector, `fill` replicates the column vector to produce the required sized matrix.

## Examples

Create a red octagon.

```
t = (1/16:1/8:1)'*2*pi;  
x = sin(t);  
y = cos(t);  
fill(x,y,'r')  
axis square
```

**See Also**

`axis`, `caxis`, `colormap`, `ColorSpec`, `fill3`, `patch`

“Polygons and Surfaces” on page 1-94 for related functions

# fill3

---

## Purpose

Filled 3-D polygons



## Syntax

```
fill3(X,Y,Z,C)
fill3(X,Y,Z,ColorSpec)
fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)
fill3(...,'PropertyName',PropertyValue)
h = fill3(...)
```

## Description

The `fill3` function creates flat-shaded and Gouraud-shaded polygons.

`fill3(X,Y,Z,C)` fills three-dimensional polygons. `X`, `Y`, and `Z` triplets specify the polygon vertices. If `X`, `Y`, or `Z` is a matrix, `fill3` creates  $n$  polygons, where  $n$  is the number of columns in the matrix. `fill3` closes the polygons by connecting the last vertex to the first when necessary.

`C` specifies color, where `C` is a vector or matrix of indices into the current colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`.

`fill3(X,Y,Z,ColorSpec)` fills three-dimensional polygons defined by `X`, `Y`, and `Z` with color specified by `ColorSpec`.

`fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)` specifies multiple filled three-dimensional areas.

`fill3(...,'PropertyName',PropertyValue)` allows you to set values for specific patch properties.

`h = fill3(...)` returns a vector of handles to patch graphics objects, one handle per patch.

## Algorithm

If `X`, `Y`, and `Z` are matrices of the same size, `fill3` forms a vertex from the corresponding elements of `X`, `Y`, and `Z` (all from the same matrix location), and creates one polygon from the data in each column.

If  $X$ ,  $Y$ , or  $Z$  is a matrix, `fill3` replicates any column vector argument to produce matrices of the required size.

If you specify color using `ColorSpec`, `fill3` generates flat-shaded polygons and sets the patch object `FaceColor` property to an RGB triple.

If you specify color using  $C$ , `fill3` scales the elements of  $C$  by the axes property `CLim`, which specifies the color axis scaling parameters, before indexing the current colormap.

If  $C$  is a row vector, `fill3` generates flat-shaded polygons and sets the `FaceColor` property of the patch objects to 'flat'. Each element becomes the `CData` property value for the respective patch object.

If  $C$  is a column vector or a matrix, `fill3` generates polygons with interpolated colors and sets the patch object `FaceColor` property to 'interp'. `fill3` uses a linear interpolation of the vertex colormap indices when generating polygons with interpolated colors. The elements in one column become the `CData` property value for the respective patch object. If  $C$  is a column vector, `fill3` replicates the column vector to produce the required sized matrix.

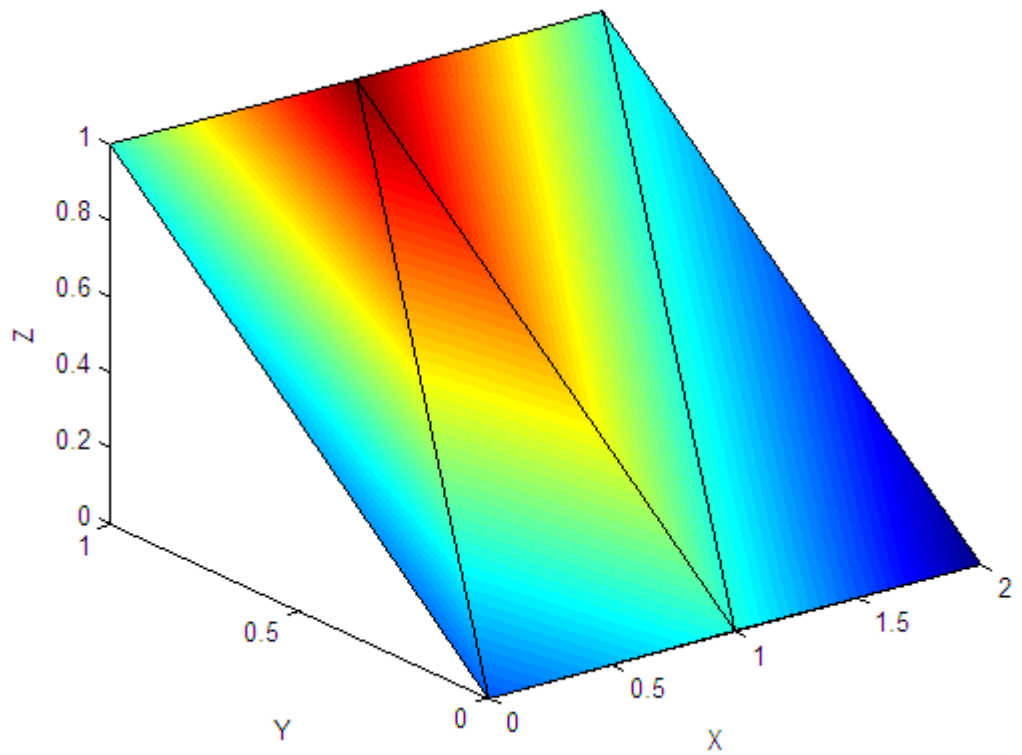
## Examples

Create four triangles with interpolated colors.

```
X = [0 1 1 2;1 1 2 2;0 0 1 1];
Y = [1 1 1 1;1 0 1 0;0 0 0 0];
Z = [1 1 1 1;1 0 1 0;0 0 0 0];
C = [0.5000 1.0000 1.0000 0.5000;
     1.0000 0.5000 0.5000 0.1667;
     0.3330 0.3330 0.5000 0.5000];
fill3(X,Y,Z,C)
```

## fill3

---



### See Also

`axis`, `caxis`, `colormap`, `ColorSpec`, `fill`, `patch`

“Polygons and Surfaces” on page 1-94 for related functions



**Purpose**

1-D digital filter

**Syntax**

```
y = filter(b,a,X)
[y,zf] = filter(b,a,X)
[y,zf] = filter(b,a,X,zi)
y = filter(b,a,X,zi,dim)
[... ] = filter(b,a,X,[],dim)
```

**Description**

The `filter` function filters a data sequence using a digital filter which works for both real and complex inputs. The filter is a *direct form II transposed* implementation of the standard difference equation (see “Algorithm”).

`y = filter(b,a,X)` filters the data in vector `X` with the filter described by numerator coefficient vector `b` and denominator coefficient vector `a`. If `a(1)` is not equal to 1, `filter` normalizes the filter coefficients by `a(1)`. If `a(1)` equals 0, `filter` returns an error.

If `X` is a matrix, `filter` operates on the columns of `X`. If `X` is a multidimensional array, `filter` operates on the first nonsingleton dimension.

`[y,zf] = filter(b,a,X)` returns the final conditions, `zf`, of the filter delays. If `X` is a row or column vector, output `zf` is a column vector of  $\max(\text{length}(a), \text{length}(b)) - 1$ . If `X` is a matrix, `zf` is an array of such vectors, one for each column of `X`, and similarly for multidimensional arrays.

`[y,zf] = filter(b,a,X,zi)` accepts initial conditions, `zi`, and returns the final conditions, `zf`, of the filter delays. Input `zi` is a vector of length  $\max(\text{length}(a), \text{length}(b)) - 1$ , or an array with the leading dimension of size  $\max(\text{length}(a), \text{length}(b)) - 1$  and with remaining dimensions matching those of `X`.

`y = filter(b,a,X,zi,dim)` and `[... ] = filter(b,a,X,[],dim)` operate across the dimension `dim`.

# filter

## Example

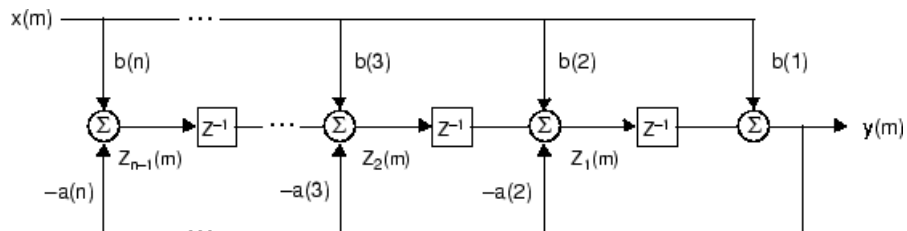
You can use `filter` to find a running average without using a `for` loop. This example finds the running average of a 16-element vector, using a window size of 5.

```
data = [1:0.2:4]';  
windowSize = 5;  
filter(ones(1,windowSize)/windowSize,1,data)
```

```
ans =  
    0.2000  
    0.4400  
    0.7200  
    1.0400  
    1.4000  
    1.6000  
    1.8000  
    2.0000  
    2.2000  
    2.4000  
    2.6000  
    2.8000  
    3.0000  
    3.2000  
    3.4000  
    3.6000
```

## Algorithm

The `filter` function is implemented as a direct form II transposed structure,



or

$$y(n) = b(1)*x(n) + b(2)*x(n-1) + \dots + b(nb+1)*x(n-nb) \\ - a(2)*y(n-1) - \dots - a(na+1)*y(n-na)$$

where  $n-1$  is the filter order, which handles both FIR and IIR filters [1],  $na$  is the feedback filter order, and  $nb$  is the feedforward filter order.

The operation of `filter` at sample  $m$  is given by the time domain difference equations

$$y(m) = b(1)x(m) + z_1(m-1)$$

$$z_1(m) = b(2)x(m) + z_2(m-1) - a(2)y(m)$$

$$\vdots = \vdots \quad \vdots$$

$$z_{n-2}(m) = b(n-1)x(m) + z_{n-1}(m-1) - a(n-1)y(m)$$

$$z_{n-1}(m) = b(n)x(m) - a(n)y(m)$$

The input-output description of this filtering operation in the  $z$ -transform domain is a rational transfer function,

$$Y(z) = \frac{b(1) + b(2)z^{-1} + \dots + b(nb+1)z^{-nb}}{1 + a(2)z^{-1} + \dots + a(na+1)z^{-na}} X(z)$$

## See Also

`filter2`

`filtfilt`, `filtic` in the Signal Processing Toolbox

## References

[1] Oppenheim, A. V. and R.W. Schaffer. *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989, pp. 311-312.

# filter (timeseries)

---

**Purpose** Shape frequency content of time series

**Syntax**  
`ts2 = filter(ts1,b,a)`  
`ts2 = filter(ts1,b,a,Index)`

**Description** `ts2 = filter(ts1,b,a)` applies the transfer function filter  $b(z^{-1})/a(z^{-1})$  to the data in the timeseries object `ts1`.  
`b` and `a` are the coefficient arrays of the transfer function numerator and denominator, respectively.  
`ts2 = filter(ts1,b,a,Index)` uses the optional `Index` integer array to specify the columns or rows to filter. 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** The time-series data must be uniformly sampled to use this filter.

The following function

$$y = \text{filter}(b,a,x)$$

creates filtered data `y` by processing the data in vector `x` with the filter described by vectors `a` and `b`.

The `filter` function is a general tapped delay-line filter, described by the difference equation

$$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \dots + b(nb)x(n-nb+1) \\ - a(2)y(n-1) - \dots - a(N_a)y(n-N_b+1)$$

Here,  $n$  is the index of the current sample,  $N_a$  is the order of the polynomial described by vector `a`, and  $N_b$  is the order of the polynomial described by vector `b`. The output  $y(n)$  is a linear combination of current and previous inputs,  $x(n) x(n-1)\dots$ , and previous outputs,  $y(n-1) y(n-2)\dots$ .

You use the discrete filter to shape the data by applying a transfer function to the input signal.

Depending on your objectives, the transfer function you choose might alter both the amplitude and the phase of the variations in the data at different frequencies to produce either a smoother or a rougher output.

In digital signal processing (DSP), it is customary to write transfer functions as rational expressions in  $z^{-1}$  and to order the numerator and denominator terms in ascending powers of  $z^{-1}$ .

Taking the z-transform of the difference equation

$$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \dots + b(nb)x(n-nb+1) \\ - a(2)y(n-1) - \dots - a(na)y(n-na+1)$$

results in the transfer function

$$Y(z) = H(z^{-1})X(z) = \frac{b(1) + b(2)z^{-1} + \dots + b(nb)z^{-nb+1}}{a(1) + a(2)z^{-1} + \dots + a(na)z^{-na+1}}X(z)$$

where  $Y(z)$  is the z-transform of the filtered output  $y(n)$ . The coefficients  $b$  and  $a$  are unchanged by the z-transform.

## Examples

Consider the following transfer function:

$$H(z^{-1}) = \frac{b(z^{-1})}{a(z^{-1})} = \frac{2 + 3z^{-1}}{1 + 0.2z^{-1}}$$

You will apply this transfer function to the data in `count.dat`.

**1** Load the matrix `count` into the workspace.

```
load count.dat;
```

**2** Create a time-series object based on this matrix.

```
count1=timeseries(count(:,1),[1:24]);
```

## filter (timeseries)

---

- 3** Enter the coefficients of the denominator ordered in ascending powers of  $z^{-1}$  to represent  $1 + 0.2z^{-1}$ .

```
a = [1 0.2];
```

- 4** Enter the coefficients of the numerator to represent  $2 + 3z^{-1}$ .

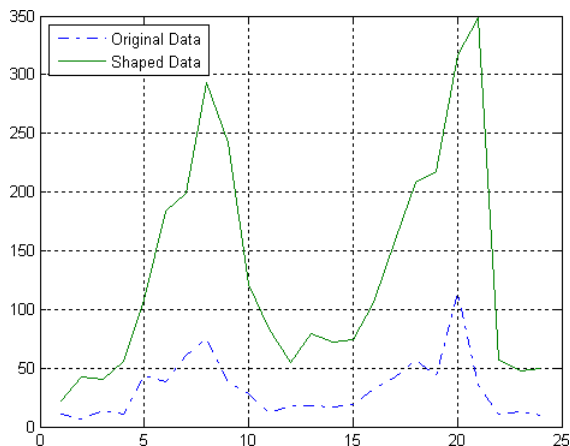
```
b = [2 3];
```

- 5** Call the filter function.

```
filter_count = filter(count1,b,a)
```

- 6** Compare the original data and the shaped data with an overlaid plot of the two curves:

```
plot(count1,'-.'), grid on, hold on  
plot(filter_count,'-')  
legend('Original Data','Shaped Data',2)
```



### See Also

`idealfilter (timeseries), timeseries, tsprops`

---

<b>Purpose</b>	2-D digital filter
<b>Syntax</b>	<pre>Y = filter2(h,X) Y = filter2(h,X,shape)</pre>
<b>Description</b>	<p><code>Y = filter2(h,X)</code> filters the data in <code>X</code> with the two-dimensional FIR filter in the matrix <code>h</code>. It computes the result, <code>Y</code>, using two-dimensional correlation, and returns the central part of the correlation that is the same size as <code>X</code>.</p> <p><code>Y = filter2(h,X,shape)</code> returns the part of <code>Y</code> specified by the <code>shape</code> parameter. <code>shape</code> is a string with one of these values:</p> <ul style="list-style-type: none"><li><code>'full'</code> Returns the full two-dimensional correlation. In this case, <code>Y</code> is larger than <code>X</code>.</li><li><code>'same'</code> (default) Returns the central part of the correlation. In this case, <code>Y</code> is the same size as <code>X</code>.</li><li><code>'valid'</code> Returns only those parts of the correlation that are computed without zero-padded edges. In this case, <code>Y</code> is smaller than <code>X</code>.</li></ul>
<b>Remarks</b>	Two-dimensional correlation is equivalent to two-dimensional convolution with the filter matrix rotated 180 degrees. See the Algorithm section for more information about how <code>filter2</code> performs linear filtering.
<b>Algorithm</b>	<p>Given a matrix <code>X</code> and a two-dimensional FIR filter <code>h</code>, <code>filter2</code> rotates your filter matrix 180 degrees to create a convolution kernel. It then calls <code>conv2</code>, the two-dimensional convolution function, to implement the filtering operation.</p> <p><code>filter2</code> uses <code>conv2</code> to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, <code>filter2</code> then extracts the central part of the convolution that is the same size as the input</p>

## filter2

---

matrix, and returns this as the result. If the shape parameter specifies an alternate part of the convolution for the result, `filter2` returns the appropriate part.

### See Also

`conv2`, `filter`



**Purpose** Find indices and values of nonzero elements

**Syntax**

```
ind = find(X)
ind = find(X, k)
ind = find(X, k, 'first')
ind = find(X, k, 'last')
[row,col] = find(X, ...)
[row,col,v] = find(X, ...)
```

**Description** `ind = find(X)` locates all nonzero elements of array `X`, and returns the linear indices of those elements in vector `ind`. If `X` is a row vector, then `ind` is a row vector; otherwise, `ind` is a column vector. If `X` contains no nonzero elements or is an empty array, then `ind` is an empty array.

`ind = find(X, k)` or `ind = find(X, k, 'first')` returns at most the first `k` indices corresponding to the nonzero entries of `X`. `k` must be a positive integer, but it can be of any numeric data type.

`ind = find(X, k, 'last')` returns at most the last `k` indices corresponding to the nonzero entries of `X`.

`[row,col] = find(X, ...)` returns the row and column indices of the nonzero entries in the matrix `X`. This syntax is especially useful when working with sparse matrices. If `X` is an `N`-dimensional array with `N > 2`, `col` contains linear indices for the columns. For example, for a 5-by-7-by-3 array `X` with a nonzero element at `X(4,2,3)`, `find` returns 4 in `row` and 16 in `col`. That is, (7 columns in page 1) + (7 columns in page 2) + (2 columns in page 3) = 16.

`[row,col,v] = find(X, ...)` returns a column or row vector `v` of the nonzero entries in `X`, as well as row and column indices. If `X` is a logical expression, then `v` is a logical array. Output `v` contains the non-zero elements of the logical array obtained by evaluating the expression `X`. For example,

```
A= magic(4)
A =
    16     2     3    13
     5    11    10     8
```

# find

---

```
     9     7     6    12
     4    14    15     1
```

```
[r,c,v]= find(A>10);
```

```
r', c', v'
ans =
     1     2     4     4     1     3
ans =
     1     2     2     3     4     4
ans =
     1     1     1     1     1     1
```

Here the returned vector  $v$  is a logical array that contains the nonzero elements of  $N$  where

```
N=(A>10)
```

## Examples

### Example 1

```
X = [1 0 4 -3 0 0 0 8 6];
indices = find(X)
```

returns linear indices for the nonzero entries of  $X$ .

```
indices =
     1     3     4     8     9
```

### Example 2

You can use a logical expression to define  $X$ . For example,

```
find(X > 2)
```

returns linear indices corresponding to the entries of  $X$  that are greater than 2.

```
ans =
     3     8     9
```

**Example 3**

The following `find` command

```
X = [3 2 0; -5 0 7; 0 0 1];  
[r,c,v] = find(X)
```

returns a vector of row indices of the nonzero entries of X

```
r =  
    1  
    2  
    1  
    2  
    3
```

a vector of column indices of the nonzero entries of X

```
c =  
    1  
    1  
    2  
    3  
    3
```

and a vector containing the nonzero entries of X.

```
v =  
    3  
   -5  
    2  
    7  
    1
```

**Example 4**

The expression

```
[r,c,v] = find(X>2)
```

# find

---

returns a vector of row indices of the nonzero entries of  $X$

```
r =  
    1  
    2
```

a vector of column indices of the nonzero entries of  $X$

```
c =  
    1  
    3
```

and a logical array that contains the non zero elements of  $N$  where  $N=(X>2)$ .

```
v =  
    1  
    1
```

Recall that when you use `find` on a logical expression, the output vector `v` does not contain the nonzero entries of the input array. Instead, it contains the nonzero values returned after evaluating the logical expression.

## Example 5

Some operations on a vector

```
x = [11 0 33 0 55]';
```

```
find(x)  
ans =  
    1  
    3  
    5
```

```
find(x == 0)  
ans =  
    2
```

```
4  
  
find(0 < x & x < 10*pi)  
ans =  
1
```

### Example 6

For the matrix

```
M = magic(3)  
M =  
    8    1    6  
    3    5    7  
    4    9    2  
  
find(M > 3, 4)
```

returns the indices of the first four entries of *M* that are greater than 3.

```
ans =  
    1  
    3  
    5  
    6
```

### Example 7

If *X* is a vector of all zeros, `find(X)` returns an empty matrix. For example,

```
indices = find([0;0;0])  
indices =  
Empty matrix: 0-by-1
```

## See Also

`nonzeros`, `sparse`, `colon`, logical operators (elementwise and short-circuit), relational operators, `ind2sub`

# findall

---

**Purpose** Find all graphics objects

**Syntax** `object_handles = findall(handle_list)`  
`object_handles = findall(handle_list, 'property', 'value', ...)`

**Description** `object_handles = findall(handle_list)` returns the handles, including hidden handles, of all objects in the hierarchy under the objects identified in `handle_list`.

`object_handles = findall(handle_list, 'property', 'value', ...)` returns the handles of all objects in the hierarchy under the objects identified in `handle_list` that have the specified properties set to the specified values.

**Remarks** `findall` is similar to `findobj`, except that it finds objects even if their `HandleVisibility` is set to off.

**Examples**

```
plot(1:10)
xlabel xlab
a = findall(gcf)
b = findobj(gcf)
c = findall(b, 'Type', 'text') % return the xlabel handle twice
d = findobj(b, 'Type', 'text') % can't find the xlabel handle
```

**See Also** `allchild`, `findobj`

**Purpose** Find visible offscreen figures

**Syntax** `findfigs`

**Description** `findfigs` finds all visible figure windows whose display area is off the screen and positions them on the screen.

A window appears to the MATLAB software to be offscreen when its display area (the area not covered by the window's title bar, menu bar, and toolbar) does not appear on the screen.

This function is useful when you are bringing an application from a larger monitor to a smaller one (or one with lower resolution). Windows visible on the larger monitor may appear offscreen on a smaller monitor. Using `findfigs` ensures that all windows appear on the screen.

**See Also** "Graphics Object Identification" on page 1-97 for related functions.

# findobj

---

## Purpose

Locate graphics objects with specific properties

## Syntax

```
h = findobj
h = findobj('PropertyName',PropertyValue,...)
h =
findobj('PropertyName',PropertyValue,'-logicaloperator',
    PropertyName',PropertyValue,...)
h = findobj('-regexp','PropertyName','regexp',...)
h = findobj('-property','PropertyName')
h = findobj(objhandles,...)
h = findobj(objhandles,'-depth',d,...)
h = findobj(objhandles,'flat','PropertyName',PropertyValue,
    ...)
```

## Description

`findobj` locates graphics objects and returns their handles. You can limit the search to objects with particular property values and along specific branches of the hierarchy.

`h = findobj` returns the handles of the root object and all its descendants.

`h = findobj('PropertyName',PropertyValue,...)` returns the handles of all graphics objects having the property *PropertyName*, set to the value *PropertyValue*. You can specify more than one property/value pair, in which case, `findobj` returns only those objects having all specified values.

`h = findobj('PropertyName',PropertyValue,'-logicaloperator',PropertyName',PropertyValue,...)` applies the logical operator to the property value matching. Possible values for *-logicaloperator* are:

- -and
- -or
- -xor
- -not



See the Examples section for examples of how to use these operators. See “Logical Operators” for an explanation of logical operators.

`h = findobj('-regex','PropertyName','regex',...)` matches objects using regular expressions as if the value of the property `PropertyName` was passed to the `regex` function as

```
regex(PropertyValue,'regex')
```

If a match occurs, `findobj` returns the object’s handle. See the `regex` function for information on how the MATLAB software uses regular expressions.

`h = findobj('-property','PropertyName')` finds all objects having the specified property.

`h = findobj(objhandles,...)` restricts the search to objects listed in `objhandles` and their descendants.

`h = findobj(objhandles,'-depth',d,...)` specified the depth of the search. The depth argument `d` controls how many levels under the handles in `objhandles` are traversed. Specifying `d` as `inf` to get the default behavior of all levels. Specify `d` as `0` to get the same behavior as using the `flat` argument.

`h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)` restricts the search to those objects listed in `objhandles` and does not search descendants.

## Remarks

`findobj` returns an error if a handle refers to a nonexistent graphics object.

`findobj` correctly matches any legal property value. For example,

```
findobj('Color','r')
```

finds all objects having a `Color` property set to `red`, `r`, or `[1 0 0]`.

When a graphics object is a descendant of more than one object identified in `objhandles`, MATLAB searches the object each time

# findobj

---

`findobj` encounters its handle. Therefore, implicit references to a graphics object can result in its handle being returned multiple times.

## Examples

Find all line objects in the current axes:

```
h = findobj(gca, 'Type', 'line')
```

Find all objects having a `Label` set to 'foo' and a `String` set to 'bar':

```
h = findobj('Label', 'foo', '-and', 'String', 'bar');
```

Find all objects whose `String` is not 'foo' and is not 'bar':

```
h = findobj('-not', 'String', 'foo', '-not', 'String', 'bar');
```

Find all objects having a `String` set to 'foo' and a `Tag` set to 'button one' and whose `Color` is not 'red' or 'blue':

```
h = findobj('String', 'foo', '-and', 'Tag', 'button one', ...  
    '-and', '-not', {'Color', 'red', '-or', 'Color', 'blue'})
```

Find all objects for which you have assigned a value to the `Tag` property (that is, the value is not the empty string ''):

```
h = findobj('-regexp', 'Tag', '[^'']')
```

Find all children of the current figure that have their `BackgroundColor` property set to a certain shade of gray ([.7 .7 .7]). Note that this statement also searches the current figure for the matching property value pair.

```
h = findobj(gcf, '-depth', 1, 'BackgroundColor', [.7 .7 .7])
```

## See Also

`copyobj`, `findall`, `gcf`, `gca`, `gcbo`, `gco`, `get`, `regexp`, `set`

See “Example — Using Logical Operators and Regular Expression” for more examples.

“Graphics Object Identification” on page 1-97 for related functions

**Purpose** Finds objects matching specified conditions

**Syntax** `Hmatch = findobj(Hobj,<conditions>)`

**Description** `Hmatch = findobj(Hobj,<conditions>)` finds handle objects that meet the specified conditions. The `Hobj` argument must be an array of handle objects and `Hmatch` is returned with an array of handles matching the conditions. Note that `findobj` has access only to public members of the objects in `Hobj`. See the Handle Graphics `findobj` function for information on specifying conditions.

**See Also** `findprop`, `handle`

# findprop (handle)

---

**Purpose** Find meta.property object associated with property name

**Syntax** `p = findprop(h, 'Name')`

**Description** `p = findprop(h, 'Name')` returns the meta.property object associated with the property Name of the object h. Name can be a property defined by the class of h or a dynamic property defined only for the object h.

**See Also** “Obtaining Information About Classes with Meta-Classes”  
“Dynamic Properties — Adding Properties to an Instance”  
`handle`, `findobj (handle)`, `meta.property`

**Purpose** Find string within another, longer string

**Syntax** `k = findstr(str1, str2)`

**Description** `k = findstr(str1, str2)` searches the longer of the two input strings for any occurrences of the shorter string, returning the starting index of each such occurrence in the double array `k`. If no occurrences are found, then `findstr` returns the empty array, `[]`.

The search performed by `findstr` is case sensitive. Any leading and trailing blanks in either input string are explicitly included in the comparison.

Unlike the `strfind` function, the order of the input arguments to `findstr` is not important. This can be useful if you are not certain which of the two input strings is the longer one.

**Examples**

```
s = 'Find the starting indices of the shorter string.';

findstr(s, 'the')
ans =
     6     30

findstr('the', s)
ans =
     6     30
```

**See Also** `strfind`, `strmatch`, `strtok`, `strcmp`, `strncmp`, `strcmpi`, `strncmpi`, `regexp`, `regexp`, `regexpi`, `regexprep`


# finish

---

**Purpose** Termination M-file for MATLAB program

**Description** When the MATLAB program quits, it runs a script called `finish.m`, if the script exists and is on the search path MATLAB uses or in the current directory. This is a file you create yourself that instructs MATLAB to perform any final tasks just prior to terminating. For example, you might want to save the data in your workspace to a MAT-file before MATLAB exits.

`finish.m` is invoked whenever you do one of the following:

- Click the Close box  in the MATLAB desktop on Microsoft Windows platforms or the equivalent on UNIX<sup>10</sup> platforms
- Select **Exit MATLAB** from the desktop **File** menu
- Type `quit` or `exit` at the Command Window prompt

**Remarks** When using Handle Graphics features in `finish.m`, use `uiwait`, `waitfor`, or `drawnow` so that figures are visible. See the reference pages for these functions for more information.

**Examples** Two sample `finish.m` files are provided with MATLAB in `matlabroot/toolbox/local`. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` and add it to the path to use it:

- `finishesav.m` — Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m` — Displays a dialog allowing you to cancel quitting and saves the workspace. See also the “Confirmation Dialogs Preferences” and the option for exiting MATLAB.

**See Also** `quit`, `exit`, `startup`

10. UNIX is a registered trademark of The Open Group in the United States and other countries.

“Quitting the MATLAB Program” in the MATLAB Desktop Tools and Development Environment documentation

# fitsinfo

---

**Purpose** Information about FITS file

**Syntax** `info = fitsinfo(filename)`

**Description** `info = fitsinfo(filename)` returns the structure, `info`, with fields that contain information about the contents of a Flexible Image Transport System (FITS) file. `filename` is a string enclosed in single quotes that specifies the name of the FITS file.

The `info` structure contains the following fields, listed in the order they appear in the structure. In addition, the `info` structure can also contain information about any number of optional file components, called *extensions* in FITS terminology. For more information, see “FITS File Extensions” on page 2-1291.

Field Name	Description	Return Type
Filename	Name of the file	String
FileModDate	File modification date	String
FileSize	Size of the file in bytes	Double
Contents	List of extensions in the file in the order that they occur	Cell array of strings
PrimaryData	Information about the primary data in the FITS file	Structure array

## PrimaryData

The `PrimaryData` field is a structure that describes the primary data in the file. The following table lists the fields in the order they appear in the structure.

Field Name	Description	Return Type
DataType	Precision of the data	String
Size	Array containing the size of each dimension	Double array



Field Name	Description	Return Type
DataSize	Size of the primary data in bytes	Double
MissingDataValue	Value used to represent undefined data	Double
Intercept	Value, used with Slope, to calculate actual pixel values from the array pixel values, using the equation: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	Double
Slope	Value, used with Intercept, to calculate actual pixel values from the array pixel values, using the equation: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	Double
Offset	Number of bytes from beginning of the file to the location of the first data value	Double
Keywords	A number-of-keywords-by-3 cell array containing keywords, values, and comments of the header in each column	Cell array of strings

## FITS File Extensions

A FITS file can also include optional extensions. If the file contains any of these extensions, the info structure can contain these additional fields.

- AsciiTable — Numeric information in tabular format, stored as ASCII characters

- BinaryTable — Numeric information in tabular format, stored in binary representation
- Image — A multidimensional array of pixels
- Unknown — Nonstandard extension

## AsciiTable Extension

The AsciiTable structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
Rows	Number of rows in the table	Double
RowSize	Number of characters in each row	Double
NFields	Number of fields in each row	Double array
FieldFormat	A 1-by-NFields cell containing formats in which each field is encoded. The formats are FORTRAN-77 format codes.	Cell array of strings
FieldPrecision	A 1-by-NFields cell containing precision of the data in each field	Cell array of strings
FieldWidth	A 1-by-NFields array containing the number of characters in each field	Double array
FieldPos	A 1-by-NFields array of numbers representing the starting column for each field	Double array
DataSize	Size of the data in the table in bytes	Double
MissingDataValue	A 1-by-NFields array of numbers used to represent undefined data in each field	Cell array of strings

Field Name	Description	Return Type
Intercept	A 1-by-NFields array of numbers used along with Slope to calculate actual data values from the array data values using the equation: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	Double array
Slope	A 1-by-NFields array of numbers used with Intercept to calculate true data values from the array data values using the equation: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	Double array
Offset	Number of bytes from beginning of the file to the location of the first data value in the table	Double
Keywords	A number-of-keywords-by-3 cell array containing all the Keywords, Values and Comments in the ASCII table header	Cell array of strings

### BinaryTable Extension

The BinaryTable structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
Rows	Number of rows in the table	Double
RowSize	Number of bytes in each row	Double
NFields	Number of fields in each row	Double

Field Name	Description	Return Type
FieldFormat	A 1-by-NFields cell array containing the data type of the data in each field. The data type is represented by a FITS binary table format code.	Cell array of strings
FieldPrecision	A 1-by-NFields cell containing precision of the data in each field	Cell array of strings
FieldSize	A 1-by-NFields array, where each element contains the number of values in the Nth field	Double array
DataSize	Size of the data in the Binary Table, in bytes. Includes any data past the main table.	Double
MissingDataValue	An 1-by-NFields array of numbers used to represent undefined data in each field	Cell array of double
Intercept	A 1-by-NFields array of numbers used along with Slope to calculate actual data values from the array data values using the equation: $actual\_value = slope * array\_value + Intercept$	Double array
Slope	A 1-by-NFields array of numbers used with Intercept to calculate true data values from the array data values using the equation: $actual\_value = Slope * array\_value + Intercept$	Double array

Field Name	Description	Return Type
Offset	Number of bytes from beginning of the file to the location of the first data value	Double
ExtensionSize	Size of any data past the main table, in bytes	Double
ExtensionOffset	Number of bytes from the beginning of the file to any data past the main table	Double
Keywords	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

### Image Extension

The Image structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
DataType	Precision of the data	String
Size	Array containing sizes of each dimension	Double array
DataSize	Size of the data in the Image extension in bytes	Double
Offset	Number of bytes from the beginning of the file to the first data value	Double
MissingDataValue	Value used to represent undefined data	Double

Field Name	Description	Return Type
Intercept	Value, used with Slope, to calculate actual pixel values from the array pixel values, using the equation: $actual\_value = Slope * array\_value + Intercept$	Double
Slope	Value, used with Intercept, to calculate actual pixel values from the array pixel values, using the equation: $actual\_value = Slope * array\_value + Intercept$	Double
Keywords	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

## Unknown Structure

The Unknown structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
DataType	Precision of the data	String
Size	Sizes of each dimension	Double array
DataSize	Size of the data in nonstandard extensions, in bytes	Double
Offset	Number of bytes from beginning of the file to the first data value	Double

Field Name	Description	Return Type
MissingDataValue	Representation of undefined data	Double
Intercept	Value, used with Slope, to calculate actual data values from the array data values, using the equation: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	Double
Slope	Value, used with Intercept, to calculate actual data values from the array data values, using the equation: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	Double
Keywords	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

## Example

Use `fitsinfo` to obtain information about the FITS file `tst0012.fits`. In addition to its primary data, the file also contains an example of the extensions `BinaryTable`, `Unknown`, `Image`, and `AsciiTable`.

```
S = fitsinfo('tst0012.fits');
S =
    Filename: [1x71 char]
    FileModDate: '12-Mar-2001 18:37:46'
    FileSize: 109440
    Contents: {'Primary' 'Binary Table' 'Unknown'
'Image' 'ASCII Table'}
    PrimaryData: [1x1 struct]
    BinaryTable: [1x1 struct]
```

```
Unknown: [1x1 struct]
Image: [1x1 struct]
AsciiTable: [1x1 struct]
```

The PrimaryData field describes the data in the file. For example, the Size field indicates the data is a 102-by-109 matrix.

```
S.PrimaryData
  DataType: 'single'
  Size: [102 109]
  DataSize: 44472
MissingDataValue: []
Intercept: 0
Slope: 1
Offset: 2880
Keywords: {25x3 cell}
```

The AsciiTable field describes the AsciiTable extension. For example, using the FieldWidth and FieldPos fields you can determine the length and location of each field within a row.

```
S.AsciiTable
ans =
  Rows: 53
  RowSize: 59
  NFields: 8
  FieldFormat: {'A9' 'F6.2' 'I3' 'E10.4' 'D20.15' 'A5' 'A1' 'I4'}
  FieldPrecision: {1x8 cell}
  FieldWidth: [9 6.2000 3 10.4000 20.1500 5 1 4]
  FieldPos: [1 11 18 22 33 54 54 55]
  DataSize: 3127
MissingDataValue: {'*' '---.--' ' *' [] '*' '*' '*' ''}
Intercept: [0 0 -70.2000 0 0 0 0 0]
Slope: [1 1 2.1000 1 1 1 1 1]
Offset: 103680
Keywords: {65x3 cell}
```

## See Also

fitsread



**Purpose** Read data from FITS file

**Syntax**

```
data = fitsread(filename)
data = fitsread(filename, extname)
data = fitsread(filename, extname, index)
data = fitsread(filename, 'raw')
```

**Description** `data = fitsread(filename)` reads the primary data of the Flexible Image Transport System (FITS) file specified by `filename`. Undefined data values are replaced by NaN. Numeric data are scaled by the slope and intercept values and are always returned in double precision. The `filename` argument is a string enclosed in single quotes.

`data = fitsread(filename, extname)` reads data from a FITS file according to the data array or extension specified in `extname`. You can specify only one `extname`. The valid choices for `extname` are shown in the following table.

#### Data Arrays or Extensions

<b>extname</b>	<b>Description</b>
'primary'	Read data from the primary data array.
'table'	Read data from the ASCII Table extension.
'bintable'	Read data from the Binary Table extension.
'image'	Read data from the Image extension.
'unknown'	Read data from the Unknown extension.

`data = fitsread(filename, extname, index)` is the same as the above syntax, except that if there is more than one of the specified extension type `extname` in the file, then only the one at the specified `index` is read.

`data = fitsread(filename, 'raw')` reads the primary or extension data of the FITS file, but, unlike the above syntaxes, does not replace

# fitsread

---

undefined data values with NaN and does not scale the data. The data returned has the same class as the data stored in the file.

## Example

Read FITS file `tst0012.fits` into a 109-by-102 matrix called `data`.

```
data = fitsread('tst0012.fits');

whos data
  Name      Size      Bytes  Class

  data     109x102    88944  double array
```

Here is the beginning of the data read from the file.

```
data(1:5,1:6)
ans =
  135.200  134.9436  134.1752  132.8980  131.1165  128.8378
  137.568  134.9436  134.1752  132.8989  131.1167  126.3343
  135.9946 134.9437  134.1752  132.8989  131.1185  128.1711
  134.0093 134.9440  134.1749  132.8983  131.1201  126.3349
  131.5855 134.9439  134.1749  132.8989  131.1204  126.3356
```

Read only the Binary Table extension from the file.

```
data = fitsread('tst0012.fits', 'bintable')

data =
  Columns 1 through 4
   {11x1 cell} [11x1 int16] [11x3 uint8] [11x2 double]
  Columns 5 through 9
   [11x3 cell] {11x1 cell} [11x1 int8] {11x1 cell} [11x3 int32]
  Columns 10 through 13
   [11x2 int32] [11x2 single] [11x1 double] [11x1 uint8]
```

## See Also

`fitsinfo`

**Purpose** Round toward zero

**Syntax**  $B = \text{fix}(A)$

**Description**  $B = \text{fix}(A)$  rounds the elements of  $A$  toward zero, resulting in an array of integers. For complex  $A$ , the imaginary and real parts are rounded independently.

**Examples**

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]
```

```
a =  
Columns 1 through 4  
-1.9000    -0.2000    3.4000    5.6000  
  
Columns 5 through 6  
7.0000    2.4000 + 3.6000i
```

```
fix(a)
```

```
ans =  
Columns 1 through 4  
-1.0000    0    3.0000    5.0000  
  
Columns 5 through 6  
7.0000    2.0000 + 3.0000i
```

**See Also** `ceil`, `floor`, `round`

# flipdim

---

**Purpose** Flip array along specified dimension

**Syntax** `B = flipdim(A,dim)`

**Description** `B = flipdim(A,dim)` returns `A` with dimension `dim` flipped.

When the value of `dim` is 1, the array is flipped row-wise down. When `dim` is 2, the array is flipped columnwise left to right. `flipdim(A,1)` is the same as `flipud(A)`, and `flipdim(A,2)` is the same as `fliplr(A)`.

**Examples** `flipdim(A,1)` where

`A =`

```
1    4
2    5
3    6
```

produces

```
3    6
2    5
1    4
```

**See Also** `fliplr`, `flipud`, `permute`, `rot90`

**Purpose**

Flip matrix left to right

**Syntax**

$B = \text{fliplr}(A)$

**Description**

$B = \text{fliplr}(A)$  returns  $A$  with columns flipped in the left-right direction, that is, about a vertical axis.

If  $A$  is a row vector, then  $\text{fliplr}(A)$  returns a vector of the same length with the order of its elements reversed. If  $A$  is a column vector, then  $\text{fliplr}(A)$  simply returns  $A$ .

**Examples**

If  $A$  is the 3-by-2 matrix,

```
A =  
    1    4  
    2    5  
    3    6
```

then  $\text{fliplr}(A)$  produces

```
    4    1  
    5    2  
    6    3
```

If  $A$  is a row vector,

```
A =  
    1    3    5    7    9
```

then  $\text{fliplr}(A)$  produces

```
    9    7    5    3    1
```

**Limitations**

The array being operated on cannot have more than two dimensions. This limitation exists because the axis upon which to flip a multidimensional array would be undefined.

**See Also**

`flipdim`, `flipud`, `rot90`

# flipud

---

**Purpose** Flip matrix up to down

**Syntax** `B = flipud(A)`

**Description** `B = flipud(A)` returns `A` with rows flipped in the up-down direction, that is, about a horizontal axis.

If `A` is a column vector, then `flipud(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a row vector, then `flipud(A)` simply returns `A`.

**Examples** If `A` is the 3-by-2 matrix,

```
A =  
    1    4  
    2    5  
    3    6
```

then `flipud(A)` produces

```
    3    6  
    2    5  
    1    4
```

If `A` is a column vector,

```
A =  
    3  
    5  
    7
```

then `flipud(A)` produces

```
A =  
    7  
    5  
    3
```

**Limitations**

The array being operated on cannot have more than two dimensions. This limitation exists because the axis upon which to flip a multidimensional array would be undefined.

**See Also**

`flipdim`, `flipplr`, `rot90`

# floor

---

**Purpose** Round toward negative infinity

**Syntax** `B = floor(A)`

**Description** `B = floor(A)` rounds the elements of `A` to the nearest integers less 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, 7.0, 2.4+3.6i]

a =
Columns 1 through 4
-1.9000          -0.2000          3.4000          5.6000

Columns 5 through 6
7.0000          2.4000 + 3.6000i

floor(a)

ans =
Columns 1 through 4
-2.0000          -1.0000          3.0000          5.0000

Columns 5 through 6
7.0000          2.0000 + 3.0000i
```

**See Also** `ceil`, `fix`, `round`



**Purpose**

Simple function of three variables

**Syntax**

```
v = flow
v = flow(n)
v = flow(x,y,z)
[x,y,z,v] = flow(...)
```

**Description**

`flow`, a function of three variables, generates fluid-flow data that is useful for demonstrating `slice`, `interp3`, and other functions that visualize scalar volume data.

`v = flow` produces a 50-by-25-by-25 array.

`v = flow(n)` produces a n-by-2n-by-n array.

`v = flow(x,y,z)` evaluates the speed profile at the points `x`, `y`, and `z`.

`[x,y,z,v] = flow(...)` returns the coordinates as well as the volume data.

**See Also**

`slice`, `interp3`

“Volume Visualization” on page 1-106 for related functions

See “Example – Slicing Fluid Flow Data” for an example that uses `flow`.

# fminbnd

---

**Purpose** Find minimum of single-variable function on fixed interval

**Syntax**

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
[x,fval] = fminbnd(...)
[x,fval,exitflag] = fminbnd(...)
[x,fval,exitflag,output] = fminbnd(...)
```

**Description** fminbnd finds the minimum of a function of one variable within a fixed interval.

`x = fminbnd(fun,x1,x2)` returns a value `x` that is a local minimizer of the function that is described in `fun` in the interval  $x_1 < x < x_2$ . `fun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information.

“Parametrizing Functions” in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`.

`x = fminbnd(fun,x1,x2,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminbnd` uses these options structure fields:

<code>Display</code>	Level of display. 'off' displays no output; 'iter' displays output at each iteration; 'final' displays just the final output; 'notify' (default) displays output only if the function does not converge.
<code>FunValCheck</code>	Check whether objective function values are valid. 'on' displays an error when the objective function returns a value that is complex or NaN. 'off' displays no error.
<code>MaxFunEvals</code>	Maximum number of function evaluations allowed.
<code>MaxIter</code>	Maximum number of iterations allowed.

<code>OutputFcn</code>	User-defined function that is called at each iteration. See “Output Function” in the Optimization Toolbox for more information.
<code>PlotFcns</code>	User-defined plot function that is called at each iteration. See “Plot Functions” in the Optimization Toolbox for more information.
<code>TolX</code>	Termination tolerance on $x$ .

`[x,fval] = fminbnd(...)` returns the value of the objective function computed in `fun` at  $x$ .

`[x,fval,exitflag] = fminbnd(...)` returns a value `exitflag` that describes the exit condition of `fminbnd`:

1	<code>fminbnd</code> converged to a solution $x$ based on <code>options.TolX</code> .
0	Maximum number of function evaluations or iterations was reached.
-1	Algorithm was terminated by the output function.
-2	Bounds are inconsistent ( $x1 > x2$ ).

`[x,fval,exitflag,output] = fminbnd(...)` returns a structure `output` that contains information about the optimization:

<code>output.algorithm</code>	Algorithm used
<code>output.funcCount</code>	Number of function evaluations
<code>output.iterations</code>	Number of iterations
<code>output.message</code>	Exit message

## Arguments

`fun` is the function to be minimized. `fun` accepts a scalar  $x$  and returns a scalar  $f$ , the objective function evaluated at  $x$ . The function `fun` can be specified as a function handle for an M-file function

# fminbnd

---

```
x = fminbnd(@myfun,x1,x2);
```

where myfun.m is an M-file function such as

```
function f = myfun(x)
f = ...           % Compute function value at x.
```

or as a function handle for an anonymous function:

```
x = fminbnd(@(x) sin(x*x),x1,x2);
```

Other arguments are described in the syntax descriptions above.

## Examples

`x = fminbnd(@cos,3,4)` computes  $\pi$  to a few decimal places and gives a message on termination.

```
[x,fval,exitflag] = ...
    fminbnd(@cos,3,4,optimset('TolX',1e-12,'Display','off'))
```

computes  $\pi$  to about 12 decimal places, suppresses output, returns the function value at  $x$ , and returns an `exitflag` of 1.

The argument `fun` can also be a function handle for an anonymous function. For example, to find the minimum of the function

$f(x) = x^3 - 2x - 5$  on the interval  $(0,2)$ , create an anonymous function `f`

```
f = @(x)x.^3-2*x-5;
```

Then invoke `fminbnd` with

```
x = fminbnd(f, 0, 2)
```

The result is

```
x =
    0.8165
```

The value of the function at the minimum is

```
y = f(x)

y =
    -6.0887
```

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = (x - a)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminbnd`. To optimize for a specific value of `a`, such as `a = 1.5`.

**1** Assign the value to `a`.

```
a = 1.5; % define parameter first
```

**2** Call `fminbnd` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

```
x = fminbnd(@(x) myfun(x,a),0,1)
```

## Algorithm

`fminbnd` is an M-file. The algorithm is based on golden section search and parabolic interpolation. Unless the left endpoint  $x_1$  is very close to the right endpoint  $x_2$ , `fminbnd` never evaluates `fun` at the endpoints, so `fun` need only be defined for  $x$  in the interval  $x_1 < x < x_2$ . If the minimum actually occurs at  $x_1$  or  $x_2$ , `fminbnd` returns an interior point at a distance of no more than  $2 \cdot \text{TolX}$  from  $x_1$  or  $x_2$ , where `TolX` is the termination tolerance. See [1] or [2] for details about the algorithm.

## Limitations

The function to be minimized must be continuous. `fminbnd` may only give local solutions.

`fminbnd` often exhibits slow convergence when the solution is on a boundary of the interval.

# fminbnd

---

fminbnd only handles real variables.

## See Also

fminsearch, fzero, optimset, function\_handle (@), anonymous function

## References

[1] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

[2] Brent, Richard. P., *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973

## Purpose

Find minimum of unconstrained multivariable function using derivative-free method

## Syntax

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,options)
[x,fval] = fminsearch(...)
[x,fval,exitflag] = fminsearch(...)
[x,fval,exitflag,output] = fminsearch(...)
```

## Description

fminsearch finds the minimum of a scalar function of several variables, starting at an initial estimate. This is generally referred to as *unconstrained nonlinear optimization*.

`x = fminsearch(fun,x0)` starts at the point `x0` and finds a local minimum `x` of the function described in `fun`. `x0` can be a scalar, vector, or matrix. `fun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information.

“Parametrizing Functions” in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`. See also “Example 2” on page 2-1410 and “Example 3” on page 2-1410 below.

`x = fminsearch(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminsearch` uses these `options` structure fields:

Display	Level of display. 'off' displays no output; 'iter' displays output at each iteration; 'final' displays just the final output; 'notify' (default) displays output only if the function does not converge.
FunValCheck	Check whether objective function values are valid. 'on' displays an error when the objective function returns a value that is complex, Inf or NaN. 'off' (the default) displays no error.
MaxFunEvals	Maximum number of function evaluations allowed

# fminsearch

---

MaxIter	Maximum number of iterations allowed
OutputFcn	User-defined function that is called at each iteration. See “Output Function” in the Optimization Toolbox for more information.
PlotFcns	User-defined plot function that is called at each iteration. See “Plot Functions” in the Optimization Toolbox for more information.
TolFun	Termination tolerance on the function value
TolX	Termination tolerance on x

`[x,fval] = fminsearch(...)` returns in `fval` the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fminsearch(...)` returns a value `exitflag` that describes the exit condition of `fminsearch`:

1	<code>fminsearch</code> converged to a solution <code>x</code> .
0	Maximum number of function evaluations or iterations was reached.
-1	Algorithm was terminated by the output function.

`[x,fval,exitflag,output] = fminsearch(...)` returns a structure `output` that contains information about the optimization:

<code>output.algorithm</code>	Algorithm used
<code>output.funcCount</code>	Number of function evaluations
<code>output.iterations</code>	Number of iterations
<code>output.message</code>	Exit message

## Arguments

`fun` is the function to be minimized. It accepts an input `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function



```
x = fminsearch(@myfun, x0)
```

where myfun is an M-file function such as

```
function f = myfun(x)
f = ...           % Compute function value at x
```

or as a function handle for an anonymous function, such as

```
x = fminsearch(@(x)sin(x^2), x0);
```

Other arguments are described in the syntax descriptions above.

## Examples

### Example 1

A classic test example for multidimensional minimization is the Rosenbrock banana function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

The minimum is at (1, 1) and has the value 0. The traditional starting point is (-1.2, 1). The anonymous function shown here defines the function and returns a function handle called banana:

```
banana = @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

Pass the function handle to fminsearch:

```
[x,fval] = fminsearch(banana,[-1.2, 1])
```

This produces

```
x =
    1.0000    1.0000
fval =
    8.1777e-010
```

This indicates that the minimizer was found to at least four decimal places with a value near zero.

## Example 2

If fun is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function myfun defined by the following M-file function.

```
function f = myfun(x,a)
f = x(1)^2 + a*x(2)^2;
```

Note that myfun has an extra parameter a, so you cannot pass it directly to fminsearch. To optimize for a specific value of a, such as a = 1.5.

**1** Assign the value to a.

```
a = 1.5; % define parameter first
```

**2** Call fminsearch with a one-argument anonymous function that captures that value of a and calls myfun with two arguments:

```
x = fminsearch(@(x) myfun(x,a),[0,1])
```

## Example 3

You can modify the first example by adding a parameter a to the second term of the banana function:

$$f(x) = 100(x_2 - x_1^2)^2 + (a - x_1)^2$$

This changes the location of the minimum to the point [a, a<sup>2</sup>]. To minimize this function for a specific value of a, for example a = sqrt(2), create a one-argument anonymous function that captures the value of a.

```
a = sqrt(2);
banana = @(x) 100*(x(2)-x(1)^2)^2+(a-x(1))^2;
```

Then the statement

```
[x,fval] = fminsearch(banana, [-1.2, 1], ...  
    optimset('TolX',1e-8));
```

seeks the minimum  $[\sqrt{2}, 2]$  to an accuracy higher than the default on  $x$ .

## Algorithm

fminsearch uses the simplex search method of [1]. This is a direct search method that does not use numerical or analytic gradients.

If  $n$  is the length of  $x$ , a simplex in  $n$ -dimensional space is characterized by the  $n+1$  distinct vectors that are its vertices. In two-space, a simplex is a triangle; in three-space, it is a pyramid. At each step of the search, a new point in or near the current simplex is generated. The function value at the new point is compared with the function's values at the vertices of the simplex and, usually, one of the vertices is replaced by the new point, giving a new simplex. This step is repeated until the diameter of the simplex is less than the specified tolerance.

## Limitations

fminsearch can often handle discontinuity, particularly if it does not occur near the solution. fminsearch may only give local solutions.

fminsearch only minimizes over the real numbers, that is,  $x$  must only consist of real numbers and  $f(x)$  must only return real numbers. When  $x$  has complex variables, they must be split into real and imaginary parts.

## See Also

fminbnd, optimset, function\_handle (@), anonymous function

## References

[1] Lagarias, J.C., J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions," *SIAM Journal of Optimization*, Vol. 9 Number 1, pp. 112-147, 1998.

# fopen

---

**Purpose** Open file, or obtain information about open files

**Syntax**

```
fid = fopen(filename)
fid = fopen(filename, permission)
fid = fopen(filename, permission_tmode)
[fid, message] = fopen(filename, permission)
[fid, message] = fopen(filename, permission, machineformat)
[fid, message] = fopen(filename, permission, machineformat,
    encoding)
fids = fopen('all')
[filename, permission, machineformat, encoding] = fopen(fid)
```

**Description**

`fid = fopen(filename)` opens the file `filename` for read access. (On Windows systems, `fopen` opens files for binary read access.) The `filename` argument is a string enclosed in single quotes. It can be a MATLABPATH relative partial pathname if the file is opened for reading only. A relative path is always searched for first with respect to the current directory. If it is not found, and reading only is specified or implied, then `fopen` does an additional search of the MATLABPATH.

`fid` is a scalar MATLAB integer, called a file identifier. You use the `fid` as the first argument to other file input/output routines. If `fopen` cannot open the file, it returns `-1`. Two file identifiers are automatically available and need not be opened. They are `fid=1` (standard output) and `fid=2` (standard error).

`fid = fopen(filename, permission)` opens the file `filename` in the specified permission. The `permission` argument can be any of the following:

### Permission Specifiers

Permission	Description
'r'	Open file for reading (default).
'w'	Open file, or create new file, for writing; discard existing contents, if any.

**Permission Specifiers (Continued)**

Permission	Description
'a'	Open file, or create new file, for writing; append data to the end of the file.
'r+'	Open file for reading and writing.
'w+'	Open file, or create new file, for reading and writing; discard existing contents, if any.
'a+'	Open file, or create new file, for reading and writing; append data to the end of the file.
'A'	Append without automatic flushing; used with tape drives.
'W'	Write without automatic flushing; used with tape drives.

---

**Note** If the file is opened in update mode ('+'), an input command like `fread`, `fscanf`, `fgets`, or `fgetl` cannot be immediately followed by an output command like `fwrite` or `fprintf` without an intervening `fseek` or `frewind`. The reverse is also true: that is, an output command like `fwrite` or `fprintf` cannot be immediately followed by an input command like `fread`, `fscanf`, `fgets`, or `fgetl` without an intervening `fseek` or `frewind`.

---

`fid = fopen(filename, permission_tmode)` on Windows systems, opens the file in text mode instead of binary mode (the default). The `permission_tmode` argument consists of any of the specifiers shown in the Permission Specifiers on page 2-1318 table above, followed by the letter `t`, for example `'rt'` or `'wt+'`. On UNIX systems, text and binary

mode are the same. (UNIX is a registered trademark of The Open Group in the United States and other countries).

## Binary and Text Modes

Mode	Behavior
Binary	No characters are given special treatment.
Text	On a read operation, whenever MATLAB encounters a carriage return followed by a newline character, it removes the carriage return from the input. On a write or append operation, MATLAB inserts a carriage return before any newline character.

`[fid, message] = fopen(filename, permission)` opens a file as above. If it cannot open the file, `fid` equals `-1` and `message` contains a system-dependent error message. If `fopen` successfully opens a file, the value of `message` is empty.

`[fid, message] = fopen(filename, permission, machineformat)` opens the file with the specified `permission` and treats data read using `fread` or data written using `fwrite` as having a format given by `machineformat`. `machineformat` is one of the following strings:

## Precision Support

'ieee be' or 'b'	IEEE floating point with big-endian byte ordering
'ieee le' or 'l'	IEEE floating point with little-endian byte ordering
'ieee-be.16' or 's'	IEEE floating point with big-endian byte ordering and 64-bit long data type

**Precision Support (Continued)**

'ieee-le.164' or 'a'	IEEE floating point with little-endian byte ordering and 64-bit long data type
'native' or 'n'	Numeric format of the machine on which MATLAB is running (the default)

[fid, message] = fopen(filename, permission, machineformat, encoding) opens the specified file using the specified permission and machineformat. encoding is a string that specifies the character encoding scheme associated with the file. It must be the empty string ('') or a name or alias for an encoding scheme. Some examples are 'UTF-8', 'latin1', 'US-ASCII', and 'Shift\_JIS'. For common names and aliases, see the Web site <http://www.iana.org/assignments/character-sets>. If encoding is unspecified or is the empty string (''), MATLAB's default encoding scheme is used.

fids = fopen('all') returns a row vector containing the file identifiers of all open files, not including 1 and 2 (standard output and standard error). The number of elements in the vector is equal to the number of open files.

[filename, permission, machineformat, encoding] = fopen(fid) returns the filename, permission, machineformat, and encoding values used by MATLAB when it opened the file associated with identifier fid. MATLAB does not determine these output values by reading information from the opened file. For any of these parameters that were not specified when the file was opened, MATLAB returns its default value. The encoding string is a standard character encoding scheme name that may not be the same as the encoding argument used in the call to fopen that opened the file. An invalid fid returns empty strings for all output arguments.

The 'W' and 'A' modes do not automatically perform a flush of the current output buffer after output operations.

# fopen

---

## Examples

The example uses `fopen` to open a file and then passes the `fid` returned by `fopen` to other file I/O functions to read data from the file and then close the file.

```
fid = fopen('fgetl.m');
while 1
    tline = fgetl(fid);
    if ~ischar(tline), break, end
    disp(tline)
end
fclose(fid);
```

## See Also

`fclose`, `ferror`, `fprintf`, `fread`, `fscanf`, `fseek`, `ftell`, `fwrite`



<b>Purpose</b>	Connect serial port object to device
<b>Syntax</b>	<code>fopen(obj)</code>
<b>Description</b>	<code>fopen(obj)</code> connects the serial port object, <code>obj</code> to the device.
<b>Remarks</b>	<p>Before you can perform a read or write operation, <code>obj</code> must be connected to the device with the <code>fopen</code> function. When <code>obj</code> is connected to the device:</p> <ul style="list-style-type: none"><li>• Data remaining in the input buffer or the output buffer is flushed.</li><li>• The <code>Status</code> property is set to <code>open</code>.</li><li>• The <code>BytesAvailable</code>, <code>ValuesReceived</code>, <code>ValuesSent</code>, and <code>BytesToOutput</code> properties are set to 0.</li></ul> <p>An error is returned if you attempt to perform a read or write operation while <code>obj</code> is not connected to the device. You can connect only one serial port object to a given device.</p> <p>Some properties are read-only while the serial port object is open (connected), and must be configured before using <code>fopen</code>. Examples include <code>InputBufferSize</code> and <code>OutputBufferSize</code>. Refer to the property reference pages to determine which properties have this constraint.</p> <p>The values for some properties are verified only after <code>obj</code> is connected to the device. If any of these properties are incorrectly configured, then an error is returned when <code>fopen</code> is issued and <code>obj</code> is not connected to the device. Properties of this type include <code>BaudRate</code>, and are associated with device settings.</p> <p>If you use the <code>help</code> command to display help for <code>fopen</code>, then you need to supply the pathname shown below.</p> <pre>help serial/fopen</pre>

# fopen (serial)

---

## Example

This example creates the serial port object `s`, connects `s` to the device using `fopen`, writes and reads text data, and then disconnects `s` from the device.

```
s = serial('COM1');  
fopen(s)  
fprintf(s, '*IDN?')  
idn = fscanf(s);  
fclose(s)
```

## See Also

### Functions

`fclose`

### Properties

`BytesAvailable`, `BytesToOutput`, `Status`, `ValuesReceived`, `ValuesSent`

**Purpose**

Execute block of code specified number of times

**Syntax**

```
for x=initval:endval, statements, end
for x=initval:stepval:endval, statements, end
```

**Description**

`for x=initval:endval, statements, end` repeatedly executes one or more MATLAB *statements* in a loop. Loop counter variable *x* is initialized to value *initval* at the start of the first pass through the loop, and automatically increments by 1 each time through the loop. The program makes repeated passes through *statements* until either *x* has incremented to the value *endval*, or MATLAB encounters a `break`, or `return` instruction, thus forcing an immediately exit of the loop. If MATLAB encounters a `continue` statement in the loop code, it immediately exits the current pass at the location of the `continue` statement, skipping any remaining code in that pass, and begins another pass at the start of the loop *statements* with the value of the loop counter incremented by 1.

The values *initval* and *endval* must be real numbers or arrays of real numbers, or can also be calls to functions that return the same. The value assigned to *x* is often used in the code within the loop, however it is recommended that you do not assign to *x* in the loop code.

`for x=initval:stepval:endval, statements, end` is the same as the above syntax, except that loop counter *x* is incremented (or decremented when *stepval* is negative) by the value *stepval* on each iteration through the loop. The value *stepval* must be a real number or can also be a call to a function that returns a real number.

The general format is

```
for variable = initval:endval
    statement
    ...
    statement
end
```

# for

---

The scope of the `for` statement is always terminated with a matching `end`.

See “Program Control Statements” in the MATLAB Programming Fundamentals documentation for more information on controlling the flow of your program code.

## Remarks

It is recommended that you do not assign to the loop control variable while in the body of a loop. If you do assign to a variable that has the same name as the loop control variable (see `k` in the example below), then the value of that variable alternates between the value assigned by the `for` statement at the start of each loop iteration and the value explicitly assigned to it in the loop code:

```
for k=1:2
    disp(sprintf(' At the start of the loop, k = %d', k))
    k = 10;
    disp(sprintf(' Following the assignment, k = %d\n', k))
end
```

```
At the start of the loop, k = 1
Following the assignment, k = 10
```

```
At the start of the loop, k = 2
Following the assignment, k = 10
```

## Examples

Assume `k` has already been assigned a value. Create the Hilbert matrix, using `zeros` to preallocate the matrix to conserve memory:

```
a = zeros(k,k) % Preallocate matrix
for m = 1:k
    for n = 1:k
        a(m,n) = 1/(m+n -1);
    end
end
```

Step `s` with increments of `-0.1`:

---

```
for s = 1.0: -0.1: 0.0, ..., end
```

Step `s` with values 1, 5, 8, and 17:

```
for s = [1,5,8,17], ..., end
```

Successively set `e` to the unit `n`-vectors:

```
for e = eye(n), ..., end
```

The line

```
for V = A, ..., end
```

has the same effect as

```
for k = 1:n, V = A(:,k); ..., end
```

except `k` is also set here.

## See Also

`end`, `while`, `break`, `continue`, `parfor`, `return`, `if`, `switch`, `colon`

# format

---

**Purpose** Set display format for output

**Graphical Interface** As an alternative to using the `format` command, you can also use the MATLAB Preferences GUI. Select **File > Preferences > Command Window** and press the **Help** button for more information.

**Syntax**

```
format
format type
format('type')
```

**Description** Use the `format` function to control the output format of numeric values displayed in the Command Window.

---

**Note** The `format` function affects only how numbers are displayed, not how MATLAB computes or saves them.

---

`format` by itself, changes the output format to the default appropriate for the class of the variable currently being used. For floating-point variables, for example, the default is `format short` (i.e., 5-digit scaled, fixed-point values).

`format type` changes the format to the specified *type*. The tables shown below list the allowable values for *type*.

`format('type')` is the function form of the syntax.

The tables below show the allowable values for *type*, and provides an example for each type using `pi`.

Use these format types to switch between different output display formats for floating-point variables.

Type	Result
short	Scaled fixed point format, with 4 digits after the decimal point. For example, 3.1416.

Type	Result
long	Scaled fixed point format with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, 3.141592653589793.
short e	Floating point format, with 4 digits after the decimal point. For example, 3.1416e+000.
long e	Floating point format, with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, 3.141592653589793e+000.
short g	Best of fixed or floating point, with 4 digits after the decimal point. For example, 3.1416.
long g	Best of fixed or floating point, with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, 3.14159265358979.
short eng	Engineering format that has 4 digits after the decimal point, and a power that is a multiple of three. For example, 3.1416e+000.
long eng	Engineering format that has exactly 16 significant digits and a power that is a multiple of three. For example, 3.14159265358979e+000.

Use these format types to switch between different output display formats for all numeric variables.

Value for type	Result
+	+, -, blank
bank	Fixed dollars and cents. For example, 3.14
hex	Hexadecimal (hexadecimal representation of a binary double-precision number). For example, 400921fb54442d18
rat	Ratio of small integers. For example, 355/113

# format

---

Use these format types to affect the spacing in the display of all variables.

Value for type	Result	Example
compact	Suppresses excess line feeds to show more output in a single screen. Contrast with loose.	theta = pi/2 theta = 1.5708
loose	Adds linefeeds to make output more readable. Contrast with compact.	theta = pi/2  theta =  1.5708

## Remarks

Computations on floating-point variables, namely `single` or `double`, are done in appropriate floating-point precision, no matter how those variables are displayed. Computations on integer variables are done natively in integer.

MATLAB always displays integer variables to the appropriate number of digits for the class. For example, MATLAB uses three digits to display numbers of type `int8` (i.e., -128:127). Setting `format` to `short` or `long` does not affect the display of integer variables.

The specified format applies only to the current MATLAB session. To maintain a format across sessions, use MATLAB preferences. “Preferences”

To see which type is currently in use, type

```
get(0, 'Format')
```

To see if `compact` or `loose` formatting is currently selected, type

```
get(0, 'FormatSpacing').
```



**Examples****Example 1**

Change the format to long by typing

```
format long
```

View the result for the value of pi by typing

```
pi
ans =
    3.14159265358979
```

View the current format by typing

```
get(0, 'format')
ans =
    long
```

Set the format to short e by typing

```
format short e
```

or use the function form of the syntax

```
format('short','e')
```

**Example 2**

When the format is set to short, both pi and single(pi) display as 5-digit values:

```
format short
```

```
pi
ans =
    3.1416
```

```
single(pi)
ans =
    3.1416
```

# format

---

Now set format to long, and pi displays a 15-digit value while single(pi) display an 8-digit value:

```
format long

pi
ans =
    3.14159265358979

single(pi)
ans =
    3.1415927
```

### Example 3

Set the format to its default, and display the maximum values for integers and real numbers in MATLAB:

```
format

intmax('uint64')
ans =
    18446744073709551615

realmax
ans =
    1.7977e+308
```

Now change the format to hexadecimal, and display these same values:

```
format hex

intmax('uint64')
ans =
    ffffffffffffffff

realmax
ans =
    7fefffffffffffffff
```

The hexadecimal display corresponds to the internal representation of the value. It is not the same as the hexadecimal notation in the C programming language.

#### Example 4

This example illustrates the `short eng` and `long eng` formats. The value assigned to variable `A` increases by a multiple of 10 each time through the `for` loop.

```
A = 5.123456789;

for k=1:10
    disp(A)
    A = A * 10;
end
```

The values displayed for `A` are shown here. The power of 10 is always a multiple of 3. The value itself is expressed in 5 or more digits for the `short eng` format, and in exactly 15 digits for `long eng`:

format short eng	format long eng
5.1235e+000	5.12345678900000e+000
51.2346e+000	51.2345678900000e+000
512.3457e+000	512.345678900000e+000
5.1235e+003	5.12345678900000e+003
51.2346e+003	51.2345678900000e+003
512.3457e+003	512.345678900000e+003
5.1235e+006	5.12345678900000e+006
51.2346e+006	51.2345678900000e+006
512.3457e+006	512.345678900000e+006
5.1235e+009	5.12345678900000e+009

## Algorithms

If the largest element of a matrix is larger than  $10^3$  or smaller than  $10^{-3}$ , MATLAB applies a common scale factor for the short and long formats. The function `format +` displays +, -, and blank characters for positive, negative, and zero elements. `format hex` displays the hexadecimal

# format

---

representation of a binary double-precision number. `format rat` uses a continued fraction algorithm to approximate floating-point values by ratios of small integers. See `rat.m` for the complete code.

## See Also

`disp`, `display`, `isnumeric`, `isfloat`, `isinteger`, `floor`, `sprintf`, `fprintf`, `num2str`, `rat`, `spy`

**Purpose**

Plot function between specified limits

**Syntax**

```
fplot(fun,limits)
fplot(fun,limits,LineStyle)
fplot(fun,limits,tol)
fplot(fun,limits,tol,LineStyle)
fplot(fun,limits,n)
fplot(fun,lims,...)
fplot(axes_handle,...)
[X,Y] = fplot(fun,limits,...)
```

**Description**

fplot plots a function between specified limits. The function must be of the form  $y = f(x)$ , where  $x$  is a vector whose range specifies the limits, and  $y$  is a vector the same size as  $x$  and contains the function's value at the points in  $x$  (see the first example). If the function returns more than one value for a given  $x$ , then  $y$  is a matrix whose columns contain each component of  $f(x)$  (see the second example).

fplot(fun,limits) plots fun between the limits specified by limits. limits is a vector specifying the  $x$ -axis limits ([xmin xmax]), or the  $x$ - and  $y$ -axes limits, ([xmin xmax ymin ymax]).

fun must be

- The name of an M-file function
- A string with variable  $x$  that may be passed to eval, such as 'sin(x)', 'diric(x,10)', or '[sin(x),cos(x)]'
- A function handle for an M-file function or an anonymous function (see “Function Handles” and “Anonymous Functions” for more information)

The function  $f(x)$  must return a row vector for each element of vector  $x$ . For example, if  $f(x)$  returns  $[f_1(x), f_2(x), f_3(x)]$  then for input  $[x_1; x_2]$  the function should return the matrix

```
f1(x1) f2(x1) f3(x1)
f1(x2) f2(x2) f3(x2)
```

`fplot(fun,limits,LineStyle)` plots `fun` using the line specification `LineStyle`.

`fplot(fun,limits,tol)` plots `fun` using the relative error tolerance `tol` (the default is  $2e-3$ , i.e., 0.2 percent accuracy).

`fplot(fun,limits,tol,LineStyle)` plots `fun` using the relative error tolerance `tol` and a line specification that determines line type, marker symbol, and color. See `LineStyle` for more information.

`fplot(fun,limits,n)` with  $n \geq 1$  plots the function with a minimum of  $n+1$  points. The default `n` is 1. The maximum step size is restricted to be  $(1/n) * (x_{\max} - x_{\min})$ .

`fplot(fun,lims,...)` accepts combinations of the optional arguments `tol`, `n`, and `LineStyle`, in any order.

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

`[X,Y] = fplot(fun,limits,...)` returns the abscissas and ordinates for `fun` in `X` and `Y`. No plot is drawn on the screen; however, you can plot the function using `plot(X,Y)`.

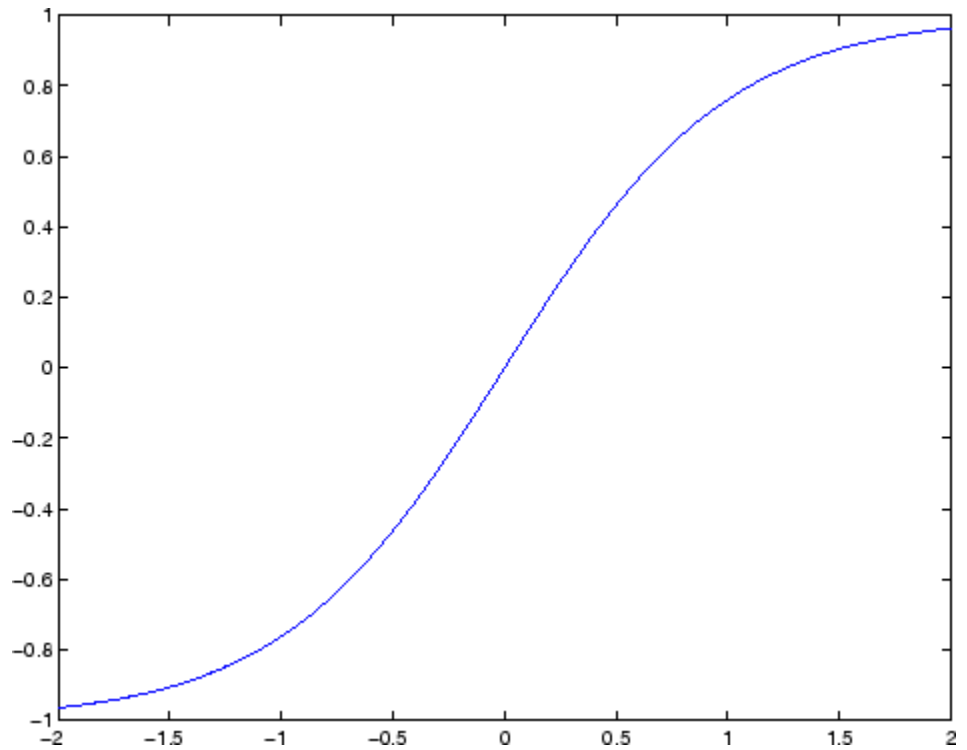
## Remarks

`fplot` uses adaptive step control to produce a representative graph, concentrating its evaluation in regions where the function's rate of change is the greatest.

## Examples

Plot the hyperbolic tangent function from -2 to 2:

```
fnch = @tanh;  
fplot(fnch,[-2 2])
```



Create an M-file, myfun, that returns a two-column matrix:

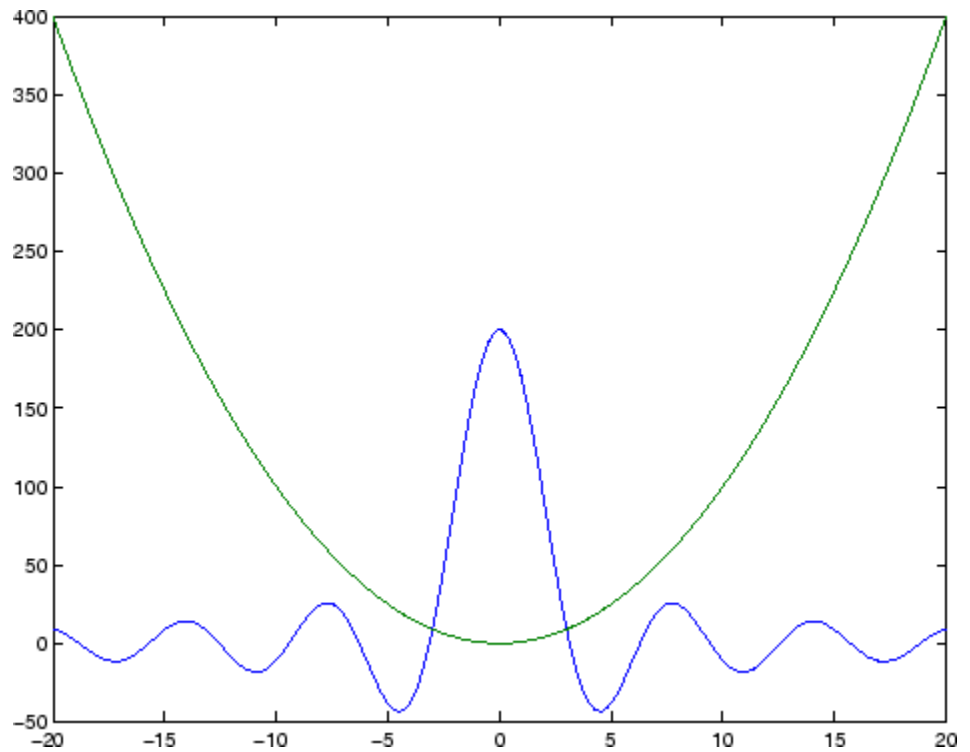
```
function Y = myfun(x)
Y(:,1) = 200*sin(x(:))./x(:);
Y(:,2) = x(:).^2;
```

Create a function handle pointing to myfun:

```
fh = @myfun;
```

Plot the function with the statement

```
fplot(fh,[-20 20])
```

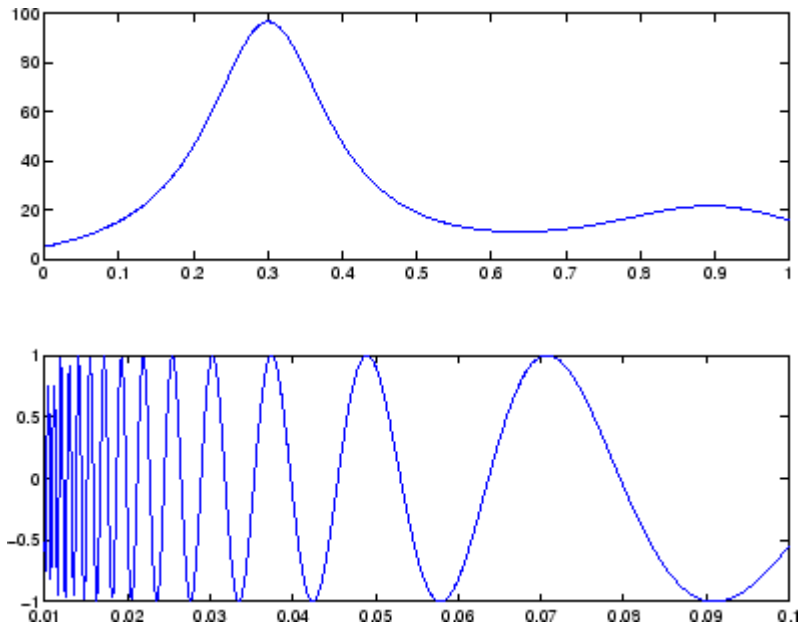


## Additional Example

This example passes function handles to `fplot`, one created from a MATLAB function and the other created from an anonymous function.

```
hmp = @humps;  
subplot(2,1,1);fplot(hmp,[0 1])  
sn = @(x) sin(1./x);  
subplot(2,1,2);fplot(sn,[.01 .1])
```



**See Also**

`eval`, `ezplot`, `feval`, `LineStyle`, `plot`

“Function Plots” on page 1-93 for related functions

# fprintf

---

**Purpose** Write formatted data to file

**Syntax** `count = fprintf(fid, format, A, ...)`

**Description** `count = fprintf(fid, format, A, ...)` formats the data in the real part of matrix A (and in any additional matrix arguments) under control of the specified format string, and writes it to the file associated with file identifier `fid`. `fprintf` returns a count of the number of bytes written.

Argument `fid` is an integer file identifier obtained from `fopen`. (It can also be 1 for standard output (the screen) or 2 for standard error. See `fopen` for more information.) Omitting `fid` causes output to appear on the screen.

For more detailed information on using string formatting commands, see “Formatting Strings” in the MATLAB Programming Fundamentals documentation.

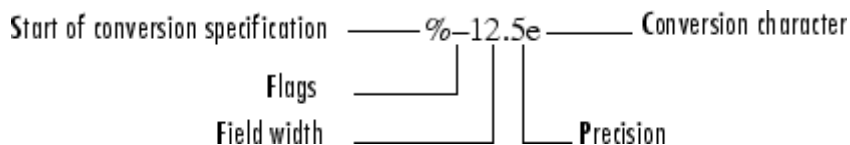
## Format String

The format argument is a string containing ordinary characters and/or C language conversion specifications. A conversion specification controls the notation, alignment, significant digits, field width, and other aspects of output format. The format string can contain escape characters to represent nonprinting characters such as newline characters and tabs.

Conversion specifications begin with the % character and contain these optional and required elements:

- Flags (optional)
- Width and precision fields (optional)
- A subtype specifier (optional)
- Conversion character (required)

You specify these elements in the following order:



## Flags

You can control the alignment of the output using any of these optional flags.

Character	Description	Example
Minus sign ( - )	Left-justifies the converted argument in its field	%-5.2d
Plus sign ( + )	Always prints a sign character ( + or - )	%+5.2d
Space character	Inserts a space before the value	% 5.2d
Zero ( 0 )	Pads with zeros rather than spaces	%05.2d

## Field Width and Precision Specifications

You can control the width and precision of the output by including these options in the format string.

Character	Description	Example
Field width	A digit string specifying the minimum number of digits to be printed	%6f
Precision	A digit string including a period ( . ) specifying the number of digits to be printed to the right of the decimal point	%6.2f

## Conversion Characters

Conversion characters specify the notation of the output.

Specifier	Description
%c	Single character
%d	Decimal notation (signed)
%e	Exponential notation (using a lowercase e as in 3.1415e+00)
%E	Exponential notation (using an uppercase E as in 3.1415E+00)
%f	Fixed-point notation
%g	The more compact of %e or %f, as defined in [2]. Insignificant zeros do not print.
%G	Same as %g, but using an uppercase E
%i	Decimal notation (signed)
%o	Octal notation (unsigned)
%s	String of characters
%u	Decimal notation (unsigned)
%x	Hexadecimal notation (using lowercase letters a–f)
%X	Hexadecimal notation (using uppercase letters A–F)

Conversion characters %o, %u, %x, and %X support subtype specifiers. See Remarks for more information.

## Escape Characters

This table lists the escape character sequences you use to specify nonprinting characters in a format specification.

Character	Description
\b	Backspace
\f	Form feed
\n	New line

Character	Description
\r	Carriage return
\t	Horizontal tab
\\	Backslash
\' or '' (two single quotes)	Single quotation mark
%%	Percent character

## Remarks

When writing text to a file on a Windows system, The MathWorks recommends that you open the file in write-text mode (e.g., `fopen(file_id, 'wt')`). This ensures that lines in the file are terminated in such a way as to be compatible with all applications that might use the file.

MATLAB writes characters using the encoding scheme associated with the file. See `fopen` for more information.

The `fprintf` function behaves like its ANSI<sup>®</sup> C language namesake with these exceptions and extensions:

- If you use `fprintf` to convert a MATLAB double into an integer, and the double contains a value that cannot be represented as an integer (for example, it contains a fraction), MATLAB ignores the specified conversion and outputs the value in exponential format. To successfully perform this conversion, use the `fix`, `floor`, `ceil`, or `round` function to change the value in the double into a value that can be represented as an integer before passing it to `sprintf`.
- The following nonstandard subtype specifiers are supported for the conversion characters `%o`, `%u`, `%x`, and `%X`.

# fprintf

---

- b The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like '%bx'.
- t The underlying C data type is a float rather than an unsigned integer.

For example, to print a double value in hexadecimal, use the format '%bx'.

- The `fprintf` function is vectorized for nonscalar arguments. The function recycles the format string through the elements of `A` (columnwise) until all the elements are used up. The function then continues in a similar manner through any additional matrix arguments.

---

**Note** `fprintf` displays negative zero (-0) differently on some platforms, as shown in the following table.

---

	Conversion Character		
Platform	%e or %E	%f	%g or %G
PC	0.000000e+000	0.000000	0
Others	-0.000000e+00	-0.000000	-0

## Examples

### Example 1

Create a text file called `exp.txt` containing a short table of the exponential function. (On Windows platforms, it is recommended that you use `fopen` with the mode set to `'wt'` to open a text file for writing.)

```
x = 0:.1:1;  
y = [x; exp(x)];  
fid = fopen('exp.txt', 'wt');  
fprintf(fid, '%6.2f %12.8f\n', y);
```

```
fclose(fid)
```

Now examine the contents of `exp.txt`:

```
type exp.txt
0.00    1.00000000
0.10    1.10517092
...
1.00    2.71828183
```

### Example 2

The command:

```
fprintf( ...
    'A unit circle has circumference %g radians.\n', 2*pi)
```

displays a line on the screen:

```
A unit circle has circumference 6.283186 radians.
```

### Example 3

To insert a single quotation mark in a string, use two single quotation marks together. For example:

```
fprintf(1, 'It' 's Friday.\n')
```

displays on the screen:

```
It's Friday.
```

### Example 4

Use `fprintf` to display a hyperlink on the screen. For example:

```
site = 'http://www.mathworks.com';
title = 'The MathWorks Web Site';
fprintf(['<a href = ' site '>' title '</a>'])
```

creates the hyperlink:

The Mathworks Web Site

in the Command Window. Click this link to display The MathWorks home page in a MATLAB Web browser.

## Example 5

The commands

```
B = [8.8 7.7; 8800 7700]
fprintf(1, 'X is %6.2f meters or %8.3f mm\n', 9.9, 9900, B)
```

display the lines

```
X is 9.90 meters or 9900.000 mm
X is 8.80 meters or 8800.000 mm
X is 7.70 meters or 7700.000 mm
```

## Example 6

Explicitly convert MATLAB double-precision variables to integer values for use with an integer conversion specifier. For instance, to convert signed 32-bit data to hexadecimal format,

```
a = [6 10 14 44];
fprintf('%9X\n', a + (a<0)*2^32)
6
A
E
2C
```

## See Also

`disp`, `fclose`, `ferror`, `fopen`, `fread`, `fscanf`, `fseek`, `ftell`, `fwrite`

## References

- [1] Kernighan, B.W., and D.M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988.
- [2] ANSI specification X3.159-1989: "Programming Language C," ANSI, 1430 Broadway, New York, NY 10018.



**Purpose** Write text to device

**Syntax**

```
fprintf(obj, 'cmd')  
fprintf(obj, 'format', 'cmd')  
fprintf(obj, 'cmd', 'mode')  
fprintf(obj, 'format', 'cmd', 'mode')
```

**Description** fprintf(obj, 'cmd') writes the string cmd to the device connected to the serial port object , obj. The default format is %s\n. The write operation is synchronous and blocks the command line until execution is complete.

fprintf(obj, 'format', 'cmd') writes the string using the format specified by format. format is a C language conversion specification. Conversion specifications involve the % character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the sprintf file I/O format specifications or a C manual for more information.

fprintf(obj, 'cmd', 'mode') writes the string with command line access specified by mode. If mode is sync, cmd is written synchronously and the command line is blocked. If mode is async, cmd is written asynchronously and the command line is not blocked. If mode is not specified, the write operation is synchronous.

fprintf(obj, 'format', 'cmd', 'mode') writes the string using the specified format. If mode is sync, cmd is written synchronously. If mode is async, cmd is written asynchronously.

**Remarks** Before you can write text to the device, it must be connected to obj with the fopen function. A connected serial port object has a Status property value of open. An error is returned if you attempt to perform a write operation while obj is not connected to the device.

The ValuesSent property value is increased by the number of values written each time fprintf is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the OutputBufferSize property.

# fprintf (serial)

---

If you use the `help` command to display help for `fprintf`, then you need to supply the pathname shown below.

```
help serial/fprintf
```

`fprintf` function will return an error message if you set the `flowcontrol` property to `hardware` on a serial object, and a hardware connection is not detected. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check you remote device's status and flow control settings to see if hardware flow control is causing errors in MATLAB.

---

**Note** If you want to check to see if the device is asserting that it is ready to receive data, set the `FlowControl` to `none`. Once you connect to the device check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is off, there is a problem on the remote device side. If `ClearToSend` is on, there is a hardware `FlowControl` device prepared to receive data and you can execute `fprintf`.

---

## Synchronous Versus Asynchronous Write Operations

By default, text is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the `mode` input argument to be `async`. For asynchronous writes:

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The M-file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in *Controlling Access to the MATLAB Command Line*.

## Rules for Completing a Write Operation with fprintf

A synchronous or asynchronous write operation using `fprintf` completes when:

- The specified data is written.
- The time specified by the `Timeout` property passes.

Additionally, you can stop an asynchronous write operation with the `stopasync` function.

## Rules for Writing the Terminator

All occurrences of `\n` in `cmd` are replaced with the `Terminator` property value. Therefore, when using the default format `%s\n`, all commands written to the device will end with this property value. The terminator required by your device will be described in its documentation.

## Example

Create the serial port object `s`, connect `s` to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');  
fopen(s)  
fprintf(s, 'RS232?')
```

Because the default format for `fprintf` is `%s\n`, the terminator specified by the `Terminator` property was automatically written. However, in some cases you might want to suppress writing the terminator. To do so, you must explicitly specify a format for the data that does not include the terminator, or configure the terminator to empty.

```
fprintf(s, '%s', 'RS232?')
```

## See Also

### Functions

`fopen`, `fwrite`, `stopasync`

# **fprintf (serial)**

---

## **Properties**

BytesToOutput, OutputBufferSize, OutputEmptyFcn,  
Status, TransferStatus, ValuesSent

**Purpose** Return image data associated with movie frame

**Syntax** [X,Map] = frame2im(F)

**Description** [X,Map] = frame2im(F) returns the indexed image X and associated colormap Map from the single movie frame F. If the frame contains true-color data, the MxNx3 matrix Map is empty. The functions getframe and im2frame create a movie frame.

**Example** Create and capture an image using getframe and frame2im:

```
peaks                                %Make figure
f = getframe;                         %Capture screen shot
[im,map] = frame2im(f);               %Return associated image data
if isempty(map)                       %Truecolor system
    rgb = im;
else                                   %Indexed system
    rgb = ind2rgb(im,map);            %Convert image data
end
```

**See Also** getframe, im2frame, movie

# fread

---

**Purpose** Read binary data from file

**Syntax**

```
A = fread(fid)
A = fread(fid, count)
A = fread(fid, count, precision)
A = fread(fid, count, precision, skip)
A = fread(fid, count, precision, skip, machineformat)
[A, count] = fread(...)
```

**Description** `A = fread(fid)` reads data in binary format from the file specified by `fid` into matrix `A`. Open the file using `fopen` before calling `fread`. The `fid` argument is the integer file identifier obtained from the `fopen` operation. The MATLAB software reads the file from beginning to end, and then positions the file pointer at the end of the file (see `feof` for details).

---

**Note** `fread` is intended primarily for binary data. When reading text files, use the `fgetl` function.

---

`A = fread(fid, count)` reads the number of elements specified by `count`. At the end of the `fread`, MATLAB sets the file pointer to the next byte to be read. A subsequent `fread` will begin at the location of the file pointer. See “Specifying the Number of Elements” on page 2-1353, below.

---

**Note** In the following syntaxes, the `count` and `skip` arguments are optional. For example, `fread(fid, precision)` is a valid syntax.

---

`A = fread(fid, count, precision)` reads the file according to the data format specified by the string `precision`. This argument commonly contains a data type specifier such as `int` or `float`, followed by an integer giving the size in bits. See “Specifying precision” on page 2-1353 and “Specifying Output Format” on page 2-1355, below.

`A = fread(fid, count, precision, skip)` includes an optional `skip` argument that specifies the number of bytes to skip after each `precision` value is read. If `precision` specifies a bit format like `'bitN'` or `'ubitN'`, the `skip` argument is interpreted as the number of bits to skip. See “Specifying a Skip Value” on page 2-1356, below.

`A = fread(fid, count, precision, skip, machineformat)` treats the data read as having a format given by `machineformat`. You can obtain the `machineformat` argument from the output of the `fopen` function. See `fopen` for possible values for `machineformat`.

`[A, count] = fread(...)` returns the data read from the file in `A`, and the number of elements successfully read in `count`.

### Specifying the Number of Elements

Valid options for `count` are

- `n` Reads `n` elements into a column vector.
- `inf` Reads to the end of the file, resulting in a column vector containing the same number of elements as are in the file. If using `inf` results in an "out of memory" error, specify a numeric count value.
- `[m,n]` Reads enough elements to fill an `m`-by-`n` matrix, filling in elements in column order, padding with zeros if the file is too small to fill the matrix. `n` can be specified as `inf`, but `m` cannot.

### Specifying precision

Any of the strings in the following table, either the MATLAB version or their C or Fortran equivalent, can be used for `precision`. If `precision` is not specified, MATLAB uses the default, which is `'uint8'`.

MATLAB	C or Fortran	Interpretation
'schar'	'signed char'	Signed integer; 8 bits
'uchar'	'unsigned char'	Unsigned integer; 8 bits

<b>MATLAB</b>	<b>C or Fortran</b>	<b>Interpretation</b>
'int8'	'integer*1'	Integer; 8 bits
'int16'	'integer*2'	Integer; 16 bits
'int32'	'integer*4'	Integer; 32 bits
'int64'	'integer*8'	Integer; 64 bits
'uint8'	'integer*1'	Unsigned integer; 8 bits
'uint16'	'integer*2'	Unsigned integer; 16 bits
'uint32'	'integer*4'	Unsigned integer; 32 bits
'uint64'	'integer*8'	Unsigned integer; 64 bits
'float32'	'real*4'	Floating-point; 32 bits
'float64'	'real*8'	Floating-point; 64 bits
'double'	'real*8'	Floating-point; 64 bits

The following platform-dependent formats are also supported, but they are not guaranteed to be the same size on all platforms.

<b>MATLAB</b>	<b>C or Fortran</b>	<b>Interpretation</b>
'char'	'char*1'	Character
'short'	'short'	Integer; 16 bits
'int'	'int'	Integer; 32 bits
'long'	'long'	Integer; 32 or 64 bits
'ushort'	'unsigned short'	Unsigned integer; 16 bits
'uint'	'unsigned int'	Unsigned integer; 32 bits
'ulong'	'unsigned long'	Unsigned integer; 32 or 64 bits
'float'	'float'	Floating-point; 32 bits



---

**Note** If the format is 'char' or 'char\*1', MATLAB reads characters using the encoding scheme associated with the file. See `fopen` for more information.

---

The following formats map to an input stream of bits rather than bytes.

<b>MATLAB</b>	<b>C or Fortran</b>	<b>Interpretation</b>
'bitN'	-	Signed integer; N bits ( $1 \leq N \leq 64$ )
'ubitN'	-	Unsigned integer; N bits ( $1 \leq N \leq 64$ )

### Specifying Output Format

By default, numeric and character values are returned in class `double` arrays. To return these values stored in classes other than `double`, create your `format` argument by first specifying your source format, then following it with the characters “=>,” and finally specifying your destination format. You are not required to use the exact name of a MATLAB class type for destination. (See `class` for details). `fread` translates the name to the most appropriate MATLAB class type. If the source and destination formats are the same, the following shorthand notation can be used.

```
*source
```

which means

```
source=>source
```

For example, '\*uint16' is the same as 'uint16=>uint16'.

---

**Note** You can also use the `*source` notation with an input stream that is specified as a number of bits (e.g., `bit4` or `ubit18`). MATLAB translates this into an output type that is a signed or unsigned integer (depending on the input type), and that is large enough to hold all of the bits in the source format. For example, `*ubit18` does not translate to `ubit18=>ubit18`, but instead to `ubit18=>uint32`.

---

This table shows some example precision format strings.

<code>'uint8=&gt;uint8'</code>	Read in unsigned 8-bit integers and save them in an unsigned 8-bit integer array.
<code>'*uint8'</code>	Shorthand version of the above.
<code>'bit4=&gt;int8'</code>	Read in signed 4-bit integers packed in bytes and save them in a signed 8-bit array. Each 4-bit integer becomes an 8-bit integer.
<code>'double=&gt;real*4'</code>	Read in doubles, convert, and save as a 32-bit floating-point array.

## Specifying a Skip Value

When `skip` is used, the precision string can contain a positive integer repetition factor of the form `'N*'`, which prefixes the source format specification, such as `'40*uchar'`.

---

**Note** Do not confuse the asterisk (\*) used in the repetition factor with the asterisk used as precision format shorthand. The format string `'40*uchar'` is equivalent to `'40*uchar=>double'`, not `'40*uchar=>uchar'`.

---

When `skip` is specified, `fread` reads in, at most, a repetition factor number of values (default is 1), skips the amount of input specified by the `skip` argument, reads in another block of values, again skips

input, and so on, until `count` number of values have been read. If a `skip` argument is not specified, the repetition factor is ignored. Use the repetition factor with the `skip` argument to extract data in noncontiguous fields from fixed-length records.

## Remarks

If the input stream is bytes and `fread` reaches the end of file (see `feof`) in the middle of reading the number of bytes required for an element, the partial result is ignored. However, if the input stream is bits, then the partial result is returned as the last value. If an error occurs before reaching the end of file, only full elements read up to that point are used.

## Examples

### Example 1

The file `alphabet.txt` contains the 26 letters of the English alphabet, all capitalized. Open the file for read access with `fopen`, and read the first five elements into output `c`. Because a precision has not been specified, MATLAB uses the default precision of `uint8`, and the output is numeric:

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, 5)
c =
    65    66    67    68    69
fclose(fid);
```

This time, specify that you want each element read as an unsigned 8-bit integer and output as a character. (Using a precision of `'char=>char'` or `'*char'` will produce the same result):

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, 5, 'uint8=>char')
c =
    ABCDE
fclose(fid);
```

When you leave out the optional count argument, MATLAB reads the file to the end, A through Z:

# fread

---

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, '*char')'
c =
    ABCDEFGHIJKLMNOPQRSTUVWXYZ
fclose(fid);
```

The `fopen` function positions the file pointer at the start of the file. So the first `fread` in this example reads the first five elements in the file, and then repositions the file pointer at the beginning of the next element. For this reason, the next `fread` picks up where the previous `fread` left off, at the character F.

```
fid = fopen('alphabet.txt', 'r');
c1 = fread(fid, 5, '*char');
c2 = fread(fid, 8, '*char');
c3 = fread(fid, 5, '*char');
fclose(fid);

sprintf('%c', c1, ' * ', c2, ' * ', c3)
ans =
    ABCDE * FGHIJKLM * NOPQR
```

Skip two elements between each read by specifying a `skip` argument of 2:

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, '*char', 2);    % Skip 2 bytes per read
fclose(fid);

sprintf('%c', c)
ans =
    ADGJMPSVY
```

## Example 2

This command displays the complete M-file containing this `fread` help entry:

```
type fread.m
```

To simulate this command using `fread`, enter the following:

```
fid = fopen('fread.m', 'r');
F = fread(fid, '*char')';
fclose(fid);
```

In the example, the `fread` command assumes the default size, `'inf'`, and precision `'*char'` (the same as `'char=>char'`). `fread` reads the entire file. To display the result as readable text, the column vector is transposed to a row vector.

### Example 3

As another example,

```
s = fread(fid, 120, '40*uchar=>uchar', 8);
```

reads in 120 bytes in blocks of 40, each separated by 8 bytes. Note that the class type of `s` is `'uint8'` since it is the appropriate class corresponding to the destination format `'uchar'`. Also, since 40 evenly divides 120, the last block read is a full block, which means that a final skip is done before the command is finished. If the last block read is not a full block, then `fread` does not finish with a skip.

See `fopen` for information about reading big and little-endian files.

### Example 4

Invoke the `fopen` function with just an `fid` input argument to obtain the machine format for the file. You can see that this file was written in IEEE floating point with little-endian byte ordering (`'ieee-le'`) format:

```
fid = fopen('A1.dat', 'r');

[fname, mode, mformat] = fopen(fid);
mformat
mformat =
    ieee-le
```

Use the MATLAB `format` function (not related to the machine format type) to have MATLAB display output using hexadecimal:

```
format hex
```

Now use the `machineformat` input with `fread` to read the data from the file using the same format:

```
x = fread(fid, 6, 'uint64', 'ieee-le')
x =
    4260800000002000
    0000000000000000
    4282000000180000
    0000000000000000
    42ca5e0000258000
    42f0000464d45200
fclose(fid);
```

Change the machine format to IEEE floating point with big-endian byte ordering (`'ieee-be'`) and verify that you get different results:

```
fid = fopen('A1.dat', 'r');
x = fread(fid, 6, 'uint64', 'ieee-be')
x =
    4370000008400000
    0000000000000000
    4308000200100000
    0000000000000000
    4352c0002f0d0000
    43c022a6a3000000
fclose(fid);
```

## Example 5

This example reads some Japanese text from a file that uses the Shift-JIS character encoding scheme. It creates a string of Unicode characters, `str`, and displays the string. Note that the computer must be configured to display Japanese (e.g., a Japanese machine running the Windows operating system) for the output of `disp(str)` to be correct.

```
fid = fopen('japanese.txt', 'r', 'n', 'Shift_JIS');
str = fread(fid, '*char')';
```

```
fclose(fid);  
disp(str);
```

**See Also**

fgetl, fscanf, fwrite, fprintf, fopen, fclose, fseek, ftell, feof

# fread (serial)

---

**Purpose** Read binary data from device

**Syntax**  
`A = fread(obj)`  
`A = fread(obj,size,'precision')`  
`[A,count] = fread(...)`  
`[A,count,msg] = fread(...)`

**Description** `A = fread(obj)` and `A = fread(obj,size)` read binary data from the device connected to the serial port object, `obj`, and returns the data to `A`. The maximum number of values to read is specified by `size`. If `size` is not specified, the maximum number of values to read is determined by the object's `InputBufferSize` property. Valid options for `size` are:

<code>n</code>	Read at most <code>n</code> values into a column vector.
<code>[m,n]</code>	Read at most <code>m</code> -by- <code>n</code> values filling an <code>m</code> -by- <code>n</code> matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. A value is defined as a byte multiplied by the *precision* (see below).

`A = fread(obj,size,'precision')` reads binary data with precision specified by *precision*.

*precision* controls the number of bits read for each value and the interpretation of those bits as integer, floating-point, or character values. If *precision* is not specified, `uchar` (an 8-bit unsigned character) is used. By default, numeric values are returned in double-precision arrays. The supported values for *precision* are listed below in Remarks.

`[A,count] = fread(...)` returns the number of values read to `count`.

`[A,count,msg] = fread(...)` returns a warning message to `msg` if the read operation was unsuccessful.



## Remarks

Before you can read data from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read, each time `fread` is issued.

If you use the `help` command to display help for `fread`, then you need to supply the pathname shown below.

```
help serial/fread
```

## Rules for Completing a Binary Read Operation

A read operation with `fread` blocks access to the MATLAB command line until:

- The specified number of values are read.
- The time specified by the `Timeout` property passes.

---

**Note** The `Terminator` property is not used for binary read operations.

---

## Supported Precisions

The supported values for *precision* are listed below.

Data Type	Precision	Interpretation
Character	<code>uchar</code>	8-bit unsigned character
	<code>schar</code>	8-bit signed character
	<code>char</code>	8-bit signed or unsigned character

# fread (serial)

---

<b>Data Type</b>	<b>Precision</b>	<b>Interpretation</b>
Integer	int8	8-bit integer
	int16	16-bit integer
	int32	32-bit integer
	uint8	8-bit unsigned integer
	uint16	16-bit unsigned integer
	uint32	32-bit unsigned integer
	short	16-bit integer
	int	32-bit integer
	long	32- or 64-bit integer
	ushort	16-bit unsigned integer
	uint	32-bit unsigned integer
	ulong	32- or 64-bit unsigned integer
Floating-point	single	32-bit floating point
	float32	32-bit floating point
	float	32-bit floating point
	double	64-bit floating point
	float64	64-bit floating point

## See Also

### Functions

fgetc1, fgets, fopen, fscanff

### Properties

BytesAvailable, BytesAvailableFcn, InputBufferSize, Status, Terminator, ValuesReceived

**Purpose** Frequency spacing for frequency response

**Syntax**

```
[f1,f2] = freqspace(n)
[f1,f2] = freqspace([m n])
[x1,y1] = freqspace(...,'meshgrid')
f = freqspace(N)
f = freqspace(N,'whole')
```

**Description** `freqspace` returns the implied frequency range for equally spaced frequency responses. `freqspace` is useful when creating desired frequency responses for various one- and two-dimensional applications.

`[f1,f2] = freqspace(n)` returns the two-dimensional frequency vectors `f1` and `f2` for an `n`-by-`n` matrix.

For `n` odd, both `f1` and `f2` are `[-n+1:2:n-1]/n`.

For `n` even, both `f1` and `f2` are `[-n:2:n-2]/n`.

`[f1,f2] = freqspace([m n])` returns the two-dimensional frequency vectors `f1` and `f2` for an `m`-by-`n` matrix.

`[x1,y1] = freqspace(...,'meshgrid')` is equivalent to

```
[f1,f2] = freqspace(...);
[x1,y1] = meshgrid(f1,f2);
```

`f = freqspace(N)` returns the one-dimensional frequency vector `f` assuming `N` evenly spaced points around the unit circle. For `N` even or odd, `f` is `(0:2/N:1)`. For `N` even, `freqspace` therefore returns  $(N+2)/2$  points. For `N` odd, it returns  $(N+1)/2$  points.

`f = freqspace(N,'whole')` returns `N` evenly spaced points around the whole unit circle. In this case, `f` is `0:2/N:2*(N-1)/N`.

**See Also** `meshgrid`

# frewind

---

<b>Purpose</b>	Move file position indicator to beginning of open file
<b>Syntax</b>	<code>frewind(fid)</code>
<b>Description</b>	<code>frewind(fid)</code> sets the file position indicator to the beginning of the file specified by <code>fid</code> , an integer file identifier obtained from <code>fopen</code> .
<b>Remarks</b>	Rewinding a <code>fid</code> associated with a tape device might not work even though <code>frewind</code> does not generate an error message.
<b>See Also</b>	<code>fclose</code> , <code>ferror</code> , <code>fopen</code> , <code>fprintf</code> , <code>fread</code> , <code>fscanf</code> , <code>fseek</code> , <code>ftell</code> , <code>fwrite</code>

**Purpose**

Read formatted data from file

**Syntax**

```
A = fscanf(fid, format)
[A,count] = fscanf(fid, format, size)
```

**Description**

`A = fscanf(fid, format)` reads data from the file specified by `fid`, converts it according to the specified `format` string, and returns it in matrix `A`. Argument `fid` is an integer file identifier obtained from `fopen`. `format` is a string specifying the format of the data to be read. See "Remarks" for details.

`[A,count] = fscanf(fid, format, size)` reads the amount of data specified by `size`, converts it according to the specified `format` string, and returns it along with a `count` of values successfully read. `size` is an argument that determines how much data is read. Valid options are

- |                    |   |
|--------------------|---|
| <code>n</code>     | Read at most <code>n</code> numbers, characters, or strings.  |
| <code>inf</code>   | Read to the end of the file.  |
| <code>[m,n]</code> | Read at most $(m*n)$ numbers, characters, or strings. Fill a matrix of at most <code>m</code> rows in column order. <code>n</code> can be <code>inf</code> , but <code>m</code> cannot. |

Characteristics of the output matrix `A` depend on the values read from the file and on the `size` argument. If `fscanf` reads only numbers, and if `size` is not of the form `[m,n]`, matrix `A` is a column vector of numbers. If `fscanf` reads only characters or strings, and if `size` is not of the form `[m,n]`, matrix `A` is a row vector of characters. See the Remarks section for more information.

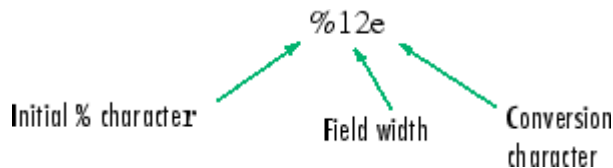
`fscanf` differs from its C language namesake `fscanf()` in an important respect — it is *vectorized* to return a matrix argument. The `format` string is cycled through the file until the first of these conditions occurs:

- The `format` string fails to match the data in the file
- The amount of data specified by `size` is read
- The end of the file is reached

## Remarks

When the MATLAB software reads a specified file, it attempts to match the data in the file to the format string. If a match occurs, the data is written into the output matrix. If a partial match occurs, only the matching data is written to the matrix, and the read operation stops.

The format string consists of ordinary characters and/or conversion specifications. Conversion specifications indicate the type of data to be matched and involve the character %, optional width fields, and conversion characters, organized as shown below.



Add one or more of these characters between the % and the conversion character:

- |                 |  |
|-----------------|--|
| An asterisk (*) | Skip over the matched value. If %*d, then the value that matches d is ignored and is not stored.   |
| A digit string  | Maximum field width. For example, %10d.  |
| A letter        | The size of the receiving object, for example, h for short, as in %hd for a short integer, or l for long, as in %ld for a long integer, or %lg for a double floating-point number. |

Valid conversion characters are

- |            |   |
|------------|---|
| %c         | Sequence of characters; number specified by field width   |
| %d         | Base 10 integers  |
| %e, %f, %g | Floating-point numbers  |
| %i         | Defaults to base 10 integers. Data starting with 0 is read as base 8. Data starting with 0x or 0X is read as base 16. |

<code>%o</code>	Signed octal integer
<code>%s</code>	A series of non-white-space characters
<code>%u</code>	Unsigned decimal
<code>%x</code>	Signed hexadecimal integer
<code>[...]</code>	Sequence of characters (scanlist)

Format specifiers `%e`, `%f`, and `%g` accept the text `'inf'`, `'-inf'`, `'nan'`, and `'-nan'`. This text is not case sensitive. The `fscanf` function converts these to the numeric representation of `Inf`, `-Inf`, `NaN`, and `-NaN`.

Use `%c` to read space characters or `%s` to skip all white space. MATLAB skips over any ordinary characters that are used in the format specifier (see Example 2 below).

MATLAB reads characters using the encoding scheme associated with the file. See `fopen` for more information. If the format string contains ordinary characters, MATLAB matches each of those characters with a character read from the file after converting both to the MATLAB internal representation of characters.

For more information about format strings, refer to the `scanf()` and `fscanf()` routines in a C language reference manual.

### **Output Characteristics: Only Numeric Values Read**

Format characters that cause `fscanf` to read numbers from the file are `%d`, `%e`, `%f`, `%g`, `%i`, `%o`, `%u`, and `%x`. When `fscanf` reads only numbers from the file, the elements of the output matrix `A` are numbers.

When there is no `size` argument or the `size` argument is `inf`, `fscanf` reads to the end of the file. The output matrix is a column vector with one element for each number read from the input.

When the `size` argument is a scalar `n`, `fscanf` reads at most `n` numbers from the file. The output matrix is a column vector with one element for each number read from the input.

When the `size` argument is a matrix  $[m,n]$ , `fscanf` reads at most  $(m*n)$  numbers from the file. The output matrix contains at most  $m$  rows and  $n$  columns. `fscanf` fills the output matrix in column order, using as many columns as it needs to contain all the numbers read from the input. Any unfilled elements in the final column contain zeros.

## **Output Characteristics: Only Character Values Read**

The format characters that cause `fscanf` to read characters and strings from the file are `%c` and `%s`. When `fscanf` reads only characters and strings from the file, the elements of the output matrix `A` are characters. When `fscanf` reads a string from the input, the output matrix includes one element for each character in the string.

When there is no `size` argument or the `size` argument is `inf`, `fscanf` reads to the end of the file. The output matrix is a row vector with one element for each character read from the input.

When the `size` argument is a scalar  $n$ , `fscanf` reads at most  $n$  character or string values from the file. The output matrix is a row vector with one element for each character read from the input. When string values are read from the input, the output matrix can contain more than  $n$  columns.

When the `size` argument is a matrix  $[m,n]$ , `fscanf` reads at most  $(m*n)$  character or string values from the file. The output matrix contains at most  $m$  rows. `fscanf` fills the output matrix in column order, using as many columns as it needs to contain all the characters read from the input. When string values are read from the input, the output matrix can contain more than  $n$  columns. Any unfilled elements in the final column contain `char(0)`.

## **Output Characteristics: Both Numeric and Character Values Read**

When `fscanf` reads a combination of numbers and either characters or strings from the file, the elements of the output matrix `A` are numbers. This is true even when a format specifier such as `'%*d %s'` tells MATLAB to ignore numbers in the input string and output only characters or strings. When `fscanf` reads a string from the input, the



output matrix includes one element for each character in the string. All characters are converted to their numeric equivalents in the output matrix.

When there is no `size` argument or the `size` argument is `inf`, `fscanf` reads to the end of the file. The output matrix is a column vector with one element for each character read from the input.

When the `size` argument is a scalar `n`, `fscanf` reads at most `n` number, character, or string values from the file. The output matrix contains at most `n` rows. `fscanf` fills the output matrix in column order, using as many columns as it needs to represent all the numbers and characters read from the input. When string values are read from the input, the output matrix can contain more than one column. Any unfilled elements in the final column contain zeros.

When the `size` argument is a matrix `[m,n]`, `fscanf` reads at most `(m*n)` number, character, or string values from the file. The output matrix contains at most `m` rows. `fscanf` fills the output matrix in column order, using as many columns as it needs to represent all the numbers and characters read from the input. When string values are read from the input, the output matrix can contain more than `n` columns. Any unfilled elements in the final column contain zeros.

---

**Note** This section applies only when `fscanf` actually reads a combination of numbers and either characters or strings from the file. Even if the format string has both format characters that would result in numbers (such as `%d`) and format characters that would result in characters or strings (such as `%s`), `fscanf` might actually read only numbers or only characters or strings. If `fscanf` reads only numbers, see “Output Characteristics: Only Numeric Values Read” on page 2-1369. If `fscanf` reads only characters or strings, see “Output Characteristics: Only Character Values Read” on page 2-1370.

---

## Examples

### Example 1

An example in `fprintf` generates a text file called `exp.txt` that looks like this:

```
0.00    1.00000000
0.10    1.10517092
...
1.00    2.71828183
```

Read this file back into a two-column MATLAB matrix:

```
fid = fopen('exp.txt', 'r');
a = fscanf(fid, '%g %g', [2 inf])    % It has two rows now.
a = a';
fclose(fid)
```

### Example 2

Start with a file `temp.dat` that contains temperature readings:

```
78 F 72 F 64 F 66 F 49 F
```

Open the file using `fopen` and read it with `fscanf`. If you include ordinary characters (such as the degree (°) and Farrenheit (F) symbols used here) in the conversion string, `fscanf` skips over those characters when reading the string:

```
fid = fopen('temps.dat', 'r');

degrees = char(176)
degrees =

fscanf(fid, ['%d' degrees 'F'])
ans =
    78
    72
    64
    66
    49
```

**See Also**      `fgetc1`, `fgets`, `fread`, `fprintf`, `fscanf`, `input`, `sscanf`, `textread`

# fscanf (serial)

---

**Purpose** Read data from device, and format as text

**Syntax**

```
A = fscanf(obj)
A = fscanf(obj,'format')
A = fscanf(obj,'format',size)
[A,count] = fscanf(...)
[A,count,msg] = fscanf(...)
```

**Description** `A = fscanf(obj)` reads data from the device connected to the serial port object, `obj`, and returns it to `A`. The data is converted to text using the `%c` format.

`A = fscanf(obj,'format')` reads data and converts it according to `format`. `format` is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters `d`, `i`, `o`, `u`, `x`, `X`, `f`, `e`, `E`, `g`, `G`, `c`, and `s`. Refer to the `sscanf` file I/O format specifications or a C manual for more information.

`A = fscanf(obj,'format',size)` reads the number of values specified by `size`. Valid options for `size` are:

<code>n</code>	Read at most <code>n</code> values into a column vector.
<code>[m,n]</code>	Read at most <code>m</code> -by- <code>n</code> values filling an <code>m</code> -by- <code>n</code> matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. If `size` is not of the form `[m,n]`, and a character conversion is specified, then `A` is returned as a row vector. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. An ASCII value is one byte.

`[A,count] = fscanf(...)` returns the number of values read to `count`.

`[A,count,msg] = fscanf(...)` returns a warning message to `msg` if the read operation did not complete successfully.

## Remarks

Before you can read data from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fscanf` is issued.

If you use the `help` command to display help for `fscanf`, then you need to supply the pathname shown below.

```
help serial/fscanf
```

## Rules for Completing a Read Operation with fscanf

A read operation with `fscanf` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The number of values specified by `size` is read.
- The input buffer is filled (unless `size` is specified)

## Example

Create the serial port object `s` and connect `s` to a Tektronix TDS 210 oscilloscope, which is displaying sine wave.

```
s = serial('COM1');  
fopen(s)
```

Use the `fprintf` function to configure the scope to measure the peak-to-peak voltage of the sine wave, return the measurement type, and return the peak-to-peak voltage.

```
fprintf(s, 'MEASUREMENT:IMMED:TYPE PK2PK')
```

## fscanf (serial)

---

```
fprintf(s, 'MEASUREMENT:IMMED:TYPE?')  
fprintf(s, 'MEASUREMENT:IMMED:VALUE?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data associated with the two query commands is automatically returned to the input buffer.

```
s.BytesAvailable  
ans =  
    21
```

Use `fscanf` to read the measurement type. The operation will complete when the first terminator is read.

```
meas = fscanf(s)  
meas =  
PK2PK
```

Use `fscanf` to read the peak-to-peak voltage as a floating-point number, and exclude the terminator.

```
pk2pk = fscanf(s, '%e', 14)  
pk2pk =  
    2.0200
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)  
delete(s)  
clear s
```

### See Also

### Functions

`fgetl`, `fgets`, `fopen`, `fread`, `strread`

## Properties

BytesAvailable, BytesAvailableFcn, InputBufferSize, Status, Terminator, Timeout

# fseek

---

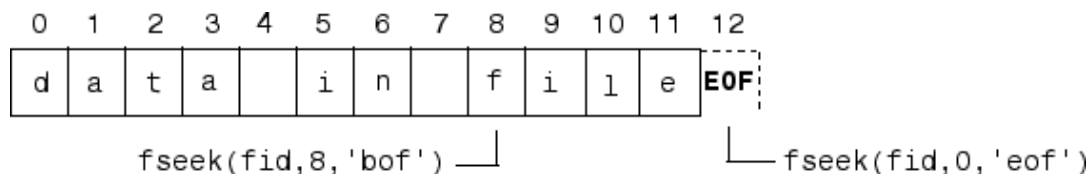
**Purpose** Set file position indicator

**Syntax** `status = fseek(fid, offset, origin)`

**Description** `status = fseek(fid, offset, origin)` repositions the file position indicator in the file with the given `fid` to the byte with the specified `offset` relative to `origin`.

For a file having `n` bytes, the bytes are numbered from 0 to `n-1`. The position immediately following the last byte is the end-of-file, or `eof`, position. You would seek to the `eof` position if you wanted to add data to the end of a file.

This figure represents a file having 12 bytes, numbered 0 through 11. The first command shown seeks to the ninth byte of data in the file. The second command seeks just past the end of the file data, to the `eof` position.



`fseek` does not seek beyond the end of file `eof` position. If you attempt to seek beyond `eof`, the MATLAB software returns an error status.

## Arguments

`fid` An integer file identifier obtained from `fopen`

`offset` A value that is interpreted as follows,

`offset > 0` Move position indicator `offset` bytes toward the end of the file.

`offset = 0` Do not change position.



	<code>offset &lt; 0</code>	Move position indicator <code>offset</code> bytes toward the beginning of the file.
<code>origin</code>	A string whose legal values are <code>'bof'</code> <code>'cof'</code> <code>'eof'</code>	<code>-1</code> : Beginning of file <code>0</code> : Current position in file <code>1</code> : End of file
<code>status</code>		A returned value that is <code>0</code> if the <code>fseek</code> operation is successful and <code>-1</code> if it fails. If an error occurs, use the function <code>ferror</code> to get more information.

## Examples

This example opens the file `test1.dat`, seeks to the 20th byte, reads fifty 32-bit unsigned integers into variable `A`, and closes the file. It then opens a second file, `test2.dat`, seeks to the end-of-file position, appends the data in `A` to the end of this file, and closes the file.

```
fid = fopen('test1.dat', 'r');
fseek(fid, 19, 'bof');
A = fread(fid, 50, 'uint32');
fclose(fid);

fid = fopen('test2.dat', 'r+');
fseek(fid, 0, 'eof');
fwrite(fid, A, 'uint32');
fclose(fid);
```

## See Also

`fopen`, `fclose`, `ferror`, `fprintf`, `fread`, `fscanf`, `ftell`, `fwrite`

# ftell

---

**Purpose** File position indicator

**Syntax** `position = ftell(fid)`

**Description** `position = ftell(fid)` returns the location of the file position indicator for the file specified by `fid`, an integer file identifier obtained from `fopen`. The `position` is a nonnegative, zero-based integer specified in bytes from the beginning of the file. A returned value of `-1` for `position` indicates that the query was unsuccessful; use `ferror` to determine the nature of the error.

**Remarks** `ftell` is likely to return an invalid `position` when *all* of the following are true. This is due to the way in which the Microsoft Windows C library currently handles its `ftell` and `fgetpos` commands:

- The file you are currently operating on is an ASCII text file.
- The file was written on a UNIX-based system, or uses the UNIX-style line terminator: a line feed (with no carriage return) at the end of each line of text. (This is the default output format for MATLAB functions `dlmwrite` and `csvwrite`.)
- You are reading the file on a Windows system.
- You opened the file with the `fopen` function with `mode` set to `'rt'`.
- The `ftell` command is directly preceded by an `fgets` or `fgetl` command.

Note that this does not affect the ability to accurately read from and write to this type of file from MATLAB.

**See Also** `fclose`, `ferror`, `fopen`, `fprintf`, `fread`, `fscanf`, `fseek`, `fwrite`

**Purpose** Connect to FTP server, creating FTP object

**Syntax** `f = ftp('host','username','password')`

**Description** `f = ftp('host','username','password')` connects to the FTP server, `host`, creating the FTP object, `f`. If a user name and password are not required for an anonymous connection, only use the `host` argument. Specify an alternate port by separating it from `host` using a colon (:). After running `ftp`, perform file operation functions on the FTP object, `f`, using methods such as `cd` and others listed under "See Also." When you're finished using the server, run `close (ftp)` to close the connection.

FTP is not a secure protocol; others can see your user name and password.

The `ftp` function is based on code from the Apache Jakarta Project.

## Examples

### Connect Without User Name

Connect to `ftp.mathworks.com`, which does not require a user name or password. Assign the resulting FTP object to `tmw`. You can access this FTP site to experiment with the FTP functions.

```
tmw=ftp('ftp.mathworks.com')
```

```
tmw =  
FTP Object  
host: ftp.mathworks.com  
user: anonymous  
dir: /  
mode: binary
```

### Connect to Specified Port

To connect to port 34, type:

```
tmw=ftp('ftp.mathworks.com:34')
```

## Connect with User Name

Connect to ftp.testsite.com and assign the resulting FTP object to test.

```
test=ftp('ftp.testsite.com','myname','mypassword')
```

```
test =  
FTP Object  
host: ftp.testsite.com  
user: myname  
dir: /  
mode: binary  
myname@ftp.testsite.com  
/
```

## See Also

ascii, binary, cd (ftp), close (ftp), delete (ftp), dir (ftp),  
mget, mkdir (ftp), mput, rename, rmdir (ftp)

**Purpose**

Convert sparse matrix to full matrix

**Syntax**

`A = full(S)`

**Description**

`A = full(S)` converts a sparse matrix `S` to full storage organization. If `S` is a full matrix, it is left unchanged. If `A` is full, `issparse(A)` is 0.

**Remarks**

Let `X` be an `m`-by-`n` matrix with `nz = nnz(X)` nonzero entries. Then `full(X)` requires space to store `m*n` real numbers while `sparse(X)` requires space to store `nz` real numbers and `(nz+n)` integers.

On most computers, a real number requires twice as much storage as an integer. On such computers, `sparse(X)` requires less storage than `full(X)` if the density, `nnz/prod(size(X))`, is less than one third. Operations on sparse matrices, however, require more execution time per element than those on full matrices, so density should be considerably less than two-thirds before sparse storage is used.

**Examples**

Here is an example of a sparse matrix with a density of about two-thirds. `sparse(S)` and `full(S)` require about the same number of bytes of storage.

```
S = sparse+(rand(200,200) < 2/3));
A = full(S);
whos
Name      Size      Bytes      Class
   A      200X200  320000  double array
   S      200X200  318432  double array (sparse)
```

**See Also**

`issparse`, `sparse`

# fullfile

---

**Purpose** Build full filename from parts

**Syntax** `f = fullfile(dir1, dir2, ..., filename)`

**Description** `f = fullfile(dir1, dir2, ..., filename)` builds a full file specification `f` from the directories and filename specified. Input arguments `dir1`, `dir2`, etc. and `filename` are each a string enclosed in single quotes. The output of the `fullfile` command is conceptually equivalent to

```
f = [dir1 filesep dir2 filesep ... filesep filename]
```

except that care is taken to handle the cases when the directories begin or end with a directory separator.

**Examples** To create the full filename from a disk name, directories, and filename,

```
f = fullfile('C:', 'Applications', 'matlab', 'myfun.m')
f =
C:\Applications\matlab\myfun.m
```

The following examples both produce the same result on UNIX<sup>11</sup> platforms, but only the second one works on all platforms.

```
fullfile(matlabroot, 'toolbox/matlab/general/Contents.m')
fullfile(matlabroot, 'toolbox', 'matlab', 'general', ...
'Contents.m')
```

**See Also** `fileparts`, `filesep`, `path`, `pathsep`, `genpath`

11. UNIX is a registered trademark of The Open Group in the United States and other countries

---

<b>Purpose</b>	Construct function name string from function handle
<b>Syntax</b>	<code>func2str(fhandle)</code>
<b>Description</b>	<p><code>func2str(fhandle)</code> constructs a string <code>s</code> that holds the name of the function to which the function handle <code>fhandle</code> belongs.</p> <p>When you need to perform a string operation, such as compare or display, on a function handle, you can use <code>func2str</code> to construct a string bearing the function name.</p> <p>The <code>func2str</code> command does not operate on nonscalar function handles. Passing a nonscalar function handle to <code>func2str</code> results in an error.</p>

## Examples

### Example 1

Convert a `sin` function handle to a string:

```
fhandle = @sin;

func2str(fhandle)
ans =
    sin
```

### Example 2

The `catcherr` function shown here accepts function handle and data arguments and attempts to evaluate the function through its handle. If the function fails to execute, `catcherr` uses `sprintf` to display an error message giving the name of the failing function. The function name must be a string for `sprintf` to display it. The code derives the function name from the function handle using `func2str`:

```
function catcherr(func, data)
try
    ans = func(data);
    disp('Answer is:');
    ans
catch
```

## func2str

---

```
        disp(sprintf('Error executing function '%s'\n', ...
                    func2str(func)))
    end
```

The first call to `catcherr` passes a handle to the `round` function and a valid data argument. This call succeeds and returns the expected answer. The second call passes the same function handle and an improper data type (a MATLAB structure). This time, `round` fails, causing `catcherr` to display an error message that includes the failing function name:

```
catcherr(@round, 5.432)
ans =
Answer is 5

xstruct.value = 5.432;
catcherr(@round, xstruct)
Error executing function "round"
```

### See Also

`function_handle`, `str2func`, `functions`



**Purpose**

Declare M-file function

**Syntax**

```
function [out1, out2, ...] = funname(in1, in2, ...)
```

**Description**

`function [out1, out2, ...] = funname(in1, in2, ...)` defines function `funname` that accepts inputs `in1`, `in2`, etc. and returns outputs `out1`, `out2`, etc.

You add new functions to the MATLAB vocabulary by expressing them in terms of existing functions. The existing commands and functions that compose the new function reside in a text file called an *M-file*.

M-files can be either *scripts* or *functions*. Scripts are simply files containing a sequence of MATLAB statements. Functions make use of their own local variables and accept input arguments.

The name of an M-file begins with an alphabetic character and has a filename extension of `.m`. The M-file name, less its extension, is what MATLAB searches for when you try to use the script or function.

A line at the top of a function M-file contains the syntax definition. The name of a function, as defined in the first line of the M-file, should be the same as the name of the file without the `.m` extension.

The variables within the body of the function are all local variables.

A *subfunction*, visible only to the other functions in the same file, is created by defining a new function with the `function` keyword after the body of the preceding function or subfunction. Subfunctions are not visible outside the file where they are defined.

You can terminate any function with an `end` statement but, in most cases, this 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.

Functions normally return when the end of the function is reached. Use a `return` statement to force an early return.

# function

---

When MATLAB does not recognize a function by name, it searches for a file of the same name on disk. If the function is found, MATLAB compiles it into memory for subsequent use. The section “Determining Which Function Gets Called” in the MATLAB Programming Fundamentals documentation explains how MATLAB interprets variable and function names that you enter, and also covers the precedence used in function dispatching.

When you call an M-file function from the command line or from within another M-file, MATLAB parses the function and stores it in memory. The parsed function remains in memory until cleared with the `clear` command or you quit MATLAB. The `pcode` command performs the parsing step and stores the result on the disk as a P-file to be loaded later.

## Examples

### Example 1

The existence of a file on disk called `stat.m` containing this code defines a new function called `stat` that calculates the mean and standard deviation of a vector:

```
function [mean,stdev] = stat(x)
n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2/n));
```

### Example 2

`avg` is a subfunction within the file `stat.m`:

```
function [mean,stdev] = stat(x)
n = length(x);
mean = avg(x,n);
stdev = sqrt(sum((x-avg(x,n)).^2)/n);

function mean = avg(x,n)
mean = sum(x)/n;
```

## See Also

`nargin`, `nargout`, `pcode`, `varargin`, `varargout`, `what`

**Purpose** Handle used in calling functions indirectly

**Syntax**  
handle = @functionname  
handle = @(arglist)anonymous\_function

**Description** handle = @functionname returns a handle to the specified MATLAB function.

A function handle is a MATLAB value that provides a means of calling a function indirectly. You can pass function handles in calls to other functions (often called *function functions*). You can also store function handles in data structures for later use (for example, as Handle Graphics callbacks). A function handle is one of the standard MATLAB data types.

At the time you create a function handle, the function you specify must be on the MATLAB path and in the current scope. This condition does not apply when you evaluate the function handle. You can, for example, execute a subfunction from a separate (out-of-scope) M-file using a function handle as long as the handle was created within the subfunction's M-file (in-scope).

handle = @(arglist)anonymous\_function constructs an anonymous function and returns a handle to that function. The body of the function, to the right of the parentheses, is a single MATLAB statement or command. arglist is a comma-separated list of input arguments. Execute the function by calling it by means of the function handle, handle.

**Remarks** The function handle is a standard MATLAB data type. As such, you can manipulate and operate on function handles in the same manner as on other MATLAB data types. This includes using function handles in structures and cell arrays:

```
S.a = @sin; S.b = @cos; S.c = @tan;  
C = {@sin, @cos, @tan};
```

# function\_handle (@)

---

However, standard matrices or arrays of function handles are not supported:

```
A = [@sin, @cos, @tan];           % This is not supported
```

For nonoverloaded functions, subfunctions, and private functions, a function handle references just the one function specified in the @functionname syntax. When you evaluate an overloaded function by means of its handle, the arguments the handle is evaluated with determine the actual function that MATLAB dispatches to.

Use `isa(h, 'function_handle')` to see if variable `h` is a function handle.

## Examples

### Example 1 – Constructing a Handle to a Named Function

The following example creates a function handle for the `humps` function and assigns it to the variable `fhandle`.

```
fhandle = @humps;
```

Pass the handle to another function in the same way you would pass any argument. This example passes the function handle just created to `fminbnd`, which then minimizes over the interval `[0.3, 1]`.

```
x = fminbnd(fhandle, 0.3, 1)
x =
    0.6370
```

The `fminbnd` function evaluates the `@humps` function handle. A small portion of the `fminbnd` M-file is shown below. In line 1, the `funfcn` input parameter receives the function handle `@humps` that was passed in. The statement, in line 113, evaluates the handle.

```
1    function [xf,fval,exitflag,output] = ...
        fminbnd(funfcn,ax,bx,options,varargin)
        .
        .
        .
```

```
113 fx = funfcn(x,varargin{:});
```

## Example 2 – Constructing a Handle to an Anonymous Function

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation  $x.^2$ :

```
sqr = @(x) x.^2;
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `sqr`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
    25
```

Because `sqr` is a function handle, you can pass it in an argument list to other functions. The code shown here passes the `sqr` anonymous function to the MATLAB `quad` function to compute its integral from zero to one:

```
quad(sqr, 0, 1)
ans =
    0.3333
```

## See Also

`str2func`, `func2str`, `functions`, `isa`

# functions

---

**Purpose** Information about function handle

**Syntax** `S = functions(funhandle)`

**Description** `S = functions(funhandle)` returns, in MATLAB structure `S`, the function name, type, filename, and other information for the function handle stored in the variable `funhandle`.

`functions` does not operate on nonscalar function handles. Passing a nonscalar function handle to `functions` results in an error.

---

**Caution** The `functions` function is provided for querying and debugging purposes. Because its behavior may change in subsequent releases, you should not rely upon it for programming purposes.

---

This table lists the standard fields of the return structure.

Field Name	Field Description
<code>function</code>	Function name
<code>type</code>	Function type (e.g., simple, overloaded)
<code>file</code>	The file to be executed when the function handle is evaluated with a nonoverloaded data type

## Examples

### Example 1

To obtain information on a function handle for the `poly` function, type

```
f = functions(@poly)
f =
    function: 'poly'
           type: 'simple'
           file: '$matlabroot\toolbox\matlab\polyfun\poly.m'
```

(The term `$matlabroot` used in this example stands for the file specification of the directory in which MATLAB software is installed for your system. Your output will display this file specification.)

Access individual fields of the returned structure using dot selection notation:

```
f.type
ans =
    simple
```

## Example 2

The function `get_handles` returns function handles for a subfunction and private function in output arguments `s` and `p` respectively:

```
function [s, p] = get_handles
s = @mysubfun;
p = @myprivatefun;
%
function mysubfun
disp 'Executing subfunction mysubfun'
```

Call `get_handles` to obtain the two function handles, and then pass each to the `functions` function. MATLAB returns information in a structure having the fields `function`, `type`, `file`, and `parentage`. The `file` field contains the file specification for the subfunction or private function:

```
[fsub fprv] = get_handles;

functions(fsub)
ans =
    function: 'mysubfun'
           type: 'scopedfunction'
           file: 'c:\matlab\get_handles.m'
    parentage: {'mysubfun' 'get_handles'}

functions(fprv)
```

# functions

---

```
ans =  
    function: 'myprivatefun'  
    type: 'scopedfunction'  
    file: 'c:\matlab\private\myprivatefun.m'  
    parentage: {'myprivatefun'}
```

## Example 3

In this example, the function `get_handles_nested.m` contains a nested function `nestfun`. This function has a single output which is a function handle to the nested function:

```
function handle = get_handles_nested(A)  
    nestfun(A);  
  
    function y = nestfun(x)  
        y = x + 1;  
    end  
  
    handle = @nestfun;  
end
```

Call this function to get the handle to the nested function. Use this handle as the input to `functions` to return the information shown here. Note that the `function` field of the return structure contains the names of the nested function and the function in which it is nested in the format. Also note that `functions` returns a `workspace` field containing the variables that are in context at the time you call this function by its handle:

```
fh = get_handles_nested(5);  
  
finfo = functions(fh)  
finfo =  
    function: 'get_handles_nested/nestfun'  
    type: 'nested'  
    file: 'c:\matlab\get_handles_nested.m'  
    workspace: [1x1 struct]
```



```
fhinfo.workspace
ans =
    handle: @get_handles_nested/nestfun
    A: 5
```

**See Also** [function\\_handle](#)

# funm

---

**Purpose** Evaluate general matrix function

**Syntax**

```
F = funm(A,fun)
F = funm(A, fun, options)
F=funm(A, fun, options, p1, p2,...)
[F, exitflag] = funm(...)
[F, exitflag, output] = funm(...)
```

**Description** `F = funm(A,fun)` evaluates the user-defined function `fun` at the square matrix argument `A`. `F = fun(x, k)` must accept a vector `x` and an integer `k`, and return a vector `f` of the same size of `x`, where `f(i)` is the `k`th derivative of the function `fun` evaluated at `x(i)`. The function represented by `fun` must have a Taylor series with an infinite radius of convergence, except for `fun = @log`, which is treated as a special case.

You can also use `funm` to evaluate the special functions listed in the following table at the matrix `A`.

Function	Syntax for Evaluating Function at Matrix A
<code>exp</code>	<code>funm(A, @exp)</code>
<code>log</code>	<code>funm(A, @log)</code>
<code>sin</code>	<code>funm(A, @sin)</code>
<code>cos</code>	<code>funm(A, @cos)</code>
<code>sinh</code>	<code>funm(A, @sinh)</code>
<code>cosh</code>	<code>funm(A, @cosh)</code>

For matrix square roots, use `sqrtm(A)` instead. For matrix exponentials, which of `expm(A)` or `funm(A, @exp)` is the more accurate depends on the matrix `A`.

The function represented by `fun` must have a Taylor series with an infinite radius of convergence. The exception is `@log`, which is treated as a special case. “Parametrizing Functions”, in the online MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

$F = \text{funm}(A, \text{fun}, \text{options})$  sets the algorithm's parameters to the values in the structure options.

The following table lists the fields of options.

Field	Description	Values
options.Display	Level of display	'off' (default), 'on', 'verbose'
options.TolBlk	Tolerance for blocking Schur form	Positive scalar. The default is 0.1.
options.TolTay	Termination tolerance for evaluating the Taylor series of diagonal blocks	Positive scalar. The default is eps.
options.MaxTerms	Maximum number of Taylor series terms	Positive integer. The default is 250.
options.MaxSqrt	When computing a logarithm, maximum number of square roots computed in inverse scaling and squaring method.	Positive integer. The default is 100.
options.Ord	Specifies the ordering of the Schur form $T$ .	A vector of length <code>length(A)</code> . <code>options.Ord(i)</code> is the index of the block into which $T(i,i)$ is placed. The default is <code>[]</code> .

$F = \text{funm}(A, \text{fun}, \text{options}, p1, p2, \dots)$  passes extra inputs  $p1, p2, \dots$  to the function.

$[F, \text{exitflag}] = \text{funm}(\dots)$  returns a scalar `exitflag` that describes the exit condition of `funm`. `exitflag` can have the following values:

- 0 — The algorithm was successful.
- 1 — One or more Taylor series evaluations did not converge, or, in the case of a logarithm, too many square roots are needed. However, the computed value of F might still be accurate. This is different from R13 and earlier versions that returned an expensive and often inaccurate error estimate as the second output argument.

[F, exitflag, output] = funm(...) returns a structure output with the following fields:

Field	Description
output.terms	Vector for which output.terms(i) is the number of Taylor series terms used when evaluating the <i>i</i> th block, or, in the case of the logarithm, the number of square roots of matrices of dimension greater than 2.
output.ind	Cell array for which the (i, j) block of the reordered Schur factor T is T(output.ind{i}, output.ind{j}).
output.ord	Ordering of the Schur form, as passed to ordschur
output.T	Reordered Schur form

If the Schur form is diagonal then output = struct('terms',ones(n,1),'ind',{1:n}).

## Examples

### Example 1

The following command computes the matrix sine of the 3-by-3 magic matrix.

```
F=funm(magic(3), @sin)
```

```
F =
```

```

-0.3850    1.0191    0.0162
 0.6179    0.2168   -0.1844
 0.4173   -0.5856    0.8185

```

### Example 2

The statements

```

S = funm(X,@sin);
C = funm(X,@cos);

```

produce the same results to within roundoff error as

```

E = expm(i*X);
C = real(E);
S = imag(E);

```

In either case, the results satisfy  $S^*S + C^*C = I$ , where  $I = \text{eye}(\text{size}(X))$ .

### Example 3

To compute the function  $\exp(x) + \cos(x)$  at  $A$  with one call to `funm`, use

```

F = funm(A,@fun_expcos)

```

where `fun_expcos` is the following M-file function.

```

function f = fun_expcos(x, k)
% Return kth derivative of exp + cos at X.
g = mod(ceil(k/2),2);
if mod(k,2)
    f = exp(x) + sin(x)*(-1)^g;
else
    f = exp(x) + cos(x)*(-1)^g;
end

```

### Algorithm

The algorithm `funm` uses is described in [1].

## See Also

expm, logm, sqrtm, function\_handle (@)

## References

- [1] Davies, P. I. and N. J. Higham, “A Schur-Parlett algorithm for computing matrix functions,” *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.
- [2] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Third Edition, Johns Hopkins University Press, 1996, p. 384.
- [3] Moler, C. B. and C. F. Van Loan, “Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later” *SIAM Review* 20, Vol. 45, Number 1, pp. 1-47, 2003.

**Purpose**

Write binary data to file

**Syntax**

```
count = fwrite(fid, A)
count = fwrite(fid, A, precision)
count = fwrite(fid, A, precision, skip)
count = fwrite(fid, A, precision, skip, machineformat)
```

**Description**

`count = fwrite(fid, A)` writes the elements of matrix `A` to the specified file. The data is written to the file in column order, and a `count` is kept of the number of elements written successfully.

`fid` is an integer file identifier obtained from `fopen`, or 1 for standard output or 2 for standard error.

`count = fwrite(fid, A, precision)` writes the elements of matrix `A` to the specified file, translating MATLAB values to the specified `precision`.

`precision` controls the form and size of the result. See `fread` for a list of allowed precisions. If `precision` is not specified, MATLAB uses the default, which is `'uint8'`. For `'bitN'` or `'ubitN'` precisions, `fwrite` sets all bits in `A` when the value is out of range. If the precision is `'char'` or `'char*1'`, MATLAB writes characters using the encoding scheme associated with the file. See `fopen` for more information.

`count = fwrite(fid, A, precision, skip)` includes an optional `skip` argument that specifies the number of bytes to skip before each `precision` value is written. With the `skip` argument present, `fwrite` skips and writes one value, skips and writes another value, etc., until all of `A` is written. If `precision` is a bit format like `'bitN'` or `'ubitN'`, `skip` is specified in bits. This is useful for inserting data into noncontiguous fields in fixed-length records.

`count = fwrite(fid, A, precision, skip, machineformat)` treats the data written as having a format given by `machineformat`. You can obtain the `machineformat` argument from the output of the `fopen` function. See `fopen` for possible values for `machineformat`.

# fwrite

---

## Remarks

You cannot view or type the contents of the file you are writing with `fwrite` until you close the file with the `fclose` function.

## Examples

### Example 1

This example creates a 100-byte binary file containing the 25 elements of the 5-by-5 magic square, stored as 4-byte integers:

```
fid = fopen('magic5.bin', 'wb');
fwrite(fid, magic(5), 'integer*4')
```

### Example 2

This example takes a string of Unicode characters, `str`, which contains Japanese text, and writes the string into a file using the Shift-JIS character encoding scheme:

```
fid = fopen('japanese_out.txt', 'w', 'n', 'Shift_JIS');
fwrite(fid, str, 'char');
fclose(fid);
```

## See Also

`fclose`, `ferror`, `fopen`, `fprintf`, `fread`, `fscanf`, `fseek`, `ftell`



**Purpose** Write binary data to device

**Syntax**

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A,'mode')
fwrite(obj,A,'precision','mode')
```

**Description** fwrite(obj,A) writes the binary data A to the device connected to the serial port object, obj.

fwrite(obj,A,'precision') writes binary data with precision specified by *precision*.

*precision* controls the number of bits written for each value and the interpretation of those bits as integer, floating-point, or character values. If *precision* is not specified, uchar (an 8-bit unsigned character) is used. The supported values for *precision* are listed below in Remarks.

fwrite(obj,A,'mode') writes binary data with command line access specified by *mode*. If *mode* is sync, A is written synchronously and the command line is blocked. If *mode* is async, A is written asynchronously and the command line is not blocked. If *mode* is not specified, the write operation is synchronous.

fwrite(obj,A,'precision','mode') writes binary data with precision specified by *precision* and command line access specified by *mode*.

**Remarks**

Before you can write data to the device, it must be connected to obj with the fopen function. A connected serial port object has a Status property value of open. An error is returned if you attempt to perform a write operation while obj is not connected to the device.

The ValuesSent property value is increased by the number of values written each time fwrite is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the OutputBufferSize property.

## fwrite (serial)

---

If you use the `help` command to display help for `fwrite`, then you need to supply the pathname shown below.

```
help serial/fwrite
```

`fwrite` will return an error message if you set the `FlowControl` property to `hardware` on a serial object, and a hardware connection is not detected. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check you remote device's status and flow control settings to see if hardware flow control is causing errors in MATLAB.

---

**Note** If you want to check to see if the device is asserting that it is ready to receive data, set the `FlowControl` to `none`. Once you connect to the device check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is off, there is a problem on the remote device side. If `ClearToSend` is on, there is a hardware `FlowControl` device prepared to receive data and you can execute `fwrite`.

---

### Synchronous Versus Asynchronous Write Operations

By default, data is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the `mode` input argument to be `async`. For asynchronous writes:

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The M-file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in [Writing Data](#).

## Rules for Completing a Write Operation with fwrite

A binary write operation using `fwrite` completes when:

- The specified data is written.
- The time specified by the `Timeout` property passes.

---

**Note** The `Terminator` property is not used with binary write operations.

---

## Supported Precisions

The supported values for *precision* are listed below.

Data Type	Precision	Interpretation
Character	<code>uchar</code>	8-bit unsigned character
	<code>schar</code>	8-bit signed character
	<code>char</code>	8-bit signed or unsigned character

# fwrite (serial)

---

<b>Data Type</b>	<b>Precision</b>	<b>Interpretation</b>
Integer	int8	8-bit integer
	int16	16-bit integer
	int32	32-bit integer
	uint8	8-bit unsigned integer
	uint16	16-bit unsigned integer
	uint32	32-bit unsigned integer
	short	16-bit integer
	int	32-bit integer
	long	32- or 64-bit integer
	ushort	16-bit unsigned integer
	uint	32-bit unsigned integer
	ulong	32- or 64-bit unsigned integer
Floating-point	single	32-bit floating point
	float32	32-bit floating point
	float	32-bit floating point
	double	64-bit floating point
	float64	64-bit floating point

## See Also

### Functions

fopen, fprintf

### Properties

BytesToOutput, OutputBufferSize, OutputEmptyFcn, Status, Timeout, TransferStatus, ValuesSent

**Purpose**

Find root of continuous function of one variable

**Syntax**

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)
[x,fval] = fzero(...)
[x,fval,exitflag] = fzero(...)
[x,fval,exitflag,output] = fzero(...)
```

**Description**

`x = fzero(fun,x0)` tries to find a zero of `fun` near `x0`, if `x0` is a scalar. `fun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. The value `x` returned by `fzero` is near a point where `fun` changes sign, or NaN if the search fails. In this case, the search terminates when the search interval is expanded until an Inf, NaN, or complex value is found.

“Parametrizing Functions” in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`. See also “Example 2” on page 2-1410 and “Example 3” on page 2-1410 below.

If `x0` is a vector of length two, `fzero` assumes `x0` is an interval where the sign of `fun(x0(1))` differs from the sign of `fun(x0(2))`. An error occurs if this is not true. Calling `fzero` with such an interval guarantees `fzero` will return a value near a point where `fun` changes sign.

`x = fzero(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fzero` uses these options structure fields:

Display	Level of display. 'off' displays no output; 'iter' displays output at each iteration; 'final' displays just the final output; 'notify' (default) displays output only if the function does not converge.
FunValCheck	Check whether objective function values are valid. 'on' displays an error when the objective function returns a value that is complex or NaN. 'off' (the default) displays no error.
OutputFcn	User-defined function that is called at each iteration. See “Output Function” in the Optimization Toolbox for more information.
PlotFcns	User-defined plot function that is called at each iteration. See “Plot Functions” in the Optimization Toolbox for more information.
TolX	Termination tolerance on x

`[x,fval] = fzero(...)` returns the value of the objective function fun at the solution x.

`[x,fval,exitflag] = fzero(...)` returns a value `exitflag` that describes the exit condition of `fzero`:

- 1 Function converged to a solution x.
- 1 Algorithm was terminated by the output function.
- 3 NaN or Inf function value was encountered during search for an interval containing a sign change.
- 4 Complex function value was encountered during search for an interval containing a sign change.
- 5 `fzero` might have converged to a singular point.
- 6 `fzero` can not detect a change in sign of the function.

`[x,fval,exitflag,output] = fzero(...)` returns a structure output that contains information about the optimization:

---

output.algorithm	Algorithm used
output.funcCount	Number of function evaluations
output.interval	Number of iterations taken to find an interval
output.iterations	Number of zero-finding iterations
output.message	Exit message

---

**Note** For the purposes of this command, zeros are considered to be points where the function actually crosses, not just touches, the  $x$ -axis.

---

## Arguments

`fun` is the function whose zero is to be computed. It accepts a vector `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fzero(@myfun,x0);
```

where `myfun` is an M-file function such as

```
function f = myfun(x)
f = ...           % Compute function value at x
```

or as a function handle for an anonymous function:

```
x = fzero(@(x)sin(x*x),x0);
```

Other arguments are described in the syntax descriptions above.

## Examples

### Example 1

Calculate  $\pi$  by finding the zero of the sine function near 3.

```
x = fzero(@sin,3)
x =
    3.1416
```

## Example 2

To find the zero of cosine between 1 and 2

```
x = fzero(@cos,[1 2])
x =
    1.5708
```

Note that  $\cos(1)$  and  $\cos(2)$  differ in sign.

## Example 3

To find a zero of the function  $f(x) = x^3 - 2x - 5$

write an anonymous function f:

```
f = @(x)x.^3-2*x-5;
```

Then find the zero near 2:

```
z = fzero(f,2)
z =
    2.0946
```

Because this function is a polynomial, the statement `roots([1 0 -2 -5])` finds the same real zero, and a complex conjugate pair of zeros.

```
    2.0946
   -1.0473 + 1.1359i
   -1.0473 - 1.1359i
```

If fun is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function myfun defined by the following M-file function.

```
function f = myfun(x,a)
f = cos(a*x);
```



Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fzero`. To optimize for a specific value of `a`, such as `a = 2`.

- 1 Assign the value to `a`.

```
a = 2; % define parameter first
```

- 2 Call `fzero` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

```
x = fzero(@(x) myfun(x,a),0.1)
```

## Algorithm

The `fzero` command is an M-file. The algorithm, which was originated by T. Dekker, uses a combination of bisection, secant, and inverse quadratic interpolation methods. An Algol 60 version, with some improvements, is given in [1]. A Fortran version, upon which the `fzero` M-file is based, is in [2].

## Limitations

The `fzero` command finds a point where the function changes sign. If the function is *continuous*, this is also a point where the function has a value near zero. If the function is not continuous, `fzero` may return values that are discontinuous points instead of zeros. For example, `fzero(@tan,1)` returns 1.5708, a discontinuous point in `tan`.

Furthermore, the `fzero` command defines a *zero* as a point where the function crosses the  $x$ -axis. Points where the function touches, but does not cross, the  $x$ -axis are not valid zeros. For example,  $y = x.^2$  is a parabola that touches the  $x$ -axis at 0. Because the function never crosses the  $x$ -axis, however, no zero is found. For functions with no valid zeros, `fzero` executes until `Inf`, `NaN`, or a complex value is detected.

## See Also

`roots`, `fminbnd`, `optimset`, `function_handle` (@), “Anonymous Functions”

## References

[1] Brent, R., *Algorithms for Minimization Without Derivatives*, Prentice-Hall, 1973.

[2] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

**Purpose** Test matrices

**Syntax**  
`[A,B,C,...] = gallery(matname,P1,P2,...)`  
`[A,B,C,...] = gallery(matname,P1,P2,...,classname)`  
`gallery(3)`  
`gallery(5)`

**Description** `[A,B,C,...] = gallery(matname,P1,P2,...)` returns the test matrices specified by the quoted string `matname`. The `matname` input is the name of a matrix family selected from the table below. `P1,P2,...` are input parameters required by the individual matrix family. The number of optional parameters `P1,P2,...` used in the calling syntax varies from matrix to matrix. The exact calling syntaxes are detailed in the individual matrix descriptions below.

`[A,B,C,...] = gallery(matname,P1,P2,...,classname)` produces a matrix of class `classname`. The `classname` input is a quoted string that must be either 'single' or 'double'. If `classname` is not specified, then the class of the matrix is determined from those arguments among `P1,P2,...` that do not specify dimensions or select an option. If any of these arguments is of class `single` then the matrix is `single`; otherwise the matrix is `double`.

`gallery(3)` is a badly conditioned 3-by-3 matrix and `gallery(5)` is an interesting eigenvalue problem.

The gallery holds over fifty different test matrix functions useful for testing algorithms and other purposes.

Test Matrices			
binomial	cauchy	chebspec	chebvand
chow	circul	clement	compar
condex	cycol	dorr	dramadah
fiedler	forsythe	frank	gearmat
gcdmat	grcar	hanowa	house

Test Matrices			
invhess	invol	ipjfact	jordbloc
kahan	kms	krylov	lauchli
lehmer	leslie	lesp	lotkin
minij	moler	neumann	orthog
parter	pei	poisson	prolate
randcolu	randcorr	randhess	randjorth
rando	randsvd	redheff	riemann
ris	smoke	toeppd	tridiag
triw	wathen	wilk	

## binomial – Multiple of involutory matrix

$A = \text{gallery}('binomial', n)$  returns an  $n$ -by- $n$  matrix, with integer entries such that  $A^2 = 2^{(n-1)} * \text{eye}(n)$ .

Thus,  $B = A * 2^{-(n-1)/2}$  is involutory, that is,  $B^2 = \text{eye}(n)$ .

## cauchy – Cauchy matrix

$C = \text{gallery}('cauchy', x, y)$  returns an  $n$ -by- $n$  matrix,  $C(i, j) = 1 / (x(i) + y(j))$ . Arguments  $x$  and  $y$  are vectors of length  $n$ . If you pass in scalars for  $x$  and  $y$ , they are interpreted as vectors  $1:x$  and  $1:y$ .

$C = \text{gallery}('cauchy', x)$  returns the same as above with  $y = x$ . That is, the command returns  $C(i, j) = 1 / (x(i) + x(j))$ .

Explicit formulas are known for the inverse and determinant of a Cauchy matrix. The determinant  $\det(C)$  is nonzero if  $x$  and  $y$  both have distinct elements.  $C$  is totally positive if  $0 < x(1) < \dots < x(n)$  and  $0 < y(1) < \dots < y(n)$ .

### **chebspec – Chebyshev spectral differentiation matrix**

`C = gallery('chebspec',n,switch)` returns a Chebyshev spectral differentiation matrix of order  $n$ . Argument `switch` is a variable that determines the character of the output matrix. By default, `switch = 0`.

For `switch = 0` (“no boundary conditions”),  $C$  is nilpotent ( $C^n = 0$ ) and has the null vector `ones(n,1)`. The matrix  $C$  is similar to a Jordan block of size  $n$  with eigenvalue zero.

For `switch = 1`,  $C$  is nonsingular and well-conditioned, and its eigenvalues have negative real parts.

The eigenvector matrix of the Chebyshev spectral differentiation matrix is ill-conditioned.

### **chebvand – Vandermonde-like matrix for the Chebyshev polynomials**

`C = gallery('chebvand',p)` produces the (primal) Chebyshev Vandermonde matrix based on the vector of points `p`, which define where the Chebyshev polynomial is calculated.

`C = gallery('chebvand',m,p)` where  $m$  is scalar, produces a rectangular version of the above, with  $m$  rows.

If `p` is a vector, then  $C(i,j) = T_{i-1}(p(j))$  where  $T_{i-1}$  is the Chebyshev polynomial of degree  $i-1$ . If `p` is a scalar, then `p` equally spaced points on the interval  $[0,1]$  are used to calculate  $C$ .

### **chow – Singular Toeplitz lower Hessenberg matrix**

`A = gallery('chow',n,alpha,delta)` returns  $A$  such that

$A = H(\alpha) + \delta \cdot \text{eye}(n)$ , where  $H_{i,j}(\alpha) = \alpha^{(i-j+1)}$  and argument  $n$  is the order of the Chow matrix. Default value for scalars `alpha` and `delta` are 1 and 0, respectively.

$H(\alpha)$  has  $p = \text{floor}(n/2)$  eigenvalues that are equal to zero. The rest of the eigenvalues are equal to  $4 \cdot \alpha \cdot \cos(k \cdot \pi / (n+2))^2$ ,  $k=1:n-p$ .

## **circul – Circulant matrix**

`C = gallery('circul',v)` returns the circulant matrix whose first row is the vector `v`.

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. It is a special Toeplitz matrix in which the diagonals “wrap around.”

If `v` is a scalar, then `C = gallery('circul',1:v)`.

The eigensystem of `C` ( $n$ -by- $n$ ) is known explicitly: If  $t$  is an  $n$ th root of unity, then the inner product of `v` and  $w = [1 \ t \ t^2 \ \dots \ t^{(n-1)}]$  is an eigenvalue of `C` and `w(n:-1:1)` is an eigenvector.

## **clement – Tridiagonal matrix with zero diagonal entries**

`A = gallery('clement',n,sym)` returns an  $n$ -by- $n$  tridiagonal matrix with zeros on its main diagonal and known eigenvalues. It is singular if order  $n$  is odd. About 64 percent of the entries of the inverse are zero. The eigenvalues include plus and minus the numbers  $n-1$ ,  $n-3$ ,  $n-5$ ,  $\dots$ , as well as (for odd  $n$ ) a final eigenvalue of 1 or 0.

Argument `sym` determines whether the Clement matrix is symmetric. For `sym = 0` (the default) the matrix is nonsymmetric, while for `sym = 1`, it is symmetric.

## **compar – Comparison matrices**

`A = gallery('compar',A,1)` returns `A` with each diagonal element replaced by its absolute value, and each off-diagonal element replaced by minus the absolute value of the largest element in absolute value in its row. However, if `A` is triangular `compar(A,1)` is too.

`gallery('compar',A)` is `diag(B) - tril(B,-1) - triu(B,1)`, where `B = abs(A)`. `compar(A)` is often denoted by  $M(A)$  in the literature.

`gallery('compar',A,0)` is the same as `gallery('compar',A)`.

**condex – Counter-examples to matrix condition number estimators**

`A = gallery('condex',n,k,theta)` returns a “counter-example” matrix to a condition estimator. It has order `n` and scalar parameter `theta` (default 100).

The matrix, its natural size, and the estimator to which it applies are specified by `k`:

<code>k = 1</code>	4-by-4	LINPACK
<code>k = 2</code>	3-by-3	LINPACK
<code>k = 3</code>	arbitrary	LINPACK (rcond) (independent of <code>theta</code> )
<code>k = 4</code>	<code>n &gt;= 4</code>	LAPACK (RCOND) (default). It is the inverse of this matrix that is a counter-example.

If `n` is not equal to the natural size of the matrix, then the matrix is padded out with an identity matrix to order `n`.

**cycol – Matrix whose columns repeat cyclically**

`A = gallery('cycol',[m n],k)` returns an `m`-by-`n` matrix with cyclically repeating columns, where one “cycle” consists of `randn(m,k)`. Thus, the rank of matrix `A` cannot exceed `k`, and `k` must be a scalar.

Argument `k` defaults to `round(n/4)`, and need not evenly divide `n`.

`A = gallery('cycol',n,k)`, where `n` is a scalar, is the same as `gallery('cycol',[n n],k)`.

**dorr – Diagonally dominant, ill-conditioned, tridiagonal matrix**

`[c,d,e] = gallery('dorr',n,theta)` returns the vectors defining an `n`-by-`n`, row diagonally dominant, tridiagonal matrix that is ill-conditioned for small nonnegative values of `theta`. The default value of `theta` is 0.01. The Dorr matrix itself is the same as `gallery('tridiag',c,d,e)`.

`A = gallery('dorr',n,theta)` returns the matrix itself, rather than the defining vectors.

## **dramadah – Matrix of zeros and ones whose inverse has large integer entries**

`A = gallery('dramadah',n,k)` returns an  $n$ -by- $n$  matrix of 0's and 1's for which  $\mu(A) = \text{norm}(\text{inv}(A), 'fro')$  is relatively large, although not necessarily maximal. An anti-Hadamard matrix  $A$  is a matrix with elements 0 or 1 for which  $\mu(A)$  is maximal.

$n$  and  $k$  must both be scalars. Argument  $k$  determines the character of the output matrix:

- $k = 1$       Default.  $A$  is Toeplitz, with  $\text{abs}(\det(A)) = 1$ , and  $\mu(A) > c(1.75)^n$ , where  $c$  is a constant. The inverse of  $A$  has integer entries.
- $k = 2$        $A$  is upper triangular and Toeplitz. The inverse of  $A$  has integer entries.
- $k = 3$        $A$  has maximal determinant among lower Hessenberg (0,1) matrices.  $\det(A) =$  the  $n$ th Fibonacci number.  $A$  is Toeplitz. The eigenvalues have an interesting distribution in the complex plane.

## **fiedler – Symmetric matrix**

`A = gallery('fiedler',c)`, where  $c$  is a length  $n$  vector, returns the  $n$ -by- $n$  symmetric matrix with elements  $\text{abs}(n(i) - n(j))$ . For scalar  $c$ , `A = gallery('fiedler',1:c)`.

Matrix  $A$  has a dominant positive eigenvalue and all the other eigenvalues are negative.

Explicit formulas for  $\text{inv}(A)$  and  $\det(A)$  are given in [Todd, J., *Basic Numerical Mathematics*, Vol. 2: Numerical Algebra, Birkhauser, Basel, and Academic Press, New York, 1977, p. 159] and attributed to Fiedler. These indicate that  $\text{inv}(A)$  is tridiagonal except for nonzero  $(1, n)$  and  $(n, 1)$  elements.



**forsythe – Perturbed Jordan block**

`A = gallery('forsythe', n, alpha, lambda)` returns the  $n$ -by- $n$  matrix equal to the Jordan block with eigenvalue  $\lambda$ , excepting that  $A(n, 1) = \alpha$ . The default values of scalars  $\alpha$  and  $\lambda$  are `sqrt(eps)` and `0`, respectively.

The characteristic polynomial of  $A$  is given by:

$$\det(A - t \cdot I) = (\lambda - t)^N - \alpha \cdot (-1)^n.$$

**frank – Matrix with ill-conditioned eigenvalues**

`F = gallery('frank', n, k)` returns the Frank matrix of order  $n$ . It is upper Hessenberg with determinant 1. If  $k = 1$ , the elements are reflected about the anti-diagonal  $(1, n) \text{ --- } (n, 1)$ . The eigenvalues of  $F$  may be obtained in terms of the zeros of the Hermite polynomials. They are positive and occur in reciprocal pairs; thus if  $n$  is odd, 1 is an eigenvalue.  $F$  has  $\text{floor}(n/2)$  ill-conditioned eigenvalues — the smaller ones.

**gcdmat – Greatest common divisor matrix**

`A = gallery('gcdmat', n)` returns the  $n$ -by- $n$  matrix with  $(i, j)$  entry  $\text{gcd}(i, j)$ . Matrix  $A$  is symmetric positive definite, and  $A.^r$  is symmetric positive semidefinite for all nonnegative  $r$ .

**gearmat – Gear matrix**

`A = gallery('gearmat', n, i, j)` returns the  $n$ -by- $n$  matrix with ones on the sub- and super-diagonals,  $\text{sign}(i)$  in the  $(1, \text{abs}(i))$  position,  $\text{sign}(j)$  in the  $(n, n+1-\text{abs}(j))$  position, and zeros everywhere else. Arguments  $i$  and  $j$  default to  $n$  and  $-n$ , respectively.

Matrix  $A$  is singular, can have double and triple eigenvalues, and can be defective.

All eigenvalues are of the form  $2 \cdot \cos(a)$  and the eigenvectors are of the form  $[\sin(w+a), \sin(w+2a), \dots, \sin(w+n \cdot a)]$ , where  $a$  and  $w$  are given in Gear, C. W., "A Simple Set of Test Matrices for Eigenvalue Programs," *Math. Comp.*, Vol. 23 (1969), pp. 119-125.

## **grcar – Toeplitz matrix with sensitive eigenvalues**

`A = gallery('grcar',n,k)` returns an  $n$ -by- $n$  Toeplitz matrix with  $-1$ s on the subdiagonal,  $1$ s on the diagonal, and  $k$  superdiagonals of  $1$ s. The default is  $k = 3$ . The eigenvalues are sensitive.

## **hanowa – Matrix whose eigenvalues lie on a vertical line in the complex plane**

`A = gallery('hanowa',n,d)` returns an  $n$ -by- $n$  block 2-by-2 matrix of the form:

$$\begin{bmatrix} d*\text{eye}(m) & -\text{diag}(1:m) \\ \text{diag}(1:m) & d*\text{eye}(m) \end{bmatrix}$$

Argument  $n$  is an even integer  $n=2*m$ . Matrix  $A$  has complex eigenvalues of the form  $d \pm k*i$ , for  $1 \leq k \leq m$ . The default value of  $d$  is  $-1$ .

## **house – Householder matrix**

`[v,beta,s] = gallery('house',x,k)` takes  $x$ , an  $n$ -element column vector, and returns  $V$  and  $\beta$  such that  $H*x = s*e_1$ . In this expression,  $e_1$  is the first column of  $\text{eye}(n)$ ,  $\text{abs}(s) = \text{norm}(x)$ , and  $H = \text{eye}(n) - \beta*v*v'$  is a Householder matrix.

$k$  determines the sign of  $s$ :

$$\begin{array}{ll} k = 0 & \text{sign}(s) = -\text{sign}(x(1)) \text{ (default)} \\ k = 1 & \text{sign}(s) = \text{sign}(x(1)) \\ k = 2 & \text{sign}(s) = 1 \text{ (x must be real)} \end{array}$$

If  $x$  is complex, then  $\text{sign}(x) = x./\text{abs}(x)$  when  $x$  is nonzero.

If  $x = 0$ , or if  $x = \alpha*e_1$  ( $\alpha \geq 0$ ) and either  $k = 1$  or  $k = 2$ , then  $V = 0$ ,  $\beta = 1$ , and  $s = x(1)$ . In this case,  $H$  is the identity matrix, which is not strictly a Householder matrix.

---

`[v, beta] = gallery('house',x)` takes  $x$ , a scalar or  $n$ -element column vector, and returns  $v$  and  $\beta$  such that  $\text{eye}(n,n) -$

$\beta v v'$  is a Householder matrix. A Householder matrix  $H$  satisfies the relationship

$$Hx = -\text{sign}(x(1)) \cdot \text{norm}(x) \cdot e_1$$

where  $e_1$  is the first column of  $\text{eye}(n, n)$ . Note that if  $x$  is complex, then  $\text{sign}(x) = \exp(i \cdot \text{arg}(x))$  (which equals  $x ./ \text{abs}(x)$  when  $x$  is nonzero).

If  $x = 0$ , then  $v = 0$  and  $\beta = 1$ .

### **invhess – Inverse of an upper Hessenberg matrix**

$A = \text{gallery}('invhess', x, y)$ , where  $x$  is a length  $n$  vector and  $y$  is a length  $n-1$  vector, returns the matrix whose lower triangle agrees with that of  $\text{ones}(n, 1) \cdot x'$  and whose strict upper triangle agrees with that of  $[1 \ y] \cdot \text{ones}(1, n)$ .

The matrix is nonsingular if  $x(1) \neq 0$  and  $x(i+1) \neq y(i)$  for all  $i$ , and its inverse is an upper Hessenberg matrix. Argument  $y$  defaults to  $-x(1:n-1)$ .

If  $x$  is a scalar,  $\text{invhess}(x)$  is the same as  $\text{invhess}(1:x)$ .

### **invol – Involutory matrix**

$A = \text{gallery}('invol', n)$  returns an  $n$ -by- $n$  involutory ( $A \cdot A = \text{eye}(n)$ ) and ill-conditioned matrix. It is a diagonally scaled version of  $\text{hilb}(n)$ .

$B = (\text{eye}(n) - A) / 2$  and  $B = (\text{eye}(n) + A) / 2$  are idempotent ( $B \cdot B = B$ ).

### **ipjfact – Hankel matrix with factorial elements**

$[A, d] = \text{gallery}('ipjfact', n, k)$  returns  $A$ , an  $n$ -by- $n$  Hankel matrix, and  $d$ , the determinant of  $A$ , which is known explicitly. If  $k = 0$  (the default), then the elements of  $A$  are  $A(i, j) = (i+j)!$ . If  $k = 1$ , then the elements of  $A$  are  $A(i, j) = 1 / (i+j)$ .

Note that the inverse of  $A$  is also known explicitly.

### **jordbloc – Jordan block**

$A = \text{gallery}('jordbloc', n, \lambda)$  returns the  $n$ -by- $n$  Jordan block with eigenvalue  $\lambda$ . The default value for  $\lambda$  is 1.

## **kahan – Upper trapezoidal matrix**

`A = gallery('kahan',n,theta,pert)` returns an upper trapezoidal matrix that has interesting properties regarding estimation of condition and rank.

If `n` is a two-element vector, then `A` is `n(1)`-by-`n(2)`; otherwise, `A` is `n`-by-`n`. The useful range of `theta` is  $0 < \theta < \pi$ , with a default value of 1.2.

To ensure that the QR factorization with column pivoting does not interchange columns in the presence of rounding errors, the diagonal is perturbed by `pert*eps*diag([n:-1:1])`. The default `pert` is 25, which ensures no interchanges for `gallery('kahan',n)` up to at least `n = 90` in IEEE arithmetic.

## **kms – Kac-Murdock-Szego Toeplitz matrix**

`A = gallery('kms',n,rho)` returns the `n`-by-`n` Kac-Murdock-Szego Toeplitz matrix such that  $A(i,j) = \rho^{(\text{abs}(i-j))}$ , for real `rho`.

For complex `rho`, the same formula holds except that elements below the diagonal are conjugated. `rho` defaults to 0.5.

The KMS matrix `A` has these properties:

- An LDL' factorization with `L = inv(gallery('tril',n,-rho,1))'`, and  $D(i,i) = (1 - \text{abs}(\rho)^2)^{\text{eye}(n)}$ , except  $D(1,1) = 1$ .
- Positive definite if and only if  $0 < \text{abs}(\rho) < 1$ .
- The inverse `inv(A)` is tridiagonal.

## **krylov – Krylov matrix**

`B = gallery('krylov',A,x,j)` returns the Krylov matrix

$[x, Ax, A^2x, \dots, A^{(j-1)}x]$

where `A` is an `n`-by-`n` matrix and `x` is a length `n` vector. The defaults are `x = ones(n,1)`, and `j = n`.

`B = gallery('krylov',n)` is the same as `gallery('krylov',(randn(n)))`.

### **lauchli – Rectangular matrix**

`A = gallery('lauchli',n,mu)` returns the  $(n+1)$ -by- $n$  matrix  
`[ones(1,n); mu*eye(n)]`

The Lauchli matrix is a well-known example in least squares and other problems that indicates the dangers of forming  $A^*A$ . Argument `mu` defaults to `sqrt(eps)`.

### **lehmer – Symmetric positive definite matrix**

`A = gallery('lehmer',n)` returns the symmetric positive definite  $n$ -by- $n$  matrix such that  $A(i,j) = i/j$  for  $j \geq i$ .

The Lehmer matrix  $A$  has these properties:

- $A$  is totally nonnegative.
- The inverse  $\text{inv}(A)$  is tridiagonal and explicitly known.
- The order  $n \leq \text{cond}(A) \leq 4*n*n$ .

### **leslie –**

`L = gallery('leslie',a,b)` is the  $n$ -by- $n$  matrix from the Leslie population model with average birth numbers `a(1:n)` and survival rates `b(1:n-1)`. It is zero, apart from the first row (which contains the `a(i)`) and the first subdiagonal (which contains the `b(i)`). For a valid model, the `a(i)` are nonnegative and the `b(i)` are positive and bounded by 1, i.e.,  $0 < b(i) \leq 1$ .

`L = gallery('leslie',n)` generates the Leslie matrix with `a = ones(n,1)`, `b = ones(n-1,1)`.

### **lesp – Tridiagonal matrix with real, sensitive eigenvalues**

`A = gallery('lesp',n)` returns an  $n$ -by- $n$  matrix whose eigenvalues are real and smoothly distributed in the interval approximately `[-2*N-3.5, -4.5]`.

The sensitivities of the eigenvalues increase exponentially as the eigenvalues grow more negative. The matrix is similar to the symmetric tridiagonal matrix with the same diagonal entries and with off-diagonal entries 1, via a similarity transformation with  $D = \text{diag}(1!, 2!, \dots, n!)$ .

## **lotkin – Lotkin matrix**

`A = gallery('lotkin', n)` returns the Hilbert matrix with its first row altered to all ones. The Lotkin matrix  $A$  is nonsymmetric, ill-conditioned, and has many negative eigenvalues of small magnitude. Its inverse has integer entries and is known explicitly.

## **minij – Symmetric positive definite matrix**

`A = gallery('minij', n)` returns the  $n$ -by- $n$  symmetric positive definite matrix with  $A(i, j) = \min(i, j)$ .

The `minij` matrix has these properties:

- The inverse `inv(A)` is tridiagonal and equal to  $-1$  times the second difference matrix, except its  $(n, n)$  element is  $1$ .
- Givens' matrix, `2*A-ones(size(A))`, has tridiagonal inverse and eigenvalues  $0.5 \cdot \sec((2 \cdot r - 1) \cdot \pi / (4 \cdot n))^2$ , where  $r = 1:n$ .
- `(n+1)*ones(size(A))-A` has elements that are  $\max(i, j)$  and a tridiagonal inverse.

## **moler – Symmetric positive definite matrix**

`A = gallery('moler', n, alpha)` returns the symmetric positive definite  $n$ -by- $n$  matrix  $U^*U$ , where `U = gallery('triw', n, alpha)`.

For the default `alpha = -1`,  $A(i, j) = \min(i, j) - 2$ , and  $A(i, i) = i$ . One of the eigenvalues of  $A$  is small.

## **neumann – Singular matrix from the discrete Neumann problem (sparse)**

`C = gallery('neumann', n)` returns the sparse  $n$ -by- $n$  singular, row diagonally dominant matrix resulting from discretizing the Neumann problem with the usual five-point operator on a regular mesh.

Argument  $n$  is a perfect square integer  $n = m^2$  or a two-element vector.  $C$  is sparse and has a one-dimensional null space with null vector `ones(n,1)`.

### orthog – Orthogonal and nearly orthogonal matrices

`Q = gallery('orthog', n, k)` returns the  $k$ th type of matrix of order  $n$ , where  $k > 0$  selects exactly orthogonal matrices, and  $k < 0$  selects diagonal scalings of orthogonal matrices. Available types are:

- $k = 1$        $Q(i, j) = \sqrt{2/(n+1)} * \sin(i*j*\pi/(n+1))$   
 Symmetric eigenvector matrix for second difference matrix. This is the default.
- $k = 2$        $Q(i, j) = 2/(\sqrt{2*n+1}) * \sin(2*i*j*\pi/(2*n+1))$   
 Symmetric.
- $k = 3$        $Q(r, s) = \exp(2*\pi*i*(r-1)*(s-1)/n) / \sqrt{n}$   
 Unitary, the Fourier matrix.  $Q^4$  is the identity. This is essentially the same matrix as `fft(eye(n))/sqrt(n)`!
- $k = 4$       Helmert matrix: a permutation of a lower Hessenberg matrix, whose first row is `ones(1:n)/sqrt(n)`.
- $k = 5$        $Q(i, j) = \sin(2*\pi*(i-1)*(j-1)/n) + \cos(2*\pi*(i-1)*(j-1)/n)$   
 Symmetric matrix arising in the Hartley transform.
- $k = 6$        $Q(i, j) = \sqrt{2/n}*\cos((i-1/2)*(j-1/2)*\pi/n)$   
 Symmetric matrix arising as a discrete cosine transform.

$k = -1$      $Q(i, j) = \cos((i-1)*(j-1)*\pi/(n-1))$   
Chebyshev Vandermonde-like matrix, based on extrema of  $T(n-1)$ .

$k = -2$      $Q(i, j) = \cos((i-1)*(j-1/2)*\pi/n)$   
Chebyshev Vandermonde-like matrix, based on zeros of  $T(n)$ .

## **parter – Toeplitz matrix with singular values near $\pi$**

$C = \text{gallery}('parter', n)$  returns the matrix  $C$  such that  $C(i, j) = 1/(i-j+0.5)$ .

$C$  is a Cauchy matrix and a Toeplitz matrix. Most of the singular values of  $C$  are very close to  $\pi$ .

## **pei – Pei matrix**

$A = \text{gallery}('pei', n, \alpha)$ , where  $\alpha$  is a scalar, returns the symmetric matrix  $\alpha*\text{eye}(n) + \text{ones}(n)$ . The default for  $\alpha$  is 1. The matrix is singular for  $\alpha$  equal to either 0 or  $-n$ .

## **poisson – Block tridiagonal matrix from Poisson’s equation (sparse)**

$A = \text{gallery}('poisson', n)$  returns the block tridiagonal (sparse) matrix of order  $n^2$  resulting from discretizing Poisson’s equation with the 5-point operator on an  $n$ -by- $n$  mesh.

## **prolate – Symmetric, ill-conditioned Toeplitz matrix**

$A = \text{gallery}('prolate', n, w)$  returns the  $n$ -by- $n$  prolate matrix with parameter  $w$ . It is a symmetric Toeplitz matrix.

If  $0 < w < 0.5$  then  $A$  is positive definite

- The eigenvalues of  $A$  are distinct, lie in  $(0, 1)$ , and tend to cluster around 0 and 1.
- The default value of  $w$  is 0.25.



### **randcolu – Random matrix with normalized cols and specified singular values**

`A = gallery('randcolu',n)` is a random  $n$ -by- $n$  matrix with columns of unit 2-norm, with random singular values whose squares are from a uniform distribution.

`A'*A` is a correlation matrix of the form produced by `gallery('randcorr',n)`.

`gallery('randcolu',x)` where  $x$  is an  $n$ -vector ( $n > 1$ ), produces a random  $n$ -by- $n$  matrix having singular values given by the vector  $x$ . The vector  $x$  must have nonnegative elements whose sum of squares is  $n$ .

`gallery('randcolu',x,m)` where  $m \geq n$ , produces an  $m$ -by- $n$  matrix.

`gallery('randcolu',x,m,k)` provides a further option:

- |                    |  |
|--------------------|--|
| <code>k = 0</code> | <code>diag(x)</code> is initially subjected to a random two-sided orthogonal transformation, and then a sequence of Givens rotations is applied (default). |
| <code>k = 1</code> | The initial transformation is omitted. This is much faster, but the resulting matrix may have zero entries.  |

For more information, see:

[1] Davies, P. I. and N. J. Higham, "Numerically Stable Generation of Correlation Matrices and Their Factors," *BIT*, Vol. 40, 2000, pp. 640-651.

### **randcorr – Random correlation matrix with specified eigenvalues**

`gallery('randcorr',n)` is a random  $n$ -by- $n$  correlation matrix with random eigenvalues from a uniform distribution. A correlation matrix is a symmetric positive semidefinite matrix with 1s on the diagonal (see `corrcoef`).

`gallery('randcorr', x)` produces a random correlation matrix having eigenvalues given by the vector  $x$ , where  $\text{length}(x) > 1$ . The vector  $x$  must have nonnegative elements summing to  $\text{length}(x)$ .

`gallery('randcorr', x, k)` provides a further option:

- |         |   |
|---------|---|
| $k = 0$ | The diagonal matrix of eigenvalues is initially subjected to a random orthogonal similarity transformation, and then a sequence of Givens rotations is applied (default). |
| $k = 1$ | The initial transformation is omitted. This is much faster, but the resulting matrix may have some zero entries.  |

For more information, see:

[1] Bendel, R. B. and M. R. Mickey, "Population Correlation Matrices for Sampling Experiments," *Commun. Statist. Simulation Comput.*, B7, 1978, pp. 163-182.

[2] Davies, P. I. and N. J. Higham, "Numerically Stable Generation of Correlation Matrices and Their Factors," *BIT*, Vol. 40, 2000, pp. 640-651.

## **randhess – Random, orthogonal upper Hessenberg matrix**

$H = \text{gallery}('randhess', n)$  returns an  $n$ -by- $n$  real, random, orthogonal upper Hessenberg matrix.

$H = \text{gallery}('randhess', x)$  if  $x$  is an arbitrary, real, length  $n$  vector with  $n > 1$ , constructs  $H$  nonrandomly using the elements of  $x$  as parameters.

Matrix  $H$  is constructed via a product of  $n-1$  Givens rotations.

## **randjorth – Random J-orthogonal matrix**

$A = \text{gallery}('randjorth', n)$ , for a positive integer  $n$ , produces a random  $n$ -by- $n$  J-orthogonal matrix  $A$ , where

- $J = \text{blkdiag}(\text{eye}(\text{ceil}(n/2)), -\text{eye}(\text{floor}(n/2)))$
- $\text{cond}(A) = \text{sqrt}(1/\text{eps})$

J-orthogonality means that  $A^*J^*A = J$ . Such matrices are sometimes called *hyperbolic*.

$A = \text{gallery}(\text{'randjorth'}, n, m)$ , for positive integers  $n$  and  $m$ , produces a random  $(n+m)$ -by- $(n+m)$  J-orthogonal matrix  $A$ , where

- $J = \text{blkdiag}(\text{eye}(n), -\text{eye}(m))$
- $\text{cond}(A) = \text{sqrt}(1/\text{eps})$

$A = \text{gallery}(\text{'randjorth'}, n, m, c, \text{symm}, \text{method})$

uses the following optional input arguments:

- $c$  — Specifies  $\text{cond}(A)$  to be the scalar  $c$ .
- $\text{symm}$  — Enforces symmetry if the scalar  $\text{symm}$  is nonzero.
- $\text{method}$  — calls `qr` to perform the underlying orthogonal transformations if the scalar  $\text{method}$  is nonzero. A call to `qr` is much faster than the default method for large dimensions

### **rando — Random matrix composed of elements -1, 0 or 1**

$A = \text{gallery}(\text{'rando'}, n, k)$  returns a random  $n$ -by- $n$  matrix with elements from one of the following discrete distributions:

- |         |  |
|---------|--|
| $k = 1$ | $A(i, j) = 0$ or $1$ with equal probability (default). |
| $k = 2$ | $A(i, j) = -1$ or $1$ with equal probability.          |
| $k = 3$ | $A(i, j) = -1, 0$ or $1$ with equal probability.       |

Argument  $n$  may be a two-element vector, in which case the matrix is  $n(1)$ -by- $n(2)$ .

## **randsvd – Random matrix with preassigned singular values**

`A = gallery('randsvd', n, kappa, mode, k1, ku)` returns a banded (multidiagonal) random matrix of order  $n$  with  $\text{cond}(A) = \text{kappa}$  and singular values from the distribution mode. If  $n$  is a two-element vector,  $A$  is  $n(1)$ -by- $n(2)$ .

Arguments  $k1$  and  $ku$  specify the number of lower and upper off-diagonals, respectively, in  $A$ . If they are omitted, a full matrix is produced. If only  $k1$  is present,  $ku$  defaults to  $k1$ .

Distribution mode can be:

- 1 One large singular value.
- 2 One small singular value.
- 3 Geometrically distributed singular values (default).
- 4 Arithmetically distributed singular values.
- 5 Random singular values with uniformly distributed logarithm.
- < 0 If mode is -1, -2, -3, -4, or -5, then `randsvd` treats mode as  $\text{abs}(\text{mode})$ , except that in the original matrix of singular values the order of the diagonal entries is reversed: small to large instead of large to small.

Condition number  $\text{kappa}$  defaults to  $\text{sqrt}(1/\text{eps})$ . In the special case where  $\text{kappa} < 0$ ,  $A$  is a random, full, symmetric, positive definite matrix with  $\text{cond}(A) = -\text{kappa}$  and eigenvalues distributed according to mode. Arguments  $k1$  and  $ku$ , if present, are ignored.

`A = gallery('randsvd', n, kappa, mode, k1, ku, method)` specifies how the computations are carried out. `method = 0` is the default, while `method = 1` uses an alternative method that is much faster for large dimensions, even though it uses more flops.

## **redheff – Redheffer's matrix of 1s and 0s**

`A = gallery('redheff', n)` returns an  $n$ -by- $n$  matrix of 0's and 1's defined by  $A(i, j) = 1$ , if  $j = 1$  or if  $i$  divides  $j$ , and  $A(i, j) = 0$  otherwise.

The Redheffer matrix has these properties:

- $(n - \text{floor}(\log_2(n)) - 1)$  eigenvalues equal to 1
- A real eigenvalue (the spectral radius) approximately  $\sqrt{n}$
- A negative eigenvalue approximately  $-\sqrt{n}$
- The remaining eigenvalues are provably “small.”
- The Riemann hypothesis is true if and only if  $\det(A) = O(n^{\frac{1}{2} + \epsilon})$  for every  $\epsilon > 0$ .

Barrett and Jarvis conjecture that “the small eigenvalues all lie inside the unit circle  $\text{abs}(Z) = 1$ ,” and a proof of this conjecture, together with a proof that some eigenvalue tends to zero as  $n$  tends to infinity, would yield a new proof of the prime number theorem.

**riemann – Matrix associated with the Riemann hypothesis**

`A = gallery('riemann', n)` returns an  $n$ -by- $n$  matrix for which the Riemann hypothesis is true if and only if

$$\det(A) = O(n! n^{-\frac{1}{2} + \epsilon})$$

for every  $\epsilon > 0$ .

The Riemann matrix is defined by:

$$A = B(2:n+1, 2:n+1)$$

where  $B(i, j) = i - 1$  if  $i$  divides  $j$ , and  $B(i, j) = -1$  otherwise.

The Riemann matrix has these properties:

- Each eigenvalue  $e(i)$  satisfies  $\text{abs}(e(i)) \leq m - 1/m$ , where  $m = n + 1$ .
- $i \leq e(i) \leq i + 1$  with at most  $m - \sqrt{m}$  exceptions.
- All integers in the interval  $(m/3, m/2]$  are eigenvalues.

## **ris** – Symmetric Hankel matrix

`A = gallery('ris', n)` returns a symmetric  $n$ -by- $n$  Hankel matrix with elements

$$A(i, j) = 0.5 / (n - i - j + 1.5)$$

The eigenvalues of  $A$  cluster around  $\pi/2$  and  $-\pi/2$ . This matrix was invented by F.N. Ris.

## **smoke** – Complex matrix with a ‘smoke ring’ pseudospectrum

`A = gallery('smoke', n)` returns an  $n$ -by- $n$  matrix with 1's on the superdiagonal, 1 in the  $(n, 1)$  position, and powers of roots of unity along the diagonal.

`A = gallery('smoke', n, 1)` returns the same except that element  $A(n, 1)$  is zero.

The eigenvalues of `gallery('smoke', n, 1)` are the  $n$ th roots of unity; those of `gallery('smoke', n)` are the  $n$ th roots of unity times  $2^{(1/n)}$ .

## **toeppd** – Symmetric positive definite Toeplitz matrix

`A = gallery('toeppd', n, m, w, theta)` returns an  $n$ -by- $n$  symmetric, positive semi-definite (SPD) Toeplitz matrix composed of the sum of  $m$  rank 2 (or, for certain  $\theta$ , rank 1) SPD Toeplitz matrices. Specifically,

$$T = w(1)*T(\theta(1)) + \dots + w(m)*T(\theta(m))$$

where  $T(\theta(k))$  has  $(i, j)$  element  $\cos(2*\pi*\theta(k)*(i-j))$ .

By default:  $m = n$ ,  $w = \text{rand}(m, 1)$ , and  $\theta = \text{rand}(m, 1)$ .

## **toeppen** – Pentadiagonal Toeplitz matrix (sparse)

`P = gallery('toeppen', n, a, b, c, d, e)` returns the  $n$ -by- $n$  sparse, pentadiagonal Toeplitz matrix with the diagonals:  $P(3, 1) = a$ ,  $P(2, 1) = b$ ,  $P(1, 1) = c$ ,  $P(1, 2) = d$ , and  $P(1, 3) = e$ , where  $a$ ,  $b$ ,  $c$ ,  $d$ , and  $e$  are scalars.

By default,  $(a,b,c,d,e) = (1, -10, 0, 10, 1)$ , yielding a matrix of Rutishauser. This matrix has eigenvalues lying approximately on the line segment  $2\cos(2*t) + 20*i*\sin(t)$ .

### **tridiag – Tridiagonal matrix (sparse)**

`A = gallery('tridiag',c,d,e)` returns the tridiagonal matrix with subdiagonal `c`, diagonal `d`, and superdiagonal `e`. Vectors `c` and `e` must have `length(d)-1`.

`A = gallery('tridiag',n,c,d,e)`, where `c`, `d`, and `e` are all scalars, yields the Toeplitz tridiagonal matrix of order `n` with subdiagonal elements `c`, diagonal elements `d`, and superdiagonal elements `e`. This matrix has eigenvalues

$$d + 2*\sqrt{c*e}*\cos(k*pi/(n+1))$$

where `k = 1:n`. (see [1].)

`A = gallery('tridiag',n)` is the same as `A = gallery('tridiag',n,-1,2,-1)`, which is a symmetric positive definite M-matrix (the negative of the second difference matrix).

### **triw – Upper triangular matrix discussed by Wilkinson and others**

`A = gallery('triw',n,alpha,k)` returns the upper triangular matrix with ones on the diagonal and alphas on the first `k >= 0` superdiagonals.

Order `n` may be a 2-element vector, in which case the matrix is `n(1)`-by-`n(2)` and upper trapezoidal.

Ostrowski [“On the Spectrum of a One-parametric Family of Matrices,” *J. Reine Angew. Math.*, 1954] shows that

$$\text{cond}(\text{gallery}('triw',n,2)) = \cot(\pi/(4*n))^2,$$

and, for large `abs(alpha)`, `cond(gallery('triw',n,alpha))` is approximately `abs(alpha)^n*sin(pi/(4*n-2))`.

Adding  $-2^{(2-n)}$  to the  $(n,1)$  element makes `triw(n)` singular, as does adding  $-2^{(1-n)}$  to all the elements in the first column.

## **wathen – Finite element matrix (sparse, random entries)**

`A = gallery('wathen', nx, ny)` returns a sparse, random,  $n$ -by- $n$  finite element matrix where  $n = 3 \cdot nx \cdot ny + 2 \cdot nx + 2 \cdot ny + 1$ .

Matrix  $A$  is precisely the “consistent mass matrix” for a regular  $n_x$ -by- $n_y$  grid of 8-node (serendipity) elements in two dimensions.  $A$  is symmetric, positive definite for any (positive) values of the “density,”  $\rho(n_x, n_y)$ , which is chosen randomly in this routine.

`A = gallery('wathen', nx, ny, 1)` returns a diagonally scaled matrix such that

$$0.25 \leq \text{eig}(\text{inv}(D) \cdot A) \leq 4.5$$

where  $D = \text{diag}(\text{diag}(A))$  for any positive integers  $n_x$  and  $n_y$  and any densities  $\rho(n_x, n_y)$ .

## **wilk – Various matrices devised or discussed by Wilkinson**

`[A,b] = gallery('wilk', n)` returns a different matrix or linear system depending on the value of  $n$ .

$n = 3$	Upper triangular system $Ux=b$ illustrating inaccurate solution.
$n = 4$	Lower triangular system $Lx=b$ , ill-conditioned.
$n = 5$	<code>hilb(6)(1:5,2:6)*1.8144</code> . A symmetric positive definite matrix.
$n = 21$	<code>W21+</code> , a tridiagonal matrix. eigenvalue problem. For more detail, see [2].

### **See Also**

`hadamard`, `hilb`, `invhilb`, `magic`, `wilkinson`

### **References**

[1] The MATLAB gallery of test matrices is based upon the work of Nicholas J. Higham at the Department of Mathematics, University of



Manchester, Manchester, England. Further background can be found in the books *MATLAB Guide, Second Edition*, Desmond J. Higham and Nicholas J. Higham, SIAM, 2005, and *Accuracy and Stability of Numerical Algorithms*, Nicholas J. Higham, SIAM, 1996.

[2] Wilkinson, J. H., *The Algebraic Eigenvalue Problem*, Oxford University Press, London, 1965, p.308.

# gamma, gammainc, gammaln

---

**Purpose** Gamma functions

**Syntax**  
Y = gamma(A)  
Y = gammainc(X,A)  
Y = gammainc(X,A,tail)  
Y = gammaln(A)

**Definition** The gamma function is defined by the integral:

$$\Gamma(a) = \int_0^{\infty} e^{-t} t^{a-1} dt$$

The gamma function interpolates the factorial function. For integer n:

$$\text{gamma}(n+1) = n! = \text{prod}(1:n)$$

The incomplete gamma function is:

$$P(x, a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

For any  $a \geq 0$ ,  $\text{gammainc}(x, a)$  approaches 1 as  $x$  approaches infinity. For small  $x$  and  $a$ ,  $\text{gammainc}(x, a)$  is approximately equal to  $x^a$ , so  $\text{gammainc}(0, 0) = 1$ .

**Description** Y = gamma(A) returns the gamma function at the elements of A. A must be real.

Y = gammainc(X,A) returns the incomplete gamma function of corresponding elements of X and A. Arguments X and A must be real and the same size (or either can be scalar).

Y = gammainc(X,A,tail) specifies the tail of the incomplete gamma function when X is non-negative. The choices for tail are 'lower' (the default) and 'upper'. The upper incomplete gamma function is defined as

$$1 - \text{gammainc}(x, a)$$

---

**Note** When  $X$  is negative,  $Y$  can be inaccurate for  $\text{abs}(X) > A+1$ .

---

$Y = \text{gammaln}(A)$  returns the logarithm of the gamma function,  $\text{gammaln}(A) = \log(\text{gamma}(A))$ . The `gammaln` command avoids the underflow and overflow that may occur if it is computed directly using  $\log(\text{gamma}(A))$ .

## Algorithm

The computations of `gamma` and `gammaln` are based on algorithms outlined in [1]. Several different minimax rational approximations are used depending upon the value of  $A$ . Computation of the incomplete gamma function is based on the algorithm in [2].

## References

- [1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.
  
- [2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

**Purpose** Current axes handle

**Syntax** `h = gca`

**Description** `h = gca` returns the handle to the current axes for the current figure. If no axes exists, the MATLAB software creates one and returns its handle. You can use the statement

```
get(gcf, 'CurrentAxes')
```

if you do not want MATLAB to create an axes if one does not already exist.

### **Current Axes**

The current axes is the target for graphics output when you create axes children. The current axes is typically the last axes used for plotting or the last axes clicked on by the mouse. Graphics commands such as `plot`, `text`, and `surf` draw their results in the current axes. Changing the current figure also changes the current axes.

### **See Also**

`axes`, `cla`, `gcf`, `findobj`

figure `CurrentAxes` property

“Graphics Object Identification” on page 1-97 for related functions

**Purpose** Handle of figure containing object whose callback is executing

**Syntax** `fig = gcbf`

**Description** `fig = gcbf` returns the handle of the figure that contains the object whose callback is currently executing. This object can be the figure itself, in which case, `gcbf` returns the figure's handle.

When no callback is executing, `gcbf` returns the empty matrix, `[]`.

The value returned by `gcbf` is identical to the figure output argument returned by `gcbo`.

**See Also** `gcbo`, `gco`, `gcf`, `gca`

<b>Purpose</b>	Handle of object whose callback is executing
<b>Syntax</b>	<pre>h = gcbo [h,figure] = gcbo</pre>
<b>Description</b>	<p><code>h = gcbo</code> returns the handle of the graphics object whose callback is executing.</p> <p><code>[h,figure] = gcbo</code> returns the handle of the current callback object and the handle of the figure containing this object.</p>
<b>Remarks</b>	<p>The MATLAB software stores the handle of the object whose callback is executing in the root <code>CallbackObject</code> property. If a callback interrupts another callback, MATLAB replaces the <code>CallbackObject</code> value with the handle of the object whose callback is interrupting. When that callback completes, MATLAB restores the handle of the object whose callback was interrupted.</p> <p>The root <code>CallbackObject</code> property is read only, so its value is always valid at any time during callback execution. The root <code>CurrentFigure</code> property, and the figure <code>CurrentAxes</code> and <code>CurrentObject</code> properties (returned by <code>gcf</code>, <code>gca</code>, and <code>gco</code>, respectively) are user settable, so they can change during the execution of a callback, especially if that callback is interrupted by another callback. Therefore, those functions are not reliable indicators of which object's callback is executing.</p> <p>When you write callback routines for the <code>CreateFcn</code> and <code>DeleteFcn</code> of any object and the figure <code>ResizeFcn</code>, you must use <code>gcbo</code> since those callbacks do not update the root's <code>CurrentFigure</code> property, or the figure's <code>CurrentObject</code> or <code>CurrentAxes</code> properties; they only update the root's <code>CallbackObject</code> property.</p> <p>When no callbacks are executing, <code>gcbo</code> returns <code>[]</code> (an empty matrix).</p>
<b>See Also</b>	<p><code>gca</code>, <code>gcf</code>, <code>gco</code>, <code>rootobject</code></p> <p>“Graphics Object Identification” on page 1-97 for related functions.</p>

**Purpose**

Greatest common divisor

**Syntax**

$G = \text{gcd}(A,B)$   
 $[G,C,D] = \text{gcd}(A,B)$

**Description**

$G = \text{gcd}(A,B)$  returns an array containing the greatest common divisors of the corresponding elements of integer arrays A and B. By convention,  $\text{gcd}(0,0)$  returns a value of 0; all other inputs return positive integers for G.

$[G,C,D] = \text{gcd}(A,B)$  returns both the greatest common divisor array G, and the arrays C and D, which satisfy the equation:  $A(i) .* C(i) + B(i) .* D(i) = G(i)$ . These are useful for solving Diophantine equations and computing elementary Hermite transformations.

**Examples**

The first example involves elementary Hermite transformations.

For any two integers a and b there is a 2-by-2 matrix E with integer entries and determinant = 1 (a *unimodular* matrix) such that:

$$E * [a;b] = [g,0],$$

where g is the greatest common divisor of a and b as returned by the command  $[g,c,d] = \text{gcd}(a,b)$ .

The matrix E equals:

$$\begin{array}{cc} c & d \\ -b/g & a/g \end{array}$$

In the case where  $a = 2$  and  $b = 4$ :

$$\begin{array}{l} [g,c,d] = \text{gcd}(2,4) \\ g = \\ \quad 2 \\ c = \\ \quad 1 \\ d = \\ \quad 0 \end{array}$$

So that

$$E = \begin{array}{cc} 1 & 0 \\ -2 & 1 \end{array}$$

In the next example, we solve for  $x$  and  $y$  in the Diophantine equation  $30x + 56y = 8$ .

$$\begin{aligned} [g, c, d] &= \text{gcd}(30, 56) \\ g &= 2 \\ c &= -13 \\ d &= 7 \end{aligned}$$

By the definition, for scalars  $c$  and  $d$ :

$$30(-13) + 56(7) = 2,$$

Multiplying through by  $8/2$ :

$$30(-13*4) + 56(7*4) = 8$$

Comparing this to the original equation, a solution can be read by inspection:

$$x = (-13*4) = -52; \quad y = (7*4) = 28$$

## See Also

1cm

## References

[1] Knuth, Donald, *The Art of Computer Programming*, Vol. 2, Addison-Wesley: Reading MA, 1973. Section 4.5.2, Algorithm X.



**Purpose** Current figure handle

**Syntax** `h = gcf`

**Description** `h = gcf` returns the handle of the current figure. The current figure is the figure window in which graphics commands such as `plot`, `title`, and `surf` draw their results. If no figure exists, the MATLAB software creates one and returns its handle. You can use the statement

```
get(0, 'CurrentFigure')
```

if you do not want MATLAB to create a figure if one does not already exist.

**See Also** `clf`, `figure`, `gca`

Root `CurrentFigure` property

“Graphics Object Identification” on page 1-97 for related functions

<b>Purpose</b>	Handle of current object
<b>Syntax</b>	<code>h = gco</code> <code>h = gco(<i>figure_handle</i>)</code>
<b>Description</b>	<code>h = gco</code> <i>returns</i> the handle of the current object. <code>h = gco(<i>figure_handle</i>)</code> returns the value of the current object for the figure specified by <i>figure_handle</i> .
<b>Remarks</b>	<p>The current object is the last object clicked on, excluding <code>uimenu</code>. If the mouse click did not occur over a figure child object, the figure becomes the current object. The MATLAB software stores the handle of the current object in the figure's <code>CurrentObject</code> property.</p> <p>The <code>CurrentObject</code> of the <code>CurrentFigure</code> does not always indicate the object whose callback is being executed. Interruptions of callbacks by other callbacks can change the <code>CurrentObject</code> or even the <code>CurrentFigure</code>. Some callbacks, such as <code>CreateFcn</code> and <code>DeleteFcn</code>, and <code>uimenu Callback</code>, intentionally do not update <code>CurrentFigure</code> or <code>CurrentObject</code>.</p> <p><code>gco</code> provides the only completely reliable way to retrieve the handle to the object whose callback is executing, at any point in the callback function, regardless of the type of callback or of any previous interruptions.</p>
<b>Examples</b>	This statement returns the handle to the current object in figure window 2:  <code>h = gco(2)</code>
<b>See Also</b>	<code>gca</code> , <code>gcbo</code> , <code>gcf</code> The root object description “Graphics Object Identification” on page 1-97 for related functions

**Purpose** Test for greater than or equal to

**Syntax** A >= B  
ge(A, B)

**Description** A >= B compares each element of array A with the corresponding element of array B, and returns an array with elements set to logical 1 (true) where A is greater than or equal to B, or set to logical 0 (false) where A is less than B. 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.

ge(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 greater than or equal to the corresponding elements of B:

```
A = magic(6);  
B = repmat(3*magic(3), 2, 2);  
  
A >= B  
ans =  
     1     0     0     1     1     1  
     0     1     0     1     1     1  
     1     0     0     1     1     1  
     0     1     1     0     1     0
```

1	0	1	1	0	0
0	1	1	1	0	1

**See Also**      gt, eq, le, lt, ne, “Relational Operators”

## Purpose

Generate path string

## Syntax

```
genpath
genpath directory
p = genpath('directory')
```

## Description

`genpath` returns a path string formed by recursively adding all the directories below *matlabroot/toolbox*.

`genpath directory` returns a path string formed by recursively adding all the directories below `directory`. This path string does not include directories named `private` or directories that begin with the `@` character.

`p = genpath('directory')` returns the path string to variable, `p`.

## Examples

To generate a path that includes *matlabroot/toolbox/images* and all directories below it, run the following:

```
p = genpath(fullfile(matlabroot,'toolbox','images'))
p =
```

```
matlabroot\toolbox\images;matlabroot\toolbox\images\
images;matlabroot\toolbox\images\images\ja;
matlabroot\toolbox\images\imdemos;matlabroot\
toolbox\images\imdemos\ja;
```

You can also use `genpath` in conjunction with `addpath` to add subdirectories to the path. The following example adds the `\control` directory and its subdirectories to the current path.

```
% Display the current path
path
```

```
MATLABPATH
```

```
K:\toolbox\matlab\general
K:\toolbox\matlab\ops
```

# genpath

---

```
K:\toolbox\matlab\lang
K:\toolbox\matlab\elmat
K:\toolbox\matlab\elfun
:
:
:

% Use GENPATH to add \control and its subdirectories
addpath(genpath('K:\toolbox\control'))

% Display the new path
path

MATLABPATH

K:\toolbox\control
K:\toolbox\control\ctrlutil
K:\toolbox\control\control
K:\toolbox\control\ctrlguis
K:\toolbox\control\ctrldemos
K:\toolbox\matlab\general
K:\toolbox\matlab\ops
K:\toolbox\matlab\lang
K:\toolbox\matlab\elmat
K:\toolbox\matlab\elfun
:
:
:
```

## See Also

`addpath`, `path`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `rmpath`, `savepath`

“Search Path” in the MATLAB Desktop Tools and Development Environment documentation

**Purpose** Construct valid variable name from string

**Syntax**  
varname = genvarname(str)  
varname = genvarname(str, exclusions)

**Description** varname = genvarname(str) constructs a string varname that is similar to or the same as the str input, and can be used as a valid variable name. str can be a single character array or a cell array of strings. If str is a cell array of strings, genvarname returns a cell array of strings in varname. The strings in a cell array returned by genvarname are guaranteed to be different from each other.

varname = genvarname(str, exclusions) returns a valid variable name that is different from any name listed in the exclusions input. The exclusions input can be a single character array or a cell array of strings. Specify the function who in the exclusions character array to create a variable name that will be unique in the current MATLAB workspace (see “Example 4” on page 2-1451, below).

---

**Note** genvarname returns a string that can be used as a variable name. It does not create a variable in the MATLAB workspace. You cannot, therefore, assign a value to the output of genvarname.

---

**Remarks** A valid MATLAB variable name is a character string of letters, digits, and underscores, such that the first character is a letter, and the length of the string is less than or equal to the value returned by the namelengthmax function. Any string that exceeds namelengthmax is truncated in the varname output. See “Example 6” on page 2-1452, below.

The variable name returned by genvarname is not guaranteed to be different from other variable names currently in the MATLAB workspace unless you use the exclusions input in the manner shown in “Example 4” on page 2-1451, below.

If you use `genvarname` to generate a field name for a structure, MATLAB does create a variable for the structure and field in the MATLAB workspace. See “Example 3” on page 2-1450, below.

If the `str` input contains any whitespace characters, `genvarname` removes them and capitalizes the next alphabetic character in `str`. If `str` contains any nonalphanumeric characters, `genvarname` translates these characters into their hexadecimal value.

## Examples

### Example 1

Create four similar variable name strings that do not conflict with each other:

```
v = genvarname({'A', 'A', 'A', 'A'})
v =
    'A'    'A1'    'A2'    'A3'
```

### Example 2

Read a column header `hdr` from worksheet `trial2` in Excel spreadsheet `myproj_apr23`:

```
[data hdr] = xlsread('myproj_apr23.xls', 'trial2');
```

Make a variable name from the text of the column header that will not conflict with other names:

```
v = genvarname(['Column ' hdr{1,3}]);
```

Assign data taken from the spreadsheet to the variable in the MATLAB workspace:

```
eval([v '= data(1:7, 3);']);
```

### Example 3

Collect readings from an instrument once every minute over the period of an hour into different fields of a structure. `genvarname` not only generates unique fieldname strings, but also creates the structure and fields in the MATLAB workspace:



```

for k = 1:60
    record.(genvarname(['reading' datestr(clock, 'HHMMSS')])) = takeReading;
    pause(60)
end

```

After the program ends, display the recorded data from the workspace:

```

record
record =
    reading090446: 27.3960
    reading090546: 23.4890
    reading090646: 21.1140
    reading090746: 23.0730
    reading090846: 28.5650
    .
    .
    .

```

#### Example 4

Generate variable names that are unique in the MATLAB workspace by putting the output from the who function in the exclusions list.

```

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, you can see that the variables created by genvarname are unique in the workspace:

```

time_elapsed =
    5.0070
time_elapsed1 =
    2.0030
time_elapsed2 =
    7.0010

```

# genvarname

---

```
time_elapsed3 =  
    8.0010  
time_elapsed4 =  
    3.0040
```

After the program completes, use the `who` function to view the workspace variables:

```
who  
  
k          time_elapsed  time_elapsed2  time_elapsed4  
t          time_elapsed1  time_elapsed3  v
```

## Example 5

If you try to make a variable name from a MATLAB keyword, `genvarname` creates a variable name string that capitalizes the keyword and precedes it with the letter `x`:

```
v = genvarname('global')  
v =  
    xGlobal
```

## Example 6

If you enter a string that is longer than the value returned by the `namelengthmax` function, `genvarname` truncates the resulting variable name string:

```
namelengthmax  
ans =  
    63  
  
vstr = genvarname(sprintf('%s%s', ...  
    'This name truncates because it contains ', ...  
    'more than the maximum number of characters'))  
vstr =  
    ThisNameTruncatesBecauseItContainsMoreThanTheMaximumNumberOfCha
```

## See Also

`isvarname`, `iskeyword`, `isletter`, `namelengthmax`, `who`, `regexp`

**Purpose**

Query Handle Graphics object properties

**Syntax**

```
get(h)
get(h, 'PropertyName')
<m-by-n value cell array> = get(H,pn)
a = get(h)
a = get(0)
a = get(0, 'Factory')
a = get(0, 'FactoryObjectTypePropertyName')
a = get(h, 'Default')
a = get(h, 'DefaultObjectTypePropertyName')
```

**Description**

---

**Note** Do not use the `get` function on Java objects as it will cause a memory leak. For more information, see “Accessing Private and Public Data”

---

`get(h)` returns all properties of the graphics object identified by the handle `h` and their current values.

`get(h, 'PropertyName')` returns the value of the property `'PropertyName'` of the graphics object identified by `h`.

`<m-by-n value cell array> = get(H,pn)` returns  $n$  property values for  $m$  graphics objects in the  $m$ -by- $n$  cell array, where  $m = \text{length}(H)$  and  $n$  is equal to the number of property names contained in `pn`.

`a = get(h)` returns a structure whose field names are the object's property names and whose values are the current values of the corresponding properties. `h` must be a scalar. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0)` returns the current values of all user-settable properties. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0, 'Factory')` returns the factory-defined values of all user-settable properties. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0, 'FactoryObjectTypePropertyName')` returns the factory-defined value of the named property for the specified object type. The argument *FactoryObjectTypePropertyName* is the word `Factory` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`)`FactoryFigureColor`.

`a = get(h, 'Default')` returns all default values currently defined on object `h`. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(h, 'DefaultObjectTypePropertyName')` returns the factory-defined value of the named property for the specified object type. The argument *DefaultObjectTypePropertyName* is the word `Default` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`).

```
DefaultFigureColor
```

## Examples

You can obtain the default value of the `LineWidth` property for line graphics objects defined on the root level with the statement

```
get(0, 'DefaultLineLineWidth')
ans =
    0.5000
```

To query a set of properties on all axes children, define a cell array of property names:

```
props = {'HandleVisibility', 'Interruptible';
         'SelectionHighlight', 'Type'};
output = get(get(gca, 'Children'), props);
```

The variable `output` is a cell array of dimension `length(get(gca, 'Children'))-by-4`.

For example, type

```
patch;surface;text;line
output = get(get(gca, 'Children'), props)
output =
'on'      'on'      'on'      'line'
'on'      'off'     'on'      'text'
'on'      'on'      'on'      'surface'
'on'      'on'      'on'      'patch'
```

## See Also

`findobj`, `gca`, `gcf`, `gco`, `set`

Handle Graphics Properties

“Graphics Object Identification” on page 1-97 for related functions

# get (COM)

---

**Purpose** Get property value from interface, or display properties

**Syntax**

```
V = h.get  
V = h.get('propertyname')  
V = get(h, ...)
```

**Description** V = h.get returns a list of all properties and their values for the object or interface, h.

If V is empty, either there are no properties in the object, or the MATLAB software cannot read the object's type library. Refer to the COM vendor's documentation. For Automation objects, if the vendor provides documentation for a specific property, use the V = get(h, ...) syntax to call it.

V = h.get('propertyname') returns the value of the property specified in the string, propertyname.

V = get(h, ...) is an alternate syntax for the same operation.

**Remarks** The meaning and type of the return value is dependent upon the specific property being retrieved. The object's documentation should describe the specific meaning of the return value. MATLAB may convert the data type of the return value. For a description of how MATLAB converts COM data types, see "Handling COM Data in MATLAB Software" in the External Interfaces documentation.

COM functions are available on Microsoft Windows systems only.

**Examples** Create a COM server running Microsoft Excel software:

```
e = actxserver ('Excel.Application');
```

Retrieve a single property value:

```
e.Path
```

Depending on your spreadsheet program, MATLAB software displays:

```
ans =  
    C:\Program Files\MSOffice\OFFICE11
```

Retrieve a list of all properties for the CommandBars interface:

```
c = e.CommandBars.get
```

MATLAB displays information similar to the following:

```
c =  
    Application: [1x1  
Interface.Microsoft_Excel_11.0_Object_Library._Application]  
    Creator: 1.4808e+009  
    ActionControl: []  
    ActiveMenuBar: [1x1  
Interface.Microsoft_Office_12.0_Object_Library.CommandBar]  
    Count: 129  
    DisplayTooltips: 1  
    DisplayKeysInTooltips: 0  
    LargeButtons: 0  
    MenuAnimationStyle: 'msoMenuAnimationNone'  
    Parent: [1x1  
Interface.Microsoft_Excel_11.0_Object_Library._Application]  
    AdaptiveMenus: 0  
    DisplayFonts: 1  
    DisableCustomize: 0  
    DisableAskAQuestionDropdown: 0
```

## See Also

set (COM), inspect, isprop, addproperty, deleteproperty

# get (hgsetget)

---

**Purpose** Query property values of handle objects derived from hgsetget class

**Syntax** CV = get(H, 'PropertyName')  
SV = get(h)

**Description** CV = get(H, 'PropertyName') returns the value of the named property from the objects in the handle array H. If H is scalar, get returns a single value; if H is an array, get returns a cell array of property values. If you specify a cell array of property names, then get returns a cell array of values, where each row in the cell corresponds to an element in H and each column in the cell corresponds to an element in the property name cell array.

SV = get(h) returns a struct array in which the field names are the object's property names and the values are the current values of the corresponding properties. h must be a scalar.

**See Also** See "Implementing a Set/Get Interface for Properties"

get, handle, hgsetget, set (hgsetget)



**Purpose** Memmapfile object properties

**Syntax**  
`s = get(obj)`  
`val = get(obj, prop)`

**Description** `s = get(obj)` returns the values of all properties of the memmapfile object `obj` in structure array `s`. Each property retrieved from the object is represented by a field in the output structure. The name and contents of each field are the same as the name and value of the property it represents.

---

**Note** Although property names of a memmapfile object are not case sensitive, field names of the output structure returned by `get` (named the same as the properties they represent) are case sensitive.

---

`val = get(obj, prop)` returns the value(s) of one or more properties specified by `prop`. The `prop` input can be a quoted string or a cell array of quoted strings, each containing a property name. If the latter is true, `get` returns the property values in a cell array.

**Examples** You can use the `get` method of the memmapfile class to return information on any or all of the object's properties. Specify one or more property names to get the values of specific properties.

This example returns the values of the `Offset`, `Repeat`, and `Format` properties for a memmapfile object. Start by constructing the object:

```
m = memmapfile('records.dat', ...
               'Offset', 2048, ...
               'Format', {
                   'int16' [2 2] 'model'; ...
                   'uint32' [1 1] 'serialno'; ...
                   'single' [1 3] 'expenses'});
```

## get (memmapfile)

---

Use the `get` method to return the specified property values in a 1-by-3 cell array `m_props`:

```
m_props = get(m, {'Offset', 'Repeat', 'Format'})
m_props =
    [2048]    [Inf]    {3x3 cell}

m_props{3}
ans =
    'int16'    [1x2 double]    'model'
    'uint32'   [1x2 double]    'serialno'
    'single'   [1x2 double]    'expenses'
```

Another way to return the same information is to use the `objname.property` syntax:

```
m_props = {m.Offset, m.Repeat, m.Format}
m_props =
    [2048]    [Inf]    {3x3 cell}
```

To return the values for all properties with `get`, pass just the object name:

```
s = get(m)
Filename: 'd:\matlab\mfiles\records.dat'
Writable: 0
Offset: 2048
Format: {3x3 cell}
Repeat: Inf
Data: [753 1]
```

To see just the `Format` field of the returned structure, type

```
s.Format
ans =
    'int16'    [1x2 double]    'model'
    'uint32'   [1x2 double]    'serialno'
    'single'   [1x2 double]    'expenses'
```

**See Also** `memmapfile`, `disp(memmapfile)`

# get (RandStream)

---

**Purpose** Random stream properties

**Class** @RandStream

**Syntax**  
`get(s)`  
`P = get(s)`  
`P = get(s, 'PropertyName')`

**Description**  
`get(s)` prints the list of properties for the random stream `s`.  
`P = get(s)` returns all properties of `s` in a scalar structure.  
`P = get(s, 'PropertyName')` returns the property 'PropertyName'.

**See Also** @RandStream, set (RandStream)

**Purpose** Serial port object properties

**Syntax**

```
get(obj)
out = get(obj)
out = get(obj, 'PropertyName')
```

**Description**

`get(obj)` returns all property names and their current values to the command line for the serial port object, `obj`.

`out = get(obj)` returns the structure `out` where each field name is the name of a property of `obj`, and each field contains the value of that property.

`out = get(obj, 'PropertyName')` returns the value `out` of the property specified by *PropertyName* for `obj`. If *PropertyName* is replaced by a 1-by-`n` or `n`-by-1 cell array of strings containing property names, then `get` returns a 1-by-`n` cell array of values to `out`. If `obj` is an array of serial port objects, then `out` will be a `m`-by-`n` cell array of property values where `m` is equal to the length of `obj` and `n` is equal to the number of properties specified.

**Remarks**

Refer to [Displaying Property Names and Property Values](#) for a list of serial port object properties that you can return with `get`.

When you specify a property name, you can do so without regard to case, and you can make use of property name completion. For example, if `s` is a serial port object, then these commands are all valid.

```
out = get(s, 'BaudRate');
out = get(s, 'baudrate');
out = get(s, 'BAUD');
```

If you use the `help` command to display help for `get`, then you need to supply the pathname shown below.

```
help serial/get
```

# get (serial)

---

## Example

This example illustrates some of the ways you can use `get` to return property values for the serial port object `s`.

```
s = serial('COM1');  
out1 = get(s);  
out2 = get(s,{'BaudRate','DataBits'});  
get(s,'Parity')  
ans =  
none
```

## See Also

### Functions

`set`

**Purpose** Timer object properties

**Syntax**  
`get(obj)`  
`V = get(obj)`  
`V = get(obj, 'PropertyName')`

**Description** `get(obj)` displays all property names and their current values for the timer object `obj`. `obj` must be a single timer object.

`V = get(obj)` returns a structure, `V`, where each field name is the name of a property of `obj` and each field contains the value of that property. If `obj` is an M-by-1 vector of timer objects, `V` is an M-by-1 array of structures.

`V = get(obj, 'PropertyName')` returns the value, `V`, of the timer object property specified in *PropertyName*.

If *PropertyName* is a 1-by-N or N-by-1 cell array of strings containing property names, `V` is a 1-by-N cell array of values. If `obj` is a vector of timer objects, `V` is an M-by-N cell array of property values where M is equal to the length of `obj` and N is equal to the number of properties specified.

**Examples**

```
t = timer;  
get(t)  
AveragePeriod: NaN  
BusyMode: 'drop'  
ErrorFcn: ''  
ExecutionMode: 'singleShot'  
InstantPeriod: NaN  
Name: 'timer-1'  
ObjectVisibility: 'on'  
Period: 1  
Running: 'off'  
StartDelay: 1  
StartFcn: ''  
StopFcn: ''  
Tag: ''
```

## get (timer)

---

```
        TasksExecuted: 0
        TasksToExecute: Inf
            TimerFcn: ''
                Type: 'timer'
                UserData: []
    get(t, {'StartDelay','Period'})
    ans =

        [0]    [1]
```

### See Also

timer, set(timer)



**Purpose** Query timeseries object property values

**Syntax** `value = get(ts, 'PropertyName')`  
`get(ts)`

**Description** `value = get(ts, 'PropertyName')` returns the value of the specified property of the timeseries object. The following syntax is equivalent:

```
value = ts.PropertyName
```

`get(ts)` displays all properties and values of the time series ts.

**See Also** `set (timeseries)`, `timeseries`, `tsprops`

# get (tscollection)

---

**Purpose** Query tscollection object property values

**Syntax** `value = get(tsc, 'PropertyName')`

**Description** `value = get(tsc, 'PropertyName')` returns the value of the specified property of the tscollection object tsc. The following syntax is equivalent:

`value = tsc.PropertyName`

`get(tsc)` displays all properties and values of the tscollection object tsc.

**See Also** `set (tscollection), tscollection`

**Purpose** Extract date-string time vector into cell array

**Syntax** `getabstime(ts)`

**Description** `getabstime(ts)` extracts the time vector from the `timeseries` object `ts` as a cell array of date strings. To define the time vector relative to a calendar date, set the `TimeInfo.StartDate` property of the `timeseries` object. When the `TimeInfo.StartDate` format is a valid `datestr` format, the output strings from `getabstime` have the same format.

**Examples** The following example shows how to extract a time vector as a cell array of date strings from a `timeseries` object.

**1** Create a `timeseries` object.

```
ts = timeseries([3 6 8 0 10]);
```

The default time vector for `ts` is `[0 1 2 3 4]`, which starts at 0 and increases in 1-second increments. The length of the time vector is equal to the length of the data.

**2** Set the `StartDate` property.

```
ts.TimeInfo.StartDate = '10/27/2005 07:05:36';
```

**3** Extract the time vector.

```
getabstime(ts)
```

```
ans =
```

```
'27-Oct-2005 07:05:36'  
'27-Oct-2005 07:05:37'  
'27-Oct-2005 07:05:38'  
'27-Oct-2005 07:05:39'  
'27-Oct-2005 07:05:40'
```

## getabstime (timeseries)

---

**4** Change the date-string format of the time vector.

```
ts.TimeInfo.Format = 'mm/dd/yy'
```

**5** Extract the time vector with the new date-string format.

```
getabstime(ts)
```

```
ans =
```

```
'10/27/05'
```

```
'10/27/05'
```

```
'10/27/05'
```

```
'10/27/05'
```

```
'10/27/05'
```

### See Also

`setabstime (timeseries)`, `timeseries`, `tsprops`

**Purpose** Extract date-string time vector into cell array

**Syntax** `getabstime(tsc)`

**Description** `getabstime(tsc)` extracts the time vector from the `tscollection` object `tsc` as a cell array of date strings. To define the time vector relative to a calendar date, set the `TimeInfo.StartDate` property of the time-series collection. When the `TimeInfo.StartDate` format is a valid `datestr` format, the output strings from `getabstime` have the same format.

**Examples** **1** Create a `tscollection` object.

```
tsc = tscollection(timeseries([3 6 8 0 10]));
```

**2** Set the `StartDate` property.

```
tsc.TimeInfo.StartDate = '10/27/2005 07:05:36';
```

**3** Extract a vector of absolute time values.

```
getabstime(tsc)
```

```
ans =
```

```
'27-Oct-2005 07:05:36'  
'27-Oct-2005 07:05:37'  
'27-Oct-2005 07:05:38'  
'27-Oct-2005 07:05:39'  
'27-Oct-2005 07:05:40'
```

**4** Change the date-string format of the time vector.

```
tsc.TimeInfo.Format = 'mm/dd/yy';
```

**5** Extract the time vector with the new date-string format.

```
getabstime(tsc)
```

# getabstime (tscollection)

---

```
ans =  
    '10/27/05'  
    '10/27/05'  
    '10/27/05'  
    '10/27/05'  
    '10/27/05'
```

**See Also**      datestr, setabstime (tscollection), tscollection

**Purpose** Value of application-defined data

**Syntax**  
`value = getappdata(h,name)`  
`values = getappdata(h)`

**Description**  
`value = getappdata(h,name)` gets the value of the application-defined data with the name specified by `name`, in the object with handle `h`. If the application-defined data does not exist, the MATLAB software returns an empty matrix in `value`.  
`values = getappdata(h)` returns all application-defined data for the object with handle `h`.

**Remarks**  
Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property.

**See Also** `setappdata`, `rmappdata`, `isappdata`

# GetCharArray

---

**Purpose** Get character array from server

**Syntax**

**MATLAB Client**

```
string = h.GetCharArray('varname', 'workspace')  
string = GetCharArray(h, 'varname', 'workspace')  
string = invoke(h, 'GetCharArray', 'varname', 'workspace')
```

**Method Signature**

```
HRESULT GetCharArray([in] BSTR varName, [in] BSTR Workspace,  
[out, retval] BSTR *mlString)
```

**Microsoft Visual Basic Client**

```
GetCharArray(varname As String, workspace As String) As String
```

**Description** GetCharArray gets the character array stored in the variable varname from the specified workspace of the server attached to handle h and returns it in string. The *workspace* argument can be either base or global.

**Remarks** If you want output from GetCharArray to be displayed at the client window, you must specify an output variable (e.g., string).

Server function names, like GetCharArray, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

**Examples** Assign a string to variable str in the base workspace of the server using PutCharArray. Read it back in the client with GetCharArray.

**MATLAB Client**

```
h = actxserver('matlab.application');  
h.PutCharArray('str', 'base', ...  
    'He jests at scars that never felt a wound.');
```

S = h.GetCharArray('str', 'base')

```
S =  
    He jests at scars that never felt a wound.
```



## Visual Basic .NET Client

This example uses the Visual Basic MsgBox command to control flow between MATLAB and the Visual Basic Client.

```
Dim Matlab As Object
Dim S As String
Matlab = CreateObject("matlab.application")
MsgBox("In MATLAB, type" & vbCrLf _
    & "str='new string';")
```

Open the MATLAB window, then type:

```
str='new string';
```

Click **Ok**.

```
Try
    S = Matlab.GetCharArray("str", "base")
    MsgBox("str = " & S)
Catch ex As Exception
    MsgBox("You did not set 'str' in MATLAB")
End Try
```

The Visual Basic MsgBox displays what you typed in MATLAB.

## See Also

PutCharArray, GetWorkspaceData, PutWorkspaceData, GetVariable, Execute

# getdatasamplesize

---

**Purpose** Size of data sample in `timeseries` object

**Syntax** `getdatasamplesize(ts)`

**Description** `getdatasamplesize(ts)` returns the size of each data sample in a `timeseries` object.

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

**Examples** The following example shows how to get the size of a data sample in a `timeseries` object.

**1** Load a 24-by-3 data array.

```
load count.dat
```

**2** Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,[1:24], 'Name', 'VehicleCount')
```

**3** Get the size of the data sample for this `timeseries` object.

```
getdatasamplesize(count_ts)
```

```
ans =
```

```
1    3
```

The size of each data sample in `count_ts` is 1-by-3, which means that each data sample is stored as a row with three values.

**See Also** `addsample`, `size (timeseries)`, `tsprops`

## getDefaultStream (RandStream)

---

**Purpose** Default random number stream

**Class** @RandStream

**Syntax** stream = RandStream.getDefaultStream

**Description** stream = RandStream.getDefaultStream returns the default random number stream. The MATLAB functions rand, randi, and randn use the default stream to generate values.

rand, randi, and randn all rely on the default stream of uniform pseudorandom numbers. randi uses one uniform value from the default stream to generate each integer value; randn uses one or more uniform values from the default stream to generate each normal value. Note that there are also rand, randi, and randn methods for which you specify a specific random stream from which to draw values.

**See Also** @RandStream, setDefaultStream (RandStream), rand, randi, randn

# getdisp (hgsetget)

---

<b>Purpose</b>	Override to change command window display
<b>Syntax</b>	getdisp(H)
<b>Description</b>	getdisp(H) called by <code>get</code> when <code>get</code> is called with no output arguments and a single input argument that is a handle array. Override this <code>hgsetget</code> class method in a subclass to change how property information is displayed in the command window.
<b>See Also</b>	See “Implementing a Set/Get Interface for Properties” <code>hgsetget</code> , <code>get</code> ( <code>hgsetget</code> )

**Purpose** Environment variable

**Syntax** `getenv 'name'`  
`N = getenv('name')`

**Description** `getenv 'name'` searches the underlying operating system's environment list for a string of the form `name=value`, where `name` is the input string. If found, the MATLAB software returns the string `value`. If the specified name cannot be found, an empty matrix is returned.

`N = getenv('name')` returns `value` to the variable `N`.

**Examples** `os = getenv('OS')`

```
os =  
Windows_NT
```

**See Also** `setenv`, `computer`, `pwd`, `ver`, `path`

# getfield

---

**Purpose** Field of structure array

**Syntax**  
`f = getfield(s, 'field')`  
`f = getfield(s, {i,j}, 'field', {k})`

**Description** `f = getfield(s, 'field')`, where `s` is a 1-by-1 structure, returns the contents of the specified field. This is equivalent to the syntax `f = s.field`.

If `s` is a structure having dimensions greater than 1-by-1, `getfield` returns the first of all output values requested in the call. That is, for structure array `s(m,n)`, `getfield` returns `f = s(1,1).field`.

`f = getfield(s, {i,j}, 'field', {k})` returns the contents of the specified field. This is equivalent to the syntax `f = s(i,j).field(k)`. All subscripts must be passed as cell arrays — that is, they must be enclosed in curly braces (similar to `{i,j}` and `{k}` above). Pass field references as strings.

**Remarks** In many cases, you can use dynamic field names in place of the `getfield` and `setfield` functions. Dynamic field names express structure fields as variable expressions that the MATLAB software evaluates at run-time. See Solution 1-19QWG for information about using dynamic field names versus the `getfield` and `setfield` functions.

**Examples** Given the structure

```
mystr(1,1).name = 'alice';  
mystr(1,1).ID = 0;  
mystr(2,1).name = 'gertrude';  
mystr(2,1).ID = 1
```

Then the command `f = getfield(mystr, {2,1}, 'name')` yields

```
f =  
    gertrude
```

To list the contents of all name (or other) fields, embed `getfield` in a loop.

```
for k = 1:2
    name{k} = getfield(mystr, {k,1}, 'name');
end
name

name =

    'alice'    'gertrude'
```

The following example starts out by creating a structure using the standard structure syntax. It then reads the fields of the structure, using `getfield` with variable and quoted field names and additional subscripting arguments.

```
class = 5;    student = 'John_Doe';
grades(class).John_Doe.Math(10,21:30) = ...
    [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];
```

Use `getfield` to access the structure fields.

```
getfield(grades, {class}, student, 'Math', {10,21:30})

ans =
    85    89    76    93    85    91    68    84    95    73
```

## See Also

`setfield`, `fieldnames`, `isfield`, `orderfields`, `rmfield`, “Using Dynamic Field Names”

# getframe

---

**Purpose** Capture movie frame

**Syntax**

```
getframe
F = getframe
F = getframe(h)
F = getframe(h,rect)
```

**Description** `getframe` returns a movie frame. The frame is a snapshot (pixmap) of the current axes or figure.

`F = getframe` gets a frame from the current axes.

`F = getframe(h)` gets a frame from the figure or axes identified by handle `h`.

`F = getframe(h,rect)` specifies a rectangular area from which to copy the pixmap. `rect` is relative to the lower left corner of the figure or axes `h`, in pixel units. `rect` is a four-element vector in the form `[left bottom width height]`, where `width` and `height` define the dimensions of the rectangle.

`getframe` returns a movie frame, which is a structure having two fields:

- `cdata` — The image data stored as a matrix of `uint8` values. The dimensions of `F.cdata` are height-by-width-by-3.
- `colormap` — The colormap stored as an `n-by-3` matrix of doubles. `F.colormap` is empty on true color systems.

To capture an image, use this approach:

```
F = getframe(gcf);
image(F.cdata)
colormap(F.colormap)
```

**Remarks** `getframe` is usually used in a `for` loop to assemble an array of movie frames for playback using `movie`. For example,

```
for j = 1:n plotting commands
    F(j) = getframe;
```



```
end  
movie(F)
```

If you are capturing frames of a plot that takes a long time to generate or are repeatedly calling `getframe` in a loop, make sure that your computer's screen saver does not activate and that your monitor does not turn off for the duration of the capture; otherwise one or more of the captured frames can contain graphics from your screen saver or nothing at all.

---

**Note** In situations where MATLAB software is running on a virtual desktop that is not currently visible on your monitor, calls to `getframe` will complete, but will capture a region on your monitor that corresponds to the position occupied by the figure or axes on the hidden desktop. Therefore, make sure that the window to be captured by `getframe` exists on the currently active desktop.

---

### Capture Regions

Note that `F = getframe` returns the contents of the current axes, exclusive of the axis labels, title, or tick labels. `F = getframe(gcf)` captures the entire interior of the current figure window. To capture the figure window menu, use the form `F = getframe(h,rect)` with a rectangle sized to include the menu.

### Resolution of Captured Frames

The resolution of the framed image depends on the size of the axes in pixels when `getframe` is called. As the `getframe` command takes a snapshot of the screen, if the axes is small in size (e.g., because you have restricted the view to a window within the axes), `getframe` will capture fewer screen pixels, and the captured image might have poor resolution if enlarged for display.

### Capturing UIControls and Information Bars

If your figure contains uicontrols or displays the linking and brushing message bar along its top, `F = getframe(figure_handle)` captures

# getframe

---

them, along with the axes and any annotations displayed on the plot. `F = getframe` does not capture the message bar or uicontrols outside of the current axes. To avoid including the message bar when capturing the entire figure, click the **X** button on the message bar to dismiss it before running `getframe`. Once you do this, the message bar does not appear on subsequent figures unless you reset a preference to show it.

## Examples

Make the peaks function vibrate.

```
Z = peaks; surf(Z)
axis tight
set(gca, 'nextplot', 'replacechildren');
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    F(j) = getframe;
end
movie(F,20) % Play the movie twenty times
```

## See Also

`frame2im`, `image`, `im2frame`, `movie`

“Bit-Mapped Images” on page 1-96 for related functions

**Purpose**

Get matrix from server

**Syntax****MATLAB Client**

```
[xreal ximag] = h.GetFullMatrix('varname', 'workspace',  
zreal, zimag)  
[xreal ximag] = GetFullMatrix(h, 'varname', 'workspace',  
zreal, zimag)  
[xreal ximag] = invoke(h, 'GetFullMatrix', 'varname', 'workspace',  
zreal, zimag)
```

**Method Signature**

```
GetFullMatrix([in] BSTR varname,  
[in] BSTR workspace, [in, out] SAFEARRAY(double) *pr,  
[in, out] SAFEARRAY(double) *pi)
```

**Microsoft Visual Basic Client**

```
GetFullMatrix(varname As String, workspace As String,  
[out] XReal As Double, [out] XImag As Double)
```

---

**Note** GetFullMatrix works only with values of type double. Use GetVariable or GetWorkspaceData for other types.

---

**Description**

GetFullMatrix gets the matrix stored in the variable *varname* from the specified *workspace* of the server attached to handle *h* and returns the real part in *xreal* and the imaginary part in *ximag*. The *workspace* argument can be either *base* or *global*.

The *zreal* and *zimag* arguments are matrices of the same size as the real and imaginary matrices (*xreal* and *ximag*) being returned from the server. The *zreal* and *zimag* matrices are commonly set to zero (see example below).

**Remarks**

If you want output from GetFullMatrix to be displayed at the client window, you must specify one or both output variables (e.g., *xreal* and/or *ximag*).

# GetFullMatrix

---

Server function names, like `GetFullMatrix`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the variant data type instead of `safearray`, which is not supported by VBScript.

## Examples

Assign a 5-by-5 real matrix to the variable `M` in the base workspace of the server, and then read it back with `GetFullMatrix`.

### MATLAB Client

```
h = actxserver('matlab.application');
h.PutFullMatrix('M','base',rand(5),zeros(5));

MReal = h.GetFullMatrix('M','base',zeros(5),zeros(5))
MReal =
    0.9501    0.7621    0.6154    0.4057    0.0579
    0.2311    0.4565    0.7919    0.9355    0.3529
    0.6068    0.0185    0.9218    0.9169    0.8132
    0.4860    0.8214    0.7382    0.4103    0.0099
    0.8913    0.4447    0.1763    0.8936    0.1389
```

### Visual Basic .NET Client

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```
Dim MatLab As Object
Dim Result As String
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim i, j As Integer

MatLab = CreateObject("matlab.application")
Result = MatLab.Execute("M = rand(5);")
```

```
MsgBox("In MATLAB, type" & vbCrLf _  
      & "M(3,4)")
```

Open the MATLAB window and type:

```
M(3,4)
```

Click **Ok**.

```
MatLab.GetFullMatrix("M", "base", XReal, XImag)  
i = 2    %0-based array  
j = 3  
  
MsgBox("XReal(" & i + 1 & "," & j + 1 & ")" & _  
      " = " & XReal(i, j))
```

Click **Ok** to close and terminate MATLAB.

## See Also

PutFullMatrix, GetWorkspaceData, PutWorkspaceData, GetVariable, Execute

# getinterpmethod

---

**Purpose** Interpolation method for `timeseries` object

**Syntax** `getinterpmethod(ts)`

**Description** `getinterpmethod(ts)` returns the interpolation method as a string that is used by the `timeseries` object `ts`. Predefined interpolation methods are 'zoh' (zero-order hold) and 'linear' (linear interpolation). The method strings are case sensitive.

**Examples** **1** Create a `timeseries` object.

```
ts = timeseries(rand(5));
```

**2** Get the interpolation method for this object.

```
getinterpmethod(ts)
```

```
ans =
```

```
linear
```

**See Also** `setinterpmethod`, `timeseries`, `tsprops`

## Purpose

Get component position in pixels

## Syntax

```
position = getpixelposition(handle)
position = getpixelposition(handle,recursive)
```

## Description

`position = getpixelposition(handle)` gets the position, in pixel units, of the component with handle `handle`. The `position` is returned as a four-element vector that specifies the location and size of the component: [distance from left, distance from bottom, width, height].

`position = getpixelposition(handle,recursive)` gets the position as above. If `recursive` is true, the returned position is relative to the parent figure of `handle`.

Use the `getpixelposition` function only to obtain coordinates for children of figures and container components (uipanel, or uibuttongroups). Results are not reliable for children of axes or other Handle Graphics objects.

## Example

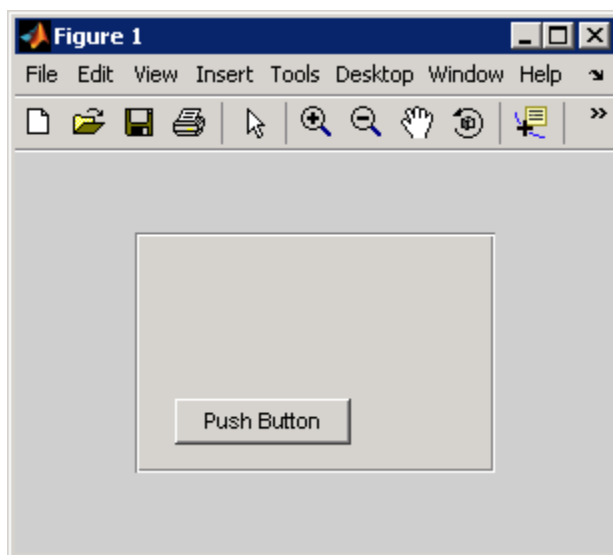
This example creates a push button within a panel, and then retrieves its position, in pixels, relative to the panel.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6]);
h1 = uicontrol(p,'Style','PushButton','Units','Normalized',...
              'String','Push Button','Position',[.1 .1 .5 .2]);
pos1 = getpixelposition(h1)

pos1 =
    18.6000    12.6000    88.0000    23.2000
```

# getpixelposition

---



The following statement retrieves the position of the push button, in pixels, relative to the figure.

```
pos1 = getpixelposition(h1,true)

pos1 =
    79.6000    53.6000    88.0000    23.2000
```

## See Also

`setpixelposition`, `uicontrol`, `uipanel`



**Purpose**

Preference

**Syntax**

```
getpref('group','pref')
getpref('group','pref',default)
getpref('group',{'pref1','pref2',... 'prefn'})
getpref('group',{'pref1',... 'prefn'},{default1,...defaultn})
getpref('group')
getpref
```

**Description**

`getpref('group','pref')` returns the value for the preference specified by `group` and `pref`. It is an error to get a preference that does not exist.

`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.

`getpref('group','pref',default)` returns the current value if the preference specified by `group` and `pref` exists. Otherwise creates the preference with the specified default value and returns that value.

`getpref('group',{'pref1','pref2',... 'prefn'})` returns a cell array containing the values for the preferences specified by `group` and the cell array of preference names. The return value is the same size as the input cell array. It is an error if any of the preferences do not exist.

`getpref('group',{'pref1',... 'prefn'},{default1,...defaultn})` returns a cell array with the current values of the preferences specified by `group` and the cell array of preference names. Any preference that does not exist is created with the specified default value and returned.

`getpref('group')` returns the names and values of all preferences in the group as a structure.

`getpref` returns all groups and preferences as a structure.

# getpref

---

---

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

---

## Examples

### Example 1

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

```
ans =
     1.0
```

### Example 2

```
rmpref('mytoolbox','version')
getpref('mytoolbox','version','1.0');
getpref('mytoolbox','version')
```

```
ans =
     1.0
```

## See Also

[addpref](#), [ispref](#), [rmpref](#), [setpref](#), [uigetpref](#), [uisetpref](#)

**Purpose** Data quality descriptions

**Syntax** `getqualitydesc(ts)`

**Description** `getqualitydesc(ts)` returns a cell array of data quality descriptions based on the `Quality` values you assigned to a `timeseries` object `ts`.

**Examples**

- 1 Create a `timeseries` object with `Data`, `Time`, and `Quality` values, respectively.

```
ts = timeseries([3; 4.2; 5; 6.1; 8], 1:5, [1; 0; 1; 0; 1]);
```

- 2 Set the `QualityInfo` property, consisting of `Code` and `Description`.

```
ts.QualityInfo.Code = [0 1];  
ts.QualityInfo.Description = {'good' 'bad'};
```

- 3 Get the data quality description strings for `ts`.

```
getqualitydesc(ts)
```

```
ans =
```

```
'bad'  
'good'  
'bad'  
'good'  
'bad'
```

**See Also** `tsprops`

# getReport (MException)

---

**Purpose** Get error message for exception

**Syntax**

```
report = getReport(ME)
report = getReport(ME, type)
report = getReport(ME, type, 'hyperlinks', value)
```

**Description** `report = getReport(ME)` returns a formatted message string, `report`, that is based on the current error (or *exception*). This exception is represented by an object `ME` of the `MException` class. The message string returned by `getReport` is the same as the error message displayed by MATLAB when it throws this same exception.

`report = getReport(ME, type)` returns a message string that either describes just the highest level error (*basic type*), or shows the error and the stack as well (*extended type*). The `type` argument, when used, must be the second argument in the input argument list. See “Examples” on page 2-1495 , below.

<b>type Option</b>	<b>Displayed Text</b>
'extended'	Display line number, error message, and cause and stack summary (default)
'basic'	Display line number and error message

`report = getReport(ME, type, 'hyperlinks', value)` returns a message string that either does or does not include active hyperlinks to the failing lines in the code. See the table below for the valid choices for `value`. The `'hyperlinks'` and `value` arguments, when used, must be the third and fourth arguments in the input argument list.

<b>value Option</b>	<b>Action</b>
'on'	Display hyperlinks to failing lines (default)

value Option	Action
'off'	Do not display hyperlinks to failing lines
'default'	Use the default for the Command Window to determine whether or not to use hyperlinks in the error message

### Examples

Try calling the MATLAB `surf` function without the required input argument. In the `catch` statement, capture the error in an `MException` object, `ME`. Then, use this object with `getReport` to retrieve a basic error string:

```
try
    surf
catch ME
    rep = getReport(ME, 'basic')
end

rep =
Error using ==> surf at 50
Not enough input arguments.
```

Run the try-catch again, this time replacing `'basic'` with `'extended'`: In this case, the error message includes information from the stack:

```
rep = getReport(ME, 'extended')

rep =
Error using ==> surf at 50
Not enough input arguments.

Error in ==> getRep>getRep3 at 9
    surf

Error in ==> getRep>getRep2 at 5
    getRep3(option, state)
```

## getReport (MException)

---

```
Error in ==> getRep at 2  
getRep2(option, state)
```

### See Also

```
try, catch, error, assert, MException, disp(MException),  
throw(MException), rethrow(MException),  
throwAsCaller(MException), addCause(MException),  
isequal(MException), eq(MException), ne(MException),  
last(MException),
```

# getsampleusingtime (timeseries)

---

**Purpose**

Extract data samples into new timeseries object

**Syntax**

```
ts2 = getsampleusingtime(ts1,Time)
ts2 = getsampleusingtime(ts1,StartTime,EndTime)
```

**Description**

ts2 = getsampleusingtime(ts1,Time) returns a new timeseries object ts2 with a single sample corresponding to the time Time in ts1.

ts2 = getsampleusingtime(ts1,StartTime,EndTime) returns a new timeseries object ts2 with samples between the times StartTime and EndTime in ts1.

**Remarks**

When the time vector in ts1 is numeric, StartTime and EndTime must also be numeric. When the times in ts1 are date strings and the StartTime and EndTime values are numeric, then the StartTime and EndTime values are treated as datenum values.

**See Also**

timeseries

## getsampleusingtime (tscollection)

---

**Purpose** Extract data samples into new tscollection object

**Syntax**  
`tsc2 = getsampleusingtime(tsc1,Time)`  
`tsc2 = getsampleusingtime(tsc1,StartTime,EndTime)`

**Description**  
`tsc2 = getsampleusingtime(tsc1,Time)` returns a new tscollection `tsc2` with a single sample corresponding to `Time` in `tsc1`.  
`tsc2 = getsampleusingtime(tsc1,StartTime,EndTime)` returns a new tscollection `tsc2` with samples between the times `StartTime` and `EndTime` in `tsc1`.

**Remarks**  
When the time vector in `ts1` is numeric, `StartTime` and `EndTime` must also be numeric. When the times in `ts1` are date strings and the `StartTime` and `EndTime` values are numeric, then the `StartTime` and `EndTime` values are treated as `datenum` values.

**See Also** `tscollection`



**Purpose** Cell array of names of `timeseries` objects in `tscollection` object

**Syntax** `names = gettimeseriesnames(tsc)`

**Description** `names = gettimeseriesnames(tsc)` returns names of `timeseries` objects in a `tscollection` object `tsc`. `names` is a cell array of strings.

**Examples** **1** Create `timeseries` objects `a` and `b`.

```
a = timeseries(rand(1000,1),'name','position');  
b = timeseries(rand(1000,1),'name','response');
```

**2** Create a `tscollection` object that includes these two time series.

```
tsc = tscollection({a,b});
```

**3** Get the names of the `timeseries` objects in `tsc`.

```
names = gettimeseriesnames(tsc)
```

```
names =  
  
    'position'    'response'
```

**See Also** `timeseries`, `tscollection`, `tsprops`

# gettsafteratevent

---

**Purpose** New timeseries object with samples occurring at or after event

**Syntax**  
`ts1 = gettsafteratevent(ts,event)`  
`ts1 = gettsafteratevent(ts,event,n)`

**Description** `ts1 = gettsafteratevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring at and after an event in `ts`, where `event` can be either a `tsdata.event` object or a string. When `event` is a `tsdata.event` object, the time defined by `event` is used. When `event` is a string, the first `tsdata.event` object in the `Events` property of the time series `ts` that matches the event name specifies the time.

`ts1 = gettsafteratevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples at and after an event in `ts`, where `n` is the number of the event occurrence with a matching event name.

**Remarks** When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

**See Also** `gettsafterevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

<b>Purpose</b>	New timeseries object with samples occurring after event
<b>Syntax</b>	<pre>ts1 = gettsafterevent(ts,event) ts1 = ttsafterevent(ts,event,n)</pre>
<b>Description</b>	<p><code>ts1 = gettsafterevent(ts,event)</code> returns a new <code>timeseries</code> object <code>ts1</code> with samples occurring after an event in <code>ts</code>, where <code>event</code> can be either a <code>tsdata.event</code> object or a string. When <code>event</code> is a <code>tsdata.event</code> object, the time defined by <code>event</code> is used. When <code>event</code> is a string, the first <code>tsdata.event</code> object in the <code>Events</code> property of <code>ts</code> that matches the event name specifies the time.</p> <p><code>ts1 = ttsafterevent(ts,event,n)</code> returns a new <code>timeseries</code> object <code>ts1</code> with samples occurring after an event in time series <code>ts</code>, where <code>n</code> is the number of the event occurrence with a matching event name.</p>
<b>Remarks</b>	<p>When the <code>timeseries</code> object <code>ts</code> contains date strings and <code>event</code> uses numeric time, the time selected by the event is treated as a date that is calculated relative to the <code>StartDate</code> property in <code>ts.TimeInfo</code>.</p> <p>When <code>ts</code> uses numeric time and <code>event</code> uses calendar dates, the time selected by the event is treated as a numeric value that is not associated with a calendar date.</p>
<b>See Also</b>	<code>gettsafteratevent</code> , <code>gettsbeforeevent</code> , <code>gettsbetweenevents</code> , <code>tsdata.event</code> , <code>tsprops</code>

# gettsatevent

---

**Purpose** New timeseries object with samples occurring at event

**Syntax**  
`ts1 = gettsatevent(ts,event)`  
`ts1 = gettsatevent(ts,event,n)`

**Description** `ts1 = gettsatevent(ts,event)` returns a new timeseries object `ts1` with samples occurring at an event in `ts`, where `event` can be either a `tsdata.event` object or a string. When `event` is a `tsdata.event` object, the time defined by `event` is used. When `event` is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsatevent(ts,event,n)` returns a new time series `ts1` with samples occurring at an event in time series `ts`, where `n` is the number of the event occurrence with a matching event name.

**Remarks** When the timeseries object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in the `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

**See Also** `gettsafterevent`, `gettsafteratevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

**Purpose** New `timeseries` object with samples occurring before or at event

**Syntax**

```
ts1 = gettsbeforeatevent(ts,event)
ts1 = gettsbeforeatevent(ts,event,n)
```

**Description** `ts1 = gettsbeforeatevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring at and before an event in `ts`, where `event` can be either a `tsdata.event` object or a string. When `event` is a `tsdata.event` object, the time defined by `event` is used. When `event` is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsbeforeatevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring at and before an event in time series `ts`, where `n` is the number of the event occurrence with a matching event name.

**Remarks** When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the event is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the event is treated as a numeric value that is not associated with a calendar date.

**See Also** `gettsafterevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

# gettsbeforeevent

---

**Purpose** New timeseries object with samples occurring before event

**Syntax**  
`ts1 = gettsbeforeevent(ts,event)`  
`ts1 = gettsbeforeevent(ts,event,n)`

**Description** `ts1 = gettsbeforeevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring before an event in `ts`, where `event` can be either a `tsdata.event` object or a string. When `event` is a `tsdata.event` object, the time defined by `event` is used. When `event` is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsbeforeevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring before an event in `ts`, where `n` is the number of the event occurrence with a matching event name.

**Remarks** When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

**See Also** `gettsafterevent`, `gettsbeforeatevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

**Purpose** New timeseries object with samples occurring between events

**Syntax**  
`ts1 = gettsbetweenevents(ts,event1,event2)`  
`ts1 = gettsbetweenevents(ts,event1,event2,n1,n2)`

**Description** `ts1 = gettsbetweenevents(ts,event1,event2)` returns a new timeseries object `ts1` with samples occurring between events in `ts`, where `event1` and `event2` can be either a `tsdata.event` object or a string. When `event1` and `event2` are `tsdata.event` objects, the time defined by the events is used. When `event1` and `event2` are strings, the first `tsdata.event` object in the `Events` property of `ts` that matches the event names specifies the time.

`ts1 = gettsbetweenevents(ts,event1,event2,n1,n2)` returns a new timeseries object `ts1` with samples occurring between events in `ts`, where `n1` and `n2` are the `n`th occurrences of the events with matching event names.

**Remarks** When the timeseries object `ts` contains date strings and `event` uses numeric time, the time selected by the event is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the event is treated as a numeric value that is not associated with a calendar date.

**See Also** `gettsafterevent`, `gettsbeforeevent`, `tsdata.event`, `tsprops`

# GetVariable

---

## Purpose

Get data from variable in server workspace

## Syntax

### MATLAB Client

```
D = h.GetVariable('varname', 'workspace')
D = GetVariable(h, 'varname', 'workspace')
D = invoke(h, 'GetVariable', 'varname', 'workspace')
```

### Method Signature

```
HRESULT GetVariable([in] BSTR varname, [in] BSTR workspace,
[out, retval] VARIANT* pdata)
```

### Microsoft Visual Basic Client

```
GetVariable(varname As String, workspace As String) As Object
```

## Description

`GetVariable` returns the data stored in the specified variable from the specified workspace of the server. Each syntax in the MATLAB Client section produce the same result. Note that the dot notation (`h.GetVariable`) is case sensitive.

`varname` from the specified workspace of the server that is attached to handle `h`. The *workspace* argument can be either `base` or `global`.

`varname` — the name of the variable whose data is returned

*workspace* — the workspace containing the variable can be either:

- `base` is the base workspace of the server
- `global` is the global workspace of the server (see `global` for more information about how to access variables in the global workspace).

---

**Note** `GetVariable` works on all MATLAB types except sparse arrays, structures, and function handles.

---

## Remarks

You can use `GetVariable` in place of `GetWorkspaceData`, `GetFullMatrix` and `GetCharArray` to get data stored in workspace



variables when you need a result returned explicitly (which might be required by some scripting languages).

## Examples

This example assigns a cell array to the variable C1 in the base workspace of the server, and then read it back with `GetVariable`, assigning it to a new variable C2.

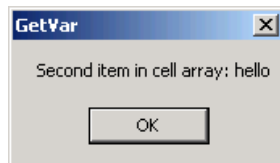
### MATLAB Client

```
h = actxserver('matlab.application');
h.PutWorkspaceData('C1', 'base', {25.72, 'hello', rand(4)});
C2 = h.GetVariable('C1','base')
C2 =
    [25.7200]    'hello'    [4x4 double]
```

### Visual Basic .NET Client

```
Dim Matlab As Object
Dim Result As String
Dim C2 As Object
Matlab = CreateObject("matlab.application")
Result = Matlab.Execute("C1 = {25.72, 'hello', rand(4)};")
C2 = Matlab.GetVariable("C1", "base")
MsgBox("Second item in cell array: " & C2(0, 1))
```

The Visual Basic Client example creates a message box displaying the second element in the cell array, which is the string `hello`.



## See Also

`GetWorkspaceData`, `PutWorkspaceData`, `GetFullMatrix`, `PutFullMatrix`, `GetCharArray`, `PutCharArray`, `Execute`

# GetWorkspaceData

---

**Purpose** Get data from server workspace

**Syntax** **MATLAB Client**  
D = h.GetWorkspaceData('varname', 'workspace')  
D = GetWorkspaceData(h, 'varname', 'workspace')  
D = invoke(h, 'GetWorkspaceData', 'varname', 'workspace')

**Method Signature**  
HRESULT GetWorkspaceData([in] BSTR varname, [in] BSTR workspace, [out] VARIANT\* pdata)

**Microsoft Visual Basic Client**  
GetWorkspaceData(varname As String, workspace As String) As Object

**Description** GetWorkspaceData gets the data stored in the variable varname from the specified workspace of the server attached to handle h and returns it in output argument D. The *workspace* argument can be either *base* or *global*.

---

**Note** GetWorkspaceData works on all MATLAB types except sparse arrays, structures, and function handles.

---

**Remarks** You can use GetWorkspaceData in place of GetFullMatrix and GetCharArray to get numeric and character array data respectively. If you want output from GetWorkspaceData to be displayed at the client window, you must specify an output variable. Server function names, like GetWorkspaceData, are case sensitive when using the first syntax shown. There is no difference in the operation of the three syntaxes shown above for the MATLAB client. The GetWorkspaceData and PutWorkspaceData functions pass numeric data as a variant data type. These functions are especially useful for

VBScript clients as VBScript does not support the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`.

## Examples

Assign a cell array to variable `C1` in the base workspace of the server, and then read it back with `GetWorkspaceData`.

### MATLAB Client

```
h = actxserver('matlab.application');
h.PutWorkspaceData('C1', 'base', ...
    {25.72, 'hello', rand(4)});
C2 = h.GetWorkspaceData('C1', 'base')

C2 =
    [25.7200]    'hello'    [4x4 double]
```

### Visual Basic .NET Client

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```
Dim Matlab, C2 As Object
Dim Result As String
Matlab = CreateObject("matlab.application")
Result = MatLab.Execute("C1 = {25.72, 'hello', rand(4)};")
MsgBox("In MATLAB, type" & vbCrLf & "C1")
Matlab.GetWorkspaceData("C1", "base", C2)
MsgBox("second value of C1 = " & C2(0, 1))
```

## See Also

`PutWorkspaceData`, `GetFullMatrix`, `PutFullMatrix`, `GetCharArray`, `PutCharArray`, `GetVariable`, `Execute`

# ginput

---

**Purpose** Graphical input from mouse or cursor

**Syntax**  
`[x,y] = ginput(n)`  
`[x,y] = ginput`  
`[x,y,button] = ginput(...)`

**Description** `ginput` enables you to select points from the figure using the mouse for cursor positioning. The figure must have focus before `ginput` receives input.

`[x,y] = ginput(n)` enables you to select `n` points from the current axes and returns the  $x$ - and  $y$ -coordinates in the column vectors `x` and `y`, respectively. Press the **Return** key to terminate the input before entering `n` points.

`[x,y] = ginput` gathers an unlimited number of points until you press the **Return** key.

---

**Note** Clicking an axes makes that axes the current axes. Although you may set the current axes before calling `ginput`, whichever axes the user clicks becomes the current axes and `ginput` returns points relative to that axes. For example, if a user selects points from multiple axes, the results returned are relative to the different axes' coordinate systems.

---

`[x,y,button] = ginput(...)` returns the  $x$ -coordinates, the  $y$ -coordinates, and the button or key designation. `button` is a vector of integers indicating which mouse buttons you pressed (1 for left, 2 for middle, 3 for right), or ASCII numbers indicating which keys on the keyboard you pressed.

**Examples** Pick 10 two-dimensional points from the figure window.

```
[x,y] = ginput(10)
```

Position the cursor with the mouse. Enter data points by pressing a mouse button or a key on the keyboard. To terminate input before entering 10 points, press the **Return** key.

## See Also

`gtext`

“Interactive Plotting” for an example

“User Interface Development” on page 1-109 for related functions

# global

---

**Purpose** Declare global variables

**Syntax** `global X Y Z`

**Description** `global X Y Z` defines X, Y, and Z as global in scope.

Ordinarily, each MATLAB function, defined by an M-file, has its own local variables, which are separate from those of other functions, and from those of the base workspace. However, if several functions, and possibly the base workspace, all declare a particular name as global, they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the functions declaring it global.

If the global variable does not exist the first time you issue the `global` statement, it is initialized to the empty matrix.

If a variable with the same name as the global variable already exists in the current workspace, MATLAB issues a warning and changes the value of that variable to match the global.

## Remarks

Use `clear global variable` to clear a global variable from the global workspace. Use `clear variable` to clear the global link from the current workspace without affecting the value of the global.

To use a global within a callback, declare the global, use it, then clear the global link from the workspace. This avoids declaring the global after it has been referenced. For example,

```
cbstr = sprintf('%s, %s, %s, %s, %s', ...
    'global MY_GLOBAL', ...
    'MY_GLOBAL = 100', ...
    'disp(MY_GLOBAL)', ...
    'MY_GLOBAL = MY_GLOBAL+1', ...
    'clear MY_GLOBAL');

uicontrol('style', 'pushbutton', 'CallBack', cbstr, ...
    'string', 'count')
```

There is no function form of the `global` command (i.e., you cannot use parentheses and quote the variable names).

## Examples

Here is the code for the functions `tic` and `toc` (some comments abridged). These functions manipulate a stopwatch-like timer. The global variable `TICTOC` is shared by the two functions, but it is invisible in the base workspace or in any other functions that do not declare it.

```
function tic
%   TIC Start a stopwatch timer.
%       TIC; any stuff; TOC
%   prints the time required.
%   See also: TOC, CLOCK.
global TICTOC
TICTOC = clock;

function t = toc
%   TOC Read the stopwatch timer.
%   TOC prints the elapsed time since TIC was used.
%   t = TOC; saves elapsed time in t, does not print.
%   See also: TIC, ETIME.
global TICTOC
if nargin < 1
    elapsed_time = etime(clock, TICTOC)
else
    t = etime(clock, TICTOC);
end
```

## See Also

`clear`, `isglobal`, `who`

**Purpose** Generalized minimum residual method (with restarts)

**Syntax**

```
x = gmres(A,b)
gmres(A,b,restart)
gmres(A,b,restart,tol)
gmres(A,b,restart,tol,maxit)
gmres(A,b,restart,tol,maxit,M)
gmres(A,b,restart,tol,maxit,M1,M2)
gmres(A,b,restart,tol,maxit,M1,M2,x0)
[x,flag] = gmres(A,b,...)
[x,flag,relres] = gmres(A,b,...)
[x,flag,relres,iter] = gmres(A,b,...)
[x,flag,relres,iter,resvec] = gmres(A,b,...)
```

**Description** `x = gmres(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. For this syntax, `gmres` does not restart; the maximum number of iterations is  $\min(n, 10)$ .

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

`gmres(A,b,restart)` restarts the method every `restart` inner iterations. The maximum number of outer iterations is  $\min(n/\text{restart}, 10)$ . The maximum number of total iterations is  $\text{restart} * \min(n/\text{restart}, 10)$ . If `restart` is `n` or `[]`, then `gmres` does not restart and the maximum number of total iterations is  $\min(n, 10)$ .



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

`gmres(A,b,restart,tol,maxit)` specifies the maximum number of outer iterations, i.e., the total number of iterations does not exceed `restart*maxit`. If `maxit` is `[]` then `gmres` uses the default,  $\min(n/\text{restart}, 10)$ . If `restart` is `n` or `[]`, then the maximum number of total iterations is `maxit` (instead of `restart*maxit`).

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

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

`[x,flag] = gmres(A,b,...)` also returns a convergence flag:

- `flag = 0`    `gmres` converged to the desired tolerance `tol` within `maxit` outer iterations.
- `flag = 1`    `gmres` iterated `maxit` times but did not converge.
- `flag = 2`    Preconditioner `M` was ill-conditioned.
- `flag = 3`    `gmres` stagnated. (Two consecutive iterates were the same.)

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

`[x,flag,relres,iter] = gmres(A,b,...)` also returns both the outer and inner iteration numbers at which `x` was computed, where  $0 \leq \text{iter}(1) \leq \text{maxit}$  and  $0 \leq \text{iter}(2) \leq \text{restart}$ .

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

## Examples

### Example 1

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = gmres(A,b,10,tol,maxit,M1);
```

displays the following message:

```
gmres(10) converged at outer iteration 2 (inner iteration 9) to
a solution with relative residual 3.3e-013
```

### 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_gmres` that

- Calls `gmres` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in `run_gmres` are available to `afun` and `mfun`.

The following shows the code for `run_gmres`:

```
function x1 = run_gmres
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
tol = 1e-12; maxit = 15;
x1 = gmres(@afun,b,10,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_gmres;
```

MATLAB software displays the message

```
gmres(10) converged at outer iteration 2 (inner iteration 9) to
a solution with relative residual 3.3e-013
```

### Example 3

```
load west0479
A = west0479
b = sum(A,2)
[x,flag] = gmres(A,b,5)
```

flag is 1 because gmres does not converge to the default tolerance  $1e-6$  within the default 10 outer iterations.

```
[L1,U1] = luinc(A,1e-5);
[x1,flag1] = gmres(A,b,5,1e-6,5,L1,U1);
```

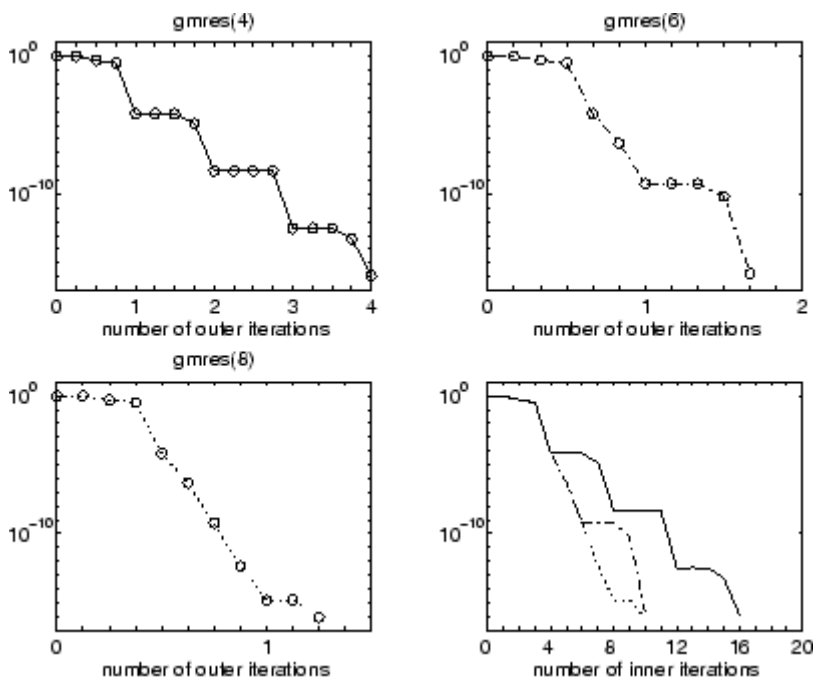
flag1 is 2 because the upper triangular U1 has a zero on its diagonal, and gmres fails in the first iteration when it tries to solve a system such as  $U1*y = r$  for y using backslash.

```
[L2,U2] = luinc(A,1e-6);
```

# gmres

```
tol = 1e-15;  
[x4,flag4,relres4,iter4,resvec4] = gmres(A,b,4,tol,5,L2,U2);  
[x6,flag6,relres6,iter6,resvec6] = gmres(A,b,6,tol,3,L2,U2);  
[x8,flag8,relres8,iter8,resvec8] = gmres(A,b,8,tol,3,L2,U2);
```

flag4, flag6, and flag8 are all 0 because gmres converged when restarted at iterations 4, 6, and 8 while preconditioned by the incomplete LU factorization with a drop tolerance of  $1e-6$ . This is verified by the plots of outer iteration number against relative residual. A combined plot of all three clearly shows the restarting at iterations 4 and 6. The total number of iterations computed may be more for lower values of restart, but the number of length  $n$  vectors stored is fewer, and the amount of work done in the method decreases proportionally.



## See Also

bicg, bicgstab, cgs, lsqr, ilu, luinc, minres, pcg, qmr, symmlq

---

function\_handle (@), mldivide (\)

## References

Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

Saad, Youcef and Martin H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, July 1986, Vol. 7, No. 3, pp. 856-869.

# gplot

---

**Purpose** Plot nodes and links representing adjacency matrix

**Syntax** `gplot(A,Coordinates)`  
`gplot(A,Coordinates,LineStyle)`

**Description** The `gplot` function graphs a set of coordinates using an adjacency matrix.

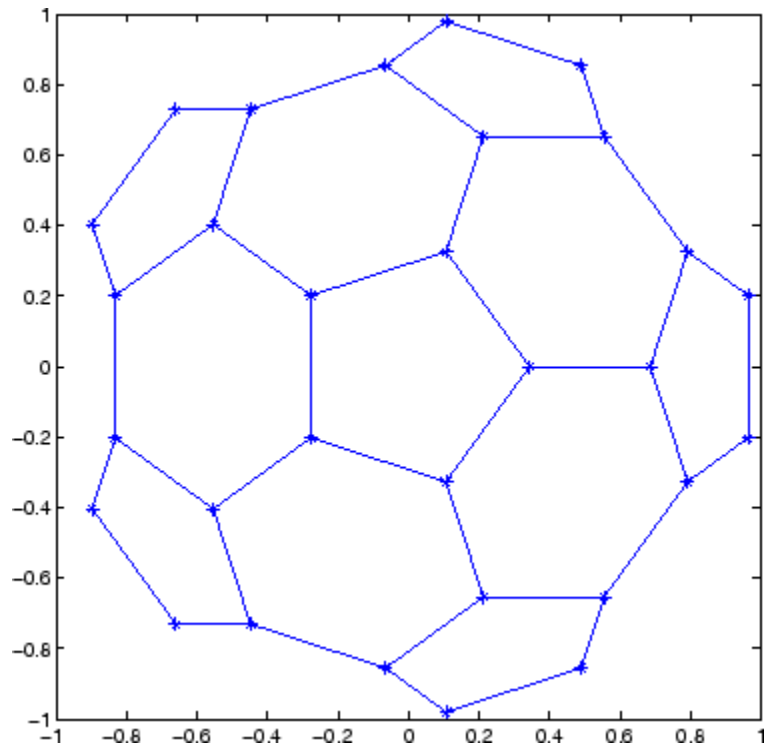
`gplot(A,Coordinates)` plots a graph of the nodes defined in `Coordinates` according to the  $n$ -by- $n$  adjacency matrix `A`, where  $n$  is the number of nodes. `Coordinates` is an  $n$ -by-2 matrix, where  $n$  is the number of nodes and each coordinate pair represents one node.

`gplot(A,Coordinates,LineStyle)` plots the nodes using the line type, marker symbol, and color specified by `LineStyle`.

**Remarks** For two-dimensional data, `Coordinates(i,:) = [x(i) y(i)]` denotes node  $i$ , and `Coordinates(j,:) = [x(j)y(j)]` denotes node  $j$ . If node  $i$  and node  $j$  are connected, `A(i,j)` or `A(j,i)` is nonzero; otherwise, `A(i,j)` and `A(j,i)` are zero.

**Examples** To draw half of a Bucky ball with asterisks at each node,

```
k = 1:30;  
[B,XY] = bucky;  
gplot(B(k,k),XY(k,:), '-*')  
axis square
```

**See Also**

LineStyleSpec, sparse, spy

“Tree Operations” on page 1-40 for related functions

# grabcode

---

**Purpose** MATLAB code from M-files published to HTML

**Syntax**

```
grabcode('name.html')
grabcode('urlname')
codeString = grabcode('name.html')
```

**Description** `grabcode('name.html')` copies MATLAB code from the file `name.html` and pastes it into an untitled document in the Editor. Use `grabcode` to get MATLAB code from demos or other published M-files when the M-file source code is not readily available. The file `name.html` was created by publishing `name.m`, an M-file containing cells. The MATLAB code from `name.m` is included at the end of `name.html` as HTML comments.

`grabcode('urlname')` copies MATLAB code from the `urlname` location and pastes it into an untitled document in the Editor.

`codeString = grabcode('name.html')` get MATLAB code from the file `name.html` and assigns it the variable `codeString`.

**Examples** Run

```
sineWaveString = grabcode('d:/mymfiles/sine_wave_.html')
```

and MATLAB displays

```
sineWaveString =

%% Simple Sine Wave Plot

%% Part One: Calculate Sine Wave
% Define the range |x|.
% Calculate the sine |y| over that range.
x = 0:.01:6*pi;
y = sin(x);

%% Part Two: Plot Sine Wave
% Graph the result.
```



`plot(x,y)`

**See Also**

demo, publish

# gradient

---

**Purpose** Numerical gradient

**Syntax**  
FX = gradient(F)  
[FX,FY] = gradient(F)  
[FX,FY,FZ,...] = gradient(F)  
[...] = gradient(F,h)  
[...] = gradient(F,h1,h2,...)

**Definition** The *gradient* of a function of two variables,  $F(x, y)$ , is defined as

$$\nabla F = \frac{\partial F}{\partial x} \hat{i} + \frac{\partial F}{\partial y} \hat{j}$$

and can be thought of as a collection of vectors pointing in the direction of increasing values of  $F$ . In MATLAB software, numerical gradients (differences) can be computed for functions with any number of variables. For a function of  $N$  variables,  $F(x, y, z, \dots)$ ,

$$\nabla F = \frac{\partial F}{\partial x} \hat{i} + \frac{\partial F}{\partial y} \hat{j} + \frac{\partial F}{\partial z} \hat{k} + \dots$$

**Description** FX = gradient(F) where F is a vector returns the one-dimensional numerical gradient of F. FX corresponds to  $\partial F / \partial x$ , the differences in x (horizontal) direction.

[FX,FY] = gradient(F) where F is a matrix returns the  $x$  and  $y$  components of the two-dimensional numerical gradient. FX corresponds to  $\partial F / \partial x$ , the differences in  $x$  (horizontal) direction. FY corresponds to  $\partial F / \partial y$ , the differences in the  $y$  (vertical) direction. The spacing between points in each direction is assumed to be one.

[FX,FY,FZ,...] = gradient(F) where F has N dimensions returns the N components of the gradient of F. There are two ways to control the spacing between values in F:

- A single spacing value, h, specifies the spacing between points in every direction.

- N spacing values ( $h_1, h_2, \dots$ ) specifies the spacing for each dimension of F. Scalar spacing parameters specify a constant spacing for each dimension. Vector parameters specify the coordinates of the values along corresponding dimensions of F. In this case, the length of the vector must match the size of the corresponding dimension.

---

**Note** The first output FX is always the gradient along the 2nd dimension of F, going across columns. The second output FY is always the gradient along the 1st dimension of F, going across rows. For the third output FZ and the outputs that follow, the Nth output is the gradient along the Nth dimension of F.

---

[...] = gradient(F,h) where h is a scalar uses h as the spacing between points in each direction.

[...] = gradient(F,h1,h2,...) with N spacing parameters specifies the spacing for each dimension of F.

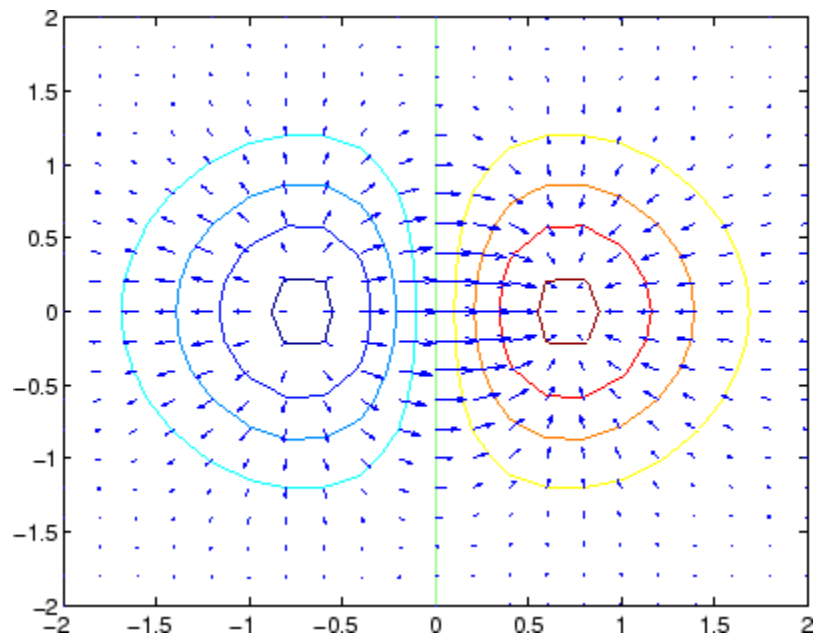
## Examples

The statements

```
v = -2:0.2:2;
[x,y] = meshgrid(v);
z = x .* exp(-x.^2 - y.^2);
[px,py] = gradient(z,.2,.2);
contour(v,v,z), hold on, quiver(v,v,px,py), hold off
```

produce

# gradient



Given,

```
F(:,:,1) = magic(3); F(:,:,2) = pascal(3);  
gradient(F)
```

takes  $dx = dy = dz = 1$ .

```
[PX,PY,PZ] = gradient(F,0.2,0.1,0.2)
```

takes  $dx = 0.2$ ,  $dy = 0.1$ , and  $dz = 0.2$ .


## See Also

del2, diff

<b>Purpose</b>	Set default figure properties for grayscale monitors
<b>Syntax</b>	<code>graymon</code>
<b>Description</b>	<code>graymon</code> sets defaults for graphics properties to produce more legible displays for grayscale monitors.
<b>See Also</b>	<code>axes</code> , <code>figure</code> “Color Operations” on page 1-102 for related functions

# grid

---

<b>Purpose</b>	Grid lines for 2-D and 3-D plots
<b>GUI Alternative</b>	To control the presence and appearance of grid lines on a graph, use the Property Editor, one of the plotting tools  . For details, see The Property Editor in the MATLAB Graphics documentation.
<b>Syntax</b>	<pre>grid on grid off grid grid(axes_handle,...) grid minor</pre>
<b>Description</b>	<p>The <code>grid</code> function turns the current axes' grid lines on and off.</p> <p><code>grid on</code> adds major grid lines to the current axes.</p> <p><code>grid off</code> removes major and minor grid lines from the current axes.</p> <p><code>grid</code> toggles the major grid visibility state.</p> <p><code>grid(axes_handle,...)</code> uses the axes specified by <code>axes_handle</code> instead of the current axes.</p>
<b>Algorithm</b>	<p><code>grid</code> sets the <code>XGrid</code>, <code>YGrid</code>, and <code>ZGrid</code> properties of the axes.</p> <p><code>grid minor</code> sets the <code>XMinorGrid</code>, <code>YMinorGrid</code>, and <code>ZMinorGrid</code> properties of the axes.</p> <p>You can set the grid lines for just one axis using the <code>set</code> command and the individual property. For example,</p> <pre>set(axes_handle, 'XGrid', 'on')</pre> <p>turns on only <i>x</i>-axis grid lines.</p> <p>You can set grid line width with the axes <code>LineWidth</code> property.</p>
<b>See Also</b>	<p><code>box</code>, <code>axes</code>, <code>set</code></p> <p>The properties of axes objects</p>

“Axes Operations” on page 1-100 for related functions

# griddata

---

**Purpose** Data gridding

**Syntax**  
`ZI = griddata(x,y,z,XI,YI)`  
`[XI,YI,ZI] = griddata(x,y,z,XI,YI)`  
`[...] = griddata(...,method)`  
`[...] = griddata(...,method,options)`

**Description** `ZI = griddata(x,y,z,XI,YI)` fits a surface of the form  $z = f(x,y)$  to the data in the (usually) nonuniformly spaced vectors  $(x,y,z)$ . `griddata` interpolates this surface at the points specified by  $(XI,YI)$  to produce `ZI`. The surface always passes through the data points. `XI` and `YI` usually form a uniform grid (as produced by `meshgrid`).

`XI` can be a row vector, in which case it specifies a matrix with constant columns. Similarly, `YI` can be a column vector, and it specifies a matrix with constant rows.

`[XI,YI,ZI] = griddata(x,y,z,XI,YI)` returns the interpolated matrix `ZI` as above, and also returns the matrices `XI` and `YI` formed from row vector `XI` and column vector `yi`. These latter are the same as the matrices returned by `meshgrid`.

`[...] = griddata(...,method)` uses the specified interpolation method:

'linear'	Triangle-based linear interpolation (default)
'cubic'	Triangle-based cubic interpolation
'nearest'	Nearest neighbor interpolation
'v4'	MATLAB 4 griddata method

The method defines the type of surface fit to the data. The 'cubic' and 'v4' methods produce smooth surfaces while 'linear' and 'nearest' have discontinuities in the first and zero'th derivatives, respectively. All the methods except 'v4' are based on a Delaunay triangulation of the data. If method is [], then the default 'linear' method is used.



[...] = griddata(...,method,options) specifies a cell array of strings options to be used in Qhull via delaunayn. If options is [], the default delaunayn options are used. If options is {' '}, no options are used, not even the default.

Occasionally, griddata might return points on or very near the convex hull of the data as NaNs. This is because roundoff in the computations sometimes makes it difficult to determine if a point near the boundary is in the convex hull.

## Remarks

XI and YI can be matrices, in which case griddata returns the values for the corresponding points (XI(i,j),YI(i,j)). Alternatively, you can pass in the row and column vectors xi and yi, respectively. In this case, griddata interprets these vectors as if they were matrices produced by the command meshgrid(xi,yi).

## Examples

Sample a function at 100 random points between  $\pm 2.0$ :

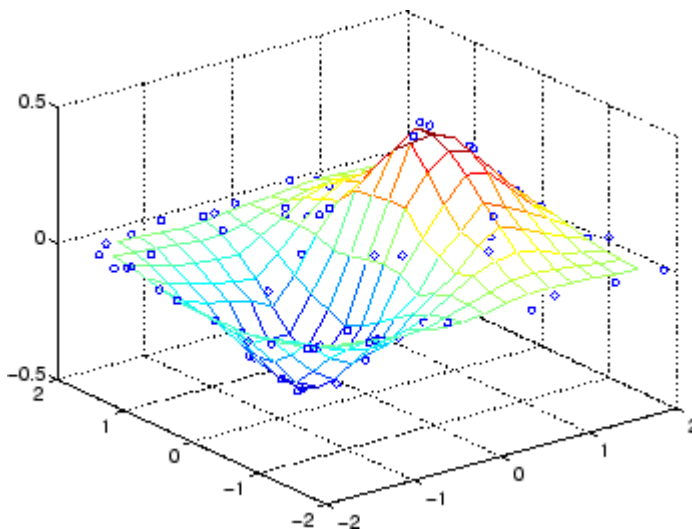
```
rand('seed',0)
x = rand(100,1)*4-2; y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

x, y, and z are now vectors containing nonuniformly sampled data. Define a regular grid, and grid the data to it:

```
ti = -2:.25:2;
[XI,YI] = meshgrid(ti,ti);
ZI = griddata(x,y,z,XI,YI);
```

Plot the gridded data along with the nonuniform data points used to generate it:

```
mesh(XI,YI,ZI), hold
plot3(x,y,z,'o'), hold off
```



## Algorithm

The `griddata(..., 'v4')` command uses the method documented in [2]. The other `griddata` methods are based on a Delaunay triangulation of the data that uses Qhull [1]. For information about Qhull, see <http://www.qhull.org/>. For copyright information, see <http://www.qhull.org/COPYING.txt>.

## See Also

`delaunay`, `griddata3`, `griddatan`, `interp2`, `meshgrid`

## References

- [1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469–483. Available in PDF format at <http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/>.
- [2] Sandwell, David T., "Biharmonic Spline Interpolation of GEOS-3 and SEASAT Altimeter Data", *Geophysical Research Letters*, 14, 2, 139–142, 1987.

[3] Watson, David E., *Contouring: A Guide to the Analysis and Display of Spatial Data*, Tarrytown, NY: Pergamon (Elsevier Science, Inc.): 1992.

# griddata3

---

**Purpose** Data gridding and hypersurface fitting for 3-D data

**Syntax**

```
w = griddata3(x,y,z,v,xi,yi,zi)
w = griddata3(x,y,z,v,xi,yi,zi,method)
w = griddata3(x,y,z,v,xi,yi,zi,method,options)
```

**Description** `w = griddata3(x,y,z,v,xi,yi,zi)` fits a hypersurface of the form  $w = f(x, y, z)$  to the data in the (usually) nonuniformly spaced vectors (x, y, z, v). `griddata3` interpolates this hypersurface at the points specified by (xi,yi,zi) to produce w. w is the same size as xi, yi, and zi.

(xi,yi,zi) is usually a uniform grid (as produced by `meshgrid`) and is where `griddata3` gets its name.

`w = griddata3(x,y,z,v,xi,yi,zi,method)` defines the type of surface that is fit to the data, where `method` is either:

'linear'            Tesselation-based linear interpolation (default)  
'nearest'          Nearest neighbor interpolation

If `method` is [], the default 'linear' method is used.

`w = griddata3(x,y,z,v,xi,yi,zi,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`.

If `options` is [], the default options are used. If `options` is {''}, no options are used, not even the default.

**Examples** Create vectors x, y, and z containing nonuniformly sampled data:

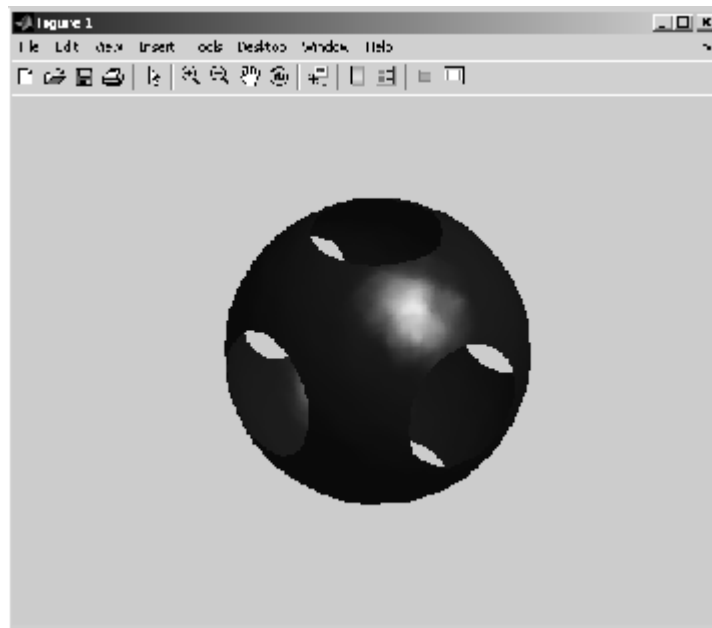
```
rand('state',0);
x = 2*rand(5000,1)-1;
y = 2*rand(5000,1)-1;
z = 2*rand(5000,1)-1;
v = x.^2 + y.^2 + z.^2;
```

Define a regular grid, and grid the data to it:

```
d = -0.8:0.05:0.8;  
[xi,yi,zi] = meshgrid(d,d,d);  
w = griddata3(x,y,z,v,xi,yi,zi);
```

Since it is difficult to visualize 4D data sets, use isosurface at 0.8:

```
p = patch(isosurface(xi,yi,zi,w,0.8));  
isonormals(xi,yi,zi,w,p);  
set(p,'FaceColor','blue','EdgeColor','none');  
view(3), axis equal, axis off, camlight, lighting phong
```



## Algorithm

The `griddata3` methods are based on a Delaunay triangulation of the data that uses Qhull [1]. For information about Qhull, see <http://www.qhull.org/>. For copyright information, see <http://www.qhull.org/COPYING.txt>.

# griddata3

---

## See Also

deleunayn, griddata, griddatan, meshgrid

## Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in PDF format at <http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/>.

**Purpose**

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

**Syntax**

```
yi = griddatan(X,y,xi)
yi = griddatan(x,y,z,v,xi,yi,zi,method)
```

**Description**

`yi = griddatan(X,y,xi)` fits a hyper-surface of the form  $y = f(X)$  to the data in the (usually) nonuniformly-spaced vectors  $(X, y)$ . `griddatan` interpolates this hyper-surface at the points specified by `xi` to produce `yi`. `xi` can be nonuniform.

$X$  is of dimension  $m$ -by- $n$ , representing  $m$  points in  $n$ -dimensional space.  $y$  is of dimension  $m$ -by-1, representing  $m$  values of the hyper-surface  $f(X)$ . `xi` is a vector of size  $p$ -by- $n$ , representing  $p$  points in the  $n$ -dimensional space whose surface value is to be fitted. `yi` is a vector of length  $p$  approximating the values  $f(xi)$ . The hypersurface always goes through the data points  $(X,y)$ . `xi` is usually a uniform grid (as produced by `meshgrid`).

`yi = griddatan(x,y,z,v,xi,yi,zi,method)` defines the type of surface fit to the data, where 'method' is one of:

'linear'      Tessellation-based linear interpolation (default)  
'nearest'     Nearest neighbor interpolation

All the methods are based on a Delaunay tessellation of the data.

If `method` is `[]`, the default 'linear' method is used.

`yi = griddatan(x,y,z,v,xi,yi,zi,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`.

If `options` is `[]`, the default options are used. If `options` is `{' '}`, no options are used, not even the default.

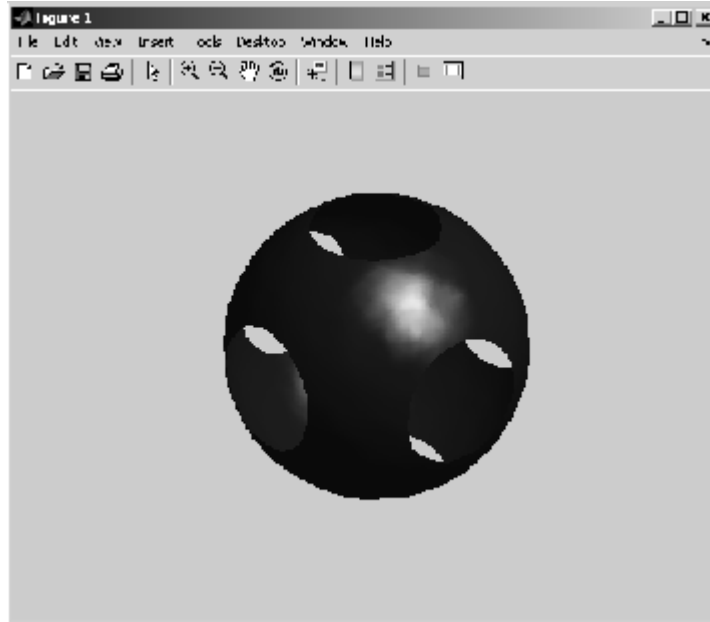
**Examples**

```
rand('state',0)
X = 2*rand(5000,3)-1;
Y = sum(X.^2,2);
d = -0.8:0.05:0.8;
```

```
[y0,x0,z0] = ndgrid(d,d,d);  
XI = [x0(:) y0(:) z0(:)];  
YI = griddatan(X,Y,XI);
```

Since it is difficult to visualize 4D data sets, use isosurface at 0.8:

```
YI = reshape(YI, size(x0));  
p = patch(isosurface(x0,y0,z0,YI,0.8));  
isonormals(x0,y0,z0,YI,p);  
set(p,'FaceColor','blue','EdgeColor','none');  
view(3), axis equal, axis off, camlight, lighting phong
```



## Algorithm

The griddatan methods are based on a Delaunay triangulation of the data that uses Qhull [1]. For information about Qhull, see <http://www.qhull.org/>. For copyright information, see <http://www.qhull.org/COPYING.txt>.



**See Also**

deleunayn, griddata, griddata3, meshgrid

**Reference**

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in PDF format at <http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/>.

**Purpose** Generalized singular value decomposition

**Syntax**  $[U,V,X,C,S] = \text{gsvd}(A,B)$   
 $\text{sigma} = \text{gsvd}(A,B)$

**Description**  $[U,V,X,C,S] = \text{gsvd}(A,B)$  returns unitary matrices  $U$  and  $V$ , a (usually) square matrix  $X$ , and nonnegative diagonal matrices  $C$  and  $S$  so that

$$\begin{aligned}A &= U * C * X' \\ B &= V * S * X' \\ C' * C + S' * S &= I\end{aligned}$$

$A$  and  $B$  must have the same number of columns, but may have different numbers of rows. If  $A$  is  $m$ -by- $p$  and  $B$  is  $n$ -by- $p$ , then  $U$  is  $m$ -by- $m$ ,  $V$  is  $n$ -by- $n$  and  $X$  is  $p$ -by- $q$  where  $q = \min(m+n, p)$ .

$\text{sigma} = \text{gsvd}(A,B)$  returns the vector of generalized singular values,  $\text{sqrt}(\text{diag}(C' * C) ./ \text{diag}(S' * S))$ .

The nonzero elements of  $S$  are always on its main diagonal. If  $m \geq p$  the nonzero elements of  $C$  are also on its main diagonal. But if  $m < p$ , the nonzero diagonal of  $C$  is  $\text{diag}(C, p-m)$ . This allows the diagonal elements to be ordered so that the generalized singular values are nondecreasing.

$\text{gsvd}(A,B,0)$ , with three input arguments and either  $m$  or  $n \geq p$ , produces the “economy-sized” decomposition where the resulting  $U$  and  $V$  have at most  $p$  columns, and  $C$  and  $S$  have at most  $p$  rows. The generalized singular values are  $\text{diag}(C) ./ \text{diag}(S)$ .

When  $B$  is square and nonsingular, the generalized singular values,  $\text{gsvd}(A,B)$ , are equal to the ordinary singular values,  $\text{svd}(A/B)$ , but they are sorted in the opposite order. Their reciprocals are  $\text{gsvd}(B,A)$ .

In this formulation of the  $\text{gsvd}$ , no assumptions are made about the individual ranks of  $A$  or  $B$ . The matrix  $X$  has full rank if and only if the matrix  $[A;B]$  has full rank. In fact,  $\text{svd}(X)$  and  $\text{cond}(X)$  are equal to  $\text{svd}([A;B])$  and  $\text{cond}([A;B])$ . Other formulations, eg. G. Golub and

C. Van Loan [1], require that  $\text{null}(A)$  and  $\text{null}(B)$  do not overlap and replace  $X$  by  $\text{inv}(X)$  or  $\text{inv}(X')$ .

Note, however, that when  $\text{null}(A)$  and  $\text{null}(B)$  do overlap, the nonzero elements of  $C$  and  $S$  are not uniquely determined.

## Examples

### Example 1

The matrices have at least as many rows as columns.

```
A = reshape(1:15,5,3)
```

```
B = magic(3)
```

```
A =
```

```

  1     6    11
  2     7    12
  3     8    13
  4     9    14
  5    10    15
```

```
B =
```

```

  8     1     6
  3     5     7
  4     9     2
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 5-by-5 orthogonal  $U$ , a 3-by-3 orthogonal  $V$ , a 3-by-3 nonsingular  $X$ ,

```
X =
```

```

  2.8284  -9.3761  -6.9346
 -5.6569  -8.3071 -18.3301
  2.8284  -7.2381 -29.7256
```

and

```
C =
```

```

  0.0000         0         0
```

$$S = \begin{bmatrix} 0 & 0.3155 & 0 \\ 0 & 0 & 0.9807 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1.0000 & 0 & 0 \\ 0 & 0.9489 & 0 \\ 0 & 0 & 0.1957 \end{bmatrix}$$

Since A is rank deficient, the first diagonal element of C is zero.

The economy sized decomposition,

$$[U, V, X, C, S] = \text{gsvd}(A, B, 0)$$

produces a 5-by-3 matrix U and a 3-by-3 matrix C.

$$U = \begin{bmatrix} 0.5700 & -0.6457 & -0.4279 \\ -0.7455 & -0.3296 & -0.4375 \\ -0.1702 & -0.0135 & -0.4470 \\ 0.2966 & 0.3026 & -0.4566 \\ 0.0490 & 0.6187 & -0.4661 \end{bmatrix}$$

$$C = \begin{bmatrix} 0.0000 & 0 & 0 \\ 0 & 0.3155 & 0 \\ 0 & 0 & 0.9807 \end{bmatrix}$$

The other three matrices, V, X, and S are the same as those obtained with the full decomposition.

The generalized singular values are the ratios of the diagonal elements of C and S.

$$\begin{aligned} \text{sigma} &= \text{gsvd}(A, B) \\ \text{sigma} &= \\ &0.0000 \\ &0.3325 \end{aligned}$$

5.0123

These values are a reordering of the ordinary singular values

```
svd(A/B)
ans =
    5.0123
    0.3325
    0.0000
```

## Example 2

The matrices have at least as many columns as rows.

```
A = reshape(1:15,3,5)
B = magic(5)
A =
     1     4     7    10    13
     2     5     8     9    14
     3     6     9    12    15
B =
    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
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 3-by-3 orthogonal U, a 5-by-5 orthogonal V, a 5-by-5 nonsingular X and

```
C =
     0     0    0.0000     0     0
     0     0     0    0.0439     0
     0     0     0     0    0.7432
```

```
S =
    1.0000    0    0    0    0
         0    1.0000    0    0    0
         0    0    1.0000    0    0
         0    0    0    0.9990    0
         0    0    0    0    0.6690
```

In this situation, the nonzero diagonal of C is `diag(C,2)`. The generalized singular values include three zeros.

```
sigma = gsvd(A,B)
sigma =
    0
    0
    0.0000
    0.0439
    1.1109
```

Reversing the roles of A and B reciprocates these values, producing two infinities.

```
gsvd(B,A)
ans =
    1.0e+016 *
    0.0000
    0.0000
    4.4126
    Inf
    Inf
```

## Algorithm

The generalized singular value decomposition uses the C-S decomposition described in [1], as well as the built-in `svd` and `qr` functions. The C-S decomposition is implemented in a subfunction in the `gsvd` M-file.

## Diagnostics

The only warning or error message produced by `gsvd` itself occurs when the two input arguments do not have the same number of columns.

**See Also**

qr, svd

**References**

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

**Purpose** Test for greater than

**Syntax** `A > B`  
`gt(A, B)`

**Description** `A > B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (**true**) where `A` is greater than `B`, or set to logical 0 (**false**) where `A` is less than or equal to `B`. 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.

`gt(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 greater than the corresponding elements of `B`:

```
A = magic(6);  
B = repmat(3*magic(3), 2, 2);
```

```
A > B  
ans =  
     1     0     0     1     1     1  
     0     1     0     1     1     1  
     1     0     0     1     0     1  
     0     1     1     0     1     0
```



---

1	0	1	1	0	0
0	1	1	1	0	1

**See Also**     lt, ge, le, ne, eq, “Relational Operators”

# gtext

---

## Purpose

Mouse placement of text in 2-D view

## Syntax

```
gtext('string')
gtext({'string1','string2','string3',...})
gtext({'string1';'string2';'string3';...})
h = gtext(...)
```

## Description

`gtext` displays a text string in the current figure window after you select a location with the mouse.

`gtext('string')` waits for you to press a mouse button or keyboard key while the pointer is within a figure window. Pressing a mouse button or any key places `'string'` on the plot at the selected location.

`gtext({'string1','string2','string3',...})` places all strings with one click, each on a separate line.

`gtext({'string1';'string2';'string3';...})` places one string per click, in the sequence specified.

`h = gtext(...)` returns the handle to a text graphics object that is placed on the plot at the location you select.

## Remarks

As you move the pointer into a figure window, the pointer becomes crosshairs to indicate that `gtext` is waiting for you to select a location. `gtext` uses the functions `ginput` and `text`.

## Examples

Place a label on the current plot:

```
gtext('Note this divergence!')
```

## See Also

`ginput`, `text`

“Annotating Plots” on page 1-91 for related functions

**Purpose**

Store or retrieve GUI data

**Syntax**

```
guidata(object_handle,data)
data = guidata(object_handle)
```

**Description**

`guidata(object_handle,data)` stores the variable `data` as GUI data. If `object_handle` is not a figure handle, then the object's parent figure is used. `data` can be any MATLAB variable, but is typically a structure, which enables you to add new fields as required.

`guidata` can manage only one variable at any time. Subsequent calls to `guidata(object_handle,data)` overwrite the previously created version of GUI data.

---

**Note for GUIDE Users** GUIDE uses `guidata` to store and maintain the `handles` structure. From a GUIDE-generated GUI M-file, do not use `guidata` to store any data other than `handles`. If you do, you may overwrite the `handles` structure and your GUI will not work. If you need to store other data with your GUI, you can add it to the `handles` structure. See GUI Data in the MATLAB documentation.

---

`data = guidata(object_handle)` returns previously stored data, or an empty matrix if nothing has been stored.

To change the data managed by `guidata`:

- 1 Get a copy of the data with the command `data = guidata(object_handle)`.
- 2 Make the desired changes to `data`.
- 3 Save the changed version of `data` with the command `guidata(object_handle,data)`.

`guidata` provides application developers with a convenient interface to a figure's application data:

- You do not need to create and maintain a hard-coded property name for the application data throughout your source code.
- You can access the data from within a subfunction callback routine using the component's handle (which is returned by `gcbo`), without needing to find the figure's handle.

If you are not using GUIDE, `guidata` is particularly useful in conjunction with `guihandles`, which creates a structure containing the handles of all the components in a GUI.

## Examples

In this example, `guidata` is used to save a structure on a GUI figure's application data from within the initialization section of the application M-file. This structure is initially created by `guihandles` and then used to save additional data as well.

```
% create structure of handles
myhandles = guihandles(figure_handle);
% add some additional data
myhandles.numberOfErrors = 0;
% save the structure
guidata(figure_handle,myhandles)
```

You can recall the data from within a subfunction callback routine and then save the structure again:

```
% get the structure in the subfunction
myhandles = guidata(gcbo);
myhandles.numberOfErrors = myhandles.numberOfErrors + 1;
% save the changes to the structure
guidata(gcbo,myhandles)
```

## See Also

`guide`, `guihandles`, `getappdata`, `setappdata`

---

<b>Purpose</b>	Open GUI Layout Editor
<b>Syntax</b>	<pre>guide guide('filename.fig') guide('fullpath') guide(HandleList)</pre>
<b>Description</b>	<p><code>guide</code> initiates the GUI design environment (GUIDE) tools that allow you to create or edit GUIs interactively.</p> <p><code>guide</code> opens the GUIDE Quick Start dialog where you can choose to open a previously created GUI or create a new one using one of the provided templates.</p> <p><code>guide('filename.fig')</code> opens the FIG-file named <code>filename.fig</code> for editing if it is on the MATLAB path.</p> <p><code>guide('fullpath')</code> opens the FIG-file at <code>fullpath</code> even if it is not on the MATLAB path.</p> <p><code>guide(HandleList)</code> opens the content of each of the figures in <code>HandleList</code> in a separate copy of the GUIDE design environment.</p>
<b>See Also</b>	<p><code>inspect</code></p> <p>Creating GUIs</p>

# guihandles

---

**Purpose** Create structure of handles

**Syntax** `handles = guihandles(object_handle)`  
`handles = guihandles`

**Description** `handles = guihandles(object_handle)` returns a structure containing the handles of the objects in a figure, using the value of their Tag properties as the fieldnames, with the following caveats:

- Objects are excluded if their Tag properties are empty, or are not legal variable names.
- If several objects have the same Tag, that field in the structure contains a vector of handles.
- Objects with hidden handles are included in the structure.

`handles = guihandles` returns a structure of handles for the current figure.

**See Also** `guidata`, `guide`, `getappdata`, `setappdata`

**Purpose** Uncompress GNU zip files

**Syntax**

```
gunzip(files)
gunzip(files,outputdir)
gunzip(url, ...)
filenames = gunzip(...)
```

**Description** `gunzip(files)` uncompresses GNU zip files from the list of files specified in `files`. Directories recursively `gunzip` all of their content. The output files have the same name, excluding the extension `.gz`, and are written to the same directory as the input files.

`files` is a string or cell array of strings containing a list of files or directories. Individual files that are on the MATLAB path can be specified as partial path names. Otherwise, an individual file can be specified relative to the current directory or with an absolute path.

# gunzip

---

Directories must be specified relative to the current directory or with absolute paths. On UNIX<sup>12</sup> systems, directories can also start with `~/` or `~username/`, which expands to the current user's home directory or the specified user's home directory, respectively. The wildcard character `*` can be used when specifying files or directories, except when relying on the MATLAB path to resolve a file name or partial path name.

`gunzip(files,outputdir)` writes the gunzipped file into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`gunzip(url, ...)` extracts the GNU zip contents from an Internet universal resource locator (URL). The URL must include the protocol type (for example, `'http://'`). MATLAB downloads the URL to the temp directory, and then deletes it.

`filenames = gunzip(...)` gunzips the files and returns the relative path names of the gunzipped files in the string cell array `filenames`.

## Examples

To gunzip all `.gz` files in the current directory, type:

```
gunzip('*.gz');
```

To gunzip Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory `ncm`, type:

```
url = 'http://www.mathworks.com/moler/ncm.tar.gz';  
gunzip(url,'ncm')  
untar('ncm/ncm.tar','ncm')
```

## See Also

`gzip`, `tar`, `untar`, `unzip`, `zip`

12. UNIX is a registered trademark of The Open Group in the United States and other countries.



**Purpose** Compress files into GNU zip files

**Syntax**

```
gzip(files)
gzip(files,outputdir)
filenames = gzip(...)
```

**Description** `gzip(files)` creates GNU zip files from the list of files specified in `files`. Directories recursively `gzip` all their contents. Each output gzipped file is written to the same directory as the input file and with the file extension `.gz`.

`files` is a string or cell array of strings containing a list of files or directories to `gzip`. Individual files that are on the MATLAB path can be specified as partial path names. Otherwise, an individual file can be specified relative to the current directory or with an absolute path.

Directories must be specified relative to the current directory or with absolute paths. On UNIX<sup>13</sup> systems, directories can also start with `~/` or `~username/`, which expands to the current user's home directory or the specified user's home directory, respectively. The wildcard character `*` can be used when specifying files or directories, except when relying on the MATLAB path to resolve a file name or partial path name.

`gzip(files,outputdir)` writes the gzipped files into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`filenames = gzip(...)` gzips the files and returns the relative path names of all gzipped files in the string cell array `filenames`.

## Example

To gzip all `.m` and `.mat` files in the current directory and store the results in the directory `archive`, type:

```
gzip({'*.m','*.mat'},'archive');
```

## See Also

`gunzip`, `tar`, `untar`, `unzip`, `zip`

13. UNIX is a registered trademark of The Open Group in the United States and other countries.

**Purpose** Hadamard matrix

**Syntax** `H = hadamard(n)`

**Description** `H = hadamard(n)` returns the Hadamard matrix of order `n`.

**Definition** Hadamard matrices are matrices of 1's and -1's whose columns are orthogonal,

$$H' * H = n * I$$

where `[n n]=size(H)` and `I = eye(n,n)` ..

They have applications in several different areas, including combinatorics, signal processing, and numerical analysis, [1], [2].

An `n`-by-`n` Hadamard matrix with `n > 2` exists only if `rem(n,4) = 0`. This function handles only the cases where `n`, `n/12`, or `n/20` is a power of 2.

**Examples** The command `hadamard(4)` produces the 4-by-4 matrix:

```
1   1   1   1
1  -1   1  -1
1   1  -1  -1
1  -1  -1   1
```

**See Also** `compan`, `hankel`, `toeplitz`

**References** [1] Ryser, H. J., *Combinatorial Mathematics*, John Wiley and Sons, 1963.

[2] Pratt, W. K., *Digital Signal Processing*, John Wiley and Sons, 1978.

# handle

---

**Purpose** Abstract class for deriving handle classes

**Syntax** `classdef myclass < handle`

**Description** The `handle` class is the superclass for all classes that follow handle semantics: a handle object indirectly references its data. If a handle object is copied, only the handle is copied and both objects refer to the same data. This behavior is equivalent to that of Handle Graphics objects, where the handle of a graphics object always refers to a particular object regardless of whether you save the handle when you create the object, store it in another variable, or obtain it with convenience functions like `findobj` or `gca`.

If you want to create a class that defines events, you must derive that class from the `handle` class.

The `handle` class is an abstract class so you cannot create an instance of this class directly. You use the `handle` class to derive other classes, which can be concrete classes whose instances are handle objects. See “Value or Handle Class — Which to Use” for information on using handle classes.

`classdef myclass < handle` makes `myclass` a subclass of the `handle` class.

## Handle Class Methods

When you derive a class from the `handle` class, your class inherits the following methods.

Method	Purpose
<code>addlistener</code>	Creates a listener for the specified event and assigns a callback function to execute when the event occurs.
<code>notify</code>	Broadcast a notice that a specific event is occurring on a specified handle object or array of handle objects.

Method	Purpose
<code>delete</code>	Handle object destructor method that is called when the object's lifecycle ends.
<code>disp</code>	Handle object <code>disp</code> method which is called by the <code>display</code> method. See the MATLAB <code>disp</code> function.
<code>display</code>	Handle object <code>display</code> method called when MATLAB software interprets an expression returning a handle object that is not terminated by a semicolon. See the MATLAB <code>display</code> function.
<code>findobj</code>	Finds objects matching the specified conditions from the input array of handle objects .
<code>findprop</code>	Returns a <code>meta.property</code> objects associated with the specified property name.
<code>fields</code>	Returns a cell array of string containing the names of public properties.
<code>fieldnames</code>	Returns a cell array of string containing the names of public properties. See the MATLAB <code>fieldnames</code> function.
<code>isvalid</code>	Returns a logical array in which elements are true if the corresponding elements in the input array are valid handles.
<code>eq</code> <code>ne</code> <code>lt</code> <code>le</code> <code>gt</code> <code>ge</code> relational operators	Relational functions return a logical array of the same size as the pair of input handle object arrays. Comparisons use a number associated with each handle. You can assume that the same two handles will compare as equal and the repeated comparison of any two handles will yield the same result in the same MATLAB session. Different handles are always not-equal. The order of handles is purely arbitrary, but consistent.

# handle

---

Method	Purpose
<code>ctranspose</code> <code>transpose</code>	Transposes the elements of the handle object array.
<code>permute</code>	Rearranges the dimensions of the handle object array. See the MATLAB <code>permute</code> function.
<code>reshape</code>	Changes the dimensions of the handle object array to the specified dimensions. See the MATLAB <code>reshape</code> function.
<code>sort</code>	Sort the handle objects in any array in ascending or descending order. The order of handles is purely arbitrary, but reproducible in a given MATLAB session. See the MATLAB <code>sort</code> function.

## Handle Class Events

The handle class defines one event:

`ObjectBeingDestroyed`

This event is triggered when the handle object is about to be destroyed. If you define a listener for this event, its callback executes before the handle object is destroyed.

You can add a listener for this event using the `addListener` method. See “Defining Events and Listeners — Syntax and Techniques” for more information on using events and listeners.

## Handle Subclasses

There are two abstract handle subclasses that you can use to derive handle classes:

- `hgsetget` — use when you want to create a handle class that inherits set and get methods having the same behavior as Handle Graphics set and get functions.

- `dynamicprops` — use when you want to create a handle class that allows you to add instance data (dynamically defined properties) to objects.

## **Useful Functions**

- `properties` — list the class public properties
- `methods` — list the class methods
- `events` — list the events defined by the class

Note that `ishandle` does not test for handle class objects. Use `isa` instead.

# hankel

---

**Purpose** Hankel matrix

**Syntax** `H = hankel(c)`  
`H = hankel(c,r)`

**Description** `H = hankel(c)` returns the square Hankel matrix whose first column is `c` and whose elements are zero below the first anti-diagonal.

`H = hankel(c,r)` returns a Hankel matrix whose first column is `c` and whose last row is `r`. If the last element of `c` differs from the first element of `r`, the last element of `c` prevails.

**Definition** A Hankel matrix is a matrix that is symmetric and constant across the anti-diagonals, and has elements  $h(i,j) = p(i+j-1)$ , where vector `p = [c r(2:end)]` completely determines the Hankel matrix.

**Examples** A Hankel matrix with anti-diagonal disagreement is

```
c = 1:3; r = 7:10;
h = hankel(c,r)
h =
     1     2     3     8
     2     3     8     9
     3     8     9    10

p = [1 2 3 8 9 10]
```

**See Also** `hadamard`, `toeplitz`, `kron`



**Purpose**

Summary of MATLAB HDF4 capabilities

**Description**

The MATLAB software provides a set of low-level functions that enable you to access the HDF4 library developed by the National Center for Supercomputing Applications (NCSA). For information about HDF4, go to the HDF Web page at <http://www.hdfgroup.org>.

---

**Note** For information about MATLAB HDF5 capabilities, which is a completely separate, incompatible format, see `hdf5`.

---

The following table lists all the HDF4 application programming interfaces (APIs) supported by MATLAB with the name of the MATLAB function used to access the API. To use these functions, you must be familiar with the HDF library. For more information about using these MATLAB functions, see *Working with Scientific Data Formats*.

<b>Application Programming Interface</b>	<b>Description</b>	<b>MATLAB Function</b>
Annotations	Stores, manages, and retrieves text used to describe an HDF file or any of the data structures contained in the file.	<code>hdfan</code>
General Raster Images	Stores, manages, and retrieves raster images, their dimensions and palettes. It can also manipulate unattached palettes.  Note: Use the MATLAB functions <code>imread</code> and <code>imwrite</code> with HDF raster image formats.	<code>hdfdf24</code> , <code>hdfdf8</code>

<b>Application Programming Interface</b>	<b>Description</b>	<b>MATLAB Function</b>
HDF-EOS	Provides functions to read HDF-EOS grid (GD), point (PT), and swath (SW) data.	hdfgd, hdfpt, hdfsw
HDF Utilities	Provides functions to open and close HDF files and handle errors.	hdfh, hdfhd, hdfhe
MATLAB HDF Utilities	Provides utility functions that help you work with HDF files in the MATLAB environment.	hdfml
Scientific Data	Stores, manages, and retrieves multidimensional arrays of character or numeric data, along with their dimensions and attributes.	hdfsd
V Groups	Creates and retrieves groups of other HDF data objects, such as raster images or V data.	hdfv
V Data	Stores, manages, and retrieves multivariate data stored as records in a table.	hdfvf, hdfvh, hdfvs

## See Also

hdfinfo, hdfread, hdftool, imread

**Purpose**

Summary of MATLAB HDF5 capabilities

**Description**

The MATLAB software provides both high-level and low-level access to HDF5 files. The high-level access functions make it easy to read a data set from an HDF5 file or write a variable from the MATLAB workspace into an HDF5 file. The MATLAB low-level interface provides direct access to the more than 200 functions in the HDF5 library. MATLAB currently supports version HDF5-1.6.5 of the library.

---

**Note** For information about MATLAB HDF4 capabilities, which is a completely separate, incompatible format, see `hdf`.

---

The following sections provide an overview of both this high- and low-level access. To use these MATLAB functions, you must be familiar with HDF5 programming concepts and, when using the low-level functions, details about the functions in the library. To get this information, go to the HDF Web page at <http://www.hdfgroup.org>.

**High-level Access**

MATLAB includes three functions that provide high-level access to HDF5 files:

- `hdf5info`
- `hdf5read`
- `hdf5write`

Using these functions you can read data and metadata from an HDF5 file and write data from the MATLAB workspace to a file in HDF5 format. For more information about these functions, see their individual reference pages.

**Low-level Access**

MATLAB provides direct access to the over 200 functions in the HDF5 Library. Using these functions, you can read and write complex

datatypes, utilize HDF5 data subsetting capabilities, and take advantage of other features present in the HDF5 library.

The HDF5 library organizes the routines in the library into interfaces. MATLAB organizes the corresponding MATLAB functions into class directories that match these HDF5 library interfaces. For example, the MATLAB functions for the HDF5 Attribute Interface are in the @H5A class directory.

The following table lists all the HDF5 library interfaces in alphabetical order by name. The table includes the name of the associated MATLAB class directory.

<b>HDF5 Library Interface</b>	<b>MATLAB Class Directory</b>	<b>Description</b>
Attribute	@H5A	Manipulate metadata associated with data sets or groups
Dataset	@H5D	Manipulate multidimensional arrays of data elements, together with supporting metadata
Dataspace	@H5S	Define and work with data spaces, which describe the the dimensionality of a data set
Datatype	@H5T	Define the type of variable that is stored in a data set
Error	@H5E	Handle errors
File	@H5F	Access files
Filters and Compression	@H5Z	Create inline data filters and data compression
Group	@H5G	Organize objects in a file; analogous to a directory structure
Identifier	@H5I	Manipulate HDF5 object identifiers

<b>HDF5 Library Interface</b>	<b>MATLAB Class Directory</b>	<b>Description</b>
Library	@H5	General-purpose functions for use with the entire HDF5 library, such as initialization
MATLAB	@H5ML	MATLAB utility functions that are not part of the HDF5 library itself.
Property	@H5P	Manipulate object property lists
Reference	@H5R	Manipulate HDF5 references, which are like UNIX links or Windows shortcuts

In most cases, the syntax of the MATLAB function is identical to the syntax of the HDF5 library function. To get detailed information about the MATLAB syntax of an HDF5 library function, view the help for the individual MATLAB function, as follows:

```
help @H5F/open
```

To view a list of all the MATLAB HDF5 functions in a particular interface, type:

```
help imagesci/@H5F
```

## See Also

`hdf`, `hdf5info`, `hdf5read`, `hdf5write`

# hdf5info

---

## Purpose

Information about HDF5 file

## Syntax

```
fileinfo = hdf5info(filename)
fileinfo = hdf5info(...,'ReadAttributes',B00L)
[...] = hdf5info(..., 'V71Dimensions', B00L)
```

## Description

`fileinfo = hdf5info(filename)` returns a structure `fileinfo` whose fields contain information about the contents of the HDF5 file `filename`. `filename` is a string that specifies the name of the HDF5 file.

`fileinfo = hdf5info(...,'ReadAttributes',B00L)` specifies whether `hdf5info` returns the values of the attributes or just information describing the attributes. By default, `hdf5info` reads in attribute values (`B00L = true`).

`[...] = hdf5info(..., 'V71Dimensions', B00L)` specifies whether to report the dimensions of data sets and attributes as they were returned in previous versions of `hdf5info` (MATLAB 7.1 [R14SP3] and earlier). If `B00L` is true, `hdf5info` swaps the first two dimensions of the data set. This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, swapping these dimensions may not correctly reflect the intent of the data in the file and may invalidate metadata. When `B00L` is false (the default), `hdf5info` returns data dimensions that correctly reflect the data ordering as it is written in the file—each dimension in the output variable matches the same dimension in the file.

---

**Note** If you use the 'V71Dimensions' parameter and intend on passing the `fileinfo` structure returned to the `hdf5read` function, you should also specify the 'V71Dimensions' parameters with `hdf5read`. If you do not, `hdf5read` uses the new behavior when reading the data set and certain metadata returned by `hdf5info` does not match the actual data returned by `hdf5read`.

---

**Examples**

```
fileinfo = hdf5info('example.h5')

fileinfo =

    Filename: 'example.h5'
  LibVersion: '1.4.5'
      Offset: 0
   FileSize: 8172
GroupHierarchy: [1x1 struct]
```

To get more information about the contents of the HDF5 file, look at the `GroupHierarchy` field in the `fileinfo` structure returned by `hdf5info`.

```
toplevel = fileinfo.GroupHierarchy

toplevel =

    Filename: [1x64 char]
      Name: '/'
   Groups: [1x2 struct]
  Datasets: []
Datatypes: []
   Links: []
Attributes: [1x2 struct]
```

To probe further into the file hierarchy, keep examining the `Groups` field.

**See also**

`hdf5read`, `hdf5write`

# hdf5read

---

**Purpose** Read HDF5 file

**Syntax**

```
data = hdf5read(filename,datasetname)
attr = hdf5read(filename,attributename)
[data, attr] = hdf5read(...,'ReadAttributes',BOOL)
data = hdf5read(hinfo)
[...] = hdf5read(..., 'V71Dimensions', BOOL)
```

**Description** `data = hdf5read(filename,datasetname)` reads all the data in the data set `datasetname` that is stored in the HDF5 file `filename` and returns it in the variable `data`. To determine the names of data sets in an HDF5 file, use the `hdf5info` function.

The return value, `data`, is a multidimensional array. `hdf5read` maps HDF5 data types to native MATLAB data types, whenever possible. If it cannot represent the data using MATLAB data types, `hdf5read` uses one of the HDF5 data type objects. For example, if an HDF5 file contains a data set made up of an enumerated data type, `hdf5read` uses the `hdf5.h5enum` object to represent the data in the MATLAB workspace. The `hdf5.h5enum` object has data members that store the enumerations (names), their corresponding values, and the enumerated data. For more information about the HDF5 data type objects, see the `hdf5` reference page.

`attr = hdf5read(filename,attributename)` reads all the metadata in the attribute `attributename`, stored in the HDF5 file `filename`, and returns it in the variable `attr`. To determine the names of attributes in an HDF5 file, use the `hdf5info` function.

`[data, attr] = hdf5read(...,'ReadAttributes',BOOL)` reads all the data, as well as all of the associated attribute information contained within that data set. By default, `BOOL` is `false`.

`data = hdf5read(hinfo)` reads all of the data in the data set specified in the structure `hinfo` and returns it in the variable `data`. The `hinfo` structure is extracted from the output returned by `hdf5info`, which specifies an HDF5 file and a specific data set.



[...] = `hdf5read(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets read from the file. If `BOOL` is true, `hdf5read` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data dimensions correctly reflect the data ordering as it is written in the file — each dimension in the output variable matches the same dimension in the file.

## Examples

Use `hdf5info` to get information about an HDF5 file and then use `hdf5read` to read a data set, using the information structure (`hinfo`) returned by `hdf5info` to specify the data set.

```
hinfo = hdf5info('example.h5');  
dset = hdf5read(hinfo.GroupHierarchy.Groups(2).Datasets(1));
```

## See Also

`hdf5`, `hdf5info`, `hdf5write`

# hdf5write

---

**Purpose** Write data to file in HDF5 format

**Syntax**

```
hdf5write(filename,location,dataset)
hdf5write(filename,details,dataset)
hdf5write(filename,details,attribute)
hdf5write(filename, details1, dataset1, details2, dataset2,
    ...)
hdf5write(filename,...,'WriteMode',mode,...)
hdf5write(..., 'V71Dimensions', BOOL)
```

**Description** `hdf5write(filename,location,dataset)` writes the data `dataset` to the HDF5 file, `filename`. If `filename` does not exist, `hdf5write` creates it. If `filename` exists, `hdf5write` overwrites the existing file, by default, but you can also append data to an existing file using an optional syntax.

`location` defines where to write the data set in the file. HDF5 files are organized in a hierarchical structure similar to a UNIX directory structure. `location` is a string that resembles a UNIX path.

`hdf5write` maps the data in `dataset` to HDF5 data types according to rules outlined below.

`hdf5write(filename,details,dataset)` writes `dataset` to `filename` using the values in the `details` structure. For a data set, the `details` structure can contain the following fields.

Field Name	Description	Data Type
Location	Location of the data set in the file	Character array
Name	Name to attach to the data set	Character array

`hdf5write(filename,details,attribute)` writes the metadata `attribute` to `filename` using the values in the `details` structure. For an attribute, the `details` structure can contain following fields.

Field Name	Description	Data Type
AttachedTo	Location of the object this attribute modifies	Structure array
AttachType	Identifies what kind of object this attribute modifies; possible values are 'group' and 'dataset'	Character array
Name	Name to attach to the data set	Character array

`hdf5write(filename, details1, dataset1, details2, dataset2, ...)` writes multiple data sets and associated attributes to `filename` in one operation. Each data set and attribute must have an associated `details` structure.

`hdf5write(filename, ..., 'WriteMode', mode, ...)` specifies whether `hdf5write` overwrites the existing file (the default) or appends data sets and attributes to the file. Possible values for `mode` are 'overwrite' and 'append'.

`hdf5write(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets written to the file. If `BOOL` is true, `hdf5write` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data written to the file correctly reflects the data ordering of the data sets — each dimension in the file's data sets matches the same dimension in the corresponding MATLAB variable.

# hdf5write

## Data Type Mappings

The following table lists how `hdf5write` maps the data type from the workspace into an HDF5 file. If the data in the workspace that is being written to the file is a MATLAB data type, `hdf5write` uses the following rules when translating MATLAB data into HDF5 data objects.

MATLAB Data Type	HDF5 Data Set or Attribute
Numeric	Corresponding HDF5 native data type. For example, if the workspace data type is <code>uint8</code> , the <code>hdf5write</code> function writes the data to the file as 8-bit integers. The size of the HDF5 dataspace is the same size as the MATLAB array.
String	Single, null-terminated string
Cell array of strings	Multiple, null-terminated strings, each the same length. Length is determined by the length of the longest string in the cell array. The size of the HDF5 dataspace is the same size as the cell array.
Cell array of numeric data	Numeric array, the same dimensions as the cell array. The elements of the array must all have the same size and type. The data type is determined by the first element in the cell array.
Structure array	HDF5 compound type. Individual fields in the structure employ the same data translation rules for individual data types. For example, a cell array of strings becomes a multiple, null-terminated strings.
HDF5 objects	If the data being written to the file is composed of HDF5 objects, <code>hdf5write</code> uses the same data type when writing to the file. For all HDF5 objects, except <code>HDF5.h5enum</code> objects, the dataspace has the same dimensions as the array of HDF5 objects passed to the function. For <code>HDF5.h5enum</code> objects, the size and dimensions of the data set in the HDF5 file is the same as the object's Data field.

## Examples

Write a 5-by-5 data set of `uint8` values to the root group.

```
hdf5write('myfile.h5', '/dataset1', uint8(magic(5)))
```

Write a 2-by-2 string data set in a subgroup.

```
dataset = {'north', 'south'; 'east', 'west'};
hdf5write('myfile2.h5', '/group1/dataset1.1', dataset);
```

Write a data set and attribute to an existing group.

```
dset = single(rand(10,10));
dset_details.Location = '/group1/dataset1.2';
dset_details.Name = 'Random';

attr = 'Some random data';
attr_details.Name = 'Description';
attr_details.AttachedTo = '/group1/dataset1.2/Random';
attr_details.AttachType = 'dataset';

hdf5write('myfile2.h5', dset_details, dset, ...
         attr_details, attr, 'WriteMode', 'append');
```

Write a data set using objects.

```
dset = hdf5.h5array(magic(5));
hdf5write('myfile3.h5', '/g1/objects', dset);
```

## See Also

[hdf5](#), [hdf5read](#), [hdf5info](#)

# hdfinfo

---

**Purpose** Information about HDF4 or HDF-EOS file

**Syntax**  
S = hdfinfo(filename)  
S = hdfinfo(filename,mode)

**Description** S = hdfinfo(filename) returns a structure S whose fields contain information about the contents of an HDF4 or HDF-EOS file. filename is a string that specifies the name of the HDF4 file.

S = hdfinfo(filename,mode) reads the file as an HDF4 file, if mode is 'hdf', or as an HDF-EOS file, if mode is 'eos'. If mode is 'eos', only HDF-EOS data objects are queried. To retrieve information on the entire contents of a file containing both HDF4 and HDF-EOS objects, mode must be 'hdf'.

---

**Note** hdfinfo can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To get information about an HDF5 file, use hdf5info.

---

The set of fields in the returned structure S depends on the individual file. Fields that can be present in the S structure are shown in the following table.

Mode	Field Name	Description	Return Type
HDF	Attributes	Attributes of the data set	Structure array
	Description	Annotation description	Cell array
	Filename	Name of the file	String
	Label	Annotation label	Cell array
	Raster8	Description of 8-bit raster images	Structure array

Mode	Field Name	Description	Return Type
	Raster24	Description of 24-bit raster images	Structure array
	SDS	Description of scientific data sets	Structure array
	Vdata	Description of Vdata sets	Structure array
	Vgroup	Description of Vgroups	Structure array
EOS	Filename	Name of the file	String
	Grid	Grid data	Structure array
	Point	Point data	Structure array
	Swath	Swath data	Structure array

Those fields in the table above that contain structure arrays are further described in the tables shown below.

## Fields Common to Returned Structure Arrays

Structure arrays returned by `hdfinfo` contain some common fields. These are shown in the table below. Not all structure arrays will contain all of these fields.

Field Name	Description	Data Type
Attributes	Data set attributes. Contains fields Name and Value.	Structure array
Description	Annotation description	Cell array
Filename	Name of the file	String
Label	Annotation label	Cell array

Field Name	Description	Data Type
Name	Name of the data set	String
Rank	Number of dimensions of the data set	Double
Ref	Data set reference number	Double
Type	Type of HDF or HDF-EOS object	String

## Fields Specific to Certain Structures

Structure arrays returned by `hdfinfo` also contain fields that are unique to each structure. These are shown in the tables below.

### Fields of the Attribute Structure

Field Name	Description	Data Type
Name	Attribute name	String
Value	Attribute value or description	Numeric or string

### Fields of the Raster8 and Raster24 Structures

Field Name	Description	Data Type
HasPalette	1 (true) if the image has an associated palette, otherwise 0 (false) (8-bit only)	Logical
Height	Height of the image, in pixels	Number
Interlace	Interlace mode of the image (24-bit only)	String
Name	Name of the image	String
Width	Width of the image, in pixels	Number



**Fields of the SDS Structure**

<b>Field Name</b>	<b>Description</b>	<b>Data Type</b>
DataType	Data precision	String
Dims	Dimensions of the data set. Contains fields Name, DataType, Size, Scale, and Attributes. Scale is an array of numbers to place along the dimension and demarcate intervals in the data set.	Structure array
Index	Index of the SDS	Number

**Fields of the Vdata Structure**

<b>Field Name</b>	<b>Description</b>	<b>Data Type</b>
DataAttributes	Attributes of the entire data set. Contains fields Name and Value.	Structure array
Class	Class name of the data set	String
Fields	Fields of the Vdata. Contains fields Name and Attributes.	Structure array
NumRecords	Number of data set records	Double
IsAttribute	1 (true) if Vdata is an attribute, otherwise 0 (false)	Logical

**Fields of the Vgroup Structure**

<b>Field Name</b>	<b>Description</b>	<b>Data Type</b>
Class	Class name of the data set	String

## Fields of the Vgroup Structure (Continued)

Field Name	Description	Data Type
Raster8	Description of the 8-bit raster image	Structure array
Raster24	Description of the 24-bit raster image	Structure array
SDS	Description of the Scientific Data sets	Structure array
Tag	Tag of this Vgroup	Number
Vdata	Description of the Vdata sets	Structure array
Vgroup	Description of the Vgroups	Structure array

## Fields of the Grid Structure

Field Name	Description	Data Type
Columns	Number of columns in the grid	Number
DataFields	Description of the data fields in each Grid field of the grid. Contains fields Name, Rank, Dims, NumberType, FillValue, and TileDims.	Structure array
LowerRight	Lower right corner location, in meters	Number
Origin Code	Origin code for the grid	Number
PixRegCode	Pixel registration code	Number

**Fields of the Grid Structure (Continued)**

<b>Field Name</b>	<b>Description</b>	<b>Data Type</b>
Projection	Projection code, zone code, sphere code, and projection parameters of the grid. Contains fields ProjCode, ZoneCode, SphereCode, and ProjParam.	Structure
Rows	Number of rows in the grid	Number
UpperLeft	Upper left corner location, in meters	Number

**Fields of the Point Structure**

<b>Field Name</b>	<b>Description</b>	<b>Data Type</b>
Level	Description of each level of the point. Contains fields Name, NumRecords, FieldNames, DataType, and Index.	Structure

**Fields of the Swath Structure**

<b>Field Name</b>	<b>Description</b>	<b>Data Type</b>
DataFields	Data fields in the swath. Contains fields Name, Rank, Dims, NumberType, and FillValue.	Structure array

## Fields of the Swath Structure (Continued)

Field Name	Description	Data Type
GeolocationFields	Geolocation fields in the swath. Contains fields Name, Rank, Dims, NumberType, and FillValue.	Structure array
IdxMapInfo	Relationship between indexed elements of the geolocation mapping. Contains fields Map and Size.	Structure
MapInfo	Relationship between data and geolocation fields. Contains fields Map, Offset, and Increment.	Structure

## Examples

To retrieve information about the file `example.hdf`,

```
fileinfo = hdfinfo('example.hdf')

fileinfo =
  Filename: 'example.hdf'
  SDS: [1x1 struct]
  Vdata: [1x1 struct]
```

And to retrieve information from this about the scientific data set in `example.hdf`,

```
sds_info = fileinfo.SDS

sds_info =
  Filename: 'example.hdf'
  Type: 'Scientific Data Set'
  Name: 'Example SDS'
```

```
Rank: 2
DataType: 'int16'
Attributes: []
  Dims: [2x1 struct]
  Label: {}
Description: {}
Index: 0
```

**See Also**      [hdfread](#), [hdf](#)

# hdfread

---

**Purpose** Read data from HDF4 or HDF-EOS file

**Syntax**

```
data = hdfread(filename, datasetname)
data = hdfread(hinfo.fieldname)
data = hdfread(...,param1,value1,param2,value2,...)
[data,map] = hdfread(...)
```

**Description** `data = hdfread(filename, datasetname)` returns all the data in the data set specified by `datasetname` from the HDF4 or HDF-EOS file specified by `filename`. To determine the name of a data set in an HDF4 file, use the `hdfinfo` function.

---

**Note** `hdfread` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To read data from and HDF5 file, use `hdf5read`.

---

`data = hdfread(hinfo.fieldname)` returns all the data in the data set specified by `hinfo.fieldname`, where `hinfo` is the structure returned by the `hdfinfo` function and `fieldname` is the name of a field in the structure that relates to a particular type of data set. For example, to read an HDF scientific data set, specify the `SDS` field, as in `hinfo.SDS`. To read HDF V data, specify the `Vdata` field, as in `hinfo.Vdata`. `hdfread` can get the name of the HDF file from these structures.

`data = hdfread(...,param1,value1,param2,value2,...)` returns subsets of the data according to the specified parameter and value pairs. See the tables below to find the valid parameters and values for different types of data sets.

`[data,map] = hdfread(...)` returns the image data and the colormap `map` for an 8-bit raster image.

## Subsetting Parameters

The following tables show the subsetting parameters that can be used with the `hdfread` function for certain types of HDF4 data. These data types are

- HDF Scientific Data (SD)
- HDF Vdata (V)
- HDF-EOS Grid Data
- HDF-EOS Point Data
- HDF-EOS Swath Data

Note the following:

- If a parameter requires multiple values, the values must be stored in a cell array. For example, the 'Index' parameter requires three values: `start`, `stride`, and `edge`. Enclose these values in curly braces as a cell array.

```
hdfread(dataset_name, 'Index', {start, stride, edge})
```

- All values that are indices are 1-based.

## **Subsetting Parameters for HDF Scientific Data (SD) Data Sets**

When you are working with HDF SD files, `hdfread` supports the parameters listed in this table.

# hdfread

Parameter	Description
'Index'	<p>Three-element cell array, {start, stride, edge}, specifying the location, range, and values to be read from the data set</p> <ul style="list-style-type: none"><li>• <b>start</b> — A 1-based array specifying the position in the file to begin reading Default: 1, start at the first element of each dimension. The values specified must not exceed the size of any dimension of the data set.</li><li>• <b>stride</b> — A 1-based array specifying the interval between the values to read Default: 1, read every element of the data set.</li><li>• <b>edge</b> — A 1-based array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions</li></ul>

For example, this code reads the data set `Example SDS` from the HDF file `example.hdf`. The 'Index' parameter specifies that `hdfread` start reading data at the beginning of each dimension, read until the end of each dimension, but only read every other data value in the first dimension.

```
hdfread('example.hdf', 'Example SDS', ...  
        'Index', {[], [2 1], []})
```

## Subsetting Parameters for HDF Vdata Sets

When you are working with HDF Vdata files, `hdfread` supports these parameters.

Parameter	Description
'Fields'	Text string specifying the name of the data set field to be read from. When specifying multiple field names, use a comma-separated list.



Parameter	Description
'FirstRecord'	1-based number specifying the record from which to begin reading
'NumRecords'	Number specifying the total number of records to read

For example, this code reads the Vdata set `Example Vdata` from the HDF file `example.hdf`.

```
hdfread('example.hdf', 'Example Vdata', 'FirstRecord', 400,...
        'NumRecords', 50)
```

## Subsetting Parameters for HDF-EOS Grid Data

When you are working with HDF-EOS grid data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive parameters — You can only specify one of these parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
<b>Required Parameter</b>	
'Fields'	String naming the data set field to be read. You can specify only one field name for a Grid data set.
<b>Mutually Exclusive Optional Parameters</b>	

# hdfread

Parameter	Description
'Index'	<p>Three-element cell array, {<code>start</code>,<code>stride</code>,<code>edge</code>}, specifying the location, range, and values to be read from the data set</p> <p><code>start</code> — An array specifying the position in the file to begin reading</p> <p>Default: 1, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set.</p> <p><code>stride</code> — An array specifying the interval between the values to read</p> <p>Default: 1, read every element of the data set.</p> <p><code>edge</code> — An array specifying the length of each dimension to read</p> <p>Default: An array containing the lengths of the corresponding dimensions</p>
'Interpolate'	<p>Two-element cell array, {<code>longitude</code>,<code>latitude</code>}, specifying the longitude and latitude points that define a region for bilinear interpolation. Each element is an N-length vector specifying longitude and latitude coordinates.</p>
'Pixels'	<p>Two-element cell array, {<code>longitude</code>,<code>latitude</code>}, specifying the longitude and latitude coordinates that define a region. Each element is an N-length vector specifying longitude and latitude coordinates. This region is converted into pixel rows and columns with the origin in the upper left corner of the grid.</p> <p>Note: This is the pixel equivalent of reading a 'Box' region.</p>
'Tile'	<p>Vector specifying the coordinates of the tile to read, for HDF-EOS Grid files that support tiles</p>
<b>Optional Parameters</b>	
'Box'	<p>Two-element cell array, {<code>longitude</code>,<code>latitude</code>}, specifying the longitude and latitude coordinates that define a region. <code>longitude</code> and <code>latitude</code> are each two-element vectors specifying longitude and latitude coordinates.</p>

Parameter	Description
'Time'	Two-element cell array, [start stop], where start and stop are numbers that specify the start and end-point for a period of time
'Vertical'	<p>Two-element cell array, {dimension, range}</p> <p>dimension — String specifying the name of the data set field to be read from. You can specify only one field name for a Grid data set.</p> <p>range — Two-element array specifying the minimum and maximum range for the subset. If dimension is a dimension name, then range specifies the range of elements to extract. If dimension is a field name, then range specifies the range of values to extract.</p> <p>'Vertical' subsetting can be used alone or in conjunction with 'Box' or 'Time'. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to hdfread.</p>

For example,

```
hdfread(grid_dataset, 'Fields', fieldname, ...
        'Vertical', {dimension, [min, max]})
```

## Subsetting Parameters for HDF-EOS Point Data

When you are working with HDF-EOS Point data, hdfread has two required parameters and three optional parameters.

Parameter	Description
<b>Required Parameters</b>	
'Fields'	String naming the data set field to be read. For multiple field names, use a comma-separated list.
'Level'	1-based number specifying which level to read from in an HDF-EOS Point data set
<b>Optional Parameters</b>	

# hdfread

Parameter	Description
'Box'	Two-element cell array, {longitude, latitude}, specifying the longitude and latitude coordinates that define a region. longitude and latitude are each two-element vectors specifying longitude and latitude coordinates.
'RecordNumbers'	Vector specifying the record numbers to read
'Time'	Two-element cell array, [start stop], where start and stop are numbers that specify the start and endpoint for a period of time

For example,

```
hdfread(point_dataset, 'Fields', {field1, field2}, ...  
        'Level', level, 'RecordNumbers', [1:50, 200:250])
```

## Subsetting Parameters for HDF-EOS Swath Data

When you are working with HDF-EOS Swath data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive

You can only use one of the mutually exclusive parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
<b>Required Parameter</b>	
'Fields'	String naming the data set field to be read. You can specify only one field name for a Swath data set.
<b>Mutually Exclusive Optional Parameters</b>	

Parameter	Description
'Index'	<p>Three-element cell array, {<code>start</code>, <code>stride</code>, <code>edge</code>}, specifying the location, range, and values to be read from the data set</p> <ul style="list-style-type: none"> <li>• <code>start</code> — An array specifying the position in the file to begin reading Default: 1, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set.</li> <li>• <code>stride</code> — An array specifying the interval between the values to read Default: 1, read every element of the data set.</li> <li>• <code>edge</code> — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions</li> </ul>
'Time'	<p>Three-element cell array, {<code>start</code>, <code>stop</code>, <code>mode</code>}, where <code>start</code> and <code>stop</code> specify the beginning and the endpoint for a period of time, and <code>mode</code> is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met:</p> <ul style="list-style-type: none"> <li>• Its midpoint is within the box (<code>mode='midpoint'</code>).</li> <li>• Either endpoint is within the box (<code>mode='endpoint'</code>).</li> <li>• Any point is within the box (<code>mode='anypoint'</code>).</li> </ul>
<b>Optional Parameters</b>	

# hdfread

Parameter	Description
'Box'	<p>Three-element cell array, {longitude, latitude, mode} specifying the longitude and latitude coordinates that define a region. longitude and latitude are two-element vectors that specify longitude and latitude coordinates. mode is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met:</p> <ul style="list-style-type: none"><li>• Its midpoint is within the box (mode='midpoint').</li><li>• Either endpoint is within the box (mode='endpoint').</li><li>• Any point is within the box (mode='anypoint').</li></ul>
'ExtMode'	<p>String specifying whether geolocation fields and data fields must be in the same swath (mode='internal'), or can be in different swaths (mode='external')</p> <p>Note: mode is only used when extracting a time period or a region.</p>
'Vertical'	<p>Two-element cell array, {dimension, range}</p> <ul style="list-style-type: none"><li>• dimension is a string specifying either a dimension name or field name to subset the data by.</li><li>• range is a two-element vector specifying the minimum and maximum range for the subset. If dimension is a dimension name, then range specifies the range of elements to extract. If dimension is a field name, then range specifies the range of values to extract.</li></ul> <p>'Vertical' subsetting can be used alone or in conjunction with 'Box' or 'Time'. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to hdfread.</p>

For example,

```
hdfread('example.hdf', swath_dataset, 'Fields', fieldname, ...  
        'Time', {start, stop, 'midpoint'})
```

## Examples

### Example 1

Specify the name of the HDF file and the name of the data set. This example reads a data set named 'Example SDS' from a sample HDF file.

```
data = hdfread('example.hdf', 'Example SDS')
```

### Example 2

Use data returned by `hdfinfo` to specify the data set to read.

- 1 Call `hdfinfo` to retrieve information about the contents of the HDF file.

```
fileinfo = hdfinfo('example.hdf')  
fileinfo =
```

```
Filename: 'N:\toolbox\matlab\demos\example.hdf'  
SDS: [1x1 struct]  
Vdata: [1x1 struct]
```

- 2 Extract the structure containing information about the particular data set you want to import from the data returned by `hdfinfo`. The example uses the structure in the `SDS` field to retrieve a scientific data set.

```
sds_info = fileinfo.SDS  
sds_info =
```

```
Filename: 'N:\toolbox\matlab\demos\example.hdf'  
Type: 'Scientific Data Set'  
Name: 'Example SDS'  
Rank: 2  
DataType: 'int16'  
Attributes: []  
Dims: [2x1 struct]  
Label: {}  
Description: {}  
Index: 0
```

**3** You can pass this structure to `hdfread` to import the data in the data set.

```
data = hdfread(sds_info)
```

### Example 3

You can use the information returned by `hdfinfo` to check the size of the data set.

```
sds_info.Dims.Size
ans =
    16
ans =
     5
```

Using the `'index'` parameter with `hdfread`, you can read a subset of the data in the data set. This example specifies a starting index of [3 3], an interval of 1 between values ([ ] meaning the default value of 1), and a length of 10 rows and 2 columns.

```
data = hdfread(sds_info, 'Index', {[3 3],[],[10 2]});

data(:,1)
ans =
     7
     8
     9
    10
    11
    12
    13
    14
    15
    16

data(:,2)
ans =
     8
```



```
9
10
11
12
13
14
15
16
17
```

#### Example 4

This example uses the `Vdata` field from the information returned by `hdfinfo` to read two fields of the data, `Idx` and `Temp`.

```
info = hdfinfo('example.hdf');

data = hdfread(info.Vdata,...
    'Fields',{'Idx','Temp'})

data =
    [1x10 int16]
    [1x10 int16]

index = data{1,1};
temp = data{2,1};

temp(1:6)
ans =
     0    12     3     5    10    -1
```

#### See Also

`hdfinfo`, `hdf`

# hdfstool

---

**Purpose** Browse and import data from HDF4 or HDF-EOS files

**Syntax**

```
hdfstool
hdfstool(filename)
h = hdfstool(...)
```

**Description** `hdfstool` starts the HDF Import Tool, a graphical user interface used to browse the contents of HDF4 and HDF-EOS files and import data and subsets of data from these files. To open an HDF4 or HDF-EOS file, select **Open** from the **File** menu. You can open multiple files in the HDF Import Tool by selecting **Open** from the **File** menu.


`hdfstool(filename)` opens the HDF4 or HDF-EOS file specified by `filename` in the HDF Import Tool.

`h = hdfstool(...)` returns a handle `h` to the HDF Import Tool. To close the tool from the command line, use `close(h)`.

**Example**

```
hdfstool('example.hdf');
```

**See Also** `hdf`, `hdfinfo`, `hdfread`, `uiimport`

<b>Purpose</b>	Help for functions in Command Window
<b>GUI Alternatives</b>	Use the Function Browser by clicking its button,  , or run <code>doc functionname</code> to view more extensive help for a function in the Help browser.
<b>Syntax</b>	<pre>help help / help functionname help modelname.mdl help toolboxname help toolboxname/functionname help classname.methodname help classname help <b>syntax</b> t = help('topic')</pre>
<b>Description</b>	<p><code>help</code> lists all primary help topics in the Command Window. Each main help topic corresponds to a directory name on the search path the MATLAB software uses.</p> <p><code>help /</code> lists all operators and special characters, along with their descriptions.</p> <p><code>help functionname</code> displays M-file help, which is a brief description and the syntax for <code>functionname</code>, in the Command Window. The output includes a link to <code>doc functionname</code>, which displays the reference page in the Help browser, often providing additional information. Output also includes <code>see also</code> links, which display help in the Command Window for related functions. If <code>functionname</code> is overloaded, that is, appears in multiple directories on the search path, <code>help</code> displays the M-file help for the first <code>functionname</code> found on the search path, and displays a hyperlinked list of the overloaded functions and their directories. If <code>functionname</code> is also the name of a toolbox, <code>help</code> also displays a list of subdirectories and hyperlinked list of functions in the toolbox, as defined in the <code>Contents.m</code> file for the toolbox.</p>

`help modelname.mdl` displays the complete description for the MDL-file `modelname` as defined in **Model Properties > Description**. If the Simulink product is installed, you do not need to specify the `.mdl` extension.

`help toolboxname` displays the `Contents.m` file for the specified directory named `toolboxname`, where `Contents.m` contains a list and corresponding description of M-files in `toolboxname`. It is not necessary to give the full pathname of the directory; the last component, or the last several components, are sufficient. If `toolboxname` is also a function name, `help` also displays the M-file help for the function `toolboxname`.

`help toolboxname/functionname` displays the M-file help for the `functionname` that resides in the `toolboxname` directory. Use this form to get direct help for an overloaded function.

`help classname.methodname` displays help for the method `methodname` of the fully qualified class `classname`. If you do not know the fully qualified class for the method, use `class(obj)`, where `methodname` is of the same class as the object `obj`.

`help classname` displays help for the fully qualified class `classname`.

`help syntax` displays M-file help describing the syntax used in MATLAB functions.

`t = help('topic')` returns the help text for `topic` as a string, with each line separated by `/n`, where `topic` is any allowable argument for `help`.

---

**Note** M-file help displayed in the Command Window uses all uppercase characters for the function and variable names to make them stand out from the rest of the text. When typing function names, however, use lowercase characters. Some functions for interfacing to Sun Microsystems Java software do use mixed case; the M-file help accurately reflects that and you should use mixed case when typing them. For example, the `javaObject` function uses mixed case.

---

## Remarks

### Prevent Scrolling of Long Help Pages

To prevent long descriptions from scrolling off the screen before you have time to read them, enter `more` on, and then enter the `help` statement.

### How the help Function Works

The `help` function lists all help topics by displaying the first line (the H1 line) of the contents files in each directory on the search path MATLAB uses. The contents files are the M-files named `Contents.m` within each directory.

Typing `helptopic`, where `topic` is a directory name, displays the comment lines in the `Contents.m` file located in that directory. If a contents file does not exist, `help` displays the H1 lines of all the files in the directory.

Typing `help topic`, where `topic` is a function name, displays help for the function by listing the first contiguous comment lines in the M-file `topic.m`.

### Help for User-Created M-Files

You can provide help information for your own M-files, so that you and others can view it using the `help` function. You can also provide `Contents.m` files for files you create. If you provide help in class definition files for classes you create, the `doc` function displays the M-file help in the Help browser. For more information, see “Help for the Files You and Other Users Create”.

## Examples

`help close` displays help for the `close` function.

`help database/close` displays help for the `close` function in the Database Toolbox™ product.

`help datafeed` displays help for the Datafeed Toolbox™ product.

`help database` lists the functions in the Database Toolbox product and displays help for the `database` function, because there are a function and a toolbox called `database`.

`help general` lists all functions in the directory `matlabroot/toolbox/matlab/general`. This illustrates how to specify a relative partial pathname rather than a full pathname.

`help f14_dap` displays the description of the Simulink `f14_dap.mdl` model file (the Simulink product must be installed).


`t = help('close')` gets help for the function `close` and stores it as a string in `t`.

## See Also

`class`, `doc`, `docsearch`, `helpbrowser`, `helpwin`, `lookfor`, `more`, `partialpath`, `path`, `what`, `which`, `whos`

Related topics in the MATLAB Desktop Tools and Development Environment documentation:

- “Assistance While Entering Statements”, including function hints and the Function Browser
- “Help and Related Resources”

<b>Purpose</b>	Open Help browser to access all online documentation and demos
<b>GUI Alternatives</b>	As an alternative to the <code>helpbrowser</code> function, select <b>Desktop &gt; Help</b> or click the Help button  on the toolbar in the MATLAB desktop.
<b>Syntax</b>	<code>helpbrowser</code>
<b>Description</b>	<code>helpbrowser</code> displays the Help browser, open to its default startup page, providing direct access to a comprehensive library of online documentation, including reference pages and user guides. For details, see the “Help Browser Overview” topic in the MATLAB Desktop Tools and Development Environment documentation.

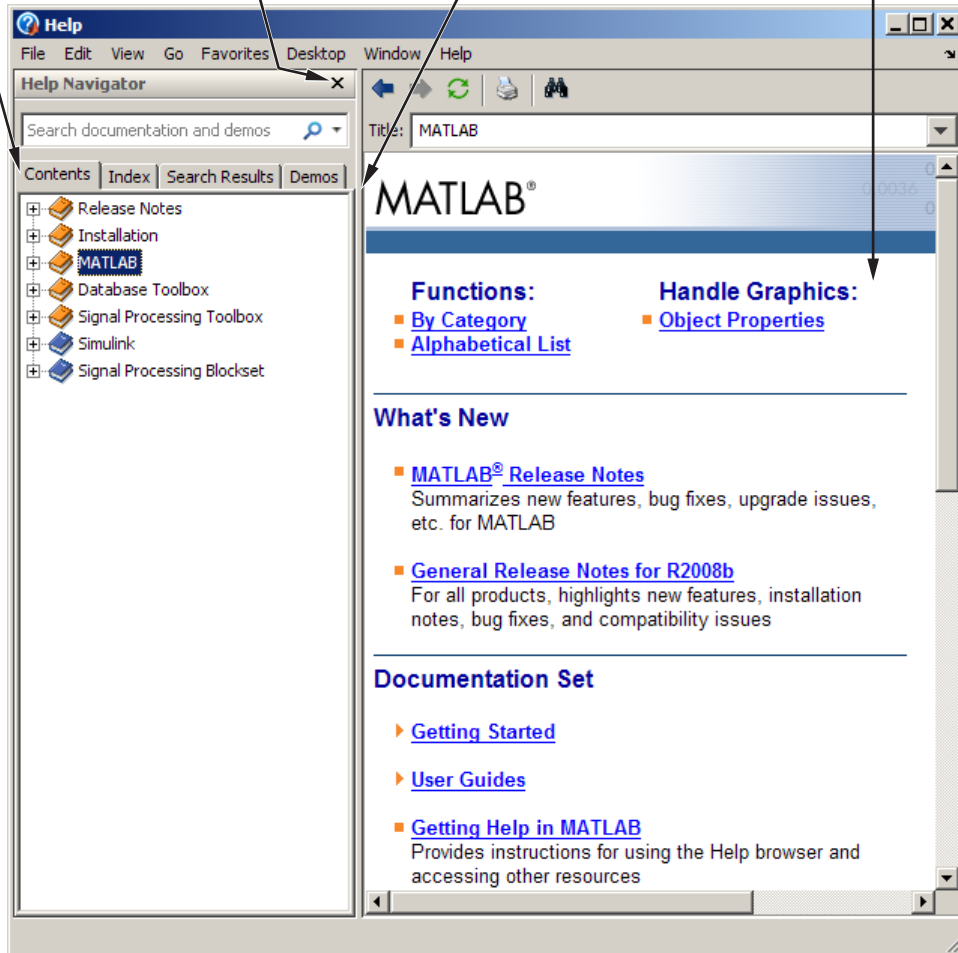
# helpbrowser

Tabs in the **Help Navigator** pane provide different ways to find information.

Use the close box to hide the pane.

Drag the separator bar to adjust the width of the panes.

View documentation in the display pane.





## See Also

builddocsearchdb, doc, docopt, docsearch, help, helpdesk, helpwin, lookfor, web

Related topics in the MATLAB Desktop Tools and Development Environment documentation:

- “Assistance While Entering Statements”, and especially “Finding Functions Using the Function Browser”
- “Help and Related Resources”

# helpdesk

---

**Purpose** Open Help browser

**Syntax** helpdesk

**Description** helpdesk displays the Help browser to its default startup page. In previous releases, helpdesk displayed the Help Desk, which was the precursor to the Help browser. In a future release, the helpdesk function will be phased out — use the doc or helpbrowser function instead.

**See Also** doc, helpbrowser

**Purpose** Create and open help dialog box

**Syntax**

```
helpdlg  
helpdlg('helpstring')  
helpdlg('helpstring','dlgname')  
h = helpdlg(...)
```

**Description** helpdlg creates a nonmodal help dialog box or brings the named help dialog box to the front.

---

**Note** A nonmodal dialog box enables the user to interact with other windows before responding. For more information, see `WindowState` in the MATLAB Figure Properties.

---

helpdlg displays a dialog box named 'Help Dialog' containing the string 'This is the default help string.'

helpdlg('helpstring') displays a dialog box named 'Help Dialog' containing the string specified by 'helpstring'.

helpdlg('helpstring','dlgname') displays a dialog box named 'dlgname' containing the string 'helpstring'.

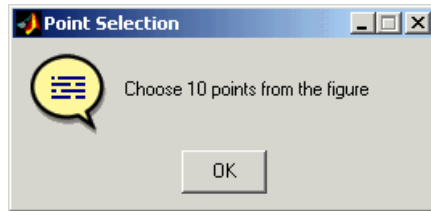
h = helpdlg(...) returns the handle of the dialog box.

**Remarks** MATLAB wraps the text in 'helpstring' to fit the width of the dialog box. The dialog box remains on your screen until you press the **OK** button or the **Enter** key. After either of these actions, the help dialog box disappears.

**Examples** The statement

```
helpdlg('Choose 10 points from the figure','Point Selection');
```

displays this dialog box:



## See Also

`dialog`, `errorDlg`, `inputdlg`, `listdlg`, `msgbox`, `questdlg`, `warndlg`  
`figure`, `uiwait`, `uiresume`

“Predefined Dialog Boxes” on page 1-108 for related functions

**Purpose** Provide access to M-file help for all functions

**Syntax** `helpwin`  
`helpwin topic`

**Description** `helpwin` lists topics for groups of functions in the Help browser. It shows brief descriptions of the topics and provides links to display M-file help for the functions in the Help browser. You cannot follow links in the `helpwin` list of functions if the MATLAB software is busy (for example, running a program).

`helpwin topic` displays help information for the topic in the Help browser. If `topic` is a directory, it displays all functions in the directory. If `topic` is a function, `helpwin` displays M-file help for that function in the Help browser. From the page, you can access a list of directories (**Default Topics** link) as well as the reference page help for the function (**Go to online doc** link). You cannot follow links in the `helpwin` list of functions if MATLAB is busy (for example, running a program).

**Examples** Typing

```
helpwin datafun
```

displays the functions in the `datafun` directory and a brief description of each.

Typing

```
helpwin fft
```

displays the M-file help for the `fft` function in the Help browser.

**See Also** `doc`, `docopt`, `help`, `helpbrowser`, `lookfor`, `web`

# hess

---

**Purpose** Hessenberg form of matrix

**Syntax**  
 $H = \text{hess}(A)$   
 $[P,H] = \text{hess}(A)$   
 $[AA, BB, Q, Z] = \text{hess}(A, B)$

**Description**  $H = \text{hess}(A)$  finds  $H$ , the Hessenberg form of matrix  $A$ .  
 $[P,H] = \text{hess}(A)$  produces a Hessenberg matrix  $H$  and a unitary matrix  $P$  so that  $A = P*H*P'$  and  $P'*P = \text{eye}(\text{size}(A))$ .  
 $[AA, BB, Q, Z] = \text{hess}(A, B)$  for square matrices  $A$  and  $B$ , produces an upper Hessenberg matrix  $AA$ , an upper triangular matrix  $BB$ , and unitary matrices  $Q$  and  $Z$  such that  $Q*A*Z = AA$  and  $Q*B*Z = BB$ .

**Definition** A Hessenberg matrix is zero below the first subdiagonal. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original, but less computation is needed to reveal them.

**Examples**  $H$  is a 3-by-3 eigenvalue test matrix:

```
H =  
  -149    -50   -154  
   537    180    546  
   -27     -9   -25
```

Its Hessenberg form introduces a single zero in the (3,1) position:

```
hess(H) =  
  -149.0000    42.2037   -156.3165  
  -537.6783   152.5511   -554.9272  
           0     0.0728     2.4489
```

**Algorithm** **Inputs of Type Double**

For inputs of type double, `hess` uses the following LAPACK routines to compute the Hessenberg form of a matrix:

<b>Matrix A</b>	<b>Routine</b>
Real symmetric	DSYTRD DSYTRD, DORGTR, (with output P)
Real nonsymmetric	DGEHRD DGEHRD, DORGHR (with output P)
Complex Hermitian	ZHETRD ZHETRD, ZUNGTR (with output P)
Complex non-Hermitian	ZGEHRD ZGEHRD, ZUNGHR (with output P)

### Inputs of Type Single

For inputs of type `single`, `hess` uses the following LAPACK routines to compute the Hessenberg form of a matrix:

<b>Matrix A</b>	<b>Routine</b>
Real symmetric	SSYTRD SSYTRD, DORGTR, (with output P)
Real nonsymmetric	SGEHRD SGEHRD, SORGHR (with output P)
Complex Hermitian	CHETRD CHETRD, CUNGTR (with output P)
Complex non-Hermitian	CGEHRD CGEHRD, CUNGHR (with output P)

### See Also

`eig`, `qz`, `schur`

### References

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling,

A. McKenney, and D. Sorensen, *LAPACK User's Guide*  
([http://www.netlib.org/lapack/lug/lapack\\_lug.html](http://www.netlib.org/lapack/lug/lapack_lug.html)), Third  
Edition, SIAM, Philadelphia, 1999.



**Purpose** Convert hexadecimal number string to decimal number

**Syntax** `d = hex2dec('hex_value')`

**Description** `d = hex2dec('hex_value')` converts *hex\_value* to its floating-point integer representation. The argument *hex\_value* is a hexadecimal integer stored in a MATLAB string. The value of *hex\_value* must be smaller than hexadecimal 10,000,000,000,000.

If *hex\_value* is a character array, each row is interpreted as a hexadecimal string.

**Examples** `hex2dec('3ff')`

```
ans =
```

```
1023
```

For a character array S,

```
S =  
0FF  
2DE  
123  
hex2dec(S)
```

```
ans =
```

```
255  
734  
291
```

**See Also** `dec2hex`, `format`, `hex2num`, `sprintf`

# hex2num

---

**Purpose** Convert hexadecimal number string to double-precision number

**Syntax** `n = hex2num(S)`

**Description** `n = hex2num(S)`, where `S` is a 16 character string representing a hexadecimal number, returns the IEEE double-precision floating-point number `n` that it represents. Fewer than 16 characters are padded on the right with zeros. If `S` is a character array, each row is interpreted as a double-precision number.

NaNs, infinities and denorms are handled correctly.

**Example** `hex2num('400921fb54442d18')`

returns `Pi`.

`hex2num('bff')`

returns

`ans =`

`-1`

**See Also** `num2hex`, `hex2dec`, `sprintf`, `format`

<b>Purpose</b>	Export figure
<b>GUI Alternative</b>	Use the <b>File</b> → <b>Saveas</b> on the figure window menu to access the <b>Export Setup</b> GUI. Use <b>Edit</b> → <b>Copy Figure</b> to copy the figure's contents to your system's clipboard. For details, see How to Print or Export in the MATLAB Graphics documentation.
<b>Syntax</b>	<code>hgexport(h,filename)</code> <code>hgexport(h,'-clipboard')</code>
<b>Description</b>	<code>hgexport(h,filename)</code> writes figure <code>h</code> to the file <code>filename</code> . <code>hgexport(h,'-clipboard')</code> writes figure <code>h</code> to the Microsoft Windows clipboard. The format in which the figure is exported is determined by which renderer you use. The Painters renderer generates a metafile. The ZBuffer and OpenGL renderers generate a bitmap.
<b>See Also</b>	<code>print</code>

# hggroup

---

**Purpose** Create hggroup object

## Syntax

**Description** An hggroup object can be the parent of any axes children except light objects, as well as other hggroup objects. You can use hggroup objects to form a group of objects that can be treated as a single object with respect to the following cases:

- **Visible** — Setting the hggroup object's `Visible` property also sets each child object's `Visible` property to the same value.
- **Selectable** — Setting each hggroup child object's `HitTest` property to `off` enables you to select all children by clicking any child object.
- **Current object** — Setting each hggroup child object's `HitTest` property to `off` enables the hggroup object to become the current object when any child object is picked. See the next section for an example.

## Examples

This example defines a callback for the `ButtonDownFcn` property of an hggroup object. In order for the hggroup to receive the mouse button down event that executes the `ButtonDownFcn` callback, the `HitTest` properties of all the line objects must be set to `off`. The event is then passed up the hierarchy to the hggroup.

The following function creates a random set of lines that are parented to an hggroup object. The subfunction `set_lines` defines a callback that executes when the mouse button is pressed over any of the lines. The callback simply increases the widths of all the lines by 1 with each button press.

---

**Note** If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

---

```
function doc_hggroup
```

```

hg = hgggroup('ButtonDownFcn',@set_lines);
hl = line(randn(5),randn(5),'HitTest','off','Parent',hg);

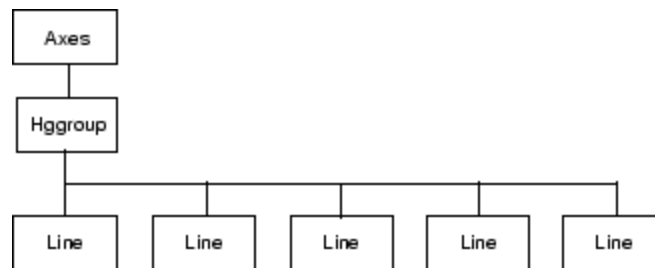
function set_lines(cb,eventdata)
hl = get(cb,'Children');% cb is handle of hgggroup object
lw = get(hl,'LineWidth');% get current line widths
set(hl,{'LineWidth'},num2cell([lw{:}]+1,[5,1]))

```

Note that selecting any one of the lines selects all the lines. (To select an object, enable plot edit mode by selecting **Plot Edit** from the **Tools** menu.)

### Instance Diagram for This Example

The following diagram shows the object hierarchy created by this example.



## Hgggroup Properties

### Setting Default Properties

You can set default hgggroup properties on the axes, figure, and root levels.

```

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

```

where *Property* is the name of the hgggroup property whose default value you want to set and *PropertyValue* is the value you are specifying. Use `set` and `get` to access the hgggroup properties.

# hggroup

---

## **See Also**

hgtransform

“Group Objects” for more information and examples.

“Function Handle Callbacks” for information on how to use function handles to define callbacks.

Hggroup Properties for property descriptions

## Purpose

Hgggroup properties

## Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands.

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

See “Group Objects” for general information on this type of object.

## Hgggroup Property Descriptions

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

### Annotation

`hg.Annotation` object Read Only

*Control the display of hgggroup objects in legends.* The `Annotation` property enables you to specify whether this hgggroup 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 hgggroup object is displayed in a figure legend:

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
<code>on</code>	Include the hgggroup object in a legend as one entry, but not its children objects

# Hgggroup Properties

---

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
off	Do not include the hgggroup or its children in a legend (default)
children	Include only the children of the hgggroup as separate entries in the legend

## 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 whether objects are in the process of being deleted. The MATLAB software sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions 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`

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

*Button press callback function.* A callback function that executes whenever you press a mouse button while the pointer is over the children of the `hgroup` object. Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with the mouse button press and an empty event structure).

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

# Hgroup Properties

---

## Children

array of graphics object handles

*Children of the hgroup object.* An array containing the handles of all objects parented to the hgroup object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not appear in the hgroup `Children` property unless you set the `Root ShowHiddenHandles` property to `on`:

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

## Clipping

{on} | off

*Clipping mode.* MATLAB clips stairs plots to the axes plot box by default. If you set `Clipping` to `off`, lines might be displayed outside the axes plot box.

## CreateFcn

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

*Callback executed during object creation.* This property defines a callback function that executes when MATLAB creates an hgroup object. You must define this property as a default value for hgroup objects or in a call to the hgroup function to create a new hgroup object. For example, the statement

```
set(0, 'DefaultHgroupCreateFcn', @myCreateFcn)
```

defines a default value on the root level that applies to every hgroup object created in that MATLAB session. Whenever you create an hgroup object, the function associated with the function handle `@myCreateFcn` executes.

MATLAB executes the callback after setting all the hgggroup object's properties. Setting the `CreateFcn` property on an existing hgggroup object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See [Function Handle Callbacks](#) for information on how to use function handles to define the callback function.

## `DeleteFcn`

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

*Callback executed during object deletion.* A callback function that executes when the hgggroup object is deleted (e.g., this might happen when you issue a `delete` command on the hgggroup 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 passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See ["Function Handle Callbacks"](#) for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

## `DisplayName`

string (default is empty string)

# Hgroup Properties

---

*String used by legend for this hgroup object.* The `legend` function uses the string defined by the `DisplayName` property to label this hgroup object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this hgroup object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, legend does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

## EraseMode

`{normal} | none | xor | background`

*Erase mode.* This property controls the technique MATLAB uses to draw and erase hgroup child objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

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

- **none** — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- **xor** — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- **background** — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

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

## Printing with Nonnormal Erase Modes

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

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

# Hgroup Properties

---

HandleVisibility  
{on} | callback | off

*Control access to object's handle by command-line users and GUIs.* This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the hgroup object.

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

## Functions Affected by Handle Visibility

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

## Properties Affected by Handle Visibility

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

the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

## Overriding Handle Visibility

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

## Handle Validity

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

`HitTest`  
{on} | off

*Pickable by mouse click.* `HitTest` determines whether the hgroup 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 hgroup child objects. Note that to pick the hgroup object, its children must have their `HitTest` property set to `off`.

If the hgroup object's `HitTest` is `off`, clicking it picks the object behind it.

`Interruptible`  
{on} | off

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

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for

# Hgroup Properties

---

events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from an `hggroup` 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.

## Parent

axes handle

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

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

## Selected

`on` | `{off}`

*Is object selected?* When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints of `hggroup` child objects if the `SelectionHighlight` property is also `on` (the default).

## 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 `hggroup` child objects. 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 `hgroup` object and set the `Tag` property:

```
t = hgroup('Tag','group1')
```

When you want to access the object, you can use `findobj` to find its handle. For example,

```
h = findobj('Tag','group1');
```

## Type

string (read only)

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

```
t = findobj(gca,'Type','hgroup');
```

## UIContextMenu

handle of a `uicontextmenu` object

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

## UserData

array

# Hgroup Properties

---

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

**Visible**

{on} | off

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

---

<b>Purpose</b>	Load Handle Graphics object hierarchy from file
<b>GUI Alternative</b>	Use the <b>File</b> → <b>Open</b> on the figure window menu to access figure files with the <b>Open</b> dialog.
<b>Syntax</b>	<pre>h = hgload('filename') [h,old_prop_values] = hgload(...,property_structure) hgload(...,'all')</pre>
<b>Description</b>	<p><code>h = hgload('filename')</code> loads Handle Graphics objects and its children (if any) from the FIG-file specified by <code>filename</code> and returns handles to the top-level objects. If <code>filename</code> contains no extension, then the MATLAB software adds the <code>.fig</code> extension.</p> <p><code>[h,old_prop_values] = hgload(...,property_structure)</code> overrides the properties on the top-level objects stored in the FIG-file with the values in <code>property_structure</code>, and returns their previous values in <code>old_prop_values</code>.</p> <p><code>property_structure</code> must be a structure having field names that correspond to property names and values that are the new property values.</p> <p><code>old_prop_values</code> is a cell array equal in length to <code>h</code>, containing the old values of the overridden properties for each object. Each cell contains a structure having field names that are property names, each of which contains the original value of each property that has been changed. Any property specified in <code>property_structure</code> that is not a property of a top-level object in the FIG-file is not included in <code>old_prop_values</code>.</p> <p><code>hgload(...,'all')</code> overrides the default behavior, which does not reload nonserializable objects saved in the file. These objects include the default toolbars and default menus.</p> <p>Nonserializable objects (such as the default toolbars and the default menus) are normally not reloaded because they are loaded from different files at figure creation time. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files.</p>

# hgload

---

Passing the string `all` to `hgload` ensures that any nonserializable objects contained in the file are also reloaded.

Note that, by default, `hgsave` excludes nonserializable objects from the FIG-file unless you use the `all` flag.

## See Also

`hgsave`, `open`

“Figure Windows” on page 1-99 for related functions

<b>Purpose</b>	Save Handle Graphics object hierarchy to file
<b>GUI Alternative</b>	Use the <b>File</b> → <b>Saves</b> on the figure window menu to access the Export Setup GUI. For details, see How to Print or Export in the MATLAB Graphics documentation.
<b>Syntax</b>	<pre>hgsave('filename') hgsave(h,'filename') hgsave(...,'all') hgsave(...,'-v6') hgsave(...,'-v7.3')</pre>
<b>Description</b>	<p><code>hgsave('filename')</code> saves the current figure to a file named <code>filename</code>.</p> <p><code>hgsave(h,'filename')</code> saves the objects identified by the array of handles <code>h</code> to a file named <code>filename</code>. If you do not specify an extension for <code>filename</code>, then the extension <code>.fig</code> is appended. If <code>h</code> is a vector, none of the handles in <code>h</code> may be ancestors or descendents of any other handles in <code>h</code>.</p> <p><code>hgsave(...,'all')</code> overrides the default behavior, which does not save nonserializable objects. Nonserializable objects include the default toolbars and default menus. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files and also reduces the size of FIG-files. Passing the string <code>all</code> to <code>hgsave</code> ensures that nonserializable objects are also saved.</p> <p>Note: the default behavior of <code>hgload</code> is to ignore nonserializable objects in the file at load time. This behavior can be overwritten using the <code>all</code> argument with <code>hgload</code>.</p> <p><code>hgsave(...,'-v6')</code> saves the FIG-file in a format that can be loaded by versions prior to MATLAB 7.</p> <p><code>hgsave(...,'-v7.3')</code> saves the FIG-file in a format that can be loaded only by MATLAB versions 7.3 and above. This format, based on HDF5 files, is intended for saving FIG-files larger than 2 GB.</p> <p>You can make <code>-v6</code> or <code>-v7.3</code> your default format for saving MAT-files and FIG-files by setting a preference, which will eliminate the need to</p>

specify the flag each time you save. See “MAT-Files Preferences” in the MATLAB Desktop Tools and Development Environment documentation.

## **Full Backward Compatibility**

When creating a figure you want to save and use in a MATLAB version prior to MATLAB 7, use the 'v6' option with the plotting function and the '-v6' option for hgsave. Check the reference page for the plotting function you are using for more information.

See “Plot Objects and Backward Compatibility” for more information.

## **See Also**

hgload, open, save

“Figure Windows” on page 1-99 for related functions

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

**Syntax** `classdef myclass < hgsetget`

**Description** `classdef myclass < hgsetget` makes *myclass* a subclass of the `hgsetget` class, which is a subclass of the `handle` class.

Use the `hgsetget` class to derive classes that inherit `set` and `get` methods that behave like Handle Graphics `set` and `get` functions.

### hgsetget Class Methods

When you derive a class from the `hgsetget` class, your class inherits the following methods.

Method	Purpose
<code>set</code>	Assigns values to the specified properties or returns a cell array of possible values for writable properties.
<code>get</code>	Returns value of specified property or a <code>struct</code> with all property values.
<code>setdisp</code>	Called when <code>set</code> is called with no output arguments and a handle array, but no property name. Override this method to change what <code>set</code> displays.
<code>getdisp</code>	Called when <code>get</code> is called with no output arguments and handle array, bet no property name. Override this method to change what <code>get</code> displays.

**See Also** See “Implementing a Set/Get Interface for Properties”

`handle`, `set (hgsetget)`, `get (hgsetget)`, `set`, `get`

# hgtransform

---

**Purpose** Create hgtransform graphics object

**Syntax**  
`h = hgtransform`  
`h = hgtransform('PropertyName',propertyvalue,...)`

**Description** `h = hgtransform` creates an hgtransform object and returns its handle.  
`h = hgtransform('PropertyName',propertyvalue,...)` creates an hgtransform object with the property value settings specified in the argument list.

hgtransform objects can contain other objects, which lets you treat the hgtransform and its children as a single entity with respect to visibility, size, orientation, etc. You can group objects by parenting them to a single hgtransform object (i.e., setting the object's `Parent` property to the hgtransform object's handle):

```
h = hgtransform;  
surface('Parent',h,...)
```

The primary advantage of parenting objects to an hgtransform object is that you can perform *transforms* (e.g., translation, scaling, rotation, etc.) on the child objects in unison.

The parent of an hgtransform object is either an axes object or another hgtransform.

Although you cannot see an hgtransform object, setting its `Visible` property to `off` makes all its children invisible as well.

## Exceptions and Limitations

- An hgtransform object can be the parent of any number of axes child objects belonging to the same axes, except for light objects.
- hgtransform objects can never be the parent of axes objects and therefore can contain objects only from a single axes.
- hgtransform objects can be the parent of other hgtransform objects within the same axes.



- You cannot transform image objects because images are not true 3-D objects. Texture mapping the image data to a surface CData enables you to produce the effect of transforming an image in 3-D space.

---

**Note** Many plotting functions clear the axes (i.e., remove axes children) before drawing the graph. Clearing the axes also deletes any hgtransform objects in the axes.

---

## More Information

- References in “See Also” on page 2-1645 provide information on types of transforms
- “Examples” on page 2-1635 provide examples that illustrate the use of transforms.

## Examples

### Transforming a Group of Objects

This example shows how to create a 3-D star with a group of surface objects parented to a single hgtransform object. The hgtransform then rotates the object about the z-axis while scaling its size.

---

**Tip** If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB Editor.

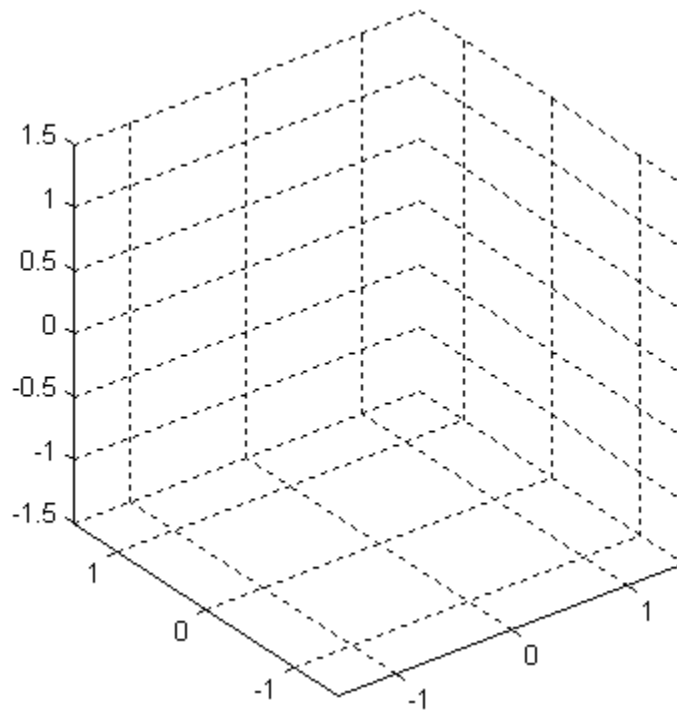
---

- 1 Create an axes and adjust the view. Set the axes limits to prevent auto limit selection during scaling.

```
ax = axes('XLim',[-1.5 1.5],'YLim',[-1.5 1.5],...  
         'ZLim',[-1.5 1.5]);  
view(3); grid on; axis equal
```

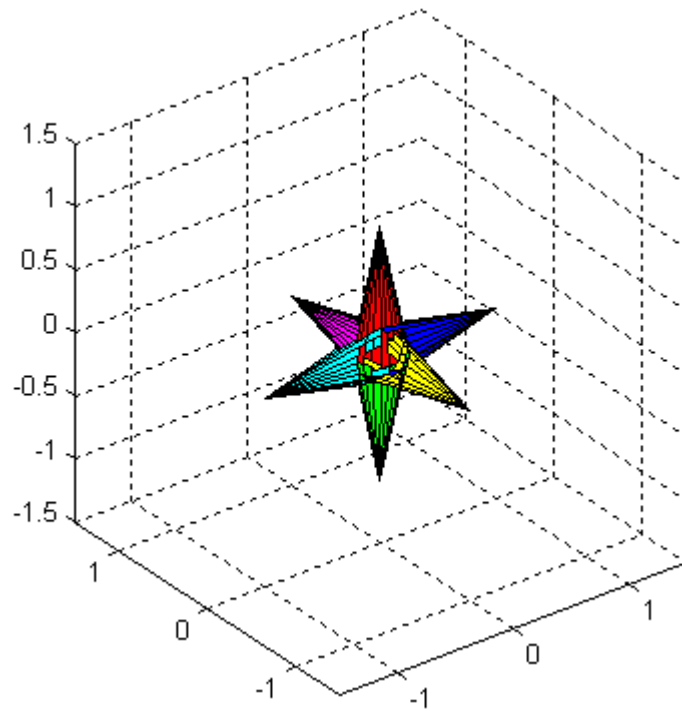
# hgtransform

---



**2** Create the objects you want to parent to the hgtransform object.

```
[x y z] = cylinder([.2 0]);  
h(1) = surface(x,y,z,'FaceColor','red');  
h(2) = surface(x,y,-z,'FaceColor','green');  
h(3) = surface(z,x,y,'FaceColor','blue');  
h(4) = surface(-z,x,y,'FaceColor','cyan');  
h(5) = surface(y,z,x,'FaceColor','magenta');  
h(6) = surface(y,-z,x,'FaceColor','yellow');
```



- 3** Create an hgtransform object and parent the surface objects to it. The figure should not change from the image above.

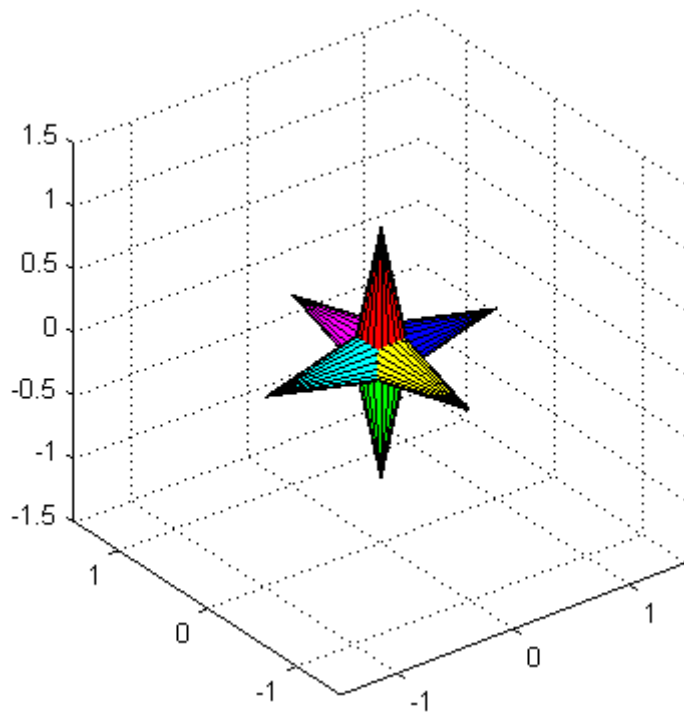
```
t = hgtransform('Parent',ax);  
set(h,'Parent',t)
```

- 4** Select a renderer and show the objects.

```
set(gcf,'Renderer','opengl')  
drawnow
```

# hgtransform

---



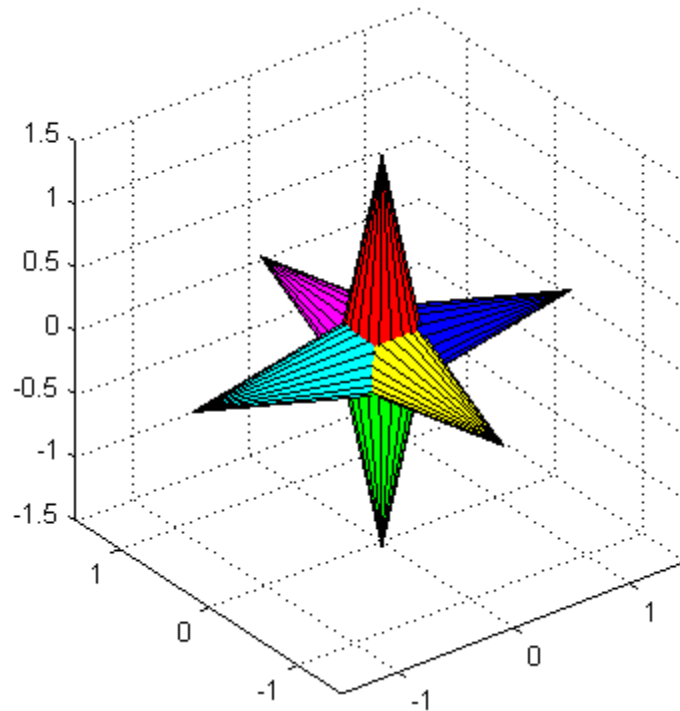
- 5 Initialize the rotation and scaling matrix to the identity matrix (`eye`). Again, the image should not change.

```
Rz = eye(4);  
Sxy = Rz;
```

- 6 Form the  $z$ -axis rotation matrix and the scaling matrix. Rotate 360 degrees ( $2\pi$  radians) and scale by using the increasing values of  $r$ .

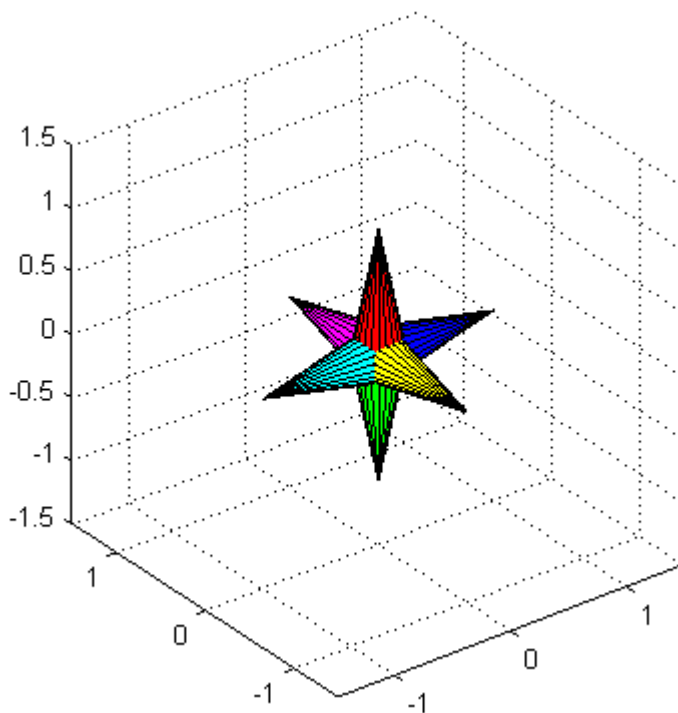
```
for r = 1:.1:2*pi  
    % Z-axis rotation matrix  
    Rz = makehgtform('zrotate',r);  
    % Scaling matrix  
    Sxy = makehgtform('scale',r/4);
```

```
% Concatenate the transforms and  
% set the hgtransform Matrix property  
set(t,'Matrix',Rz*Sxy)  
drawnow  
end  
pause(1)
```



7 Reset to the original orientation and size using the identity matrix.

```
set(t,'Matrix',eye(4))
```



## Transforming Objects Independently

This example creates two `hgtransform` objects to illustrate how to transform each independently within the same axes. A translation transformation moves one `hgtransform` object away from the origin.

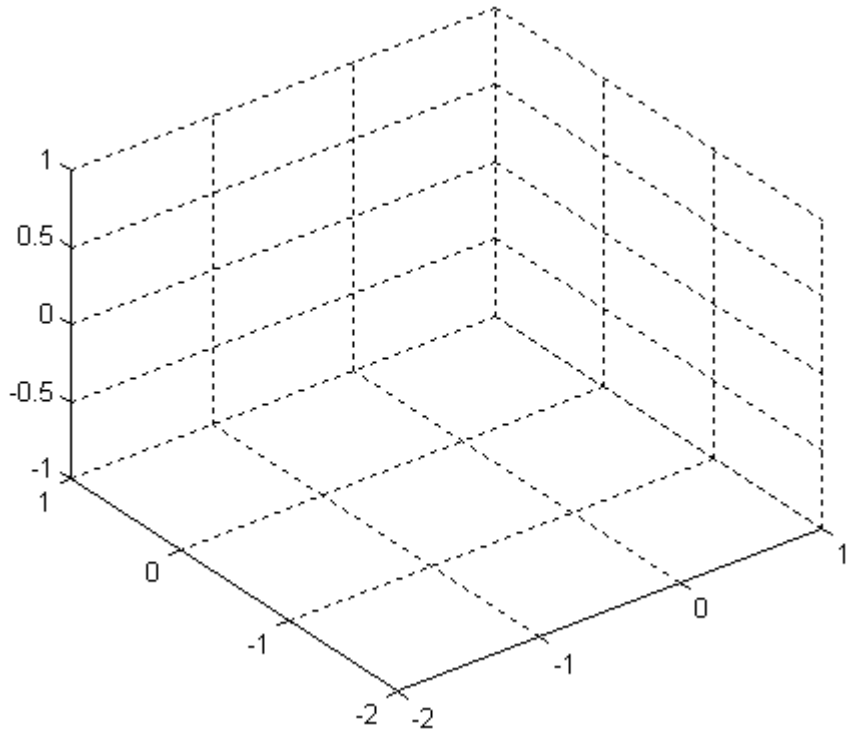
---

**Tip** If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB Editor.

---

- 1 Create and set up the axes object that will be the parent of both `hgtransform` objects. Set the limits to accommodate the translated object.

```
ax = axes('XLim',[-2 1],'YLim',[-2 1],'ZLim',[-1 1]);  
view(3); grid on; axis equal
```

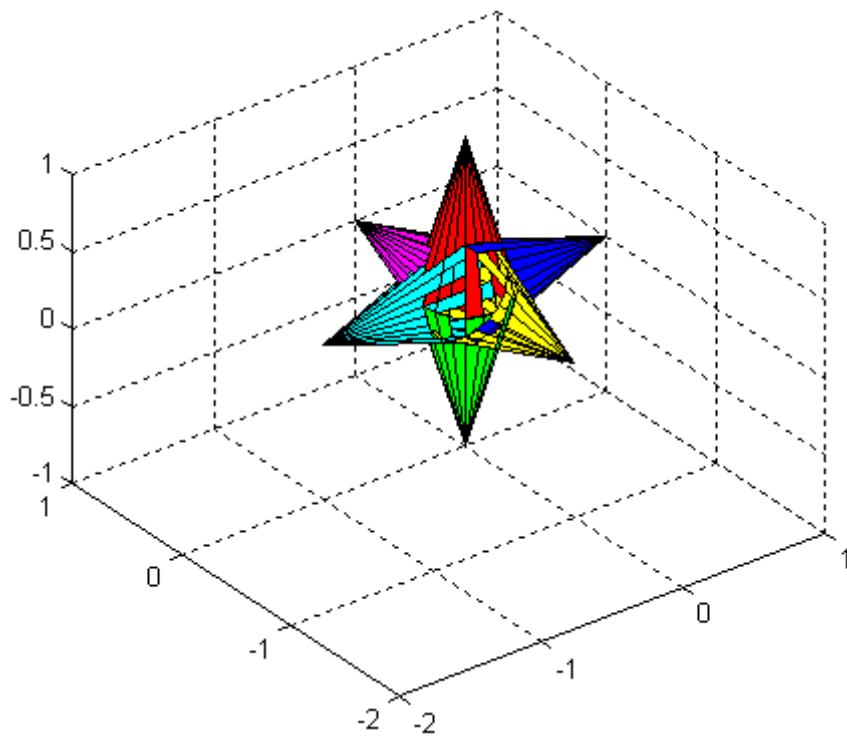


**2** Create the surface objects to group.

```
[x y z] = cylinder([.3 0]);  
h(1) = surface(x,y,z,'FaceColor','red');  
h(2) = surface(x,y,-z,'FaceColor','green');  
h(3) = surface(z,x,y,'FaceColor','blue');  
h(4) = surface(-z,x,y,'FaceColor','cyan');  
h(5) = surface(y,z,x,'FaceColor','magenta');  
h(6) = surface(y,-z,x,'FaceColor','yellow');
```

# hgtransform

---



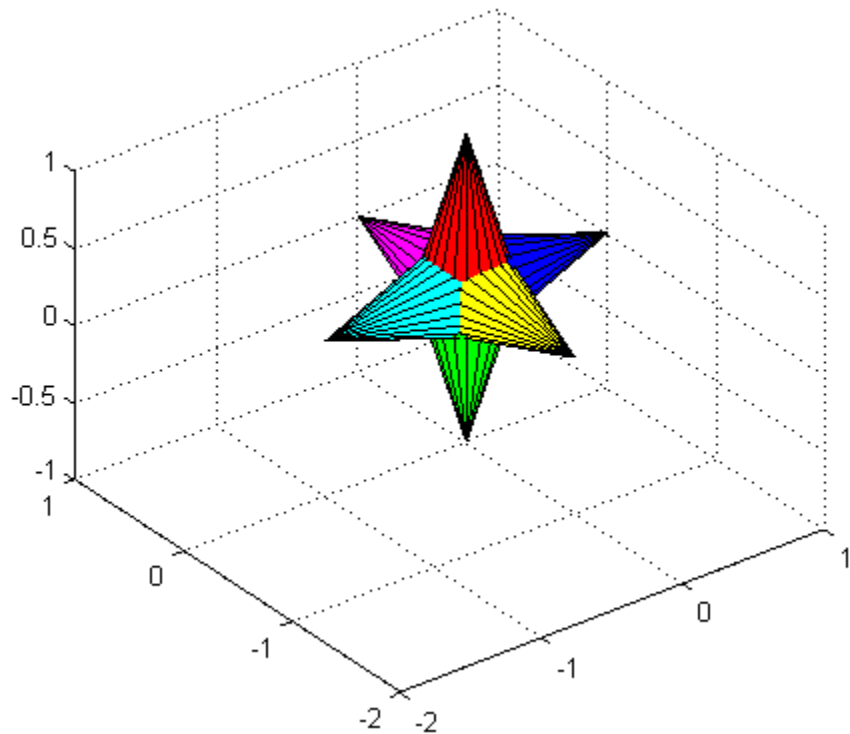
- 3** Create the hgtransform objects and parent them to the same axes. The figure should not change.

```
t1 = hgtransform('Parent',ax);  
t2 = hgtransform('Parent',ax);
```

- 4** Set the renderer to use OpenGL.

```
set(gcf,'Renderer','opengl')
```





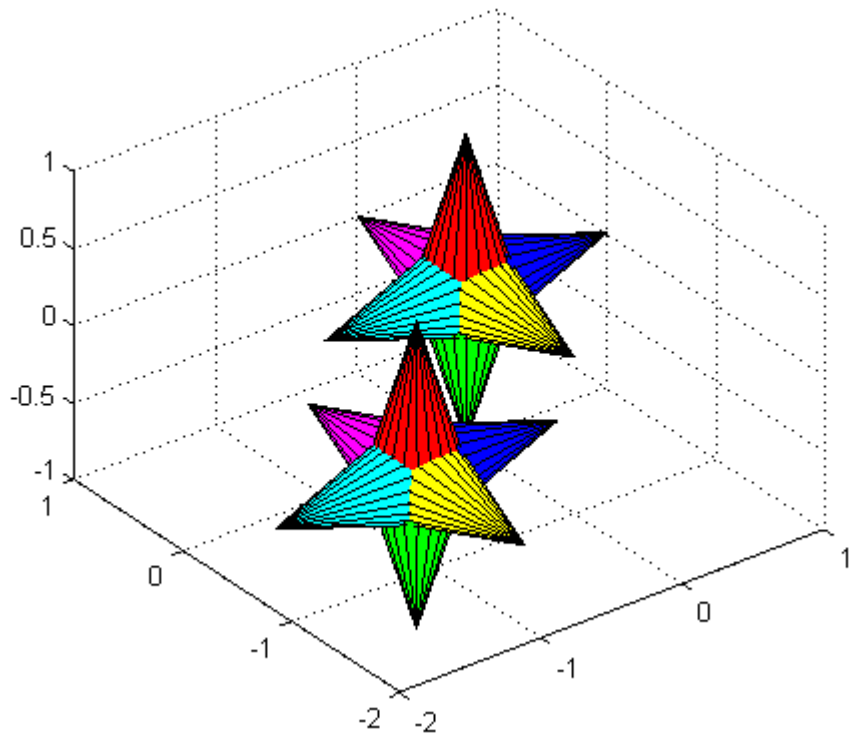
- 5** Parent the surfaces to hgtransform t1, then copy the surface objects and parent the copies to hgtransform t2. This figure should not change.

```
set(h,'Parent',t1)
h2 = copyobj(h,t2);
```

- 6** Translate the second hgtransform object away from the first hgtransform object and display the result.

```
Txy = makehgtform('translate',[-1.5 -1.5 0]);
set(t2,'Matrix',Txy)
drawnow
```

# hgtransform



- 7 Rotate both hgtransform objects in opposite directions. The final image for this step is the same as for step 6. However, you should run the code to see the rotations.

```
% Rotate 10 times (2pi radians = 1 rotation)
for r = 1:.1:20*pi
    % Form z-axis rotation matrix
    Rz = makehgtform('zrotate',r);
    % Set transforms for both hgtransform objects
    set(t1,'Matrix',Rz)
    set(t2,'Matrix',Txy*inv(Rz))
    drawnow
end
```

## Setting Default Properties

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

```
set(0, 'DefaultHgtransformPropertyname', propertyvalue, ...)
set(gcf, 'DefaultHgtransformPropertyname', propertyvalue, ...)
set(gca, 'DefaultHgtransformPropertyname', propertyvalue, ...)
```

*Propertyname* is the name of the hgtransform property and *propertyvalue* is the specified value. Use `set` and `get` to access hgtransform properties.

## See Also

hggroup, makehgtform

Tomas Moller and Eric Haines, *Real-Time Rendering*, A K Peters, Ltd., 1999 for more information about transforms.

“Group Objects” in MATLAB Graphics documentation for more information and examples.

Hgtransform Properties for property descriptions.

# Hgtransform Properties

---

**Purpose** Hgtransform properties

## Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands.

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

See “Group Objects” for general information on this type of object.

## Hgtransform Property Descriptions

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

Annotation

`hg.Annotation` object Read Only

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

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
<code>on</code>	Include the hgtransform object in a legend as one entry, but not its children objects

IconDisplayStyle Value	Purpose
off	Do not include the hgtransform or its children in a legend (default)
children	Include only the children of the hgtransform as separate entries in the legend

## 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 whether objects are in the process of being deleted. The MATLAB software sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to

# Hgtransform Properties

---

be deleted, and therefore can check the object's `BeingDeleted` property before acting.

`BusyAction`  
cancel | {queue}

*Callback routine interruption.* The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback functions. 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

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

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

*Button press callback function.* A callback function that executes whenever you press a mouse button while the pointer is within the extent of the `hgtransform` object, but not over another graphics object. The extent of an `hgtransform` object is the smallest rectangle that encloses all the children. Note that you cannot execute the `hgtransform` object's button down function if it has no children.

Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments (handle of figure

associated with the mouse button press and an empty event structure).

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

## Children

array of graphics object handles

*Children of the hgtransform object.* An array containing the handles of all graphics objects parented to the hgtransform object (whether visible or not).

The graphics objects that can be children of an hgtransform are images, lights, lines, patches, rectangles, surfaces, and text. You can change the order of the handles and thereby change the stacking of the objects on the display.

Note that if a child object’s `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in the hgtransform `Children` property unless you set the `Root ShowHiddenHandles` property to `on`.

## Clipping

{on} | off

This property has no effect on hgtransform objects.

## CreateFcn

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

*Callback executed during object creation.* This property defines a callback function that executes when MATLAB creates an hgtransform object. You must define this property as a default value for hgtransform objects. For example, the statement

```
set(0, 'DefaultHgtransformCreateFcn', @myCreateFcn)
```

# Hgtransform Properties

---

defines a default value on the root level that applies to every hgtransform object created in a MATLAB session. Whenever you create an hgtransform object, the function associated with the function handle `@myCreateFcn` executes.

MATLAB executes the callback after setting all the hgtransform object's properties. Setting the `CreateFcn` property on an existing hgtransform object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

## `DeleteFcn`

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

*Callback executed during object deletion.* A callback function that executes when the hgtransform object is deleted (e.g., this might happen when you issue a `delete` command on the hgtransform 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 passed by MATLAB as the first argument to the callback function and is 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.

See the `BeingDeleted` property for related information.



## DisplayName

string (default is empty string)

*String used by legend for this hgtransform object.* The legend function uses the string defined by the `DisplayName` property to label this hgtransform object in the legend.

- If you specify string arguments with the legend function, `DisplayName` is set to this hgtransform object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where `n` is the number assigned to the object based on its location in the list of legend entries. However, legend does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call legend with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

## EraseMode

{normal} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase hgtransform child objects (light objects have no erase mode). 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

# Hgtransform Properties

---

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.

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

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

## Printing with Nonnormal Erase Modes

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

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

## HandleVisibility

{on} | callback | off

*Control access to object's handle by command-line users and GUIs.*

This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the hgtransform object.

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

## Functions Affected by Handle Visibility

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

## Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property,

# Hgtransform Properties

---

figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

## Overriding Handle Visibility

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

## Handle Validity

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

`HitTest`  
{on} | off

*Pickable by mouse click.* `HitTest` determines whether the hgtransform object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click within the limits of the hgtransform object. If `HitTest` is off, clicking the hgtransform picks the object behind it.

`Interruptible`  
{on} | off

*Callback routine interruption mode.* The `Interruptible` property controls whether an hgtransform object callback can be interrupted by callbacks invoked subsequently. Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`,

`getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from an `hgtransform` 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.

## Matrix

4-by-4 matrix

*Transformation matrix applied to `hgtransform` object and its children.* The `hgtransform` object applies the transformation matrix to all its children.

See “Group Objects” for more information and examples.

## Parent

figure handle

*Parent of `hgtransform` object.* This property contains the handle of the `hgtransform` object's parent object. The parent of an `hgtransform` object is the axes, `hgroup`, or `hgtransform` object that contains it.

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

## Selected

`on` | `{off}`

*Is object selected?* When you set this property to `on`, MATLAB displays selection handles on all child objects of the `hgtransform` if the `SelectionHighlight` property is also `on` (the default).

## SelectionHighlight

`{on}` | `off`

# Hgtransform Properties

---

*Objects are highlighted when selected.* When the **Selected** property is on, MATLAB indicates the selected state by drawing selection handles on the objects parented to the hgtransform. 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 hgtransform object and set the **Tag** property:

```
t = hgtransform('Tag','subgroup1')
```

When you want to access the hgtransform object to add another object, you can use `findobj` to find the hgtransform object's handle. The following statement adds a line to `subgroup1` (assuming `x` and `y` are defined).

```
line('XData',x,'YData',y,'Parent',findobj('Tag','subgroup1'))
```

## Type

string (read only)

*Type of graphics object.* This property contains a string that identifies the class of graphics object. For hgtransform objects, **Type** is set to `'hgtransform'`. The following statement finds all the hgtransform objects in the current axes.

```
t = findobj(gca,'Type','hgtransform');
```

## UIContextMenu

handle of a uicontextmenu object

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

**UserData**

array

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

**Visible**

{on} | off

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

# hidden

---

<b>Purpose</b>	Remove hidden lines from mesh plot
<b>Syntax</b>	<code>hidden on</code> <code>hidden off</code> <code>hidden</code>
<b>Description</b>	<p>Hidden line removal draws only those lines that are not obscured by other objects in the field of view.</p> <p><code>hidden on</code> turns on hidden line removal for the current graph so lines in the back of a mesh are hidden by those in front. This is the default behavior.</p> <p><code>hidden off</code> turns off hidden line removal for the current graph.</p> <p><code>hidden</code> toggles the hidden line removal state.</p>
<b>Algorithm</b>	<code>hidden on</code> sets the <code>FaceColor</code> property of a surface graphics object to the background <code>Color</code> of the axes (or of the figure if axes <code>Color</code> is none).
<b>Examples</b>	Set hidden line removal <code>off</code> and <code>on</code> while displaying the <code>peaks</code> function. <pre>mesh(peaks) hidden off hidden on</pre>
<b>See Also</b>	<code>shading</code> , <code>mesh</code> The surface properties <code>FaceColor</code> and <code>EdgeColor</code> “Surface and Mesh Creation” on page 1-101 for related functions



---

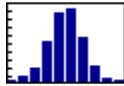
<b>Purpose</b>	Hilbert matrix
<b>Syntax</b>	<code>H = hilb(n)</code>
<b>Description</b>	<code>H = hilb(n)</code> returns the Hilbert matrix of order <code>n</code> .
<b>Definition</b>	The Hilbert matrix is a notable example of a poorly conditioned matrix [1]. The elements of the Hilbert matrices are $H(i, j) = 1/(i + j - 1)$ .
<b>Examples</b>	Even the fourth-order Hilbert matrix shows signs of poor conditioning. <pre>cond(hilb(4)) = 1.5514e+04</pre>
<b>See Also</b>	<code>invhilb</code>
<b>References</b>	[1] Forsythe, G. E. and C. B. Moler, <i>Computer Solution of Linear Algebraic Systems</i> , Prentice-Hall, 1967, Chapter 19.

# hist


---

## Purpose

Histogram plot



## GUI Alternatives

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

## Syntax

```
n = hist(Y)
n = hist(Y,x)
n = hist(Y,nbins)
[n,xout] = hist(...)
hist(...)
hist(axes_handle,...)
```

## Description

A histogram shows the distribution of data values.

`n = hist(Y)` bins the elements in vector `Y` into 10 equally spaced containers and returns the number of elements in each container as a row vector. If `Y` is an `m`-by-`p` matrix, `hist` treats the columns of `Y` as vectors and returns a 10-by-`p` matrix `n`. Each column of `n` contains the results for the corresponding column of `Y`. No elements of `Y` can be complex.

`n = hist(Y,x)` where `x` is a vector, returns the distribution of `Y` among `length(x)` bins with centers specified by `x`. For example, if `x` is a 5-element vector, `hist` distributes the elements of `Y` into five bins centered on the `x`-axis at the elements in `x`, none of which can be complex. Note: use `histc` if it is more natural to specify bin edges instead of centers.

`n = hist(Y,nbins)` where `nbins` is a scalar, uses `nbins` number of bins.

`[n,xout] = hist(...)` returns vectors `n` and `xout` containing the frequency counts and the bin locations. You can use `bar(xout,n)` to plot the histogram.

`hist(...)` without output arguments produces a histogram plot of the output described above. `hist` distributes the bins along the  $x$ -axis between the minimum and maximum values of  $Y$ .

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

## Remarks

All elements in vector  $Y$  or in one column of matrix  $Y$  are grouped according to their numeric range. Each group is shown as one bin.

The histogram's  $x$ -axis reflects the range of values in  $Y$ . The histogram's  $y$ -axis shows the number of elements that fall within the groups; therefore, the  $y$ -axis ranges from 0 to the greatest number of elements deposited in any bin. The  $x$ -range of the leftmost and rightmost bins extends to include the entire data range in the case when the user-specified range does not cover the data range; this often results in "boxes" at either or both edges of the distribution. If you want a plot in which this does not happen (that is, all bins have equal width), you can create a histogram-like display using the `bar` command.

Histograms bins are created as patch objects and always plotted with a face color that maps to the first color in the current colormap (by default, blue) and with black edges. To change colors or other patch properties, use code similar to that given in the example.

The `hist` function does not work with data that contain `inf` values.

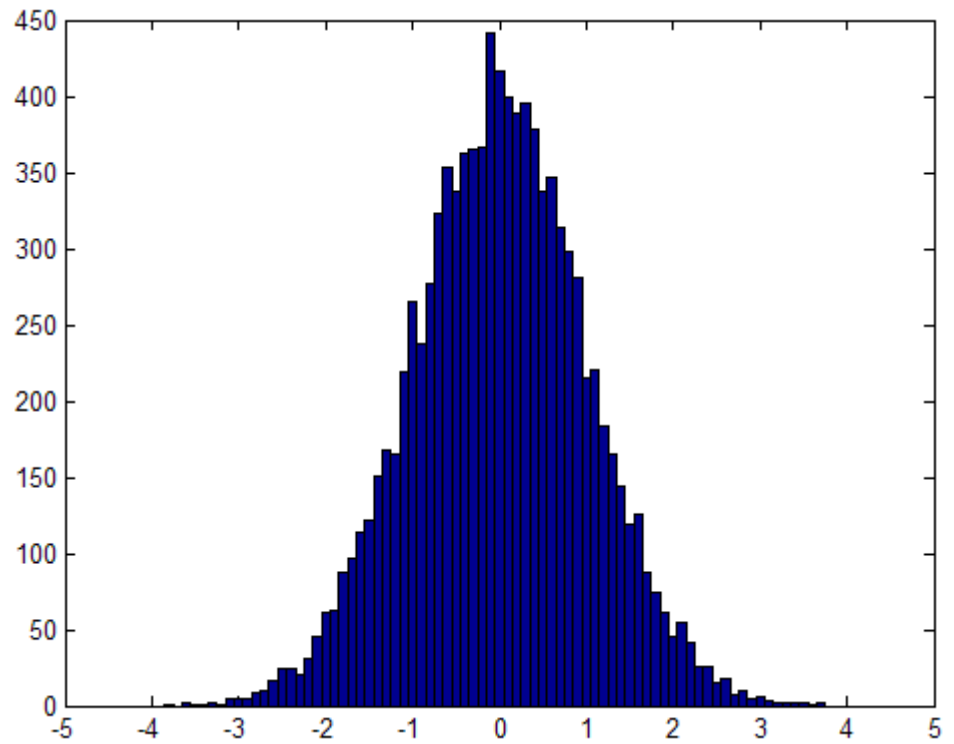
## Example

Generate a bell-curve histogram from Gaussian data.

```
x = -4:0.1:4;  
y = randn(10000,1);  
hist(y,x)
```

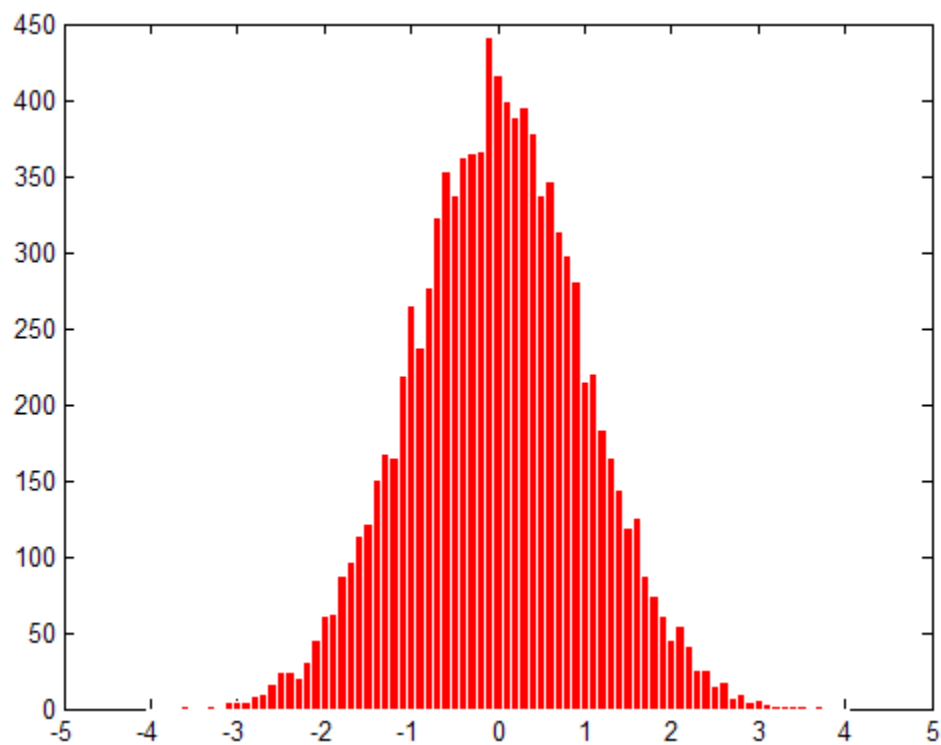
# hist

---



Change the color of the graph so that the bins are red and the edges of the bins are white.

```
h = findobj(gca,'Type','patch');  
set(h,'FaceColor','r','EdgeColor','w')
```

**See Also**

`bar`, `ColorSpec`, `histc`, `mode`, `patch`, `rose`, `stairs`

“Specialized Plotting” on page 1-92 for related functions

“Histograms” for examples

# histc

---

**Purpose** Histogram count

**Syntax**

```
n = histc(x,edges)
n = histc(x,edges,dim)
[n,bin] = histc(...)
```

**Description** `n = histc(x,edges)` counts the number of values in vector `x` that fall between the elements in the `edges` vector (which must contain monotonically nondecreasing values). `n` is a `length(edges)` vector containing these counts. No elements of `x` can be complex.

`n(k)` counts the value `x(i)` if `edges(k) <= x(i) < edges(k+1)`. The last bin counts any values of `x` that match `edges(end)`. Values outside the values in `edges` are not counted. Use `-inf` and `inf` in `edges` to include all non-NaN values.

For matrices, `histc(x,edges)` returns a matrix of column histogram counts. For N-D arrays, `histc(x,edges)` operates along the first nonsingleton dimension.

`n = histc(x,edges,dim)` operates along the dimension `dim`.

`[n,bin] = histc(...)` also returns an index matrix `bin`. If `x` is a vector, `n(k) = sum(bin==k)`. `bin` is zero for out of range values. If `x` is an M-by-N matrix, then

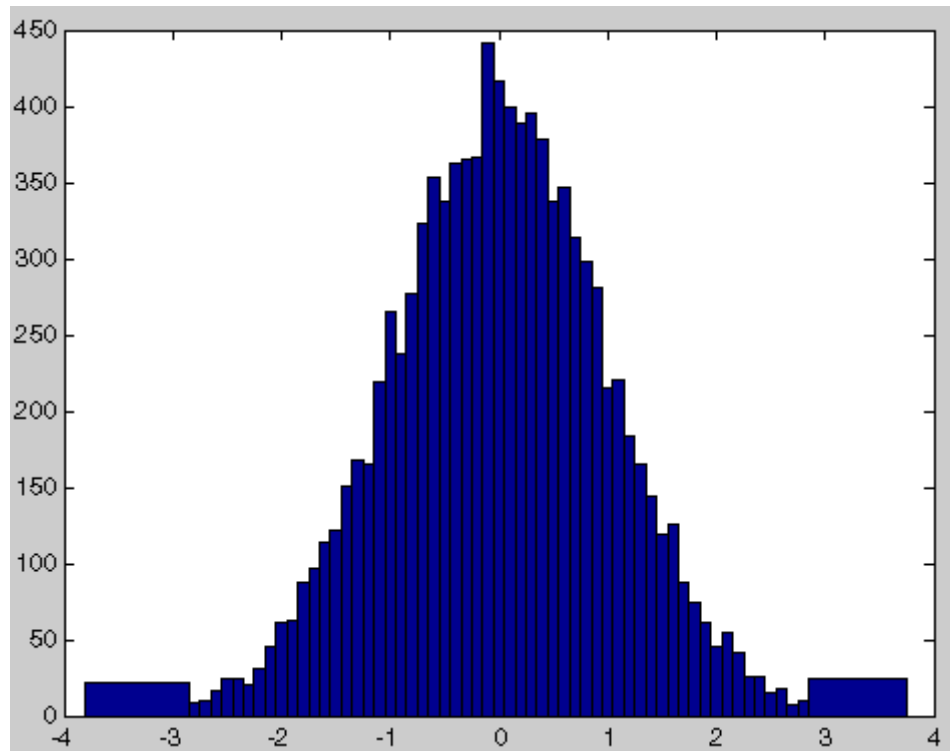
```
for j=1:N,
    n(k,j) = sum(bin(:,j)==k);
end
```

To plot the histogram, use the `bar` command.

**Examples** Generate a cumulative histogram of a distribution.

Consider the following distribution:

```
x = -2.9:0.1:2.9;
y = randn(10000,1);
figure(1), hist(y,x)
```



Calculate number of elements in each bin

```
n_elements = histc(y,x);
```

Calculate the cumulative sum of these elements using cumsum

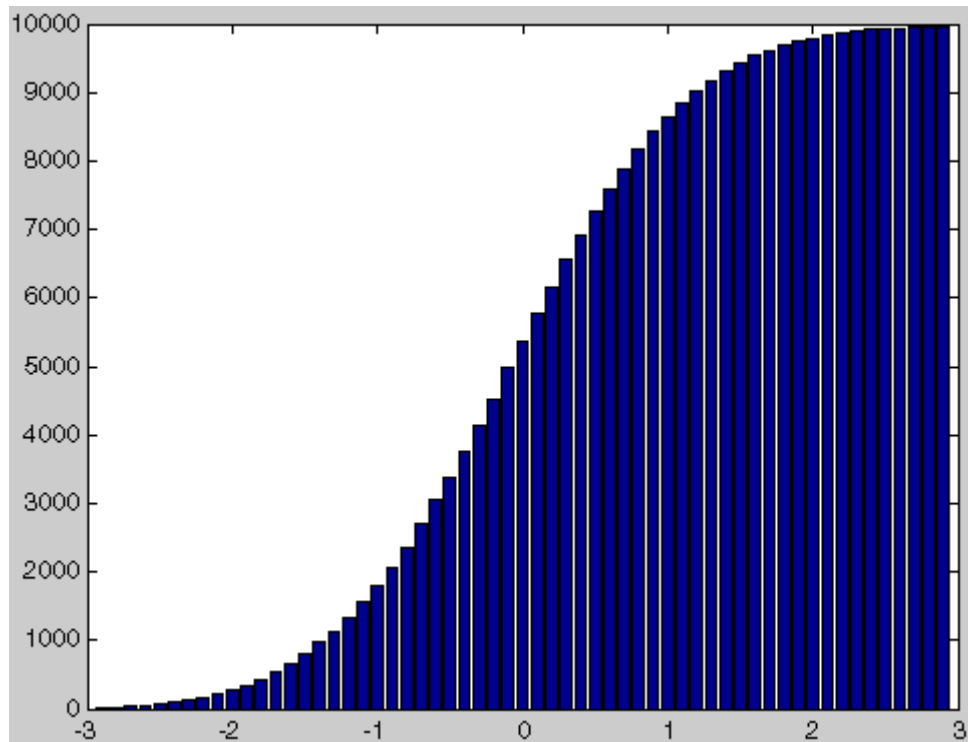
```
c_elements = cumsum(n_elements)
```

Plot the cumulative histogram

```
figure(2),bar(x,c_elements)
```

# histc

---



## See Also

`hist`, `mode`

“Specialized Plotting” on page 1-92 for related functions



**Purpose** Retain current graph in figure

**Syntax** `hold on`  
`hold off`  
`hold all`  
`hold`  
`hold(axes_handle,...)`

**Description** The `hold` function determines whether new graphics objects are added to the graph or replace objects in the graph.

`hold on` retains the current plot and certain axes properties so that subsequent graphing commands add to the existing graph.

`hold off` resets axes properties to their defaults before drawing new plots. `hold off` is the default.

`hold all` holds the plot and the current line color and line style so that subsequent plotting commands do not reset the `ColorOrder` and `LineStyleOrder` property values to the beginning of the list. Plotting commands continue cycling through the predefined colors and linestyles from where the last plot stopped in the list.

`hold` toggles the hold state between adding to the graph and replacing the graph.

`hold(axes_handle,...)` applies the hold to the axes identified by the handle `axes_handle`.

**Remarks** Test the hold state using the `ishold` function.

Although the hold state is on, some axes properties change to accommodate additional graphics objects. For example, the axes' limits increase when the data requires them to do so.

The `hold` function sets the `NextPlot` property of the current figure and the current axes. If several axes objects exist in a figure window, each axes has its own hold state. `hold` also creates an axes if one does not exist.

# hold

---

`hold on` sets the `NextPlot` property of the current figure and axes to `add`.

`hold off` sets the `NextPlot` property of the current axes to `replace`.

`hold` toggles the `NextPlot` property between the `add` and `replace` states.

## See Also

`axis`, `cla`, `ishold`, `newplot`

The `NextPlot` property of axes and figure graphics objects

“Basic Plots and Graphs” on page 1-90 for related functions

<b>Purpose</b>	Move cursor to upper-left corner of Command Window
<b>Syntax</b>	home
<b>Description</b>	home moves the cursor to the upper-left corner of the Command Window. You can use the scroll bar to see the history of previous functions.
<b>Examples</b>	Use home in an M-file to return the cursor to the upper-left corner of the screen.
<b>See Also</b>	clc

# horzcat

---

**Purpose** Concatenate arrays horizontally

**Syntax** `C = horzcat(A1, A2, ...)`

**Description** `C = horzcat(A1, A2, ...)` horizontally concatenates matrices `A1`, `A2`, and so on. All matrices in the argument list must have the same number of rows.

`horzcat` concatenates N-dimensional arrays along the second dimension. The first and remaining dimensions must match.

MATLAB calls `C = horzcat(A1, A2, ...)` for the syntax `C = [A1 A2 ...]` when any of `A1`, `A2`, etc., is an object.

## Examples

Create a 3-by-5 matrix, `A`, and a 3-by-3 matrix, `B`. Then horizontally concatenate `A` and `B`.

```
A = magic(5);           % Create 3-by-5 matrix, A
A(4:5,:) = []
```

```
A =
```

```
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
```

```
B = magic(3)*100       % Create 3-by-3 matrix, B
```

```
B =
```

```
    800    100    600
    300    500    700
    400    900    200
```

```
C = horzcat(A, B)      % Horizontally concatenate A and B
```

C =

17	24	1	8	15	800	100	600
23	5	7	14	16	300	500	700
4	6	13	20	22	400	900	200

## See Also

vertcat, cat, strcat, strvcat, special character []

# horzcat (tscollection)

---

**Purpose** Horizontal concatenation for `tscollection` objects

**Syntax** `tsc = horzcat(tsc1,tsc2,...)`

**Description** `tsc = horzcat(tsc1,tsc2,...)` performs horizontal concatenation for `tscollection` objects:

```
tsc = [tsc1 tsc2 ...]
```

This operation combines multiple `tscollection` objects, which must have the same time vectors, into one `tscollection` containing `timeseries` objects from all concatenated collections.

**See Also** `tscollection`, `vertcat` (`tscollection`)

**Purpose** Server host identification number

**Syntax** `id = hostid`

# hostid

---

## Description

`id = hostid` usually returns a single element cell array containing the MATLAB server host identifier as a string. On UNIX<sup>14</sup> platforms, there can be more than one identifier. In that case, `hostid` returns a cell array with an identifier in each cell.

14. UNIX is a registered trademark of The Open Group in the United States and other countries.



**Purpose** Convert HSV colormap to RGB colormap

**Syntax** `M = hsv2rgb(H)`  
`rgb_image = hsv2rgb(hsv_image)`

**Description** `M = hsv2rgb(H)` converts a hue-saturation-value (HSV) colormap to a red-green-blue (RGB) colormap. `H` is an  $m$ -by-3 matrix, where  $m$  is the number of colors in the colormap. The columns of `H` represent hue, saturation, and value, respectively. `M` is an  $m$ -by-3 matrix. Its columns are intensities of red, green, and blue, respectively.

`rgb_image = hsv2rgb(hsv_image)` converts the HSV image to the equivalent RGB image. HSV is an  $m$ -by- $n$ -by-3 image array whose three planes contain the hue, saturation, and value components for the image. RGB is returned as an  $m$ -by- $n$ -by-3 image array whose three planes contain the red, green, and blue components for the image.

**Remarks** As `H(:,1)` varies from 0 to 1, the resulting color varies from red through yellow, green, cyan, blue, and magenta, and returns to red. When `H(:,2)` is 0, the colors are unsaturated (i.e., shades of gray). When `H(:,2)` is 1, the colors are fully saturated (i.e., they contain no white component). As `H(:,3)` varies from 0 to 1, the brightness increases.

The MATLAB hsv colormap uses `hsv2rgb([huesaturationvalue])` where hue is a linear ramp from 0 to 1, and saturation and value are all 1's.

**See Also** `brighten`, `colormap`, `rgb2hsv`

“Color Operations” on page 1-102 for related functions

# hypot

---

**Purpose** Square root of sum of squares

**Syntax** `c = hypot(a,b)`

**Description** `c = hypot(a,b)` returns the element-wise result of the following equation, computed to avoid underflow and overflow:

$$c = \text{sqrt}(\text{abs}(a).^2 + \text{abs}(b).^2)$$

Inputs `a` and `b` must follow these rules:

- Both `a` and `b` must be single- or double-precision, floating-point arrays.
- The sizes of the `a` and `b` arrays must either be equal, or one a scalar and the other nonscalar. In the latter case, `hypot` expands the scalar input to match the size of the nonscalar input.
- If `a` or `b` is an empty array (0-by-N or N-by-0), the other must be the same size or a scalar. The result `c` is an empty array having the same size as the empty input(s).

`hypot` returns the following in output `c`, depending upon the types of inputs:

- If the inputs to `hypot` are complex (`w+xi` and `y+zi`), then the statement `c = hypot(w+xi,y+zi)` returns the *positive real* result

$$c = \text{sqrt}(\text{abs}(w).^2 + \text{abs}(x).^2 + \text{abs}(y).^2 + \text{abs}(z).^2)$$

- If `a` or `b` is `-Inf`, `hypot` returns `Inf`.
- If neither `a` nor `b` is `Inf`, but one or both inputs is `NaN`, `hypot` returns `NaN`.
- If all inputs are finite, the result is finite. The one exception is when both inputs are very near the value of the MATLAB constant `realmax`. The reason for this is that the equation `c =`

`hypot(realmax,realmax)` is theoretically  $\sqrt{2} * \text{realmax}$ , which overflows to `Inf`.

## Examples

### Example 1

To illustrate the difference between using the `hypot` function and coding the basic `hypot` equation in M-code, create an anonymous function that performs the same function as `hypot`, but without the consideration to underflow and overflow that `hypot` offers:

```
myhypot = @(a,b) sqrt(abs(a).^2+abs(b).^2);
```

Find the upper limit at which your coded function returns a useful value. You can see that this test function reaches its maximum at about `1e154`, returning an infinite result at that point:

```
myhypot(1e153,1e153)
ans =
    1.4142e+153
```

```
myhypot(1e154,1e154)
ans =
    Inf
```

Do the same using the `hypot` function, and observe that `hypot` operates on values up to about `1e308`, which is approximately equal to the value for `realmax` on your computer (the largest double-precision floating-point number you can represent on a particular computer):

```
hypot(1e308,1e308)
ans =
    1.4142e+308
```

```
hypot(1e309,1e309)
ans =
    Inf
```

# hypot

---

## Example 2

`hypot(a, a)` theoretically returns  $\sqrt{2} * \text{abs}(a)$ , as shown in this example:

```
x = 1.271161e308;
```

```
y = x * sqrt(2)
```

```
y =  
1.7977e+308
```

```
y = hypot(x, x)
```

```
y =  
1.7977e+308
```

## Algorithm

`hypot` 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

`sqrt`, `abs`, `norm`

---

**Purpose** Imaginary unit

**Syntax**  
i  
a+bi  
x+i\*y

**Description** As the basic imaginary unit  $\sqrt{-1}$ , `i` is used to enter complex numbers. Since `i` is a function, it can be overridden and used as a variable. This permits you to use `i` as an index in for loops, etc.

If desired, use the character `i` without a multiplication sign as a suffix in forming a complex numerical constant.

You can also use the character `j` as the imaginary unit.

**Examples**

```
Z = 2+3i
Z = x+i*y
Z = r*exp(i*theta)
```

**See Also** `conj`, `imag`, `j`, `real`

# idealfilter (timeseries)

---

**Purpose** Apply ideal (noncausal) filter to timeseries object

**Syntax**  
`ts2 = idealfilter(ts1,Interval,FilterType)`  
`ts2 = idealfilter(ts1,Interval,FilterType,Index)`

**Description** `ts2 = idealfilter(ts1,Interval,FilterType)` applies an ideal filter of `FilterType` 'pass' or 'notch' to one or more frequency intervals specified by `Interval` for the timeseries object `ts1`. You specify several frequency intervals as an n-by-2 array of start and end frequencies, where n represents the number of intervals.

`ts2 = idealfilter(ts1,Interval,FilterType,Index)` applies an ideal filter and uses the optional `Index` integer array to specify the columns or rows to filter. When `ts.IsTimeFirst` is set to true, `Index` specifies one or more data columns. When `ts.IsTimeFirst` is set to false, `Index` specifies one or more data rows.

## Remarks

### When to Use the Ideal Filter

You use the ideal *notch* filter when you want to remove variations in a specific frequency range. Alternatively, you use the ideal *pass* filter to allow only the variations in a specific frequency range.

These filters are ideal in the sense that they are not realizable; an ideal filter is noncausal and the ends of the filter amplitude are perfectly flat in the frequency domain.

### Requirement for Uniform Samples in Time

If the time-series data is sampled nonuniformly, filtering resamples this data on a uniform time vector.

### Interpolation of NaN Values

All NaNs in the time series are interpolated before filtering using the interpolation method you assigned to the timeseries object.

## Examples

You will apply an ideal notch filter to the data in `count.dat`.

**1** Load the matrix `count` into the workspace.

```
load count.dat;
```

- 2 Create a timeseries object based on this matrix. The time vector ranges from 1 to 24 seconds in 1-second intervals.

```
count1=timeseries(count(:,1),1:24);
```

- 3 Enter the frequency interval in hertz.

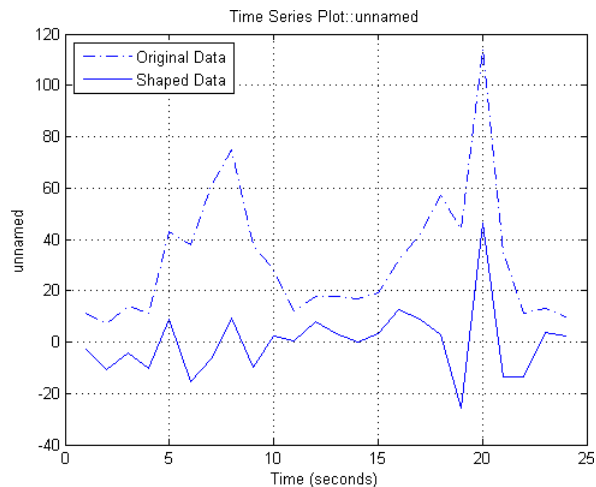
```
interval=[0.08 0.2];
```

- 4 Call the filter function:

```
idealfilter_count = idealfilter(count1,interval,'notch')
```

- 5 Compare the original data and the shaped data with an overlaid plot of the two curves.

```
plot(count1,'-.'), grid on, hold on  
plot(filter_count,'-')  
legend('Original Data','Shaped Data',2)
```



## idealfilter (timeseries)

---

**See Also**      `filter (timeseries), timeseries`



**Purpose** Integer division with rounding option

**Syntax**

```
C = idivide(A, B, opt)
C = idivide(A, B)
C = idivide(A, B, 'fix')
C = idivide(A, B, 'round')
C = idivide(A, B, 'floor')
C = idivide(A, B, 'ceil')
```

**Description** `C = idivide(A, B, opt)` is the same as `A./B` for integer classes except that fractional quotients are rounded to integers using the optional rounding mode specified by `opt`. The default rounding mode is `'fix'`. Inputs `A` and `B` must be real and must have the same dimensions unless one is a scalar. At least one of the arguments `A` and `B` must belong to an integer class, and the other must belong to the same integer class or be a scalar double. The result `C` belongs to the integer class.

`C = idivide(A, B)` is the same as `A./B` except that fractional quotients are rounded toward zero to the nearest integers.

`C = idivide(A, B, 'fix')` is the same as the syntax shown immediately above.

`C = idivide(A, B, 'round')` is the same as `A./B` for integer classes. Fractional quotients are rounded to the nearest integers.

`C = idivide(A, B, 'floor')` is the same as `A./B` except that fractional quotients are rounded toward negative infinity to the nearest integers.

`C = idivide(A, B, 'ceil')` is the same as `A./B` except that the fractional quotients are rounded toward infinity to the nearest integers.

**Examples**

```
a = int32([-2 2]);
b = int32(3);

idivide(a,b)           % Returns [0 0]
idivide(a,b,'floor')  % Returns [-1 0]
idivide(a,b,'ceil')   % Returns [0 1]
```

# idivide

---

```
idivide(a,b,'round') % Returns [-1 1]
```

## **See Also**

ldivide, rdivide, mldivide, mrdivide

**Purpose** Execute statements if condition is true

**Syntax** `if expression, statements, end`

**Description** `if expression, statements, end` evaluates *expression* and, if the evaluation yields logical 1 (true) or a nonzero result, executes one or more MATLAB commands denoted here as *statements*.

*expression* is a MATLAB expression, usually consisting of variables or smaller expressions joined by relational operators (e.g., `count < limit`), or logical functions (e.g., `isreal(A)`). Simple expressions can be combined by logical operators (`&&`, `||`, `~`) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

```
(count < limit) && ((height - offset) >= 0)
```

Nested `if` statements must each be paired with a matching `end`.

The `if` function can be used alone or with the `else` and `elseif` functions. When using `elseif` and/or `else` within an `if` statement, the general form of the statement is

```
if expression1
    statements1
elseif expression2
    statements2
else
    statements3
end
```

See “Program Control Statements” in the MATLAB Programming Fundamentals documentation for more information on controlling the flow of your program code.

**Remarks****Nonscalar Expressions**

If the evaluated expression yields a nonscalar value, then every element of this value must be true or nonzero for the entire expression to be considered true. For example, the statement `if (A < B)` is true only if each element of matrix A is less than its corresponding element in matrix B. See Example 2, below.

**Partial Evaluation of the expression Argument**

Within the context of an `if` or `while` expression, MATLAB does not necessarily evaluate all parts of a logical expression. In some cases it is possible, and often advantageous, to determine whether an expression is true or false through only partial evaluation.

For example, if A equals zero in statement 1 below, then the expression evaluates to `false`, regardless of the value of B. In this case, there is no need to evaluate B and MATLAB does not do so. In statement 2, if A is nonzero, then the expression is true, regardless of B. Again, MATLAB does not evaluate the latter part of the expression.

```
1)   if (A && B)           2)   if (A || B)
```

You can use this property to your advantage to cause MATLAB to evaluate a part of an expression only if a preceding part evaluates to the desired state. Here are some examples.

```
while (b ~= 0) && (a/b > 18.5)
    if exist('myfun.m') && (myfun(x) >= y)
        if iscell(A) && all(cellfun('isreal', A))
```

**Empty Arrays**

In most cases, using `if` on an empty array treats the array as `false`. There are some conditions however under which `if` evaluates as `true` on an empty array. Two examples of this, where A is equal to `[]`, are

```
if all(A), do_something, end
if 1|A, do_something, end
```

The latter expression is true because of short-circuiting, which causes MATLAB to ignore the right side operand of an OR statement whenever the left side evaluates to true.

### Short-Circuiting Behavior

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

See “Short-Circuiting in Elementwise Operators” for more information on this.

## Examples

### Example 1 - Simple if Statement

In this example, if both of the conditions are satisfied, then the student passes the course.

```
if ((attendance >= 0.90) && (grade_average >= 60))
    pass = 1;
end;
```

### Example 2 - Nonscalar Expression

Given matrices `A` and `B`,

```
A =           B =
     1         0         1         1
     2         3         3         4
```

Expression	Evaluates As	Because
<code>A &lt; B</code>	false	<code>A(1,1)</code> is not less than <code>B(1,1)</code> .
<code>A &lt; (B + 1)</code>	true	Every element of <code>A</code> is less than that same element of <code>B</code> with 1 added.
<code>A &amp; B</code>	false	<code>A(1,2)</code> is false, and <code>B</code> is ignored due to short-circuiting.
<code>B &lt; 5</code>	true	Every element of <code>B</code> is less than 5.

# if

---

## **See Also**

else, elseif, end, for, while, switch, break, return, relational operators, logical operators (elementwise and short-circuit),

**Purpose**

Inverse discrete Fourier transform

**Syntax**

```
y = ifft(X)
y = ifft(X,n)
y = ifft(X,[],dim)
y = ifft(X,n,dim)
y = ifft(..., 'symmetric')
y = ifft(..., 'nonsymmetric')
```

**Description**

`y = ifft(X)` returns the inverse discrete Fourier transform (DFT) of vector `X`, computed with a fast Fourier transform (FFT) algorithm. If `X` is a matrix, `ifft` returns the inverse DFT of each column of the matrix.

`ifft` tests `X` to see whether vectors in `X` along the active dimension are *conjugate symmetric*. If so, the computation is faster and the output is real. An `N`-element vector `x` is conjugate symmetric if `x(i) = conj(x(mod(N-i+1,N)+1))` for each element of `x`.

If `X` is a multidimensional array, `ifft` operates on the first non-singleton dimension.

`y = ifft(X,n)` returns the `n`-point inverse DFT of vector `X`.

`y = ifft(X,[],dim)` and `y = ifft(X,n,dim)` return the inverse DFT of `X` across the dimension `dim`.

`y = ifft(..., 'symmetric')` causes `ifft` to treat `X` as conjugate symmetric along the active dimension. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft(..., 'nonsymmetric')` is the same as calling `ifft(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft(fft(X))` equals `X` to within roundoff error.

**Algorithm**

The algorithm for `ifft(X)` is the same as the algorithm for `fft(X)`, except for a sign change and a scale factor of `n = length(X)`. As for `fft`, the execution time for `ifft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have

only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

---

**Note** You might be able to increase the speed of `ifft` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

---

## Data Type Support

`ifft` supports inputs of data types `double` and `single`. If you call `ifft` with the syntax `y = ifft(X, ...)`, the output `y` has the same data type as the input `X`.

## See Also

`fft`, `fft2`, `ifft2`, `ifftn`, `ifftshift`, `fftw`, `ifft2`, `ifftn`  
`dftmtx` and `freqz`, in the Signal Processing Toolbox software.



**Purpose** 2-D inverse discrete Fourier transform

**Syntax**

```
Y = ifft2(X)
Y = ifft2(X,m,n)
y = ifft2(..., 'symmetric')
y = ifft2(..., 'nonsymmetric')
```

**Description** `Y = ifft2(X)` returns the two-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifft2` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `M`-by-`N` matrix `X` is conjugate symmetric if  $X(i,j) = \text{conj}(X(\text{mod}(M-i+1, M) + 1, \text{mod}(N-j+1, N) + 1))$  for each element of `X`.

`Y = ifft2(X,m,n)` returns the `m`-by-`n` inverse fast Fourier transform of matrix `X`.

`y = ifft2(..., 'symmetric')` causes `ifft2` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft2(..., 'nonsymmetric')` is the same as calling `ifft2(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft2(fft2(X))` equals `X` to within roundoff error.

**Algorithm** The algorithm for `ifft2(X)` is the same as the algorithm for `fft2(X)`, except for a sign change and scale factors of `[m,n] = size(X)`. The execution time for `ifft2` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

# ifft2

---

---

**Note** You might be able to increase the speed of `ifft2` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

---

## Data Type Support

`ifft2` supports inputs of data types `double` and `single`. If you call `ifft2` with the syntax `y = ifft2(X, ...)`, the output `y` has the same data type as the input `X`.

## See Also

`dftmtx` and `freqz` in the Signal Processing Toolbox, and:  
`fft2`, `fftw`, `fftshift`, `ifft`, `ifftn`, `ifftshift`

**Purpose** N-D inverse discrete Fourier transform

**Syntax**

```
Y = ifftn(X)
Y = ifftn(X,siz)
y = ifftn(..., 'symmetric')
y = ifftn(..., 'nonsymmetric')
```

**Description** `Y = ifftn(X)` returns the n-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifftn` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `N1-by-N2-by- ... Nk` array `X` is conjugate symmetric if

$$X(i_1, i_2, \dots, i_k) = \text{conj}(X(\text{mod}(N_1 - i_1 + 1, N_1) + 1, \text{mod}(N_2 - i_2 + 1, N_2) + 1, \dots, \text{mod}(N_k - i_k + 1, N_k) + 1))$$

for each element of `X`.

`Y = ifftn(X,siz)` pads `X` with zeros, or truncates `X`, to create a multidimensional array of size `siz` before performing the inverse transform. The size of the result `Y` is `siz`.

`y = ifftn(..., 'symmetric')` causes `ifftn` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifftn(..., 'nonsymmetric')` is the same as calling `ifftn(...)` without the argument `'nonsymmetric'`.

**Remarks** For any `X`, `ifftn(fftn(X))` equals `X` within roundoff error.

**Algorithm** `ifftn(X)` is equivalent to

```
Y = X;
for p = 1:length(size(X))
    Y = ifft(Y,[],p);
end
```

This computes in-place the one-dimensional inverse DFT along each dimension of  $X$ .

The execution time for `ifftn` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

---

**Note** You might be able to increase the speed of `ifftn` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

---

## Data Type Support

`ifftn` supports inputs of data types `double` and `single`. If you call `ifftn` with the syntax `y = ifftn(X, ...)`, the output `y` has the same data type as the input `X`.

## See Also

`fftn`, `fftw`, `ifft`, `ifft2`, `ifftshift`

**Purpose** Inverse FFT shift

**Syntax** `ifftshift(X)`  
`ifftshift(X,dim)`

**Description** `ifftshift(X)` swaps the left and right halves of the vector  $X$ . For matrices, `ifftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth. If  $X$  is a multidimensional array, `ifftshift(X)` swaps “half-spaces” of  $X$  along each dimension.

`ifftshift(X,dim)` applies the `ifftshift` operation along the dimension `dim`.

---

**Note** `ifftshift` undoes the results of `fftshift`. If the matrix  $X$  contains an odd number of elements, `ifftshift(fftshift(X))` must be done to obtain the original  $X$ . Simply performing `fftshift(X)` twice will not produce  $X$ .

---

**See Also** `fft`, `fft2`, `fftn`, `fftshift`

**Purpose** Sparse incomplete LU factorization

**Syntax**  
`ilu(A,setup)`  
`[L,U] = ilu(A,setup)`  
`[L,U,P] = ilu(A,setup)`

**Description** `ilu` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`ilu(A,setup)` computes the incomplete LU factorization of `A`. `setup` is an input structure with up to five setup options. The fields must be named exactly as shown in the table below. You can include any number of these fields in the structure and define them in any order. Any additional fields are ignored.

Field Name	Description
<code>type</code>	Type of factorization. Values for <code>type</code> include: <ul style="list-style-type: none"><li>'<code>nofill</code>'—Performs ILU factorization with 0 level of fill in, known as ILU(0). With <code>type</code> set to '<code>nofill</code>', only the <code>milu</code> setup option is used; all other fields are ignored.</li><li>'<code>crout</code>'—Performs the Crout version of ILU factorization, known as ILUC. With <code>type</code> set to '<code>crout</code>', only the <code>droptol</code> and <code>milu</code> setup options are used; all other fields are ignored.</li><li>'<code>ilutp</code>' (default)—Performs ILU factorization with threshold and pivoting.</li></ul> If <code>type</code> is not specified, the ILU factorization with pivoting ILUTP is performed. Pivoting is never performed with <code>type</code> set to ' <code>nofill</code> ' or ' <code>crout</code> '.

Field Name	Description
droptol	<p>Drop tolerance of the incomplete LU factorization. droptol is a non-negative scalar. The default value is 0, which produces the complete LU factorization.</p> <p>The nonzero entries of U satisfy</p> $\text{abs}(U(i, j)) \geq \text{droptol} * \text{norm}(A(:, j)),$ <p>with the exception of the diagonal entries, which are retained regardless of satisfying the criterion. The entries of L are tested against the local drop tolerance before being scaled by the pivot, so for nonzeros in L</p> $\text{abs}(L(i, j)) \geq \text{droptol} * \text{norm}(A(:, j)) / U(j, j).$
milu	<p>Modified incomplete LU factorization. Values for milu include:</p> <ul style="list-style-type: none"> <li>• 'row'—Produces the row-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, U, preserving column sums. That is, <math>A * e = L * U * e</math>, where e is the vector of ones.</li> <li>• 'col'—Produces the column-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, U, preserving column sums. That is, <math>e' * A = e' * L * U</math>.</li> <li>• 'off' (default)—No modified incomplete LU factorization is produced.</li> </ul>

Field Name	Description
udiag	If udiag is 1, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is 0.
thresh	Pivot threshold between 0 (forces diagonal pivoting) and 1, the default, which always chooses the maximum magnitude entry in the column to be the pivot.

`ilu(A,setup)` returns `L+U-speye(size(A))`, where `L` is a unit lower triangular matrix and `U` is an upper triangular matrix.

`[L,U] = ilu(A,setup)` returns a unit lower triangular matrix in `L` and an upper triangular matrix in `U`.

`[L,U,P] = ilu(A,setup)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`.

## Remarks

These incomplete factorizations may be useful as preconditioners for a system of linear equations being solved by iterative methods such as BICG (BiConjugate Gradients), GMRES (Generalized Minimum Residual Method).

## Limitations

`ilu` works on sparse square matrices only.

## Examples

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'crout';
setup.milu = 'row';
setup.droptol = 0.1;
[L,U] = ilu(A,setup);
e = ones(size(A,2),1);
norm(A*e-L*U*e)
```

```
ans =
```



```
1.4251e-014
```

This shows that  $A$  and  $L*U$ , where  $L$  and  $U$  are given by the modified Crout ILU, have the same row-sum.

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'nofill';
nnz(A)
ans =
```

```
7840
```

```
nnz(lu(A))
ans =
```

```
126478
```

```
nnz(ilu(A,setup))
ans =
```

```
7840
```

This shows that  $A$  has 7840 nonzeros, the complete LU factorization has 126478 nonzeros, and the incomplete LU factorization, with 0 level of fill-in, has 7840 nonzeros, the same amount as  $A$ .

## See Also

`bicg`, `cholinc`, `gmres`, `luinc`

## References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

# im2frame

---

**Purpose** Convert image to movie frame

**Syntax** `f = im2frame(X,map)`  
`f = im2frame(X)`

**Description** `f = im2frame(X,map)` converts the indexed image `X` and associated colormap `map` into a movie frame `f`. If `X` is a truecolor (m-by-n-by-3) image, then `map` is optional and has no effect.

Typical usage:

```
M(1) = im2frame(X1,map);  
M(2) = im2frame(X2,map);  
...  
M(n) = im2frame(Xn,map);  
movie(M)
```

`f = im2frame(X)` converts the indexed image `X` into a movie frame `f` using the current colormap if `X` contains an indexed image.

**See Also** `frame2im`, `movie`

“Bit-Mapped Images” on page 1-96 for related functions

**Purpose** Convert image to Java image

**Syntax**

```
jimage = im2java(I)
jimage = im2java(X,MAP)
jimage = im2java(RGB)
```

**Description** To work with a MATLAB image in the Java environment, you must convert the image from its MATLAB representation into an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(I)` converts the intensity image `I` to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(X,MAP)` converts the indexed image `X`, with colormap `MAP`, to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(RGB)` converts the RGB image `RGB` to an instance of the Java image class, `java.awt.Image`.

## Class Support

The input image can be of class `uint8`, `uint16`, or `double`.

---

**Note** Java requires `uint8` data to create an instance of the Java image class, `java.awt.Image`. If the input image is of class `uint8`, `jimage` contains the same `uint8` data. If the input image is of class `double` or `uint16`, `im2java` makes an equivalent image of class `uint8`, rescaling or offsetting the data as necessary, and then converts this `uint8` representation to an instance of the Java image class, `java.awt.Image`.

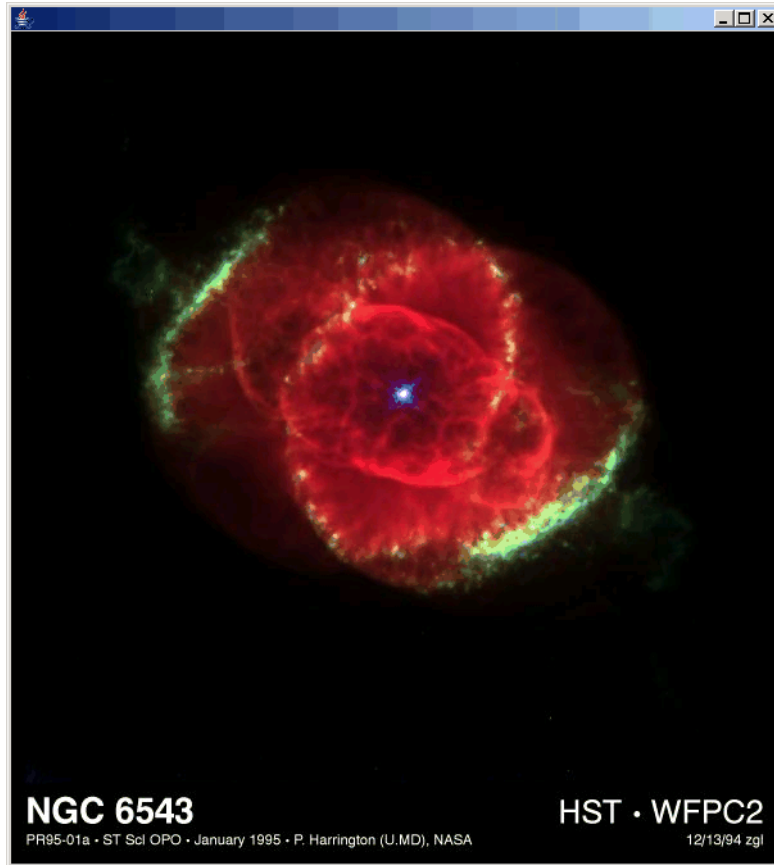
---

## Example

This example reads an image into the MATLAB workspace and then uses `im2java` to convert it into an instance of the Java image class.

```
I = imread('ngc6543a.jpg');
javaImage = im2java(I);
frame = javax.swing.JFrame;
icon = javax.swing.ImageIcon(javaImage);
label = javax.swing.JLabel(icon);
```

```
frame.getContentPane.add(label);  
frame.pack  
frame.show
```



## See Also

“Bit-Mapped Images” on page 1-96 for related functions

**Purpose** Imaginary part of complex number

**Syntax** `Y = imag(Z)`

**Description** `Y = imag(Z)` returns the imaginary part of the elements of array `Z`.

**Examples**

```
imag(2+3i)

ans =

     3
```

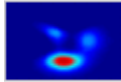
**See Also** `conj`, `i`, `j`, `real`

# image


---

## Purpose

Display image object



## GUI Alternatives

To plot a selected matrix as an image use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate image characteristics 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

```
image(C)
image(x,y,C)
image(x,y,C,'PropertyName',PropertyValue,...)
image('PropertyName',PropertyValue,...)
handle = image(...)
```

## Description

`image` creates an image graphics object by interpreting each element in a matrix as an index into the figure's colormap or directly as RGB values, depending on the data specified.

The `image` function has two forms:

- A high-level function that calls `newplot` to determine where to draw the graphics objects and sets the following axes properties:
  - `XLim` and `YLim` to enclose the image
  - `Layer` to top to place the image in front of the tick marks and grid lines
  - `YDir` to reverse
  - `View` to [0 90]

- A low-level function that adds the image to the current axes without calling `newplot`. The low-level function argument list can contain only property name/property value pairs.

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

`image(C)` displays matrix `C` as an image. Each element of `C` specifies the color of a rectangular segment in the image.

`image(x,y,C)`, where `x` and `y` are two-element vectors, specifies the range of the  $x$ - and  $y$ -axis labels, but produces the same image as `image(C)`. This can be useful, for example, if you want the axis tick labels to correspond to real physical dimensions represented by the image. If `x(1) > x(2)` or `y(1) > y(2)`, the image is flipped left-right or up-down, respectively. It can also be useful when you want to place the image within a set of axes already created. In this case, use `hold on` with the current figure and enter `x` and `y` values corresponding to the corners of the desired image location. The image is stretched and oriented as applicable.

`image(x,y,C,'PropertyName',PropertyValue,...)` is a high-level function that also specifies property name/property value pairs. This syntax calls `newplot` before drawing the image.

`image('PropertyName',PropertyValue,...)` is the low-level syntax of the `image` function. It specifies only property name/property value pairs as input arguments.

`handle = image(...)` returns the handle of the image object it creates. You can obtain the handle with all forms of the `image` function.

## Remarks

Image data can be either indexed or true color. An indexed image stores colors as an array of indices into the figure colormap. A true color image does not use a colormap; instead, the color values for each pixel are stored directly as RGB triplets. In MATLAB graphics, the `CData` property of a truecolor image object is a three-dimensional ( $m$ -by- $n$ -by-3)

array. This array consists of three  $m$ -by- $n$  matrices (representing the red, green, and blue color planes) concatenated along the third dimension.

The `imread` function reads image data into MATLAB arrays from graphics files in various standard formats, such as TIFF. You can write MATLAB image data to graphics files using the `imwrite` function. `imread` and `imwrite` both support a variety of graphics file formats and compression schemes.

When you read image data into the MATLAB workspace using `imread`, the data is usually stored as an array of 8-bit integers. However, `imread` also supports reading 16-bit-per-pixel data from TIFF and PNG files. These are more efficient storage methods than the double-precision (64-bit) floating-point numbers that MATLAB typically uses. However, it is necessary to interpret 8-bit and 16-bit image data differently from 64-bit data. This table summarizes these differences.

You cannot interactively pan or zoom outside the  $x$ -limits or  $y$ -limits of an image, unless the axes limits are already been set outside the bounds of the image, in which case there is no such restriction. If other objects (such as lineseries) occupy the axes and extend beyond the bounds of the image, you can pan or zoom to the bounds of the other objects, but no further.



<b>Image Type</b>	<b>Double-Precision Data (double Array)</b>	<b>8-Bit Data (uint8 Array) 16-Bit Data (uint16 Array)</b>
Indexed (colormap)	Image is stored as a two-dimensional (m-by-n) array of integers in the range [1, length(colormap)]; colormap is an m-by-3 array of floating-point values in the range [0, 1].	Image is stored as a two-dimensional (m-by-n) array of integers in the range [0, 255] (uint8) or [0, 65535] (uint16); colormap is an m-by-3 array of floating-point values in the range [0, 1].
True color (RGB)	Image is stored as a three-dimensional (m-by-n-by-3) array of floating-point values in the range [0, 1].	Image is stored as a three-dimensional (m-by-n-by-3) array of integers in the range [0, 255] (uint8) or [0, 65535] (uint16).

## Indexed Images

In an indexed image of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. In a `uint8` or `uint16` indexed image, there is an offset; the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on.

If you want to convert a `uint8` or `uint16` indexed image to `double`, you need to add 1 to the result. For example,

```
X64 = double(X8) + 1;
```

or

```
X64 = double(X16) + 1;
```

To convert from `double` to `uint8` or `uint16`, you need to first subtract 1, and then use `round` to ensure all the values are integers.

```
X8 = uint8(round(X64 - 1));
```

or

```
X16 = uint16(round(X64 - 1));
```

When you write an indexed image using `imwrite`, values are automatically converted if necessary.

## Colormaps

MATLAB colormaps are always *m*-by-3 arrays of double-precision floating-point numbers in the range [0, 1]. In most graphics file formats, colormaps are stored as integers, but MATLAB colormaps cannot have integer values. `imread` and `imwrite` automatically convert colormap values when reading and writing files.

## True Color Images

In a true color image of class `double`, the data values are floating-point numbers in the range [0, 1]. In a true color image of class `uint8`, the data values are integers in the range [0, 255], and for true color images of class `uint16` the data values are integers in the range [0, 65535].

If you want to convert a true color image from one data type to the other, you must rescale the data. For example, this statement converts a `uint8` true color image to `double`.

```
RGB64 = double(RGB8)/255;
```

or for `uint16` images,

```
RGB64 = double(RGB16)/65535;
```

This statement converts a `double` true color image to `uint8`:

```
RGB8 = uint8(round(RGB64*255));
```

or to obtain `uint16` images, type

```
RGB16 = uint16(round(RGB64*65535));
```

When you write a true color image using `imwrite`, values are automatically converted if necessary..

## Example

### Example 1

Load a mat-file containing a photograph of a colorful primate. Display the indexed image using its associated colormap.

```
load mandrill
figure('color','k')
image(X)
colormap(map)
axis off           % Remove axis ticks and numbers
axis image       % Set aspect ratio to
                 obtain square pixels
```



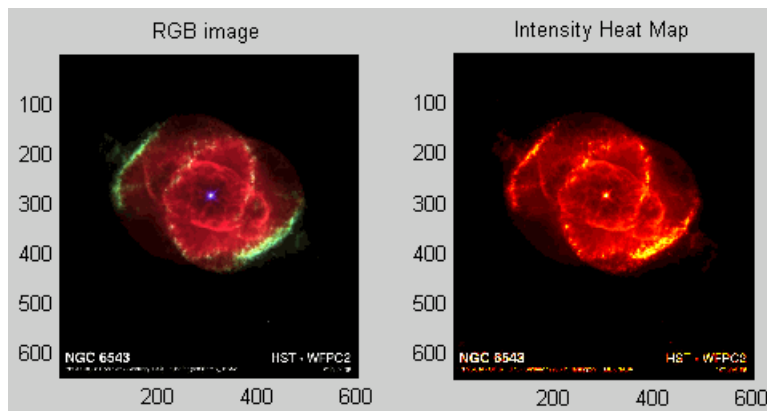
### Example 2

Load a JPEG image file of the Cat's Eye Nebula from the Hubble Space Telescope (image courtesy NASA). Display the original image using its RGB color values (left) as a subplot. Create a linked subplot (same

# image

size and scale) to display the transformed intensity image as a heat map (right).

```
figure
ax(1) = subplot(1,2,1);
rgb = imread('ngc6543a.jpg');
image(rgb); title('RGB image')
ax(2) = subplot(1,2,2);
im = mean(rgb,3);
image(im); title('Intensity Heat Map')
colormap(hot(256))
linkaxes(ax,'xy')
axis(ax,'image')
```



## Setting Default Properties

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

```
set(0,'DefaultImageProperty',PropertyValue...)
set(gcf,'DefaultImageProperty',PropertyValue...)
set(gca,'DefaultImageProperty',PropertyValue...)
```

where *Property* is the name of the image property and *PropertyValue* is the value you are specifying. Use `set` and `get` to access image properties.

## See Also

colormap, imagesc, imfinfo, imread, imwrite, newplot, pcolor, surface

“Displaying Bit-Mapped Images”

“Bit-Mapped Images” on page 1-96 for related functions

Image Properties for property descriptions

# Image Properties

---

## Purpose

Define image properties

## Modifying Properties

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

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

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

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

## Image Properties

This section lists property names along with the types of values each property accepts.

### AlphaData

m-by-n matrix of double or uint8

*The transparency data.* A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The AlphaData can be of class double or uint8.

MATLAB software determines the transparency in one of three ways:

- Using the elements of AlphaData as transparency values (AlphaDataMapping set to none)
- Using the elements of AlphaData as indices into the current alphamap (AlphaDataMapping set to direct)
- Scaling the elements of AlphaData to range between the minimum and maximum values of the axes ALim property (AlphaDataMapping set to scaled, the default)

## AlphaDataMapping

{none} | direct | scaled

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

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

## Annotation

hg.Annotation object Read Only

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

# Image Properties

---

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
on	Represent this image object in a legend (default)
off	Do not include this image object in a legend
children	Same as on because image objects do not have children

## Setting the **IconDisplayStyle** property

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

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

## Using the **IconDisplayStyle** property

See “Controlling Legends” for more information and examples.

**BeingDeleted**  
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 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

# Image Properties

---

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

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

## CData

matrix or m-by-n-by-3 array

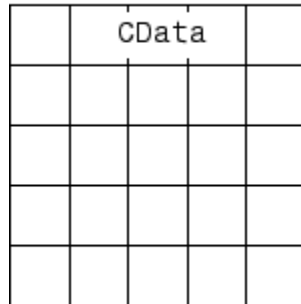
*The image data.* A matrix or 3-D array of values specifying the color of each rectangular area defining the image. `image(C)` assigns the values of `C` to `CData`. MATLAB determines the coloring of the image in one of three ways:

- Using the elements of `CData` as indices into the current colormap (the default) (`CDataMapping` set to `direct`)
- Scaling the elements of `CData` to range between the values `min(get(gca, 'CLim'))` and `max(get(gca, 'CLim'))` (`CDataMapping` set to `scaled`)
- Interpreting the elements of `CData` directly as RGB values (true color specification)

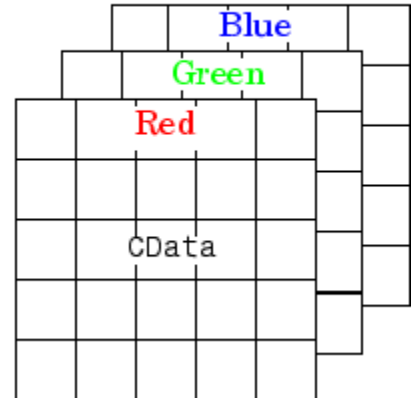
Note that the behavior of NaNs in image `CData` is not defined. See the image `AlphaData` property for information on using transparency with images.

A true color specification for `CData` requires an m-by-n-by-3 array of RGB values. The first page contains the red component, the second page the green component, and the third page the blue component of each element in the image. RGB values range from 0 to 1. The following picture illustrates the relative dimensions of `CData` for the two color models.

## Indexed Colors



## True Colors



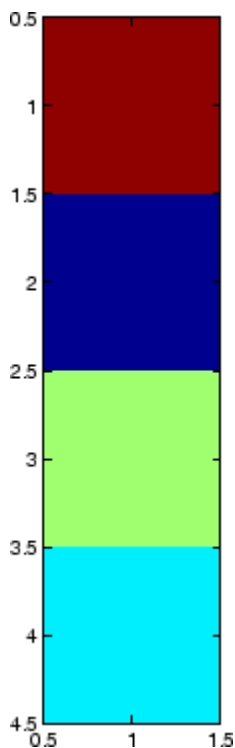
If CData has only one row or column, the height or width respectively is always one data unit and is centered about the first YData or XData element respectively. For example, using a 4-by-1 matrix of random data,

```
C = rand(4,1);  
image(C,'CDataMapping','scaled')  
axis image
```

produces

# Image Properties

---



CDataMapping  
scaled | {direct}

*Direct or scaled indexed colors.* This property determines whether MATLAB interprets the values in CData as indices into the figure colormap (the default) or scales the values according to the values of the axes CLim property.

When CDataMapping is direct, the values of CData should be in the range 1 to `length(get(gcf, 'Colormap'))`. If you use true color specification for CData, this property has no effect.

Children  
handles

The empty matrix; image objects have no children.

## Clipping

on | off

*Clipping mode.* By default, MATLAB clips images to the axes rectangle. If you set `Clipping` to `off`, the image can be displayed outside the axes rectangle. For example, if you create an image, set `hold` to `on`, freeze axis scaling (with `axis manual`), and then create a larger image, it extends beyond the axis limits.

## CreateFcn

string or function handle

*Callback routine executed during object creation.* This property defines a callback routine that executes when MATLAB creates an image object. You must define this property as a default value for images or in a call to the `image` function to create a new image object. For example, the statement

```
set(0, 'DefaultImageCreateFcn', 'axis image')
```

defines a default value on the root level that sets the aspect ratio and the axis limits so the image has square pixels. MATLAB executes this routine after setting all image properties. Setting this property on an existing image 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

# Image Properties

---

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 image object.* The `legend` function uses the string defined by the `DisplayName` property to label this image object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this image object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where `n` is the number assigned to the object based on its location in the list of legend entries. However, legend does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

## EraseMode

{normal} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- **none** — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- **xor** — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- **background** — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

## Printing with Nonnormal Erase Modes

# Image Properties

---

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

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

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

`HandleVisibility`  
{on} | callback | off

*Control access to object's handle by command-line users and GUIs.* This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

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



## Functions Affected by Handle Visibility

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

## Properties Affected by Handle Visibility

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

## Overriding Handle Visibility

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

## Handle Validity

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

---

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

---

# Image Properties

---

## HitTest

{on} | off

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

## 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.

## Parent

handle of parent axes, `hgroup`, or `hgtransform`

*Parent of this object.* This property contains the handle of the object's parent. The parent is normally the axes, `hgroup`, or `hgtransform` object that contains the object.

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

## Selected

on | {off}

*Is object selected?* When you set this property to on, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also on (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

## SelectionHighlight

{on} | off

*Objects are highlighted when selected.* When the `Selected` property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

## Tag

string

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

For example, you might create an `areaseries` object and set the `Tag` property.

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

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

# Image Properties

---

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

## Type

string (read only)

*Type of graphics object.* This property contains a string that identifies the class of graphics object. For image objects, Type is always 'image'.

## 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

[1 size(CData,2)] by default

*Control placement of image along x-axis.* A vector specifying the locations of the centers of the elements `CData(1,1)` and `CData(m,n)`, where `CData` has a size of `m`-by-`n`. Element `CData(1,1)` is centered over the coordinate defined by the first elements in `XData` and `YData`. Element `CData(m,n)` is centered over the coordinate defined by the last elements in `XData` and `YData`. The centers of the remaining elements of `CData` are evenly distributed between those two points.

The width of each `CData` element is determined by the expression

$$(XData(2) - XData(1)) / (size(CData,2) - 1)$$

You can also specify a single value for `XData`. In this case, `image` centers the first element at this coordinate and centers each following element one unit apart.

`YData`

`[1 size(CData,1)]` by default

*Control placement of image along y-axis.* A vector specifying the locations of the centers of the elements `CData(1,1)` and `CData(m,n)`, where `CData` has a size of `m`-by-`n`. Element `CData(1,1)` is centered over the coordinate defined by the first elements in `XData` and `YData`. Element `CData(m,n)` is centered over the coordinate defined by the last elements in `XData` and `YData`. The centers of the remaining elements of `CData` are evenly distributed between those two points.

The height of each `CData` element is determined by the expression

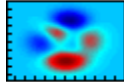
$$(YData(2) - YData(1)) / (size(CData,1) - 1)$$

You can also specify a single value for `YData`. In this case, `image` centers the first element at this coordinate and centers each following element one unit apart.


# imagesc

---

**Purpose** Scale data and display image object



## GUI Alternatives

To plot a selected matrix as an image use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate image characteristics 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

```
imagesc(C)
imagesc(x,y,C)
imagesc(...,clims)
h = imagesc(...)
```

## Description

The `imagesc` function scales image data to the full range of the current colormap and displays the image. (See “Examples” on page 2-1729 for an illustration.)

`imagesc(C)` displays `C` as an image. Each element of `C` corresponds to a rectangular area in the image. The values of the elements of `C` are indices into the current colormap that determine the color of each patch.

`imagesc(x,y,C)` displays `C` as an image and specifies the bounds of the  $x$ - and  $y$ -axis with vectors `x` and `y`. If `x(1) > x(2)` or `y(1) > y(2)`, the image is flipped left-right or up-down, respectively.

`imagesc(...,clims)` normalizes the values in `C` to the range specified by `clims` and displays `C` as an image. `clims` is a two-element vector that limits the range of data values in `C`. These values map to the full range of values in the current colormap.

`h = imagesc(...)` returns the handle for an image graphics object.

## Remarks

$x$  and  $y$  do not affect the elements in  $C$ ; they only affect the annotation of the axes. If  $\text{length}(x) > 2$  or  $\text{length}(y) > 2$ , `imagesc` ignores all except the first and last elements of the respective vector.

`imagesc` creates an image with `CDataMapping` set to `scaled`, and sets the axes `CLim` property to the value passed in `clims`.

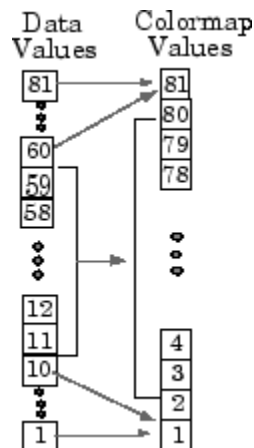
You cannot interactively pan or zoom outside the  $x$ -limits or  $y$ -limits of an image.

## Examples

You can expand midrange color resolution by mapping low values to the first color and high values to the last color in the colormap by specifying color value limits (`clims`). If the size of the current colormap is 81-by-3, the statements

```
clims = [ 10 60 ]
imagesc(C,clims)
```

map the data values in  $C$  to the colormap as shown in this illustration and the code that follows:



In this example, the left image maps to the `gray` colormap using the statements

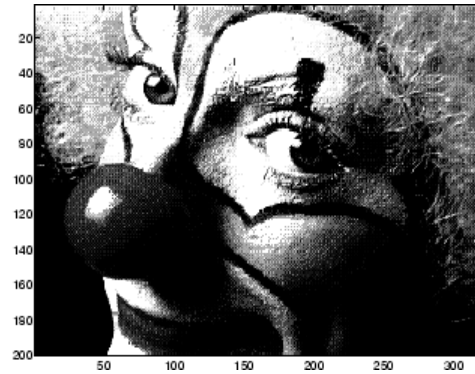
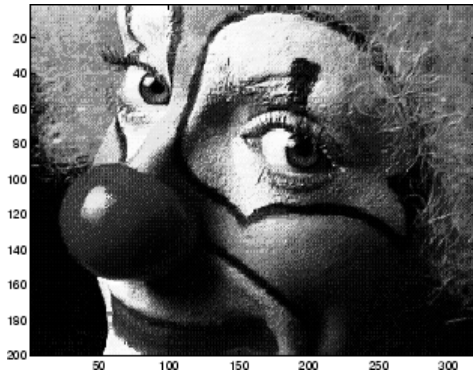
# imagesc

---

```
load clown
imagesc(X)
colormap(gray)
```

The right image has values between 10 and 60 scaled to the full range of the gray colormap using the statements

```
load clown
clims = [10 60];
imagesc(X,clims)
colormap(gray)
```



## See Also

image, imfinfo, imread, imwrite, colorbar, colormap, pcolor, surface, surf

“Bit-Mapped Images” on page 1-96 for related functions



**Purpose** Information about graphics file

**Syntax**

```
info = imfinfo(filename,fmt)
info = imfinfo(filename)
info = imfinfo(URL,...)
```

**Description** `info = imfinfo(filename,fmt)` returns a structure whose fields contain information about an image in a graphics file. `filename` is a string that specifies the name of the graphics file, and `fmt` is a string that specifies the format of the file. The file must be in the current directory or in a directory on the MATLAB path. If `imfinfo` cannot find a file named `filename`, it looks for a file named `filename.fmt`. The possible values for `fmt` are contained in the MATLAB file format registry. To view of list of these formats, run the `imformats` command.

If `filename` is a TIFF, HDF, ICO, GIF, or CUR file containing more than one image, `info` is a structure array with one element for each image in the file. For example, `info(3)` would contain information about the third image in the file.

`info = imfinfo(filename)` attempts to infer the format of the file from its contents.

`info = imfinfo(URL,...)` reads the image from the specified Internet URL. The URL must include the protocol type (e.g., `http://`)

### Information Returned

The set of fields in `info` depends on the individual file and its format. However, the first nine fields are always the same. This table lists these common fields, in the order they appear in the structure, and describes their values. “Format-Specific Notes” on page 2-1732 contains information about some fields returned by certain formats.

Field	Value
Filename	A string containing the name of the file; if the file is not in the current directory, the string contains the full pathname of the file.

Field	Value
FileModDate	A string containing the date when the file was last modified
FileSize	An integer indicating the size of the file in bytes
Format	A string containing the file format, as specified by <i>fmt</i> ; for formats with more than one possible extension (e.g. JPEG and TIFF files), <code>imfinfo</code> returns the first variant in the file format registry.
FormatVersion	A string or number describing the file format version
Width	An integer indicating the width of the image in pixels
Height	An integer indicating the height of the image in pixels
BitDepth	An integer indicating the number of bits per pixel
ColorType	A string indicating the type of image; this can include, but is not limited to, 'truecolor' for a truecolor (RGB) image, 'grayscale' for a grayscale intensity image, or 'indexed' for an indexed image

## Format-Specific Notes

- **JPEG and TIFF only** — If filename contains Exchangeable Image File Format (EXIF) tags, the `info` structure returned by `imfinfo` might also contain 'DigitalCamera' or 'GPSInfo' (global positioning system information) fields.
- **GIF only** — `imfinfo` returns the value of the 'DelayTime' field in hundredths of seconds.

## Example

```
info = imfinfo('canoe.tif')  
  
info =
```

```
Filename: [1x76 char]
FileModDate: '04-Dec-2000 13:57:55'
FileSize: 69708
Format: 'tif'
FormatVersion: []
Width: 346
Height: 207
BitDepth: 8
ColorType: 'indexed'
FormatSignature: [73 73 42 0]
ByteOrder: 'little-endian'
NewSubFileType: 0
BitsPerSample: 8
Compression: 'PackBits'
PhotometricInterpretation: 'RGB Palette'
StripOffsets: [9x1 double]
SamplesPerPixel: 1
RowsPerStrip: 23
StripByteCounts: [9x1 double]
XResolution: 72
YResolution: 72
ResolutionUnit: 'Inch'
Colormap: [256x3 double]
PlanarConfiguration: 'Chunky'
TileWidth: []
TileLength: []
TileOffsets: []
TileByteCounts: []
Orientation: 1
FillOrder: 1
GrayResponseUnit: 0.0100
MaxSampleValue: 255
MinSampleValue: 0
Thresholding: 1
```

**See Also** `imformats`, `imread`, `imwrite`

“Bit-Mapped Images” on page 1-96 for related functions

**Purpose** Manage image file format registry

**Syntax**

```

imformats
formats = imformats
formats = imformats('fmt')
formats = imformats(format_struct)
formats = imformats('factory')

```

**Description** `imformats` displays a table of information listing all the values in the MATLAB file format registry. This registry determines which file formats are supported by the `imfinfo`, `imread`, and `imwrite` functions. `formats = imformats` returns a structure containing all the values in the MATLAB file format registry. The following table lists the fields in the order they appear in the structure.

Field	Value
<code>ext</code>	A cell array of strings that specify filename extensions that are valid for this format
<code>isa</code>	A string specifying the name of the function that determines if a file is a certain format. This can also be a function handle.
<code>info</code>	A string specifying the name of the function that reads information about a file. This can also be a function handle.
<code>read</code>	A string specifying the name of the function that reads image data in a file. This can also be a function handle.
<code>write</code>	A string specifying the name of the function that writes MATLAB data to a file. This can also be a function handle.
<code>alpha</code>	Returns 1 if the format has an alpha channel, 0 otherwise
<code>description</code>	A text description of the file format

---

**Note** The values for the `isa`, `info`, `read`, and `write` fields must be functions on the MATLAB search path or function handles.

---

`formats = imformats('fmt')` searches the known formats in the MATLAB file format registry for the format associated with the filename extension `'fmt'`. If found, `imformats` returns a structure containing the characteristics and function names associated with the format. Otherwise, it returns an empty structure.

`formats = imformats(format_struct)` sets the MATLAB file format registry to the values in `format_struct`. The output structure, `formats`, contains the new registry settings.

---

**Caution** Using `imformats` to specify values in the MATLAB file format registry can result in the inability to load any image files. To return the file format registry to a working state, use `imformats` with the `'factory'` setting.

---

`formats = imformats('factory')` resets the MATLAB file format registry to the default format registry values. This removes any user-specified settings.

Changes to the format registry do not persist between MATLAB sessions. To have a format always available when you start MATLAB, add the appropriate `imformats` command to the MATLAB startup file, `startup.m`, located in `$MATLAB/toolbox/local` on UNIX systems, or `$MATLAB\toolbox\local` on Windows systems.

## Example

```
formats = imformats;
formats(1)

ans =

        ext: {'bmp'}
```

```
isa: @isbmp
info: @imbmpinfo
read: @readbmp
write: @writebmp
alpha: 0
description: 'Windows Bitmap (BMP)'
```

## See Also

fileformats, imfinfo, imread, imwrite, path

“Bit-Mapped Images” on page 1-96 for related functions

# import

---

## Purpose

Add package or class to current import list

## Syntax

```
import package_name. *  
import class_name  
import cls_or_pkg_name1 cls_or_pkg_name2...  
import  
L = import
```

## Description

`import package_name. *` adds specified *package\_name* to the current import list. Note that *package\_name* must be followed by `.*`.

`import class_name` adds a single class to the current import list. Note that *class\_name* must be fully qualified (that is, it must include the package name).

`import cls_or_pkg_name1 cls_or_pkg_name2...` adds all named classes and packages to the current import list. Note that each class name must be fully qualified, and each package name must be followed by `.*`.

`import` with no input arguments displays the current import list, without adding to it.

`L = import` with no input arguments returns a cell array of strings containing the current import list, without adding to it.

The `import` function only affects the import list of the function within which it is used. When invoked at the command prompt, `import` uses the import list for the MATLAB command environment. If `import` is used in a script invoked from a function, it affects the import list of the function. If `import` is used in a script that is invoked from the command prompt, it affects the import list for the command environment.

The import list of a function is persistent across calls to that function and is only cleared when the function is cleared.

To clear the current import list, use the following command.

```
clear import
```



This command may only be invoked at the command prompt. Attempting to use `clear import` within a function results in an error.

## Remarks

The `import` function allows your code to refer to an imported class by class name only, rather than with the fully qualified class name. `import` is particularly useful in streamlining calls to constructors, where most references to Java classes occur.

## Examples

### Add Meta Package to Current Import List

```
import meta.*
```

### Add Java Class to Current Import List

```
import java.lang.String
s = String('hello'); % Create java.lang.String object
```

### Add Multiple Java Packages to Current Import List

```
import java.util.* java.awt.*
f = Frame; % Create java.awt.Frame object
methods Enumeration % List java.util.Enumeration methods
```

## See Also

`clear`, `load`, `importdata`

# importdata

---

**Purpose** Load data from disk file

**Syntax**

```
importdata(filename)
A = importdata(filename)
A = importdata(filename,delimiter)
A = importdata(filename,delimiter,headerline)
[A D] = importdata(...)
[A D H] = importdata(...)
[...] = importdata('-pastespecial', ...)
```

**Description**

`importdata(filename)` loads data from `filename` into the workspace. The `filename` input is a string enclosed in single quotes.

`A = importdata(filename)` loads data from `filename` into structure `A`.

`A = importdata(filename,delimiter)` loads data from `filename` using `delimiter` as the column separator. The `delimiter` argument must be a string enclosed in single quotes. Use `'\t'` for tab. When importing from an ASCII file, `delimiter` only separates numeric data.

`A = importdata(filename,delimiter,headerline)` where `headerline` is a number that indicates on which line of the file the header text is located, loads data from line `headerline+1` to the end of the file.

`[A D] = importdata(...)` returns the output structure in `A`, and the delimiter character in `D`.

`[A D H] = importdata(...)` returns the output structure in `A`, the delimiter character in `D`, and the line number of the header in `H`.

`[...] = importdata('-pastespecial', ...)` loads data from your computer's paste buffer rather than from a file.

**Remarks**

`importdata` looks at the file extension to determine which helper function to use. If it can recognize the file extension, `importdata` calls the appropriate helper function, specifying the maximum number of output arguments. If it cannot recognize the file extension, `importdata` calls `finfo` to determine which helper function to use. If no helper

function is defined for this file extension, `importdata` treats the file as delimited text. `importdata` removes from the result empty outputs returned from the helper function.

## Examples

### Example 1 – A Simple Import

Import data from file `ding.wav`:

```
s = importdata('ding.wav')
s =

    data: [11554x1 double]
    fs: 22050
```

### Example 2 – Importing with Delimiter and Header

Use `importdata` to read in a text file. The third input argument is `colheaders`, which is the number of lines that belong to the header:

```
type 'myfile.txt'

    Day1 Day2 Day3 Day4 Day5 Day6 Day7
95.01 76.21 61.54 40.57 5.79 20.28 1.53
23.11 45.65 79.19 93.55 35.29 19.87 74.68
60.68 1.85 92.18 91.69 81.32 60.38 44.51
48.60 82.14 73.82 41.03 0.99 27.22 93.18
89.13 44.47 17.63 89.36 13.89 19.88 46.60
```

Import from the file, specifying the space character as the delimiter and 1 row for the column header. Assign the output to variable `M`:

```
M = importdata('myfile.txt', ' ', 1);
```

Print out columns 3 and 5, including the header for those columns:

```
for k=3:2:5
    M.colheaders(1,k)
    M.data(:,k)
    disp ' '
end
```

# importdata

---

```
ans =  
    'Day3'  
ans =  
    61.5400  
    79.1900  
    92.1800  
    73.8200  
    17.6300
```

```
ans =  
    'Day5'  
ans =  
    5.7900  
    35.2900  
    81.3200  
    0.9900  
    13.8900
```

## See Also

load

**Purpose**

Read image from graphics file

**Syntax**

```
A = imread(filename, fmt)
[X, map] = imread(...)
[...] = imread(filename)
[...] = imread(URL,...)
[...] = imread(...,Param1,Val1,Param2,Val2...)
```

**Description**

`A = imread(filename, fmt)` reads a grayscale or color image from the file specified by the string `filename`. If the file is not in the current directory, or in a directory on the MATLAB path, specify the full pathname.

The text string `fmt` specifies the format of the file by its standard file extension. For example, specify 'gif' for Graphics Interchange Format files. To see a list of supported formats, with their file extensions, use the `imformats` function. If `imread` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

The return value `A` is an array containing the image data. If the file contains a grayscale image, `A` is an M-by-N array. If the file contains a truecolor image, `A` is an M-by-N-by-3 array. For TIFF files containing color images that use the CMYK color space, `A` is an M-by-N-by-4 array. See TIFF in the Format-Specific Information section for more information.

The class of `A` depends on the bits-per-sample of the image data, rounded to the next byte boundary. For example, `imread` returns 24-bit color data as an array of `uint8` data because the sample size for each color component is 8 bits. See “Remarks” on page 2-1744 for a discussion of bitdepths, and see “Format-Specific Information” on page 2-1744 for more detail about supported bitdepths and sample sizes for a particular format.

`[X, map] = imread(...)` reads the indexed image in `filename` into `X` and its associated colormap into `map`. Colormap values in the image file are automatically rescaled into the range `[0, 1]`.

[...] = imread(filename) attempts to infer the format of the file from its content.

[...] = imread(URL,...) reads the image from an Internet URL. The URL must include the protocol type (e.g., http://).

[...] = imread(...,Param1,Val1,Param2,Val2...) specifies parameters that control various characteristics of the operations for specific formats. For more information, see “Format-Specific Information” on page 2-1744.

## Remarks

Bitdepth is the number of bits used to represent each image pixel. Bitdepth is calculated by multiplying the bits-per-sample with the samples-per-pixel. Thus, a format that uses 8-bits for each color component (or sample) and three samples per pixel has a bitdepth of 24. Sometimes the sample size associated with a bitdepth can be ambiguous: does a 48-bit bitdepth represent six 8-bit samples, four 12-bit samples, or three 16-bit samples? The following format-specific sections provide sample size information to avoid this ambiguity.

**Format-Specific Information** The following sections provide information about the support for specific formats, listed in alphabetical order by format name. These sections include information about format-specific syntaxes, if they exist.

“BMP — Windows Bitmap” on page 2-1745	“JPEG — Joint Photographic Experts Group” on page 2-1747	“PNG — Portable Network Graphics” on page 2-1749
“CUR — Cursor File” on page 2-1745	“JPEG 2000 — Joint Photographic Experts Group 2000” on page 2-1748	“PPM — Portable Pixmap” on page 2-1750
“GIF — Graphics Interchange Format” on page 2-1746	“PBM — Portable Bitmap” on page 2-1748	“RAS — Sun Raster” on page 2-1750

“HDF4 — Hierarchical Data Format” on page 2-1747

“PCX — Windows Paintbrush” on page 2-1749

“TIFF — Tagged Image File Format” on page 2-1751

“ICO — Icon File” on page 2-1747

“PGM — Portable Graymap” on page 2-1749

“XWD — X Window Dump” on page 2-1752

### BMP — Windows Bitmap

Supported Bitdepths	No Compression	RLE Compression	Output Class	Notes
1-bit	x	—	logical	
4-bit	x	x	uint8	
8-bit	x	x	uint8	
16-bit	x	—	uint8	1 sample/pixel
24-bit	x	—	uint8	3 samples/pixel
32-bit	x	—	uint8	3 samples/pixel (1 byte padding)

### CUR — Cursor File

Supported Bitdepths	No Compression	Compression	Output Class
1-bit	x	—	logical
4-bit	x	—	uint8
8-bit	x	—	uint8

Format-specific syntaxes:

`[...] = imread(..., idx)` reads in one image from a multi-image icon or cursor file. `idx` is an integer value that specifies the order that the image appears in the file. For example, if `idx` is 3, `imread` reads

the third image in the file. If you omit this argument, `imread` reads the first image in the file.

`[A, map, alpha] = imread(...)` returns the AND mask for the resource, which can be used to determine the transparency information. For cursor files, this mask may contain the only useful data.

---

**Note** By default, Microsoft Windows cursors are 32-by-32 pixels. MATLAB pointers must be 16-by-16. You will probably need to scale your image. If you have Image Processing Toolbox™, you can use the `imresize` function.

---

## GIF – Graphics Interchange Format

Supported Bitdepths	No Compression	Compression	Output Class
1-bit	x	–	logical
2-bit to 8-bit	x	–	uint8

Format-specific syntaxes:

`[...] = imread(..., idx)` reads in one or more frames from a multiframe (i.e., animated) GIF file. `idx` must be an integer scalar or vector of integer values. For example, if `idx` is 3, `imread` reads the third image in the file. If `idx` is 1:5, `imread` returns only the first five frames.

`[...] = imread(..., 'frames', idx)` is the same as the syntax above except that `idx` can be 'all'. In this case, all the frames are read and returned in the order that they appear in the file.

---

**Note** Because of the way that GIF files are structured, all the frames must be read when a particular frame is requested. Consequently, it is much faster to specify a vector of frames or 'all' for `idx` than to call `imread` in a loop when reading multiple frames from the same GIF file.

---



## HDF4 – Hierarchical Data Format

Supported Bitdepths	Raster Image with colormap	Raster image without colormap	Output Class	Notes
8-bit	x	x	uint8	
24-bit	–	x	uint8	3 samples/pixel

Format-specific syntaxes:

`[...] = imread(..., ref)` reads in one image from a multi-image HDF4 file. `ref` is an integer value that specifies the reference number used to identify the image. For example, if `ref` is 12, `imread` reads the image whose reference number is 12. (Note that in an HDF4 file the reference numbers do not necessarily correspond to the order of the images in the file. You can use `imfinfo` to match image order with reference number.) If you omit this argument, `imread` reads the first image in the file.

## ICO – Icon File

See “CUR — Cursor File” on page 2-1745

## JPEG – Joint Photographic Experts Group

`imread` can read any baseline JPEG image as well as JPEG images with some commonly used extensions. For information about support for JPEG 2000 files, see JPEG 2000.

Supported Bitdepths	Lossy Compression	Lossless Compression	Output Class	Notes
8-bit	x	x	uint8	Grayscale or RGB
12-bit	x	x	uint16	Grayscale

Supported Bitdepths	Lossy Compression	Lossless Compression	Output Class	Notes
16-bit	–	x	uint16	Grayscale
36-bit	x	x	uint16	RGB Three 12-bit samples/pixel

## JPEG 2000 – Joint Photographic Experts Group 2000

For information about JPEG files, see JPEG.

---

**Note** Only 1- and 3-sample images are supported. Indexed JPEG 2000 images are not supported.

---

Supported Bitdepths (Bits-per-sample)	Lossy Compression	Lossless Compression	Output Class	Notes
1-bit	x	x	logical	Grayscale only
2- to 8-bit	x	x	uint8	Grayscale or RGB
9- to 16-bit	–	x	uint16	Grayscale or RGB
17- to 32-bit	x	x	uint32	Grayscale or RGB

## PBM – Portable Bitmap

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
1-bit	x	x	logical

**PCX – Windows Paintbrush**

Supported Bitdepths	Output Class	Notes
1-bit	logical	Grayscale only
8-bit	uint8	Grayscale or indexed
24-bit	uint8	RGB Three 8-bit samples/pixel

**PGM – Portable Graymap**

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
Up to 16-bit	x	–	uint8
Arbitrary	–	x	

**PNG – Portable Network Graphics**

Supported Bitdepths	Output Class	Notes
1-bit	logical	Grayscale
2-bit	uint8	Grayscale
4-bit	uint8	Grayscale
8-bit	uint8	Grayscale or Indexed
16-bit	uint16	Grayscale or Indexed
24-bit	uint8	RGB Three 8-bit samples/pixel.
48-bit	uint16	RGB Three 16-bit samples/pixel.

Format-specific syntaxes:

`[...] = imread(..., 'BackgroundColor', BG)` composites any transparent pixels in the input image against the color specified in BG. If BG is 'none', then no compositing is performed. If the input image is indexed, BG must be an integer in the range  $[1, P]$  where P is the colormap length. If the input image is grayscale, BG should be an integer in the range  $[0, 1]$ . If the input image is RGB, BG should be a three-element vector whose values are in the range  $[0, 1]$ . The string 'BackgroundColor' may be abbreviated.

`[A, map, alpha] = imread(...)` returns the alpha channel if one is present; otherwise alpha is `[]`. Note that map may be empty if the file contains a grayscale or truecolor image.

If you specify the alpha output argument, BG defaults to 'none', if not specified. Otherwise, if the PNG file contains a background color chunk, that color is used as the default value for BG. If alpha is not used and the file does not contain a background color chunk, then the default value for BG is 1 for indexed images; 0 for grayscale images; and `[0 0 0]` for truecolor (RGB) images.

## PPM – Portable Pixmap

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
Up to 16-bit	x	–	uint8
Arbitrary	–	x	

## RAS – Sun Raster

The following table lists the supported bitdepths, compression, and output classes for RAS files.

Supported Bitdepths	Output Class	Notes
1-bit	logical	Bitmap

Supported Bitdepths	Output Class	Notes
8-bit	uint8	Indexed
24-bit	uint8	RGB Three 8-bit samples/pixel
32-bit	uint8	RGB with Alpha Four 8-bit samples/pixel

### TIFF — Tagged Image File Format

imread supports the following TIFF capabilities:

- Any number of samples-per-pixel
- CCITT group 3 and 4 FAX, Packbits, JPEG, LZW, Deflate, ThunderScan compression, and uncompressed images
- Logical, grayscale, indexed color, truecolor and hyperspectral images
- RGB, CMYK, CIELAB, ICCLAB color spaces. If the color image uses the CMYK color space, A is an M-by-N-by-4 array. To determine which color space is used, use `imfinfo` to get information about the graphics file and look at the value of the `PhotometricInterpretation` field. If a file contains CIELAB color data, `imread` converts it to ICCLAB before bringing it into the MATLAB workspace because 8- or 16-bit TIFF CIELAB-encoded values use a mixture of signed and unsigned data types that cannot be represented as a single MATLAB array.
- Data organized into tiles or scanlines

The following table lists the supported bit/sample and corresponding output classes for TIFF files.

Bits-per-Sample	Sample Format	Output Class
1	integer	logical
2 – 8	integer	uint8

# imread

Bits-per-Sample	Sample Format	Output Class
9 – 16	integer	uint16
17 – 32	integer	uint32
32	float	single
33 – 64	integer	uint64
64	float	double

The following are format-specific syntaxes for TIFF files.

`[...] = imread(..., idx)` reads in one image from a multi-image TIFF file. `idx` is an integer value that specifies the order in which the image appears in the file. For example, if `idx` is 3, `imread` reads the third image in the file. If you omit this argument, `imread` reads the first image in the file.

`[...] = imread(..., 'PixelRegion', {ROWS, COLS})` returns the subimage specified by the boundaries in `ROWS` and `COLS`. For tiled TIFF images, `imread` reads only the tiles that encompass the region specified by `ROWS` and `COLS`, improving memory efficiency and performance. `ROWS` and `COLS` must be either two or three element vectors. If two elements are provided, they denote the 1-based indices `[START STOP]`. If three elements are provided, the indices `[START INCREMENT STOP]` allow image downsampling.

For TIFF files, `imread` can read color data represented in the RGB, CIELAB, or ICCLAB color spaces.

## XWD – X Window Dump

The following table lists the supported bitdepths, compression, and output classes for XWD files.

Supported Bitdepths	ZPixmap	XYBitmap	XPixmap	Output Class
1-bit	x	–	x	logical
8-bit	x	–	–	uint8

## Class Support

For most image file formats, `imread` uses 8 or fewer bits per color plane to store image pixels. The following table lists the class of the returned array for the data types used by the file formats.

Data Type Used in File	Class of Array Returned by <code>imread</code>
1-bit per pixel	<code>logical</code>
2- to 8-bits per color plane	<code>uint8</code>
9- to 16-bit per pixel	<code>uint16</code> (BMP, JPEG, PNG, and TIFF) For the 16-bit BMP packed format (5-6-5), MATLAB returns <code>uint8</code>

---

**Note** For indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself may be of class `uint8` or `uint16`.

---

## Examples

This example reads the sixth image in a TIFF file.

```
[X,map] = imread('your_image.tif',6);
```

This example reads the fourth image in an HDF4 file.

```
info = imfinfo('your_hdf_file.hdf');
[X,map] = imread('your_hdf_file.hdf',info(4).Reference);
```

This example reads a 24-bit PNG image and sets any of its fully transparent (alpha channel) pixels to red.

```
bg = [255 0 0];
A = imread('your_image.png','BackgroundColor',bg);
```

This example returns the alpha channel (if any) of a PNG image.

# imread

---

```
[A,map,alpha] = imread('your_image.png');
```

This example reads an ICO image, applies a transparency mask, and then displays the image.

```
[a,b,c] = imread('your_icon.ico');  
% Augment colormap for background color (white).  
b2 = [b; 1 1 1];  
% Create new image for display.  
d = ones(size(a)) * (length(b2) - 1);  
% Use the AND mask to mix the background and  
% foreground data on the new image  
d(c == 0) = a(c == 0);  
% Display new image  
image(uint8(d)), colormap(b2)
```

## See Also

double, fread, image, imfinfo, imformats, imwrite, uint8, uint16  
“Bit-Mapped Images” on page 1-96 for related functions



**Purpose**

Write image to graphics file

**Syntax**

```
imwrite(A,filename,fmt)
imwrite(X,map,filename,fmt)
imwrite(...,filename)
imwrite(...,Param1,Val1,Param2,Val2...)
```

**Description**

`imwrite(A,filename,fmt)` writes the image `A` to the file specified by `filename` in the format specified by `fmt`.

`A` can be an `M`-by-`N` (grayscale image) or `M`-by-`N`-by-3 (truecolor image) array, but it cannot be an empty array. For TIFF files, `A` can be an `M`-by-`N`-by-4 array containing color data that uses the CMYK color space. For GIF files, `A` can be an `M`-by-`N`-by-1-by-`P` array containing grayscale or indexed images — RGB images are not supported. For information about the class of the input array and the output image, see “Class Support” on page 2-1756.

`filename` is a string that specifies the name of the output file.

`fmt` can be any of the text strings listed in the table in “Supported Image Types” on page 2-1757. This list of supported formats is determined by the MATLAB image file format registry. See `imformats` for more information about this registry.

`imwrite(X,map,filename,fmt)` writes the indexed image in `X` and its associated colormap `map` to `filename` in the format specified by `fmt`. If `X` is of class `uint8` or `uint16`, `imwrite` writes the actual values in the array to the file. If `X` is of class `double`, `imwrite` offsets the values in the array before writing, using `uint8(X-1)`. `map` must be a valid MATLAB colormap. Note that most image file formats do not support colormaps with more than 256 entries.

When writing multiframe GIF images, `X` should be an 4-dimensional `M`-by-`N`-by-1-by-`P` array, where `P` is the number of frames to write.

`imwrite(...,filename)` writes the image to `filename`, inferring the format to use from the filename’s extension. The extension must be one of the values for `fmt`, listed in “Supported Image Types” on page 2-1757.

`imwrite(...,Param1,Val1,Param2,Val2...)` specifies parameters that control various characteristics of the output file for HDF, JPEG, PBM, PGM, PNG, PPM, and TIFF files. For example, if you are writing a JPEG file, you can specify the quality of the output image. For the lists of parameters available for each format, see “Format-Specific Parameters” on page 2-1759.

## Class Support

The input array `A` can be of class `logical`, `uint8`, `uint16`, or `double`. Indexed images (`X`) can be of class `uint8`, `uint16`, or `double`; the associated colormap, `map`, must be of class `double`. Input values must be full (non-sparse).

The class of the image written to the file depends on the format specified. For most formats, if the input array is of class `uint8`, `imwrite` outputs the data as 8-bit values. If the input array is of class `uint16` and the format supports 16-bit data (JPEG, PNG, and TIFF), `imwrite` outputs the data as 16-bit values. If the format does not support 16-bit values, `imwrite` issues an error. Several formats, such as JPEG and PNG, support a parameter that lets you specify the bit depth of the output data.

If the input array is of class `double`, and the image is a grayscale or RGB color image, `imwrite` assumes the dynamic range is `[0,1]` and automatically scales the data by 255 before writing it to the file as 8-bit values.

If the input array is of class `double`, and the image is an indexed image, `imwrite` converts the indices to zero-based indices by subtracting 1 from each element, and then writes the data as `uint8`.

If the input array is of class `logical`, `imwrite` assumes the data is a binary image and writes it to the file with a bit depth of 1, if the format allows it. BMP, PNG, or TIFF formats accept binary images as input arrays.

## Supported Image Types

This table summarizes the types of images that `imwrite` can write. The MATLAB file format registry determines which file formats are supported. See `imformats` for more information about this registry. Note that, for certain formats, `imwrite` may take additional parameters, described in “Format-Specific Parameters” on page 2-1759.

Format	Full Name	Variants
'bmp'	Windows Bitmap (BMP)	1-bit, 8-bit, and 24-bit uncompressed images
'gif'	Graphics Interchange Format (GIF)	8-bit images
'hdf'	Hierarchical Data Format (HDF4)	8-bit raster image data sets, with or without associated colormap, 24-bit raster image data sets; uncompressed or with RLE or JPEG compression
'jpg' or 'jpeg'	Joint Photographic Experts Group (JPEG)	8-bit, 12-bit, and 16-bit Baseline JPEG images  <b>Note</b> <code>imwrite</code> converts indexed images to RGB before writing data to JPEG files, because the JPEG format does not support indexed images.
pbm	Portable Bitmap (PBM)	Any 1-bit PBM image, ASCII (plain) or raw (binary) encoding
'pcx'	Windows Paintbrush (PCX)	8-bit images

<b>Format</b>	<b>Full Name</b>	<b>Variants</b>
'pgm'	Portable Graymap (PGM)	Any standard PGM image; ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per gray value
'png'	Portable Network Graphics (PNG)	1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit grayscale images with alpha channels; 1-bit, 2-bit, 4-bit, and 8-bit indexed images; 24-bit and 48-bit truecolor images; 24-bit and 48-bit truecolor images with alpha channels
'pnm'	Portable Anymap (PNM)	Any of the PPM/PGM/PBM formats, chosen automatically
'ppm'	Portable Pixmap (PPM)	Any standard PPM image. ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per color component
'ras'	Sun Raster (RAS)	Any RAS image, including 1-bit bitmap, 8-bit indexed, 24-bit truecolor and 32-bit truecolor with alpha
'tif' or 'tiff'	Tagged Image File Format (TIFF)	Baseline TIFF images, including 1-bit, 8-bit, 16-bit, and 24-bit uncompressed images, images with packbits compression, images with LZW compression, and images with Deflate compression; 1-bit images with CCITT 1D, Group 3, and Group 4 compression; CIELAB, ICCLAB, and CMYK images
'xwd'	X Windows Dump (XWD)	8-bit ZPixmaps

## Format-Specific GIF-Specific Parameters

### Parameters

Parameter	Values
'BackgroundColor'	A scalar integer that specifies which index in the colormap should be treated as the transparent color for the image and is used for certain disposal methods in animated GIFs. If <i>X</i> is <code>uint8</code> or <code>logical</code> , indexing starts at 0. If <i>X</i> is <code>double</code> , indexing starts at 1.
'Comment'	A string or cell array of strings containing a comment to be added to the image. For a cell array of strings, a carriage return is added after each row.
'DelayTime'	A scalar value between 0 and 655 inclusive, that specifies the delay in seconds before displaying the next image.
'DisposalMethod'	One of the following strings, which sets the disposal method of an animated GIF: 'leaveInPlace', 'restoreBG', 'restorePrevious', or 'doNotSpecify'.
'Location'	A two-element vector specifying the offset of the top left corner of the screen relative to the top left corner of the image. The first element is the offset from the top, and the second element is the offset from the left.
'LoopCount'	<p>A finite integer between 0 and 65535 or the value <code>Inf</code> (the default) which specifies the number of times to repeat the animation. By default, the animation loops continuously. If you specify 0, the animation will be played once. If you specify the value 1, the animation will be played twice, etc.</p> <p>To enable animation within Microsoft® PowerPoint®, specify a value for the 'LoopCount' parameter within the range [1 65535]. Some Microsoft applications interpret the value 0 to mean do not loop at all.</p>

Parameter	Values
'ScreenSize'	A two-element vector specifying the height and width of the frame. When used with 'Location', ScreenSize provides a way to write frames to the image that are smaller than the whole frame. 'DisposalMethod' determines the fill value used for pixels outside the frame.
'TransparentColor'	A scalar integer. This value specifies which index in the colormap should be treated as the transparent color for the image. If X (the image) is uint8 or logical, indexing starts at 0. If X is double, indexing starts at 1.
'WriteMode'	'overwrite' (the default) or 'append'. In append mode, imwrite adds a single frame to the existing file.

## HDF4-Specific Parameters

Parameter	Values
'Compression'	'none' (the default)  'jpeg' (valid only for grayscale and RGB images)  'rle' (valid only for grayscale and indexed images)
'Quality'	A number between 0 and 100; this parameter applies only if 'Compression' is 'jpeg'.  Higher numbers mean higher <i>quality</i> (less image degradation due to compression), but the resulting file size is larger. The default value is 75.
'WriteMode'	'overwrite' (the default)  'append'

### JPEG-Specific Parameters

Parameter	Values	Default
'Bitdepth'	A scalar value indicating desired bitdepth; for grayscale images this can be 8, 12, or 16; for color images this can be 8 or 12.	8 (grayscale) and 8 bit per plane for color images
'Comment'	A column vector cell array of strings or a character matrix. <code>imwrite</code> writes each row of input as a comment in the JPEG file.	Empty
'Mode'	Specifies the type of compression used: 'lossy' or 'lossless'	'lossy'
'Quality'	A number between 0 and 100; higher numbers mean higher quality (less image degradation due to compression), but the resulting file size is larger.	75

### PBM-, PGM-, and PPM-Specific Parameters

This table describes the available parameters for PBM, PGM, and PPM files.

Parameter	Values	Default
'Encoding'	One of these strings: 'ASCII' for plain encoding 'rawbits' for binary encoding	'rawbits'
'MaxValue'	A scalar indicating the maximum gray or color value. Available only for PGM and PPM files. For PBM files, this value is always 1.	Default is 65535 if image array is 'uint16'; 255 otherwise.

## PNG-Specific Parameters

The following table lists the available parameters for PNG files, in alphabetical order. In addition to these PNG parameters, you can use any parameter name that satisfies the PNG specification for keywords; that is, uses only printable characters, contains 80 or fewer characters, and no contains no leading or trailing spaces. The value corresponding to these user-specified parameters must be a string that contains no control characters other than linefeed.

Parameter	Values
'Alpha'	A matrix specifying the transparency of each pixel individually. The row and column dimensions must be the same as the data array; they can be <code>uint8</code> , <code>uint16</code> , or <code>double</code> , in which case the values should be in the range <code>[0,1]</code> .
'Author'	A string
'Background'	The value specifies background color to be used when compositing transparent pixels. For indexed images: an integer in the range <code>[1,P]</code> , where <code>P</code> is the colormap length. For grayscale images: a scalar in the range <code>[0,1]</code> . For truecolor images: a three-element vector in the range <code>[0,1]</code> .
'bitdepth'	A scalar value indicating desired bit depth.  For grayscale images this can be 1, 2, 4, 8, or 16.  For grayscale images with an alpha channel this can be 8 or 16.  For indexed images this can be 1, 2, 4, or 8.  For truecolor images with or without an alpha channel this can be 8 or 16.  By default, <code>imwrite</code> uses 8 bits per pixel, if image is <code>double</code> or <code>uint8</code> ; 16 bits per pixel if image is <code>uint16</code> ; 1 bit per pixel if image is <code>logical</code> .



Parameter	Values
'Chromaticities'	An eight-element vector [wx wy rx ry gx gy bx by] that specifies the reference white point and the primary chromaticities
'Comment'	A string
'Copyright'	A string
'CreationTime'	A string
'Description'	A string
'Disclaimer'	A string
'Gamma'	A nonnegative scalar indicating the file gamma
'ImageModTime'	A MATLAB serial date number (see the <code>datenum</code> function) or a string convertible to a date vector via the <code>datevec</code> function. Values should be in Coordinated Universal Time (UTC).
'InterlaceType'	Either 'none' (the default) or 'adam7'
'ResolutionUnit'	Either 'unknown' or 'meter'
'SignificantBits'	A scalar or vector indicating how many bits in the data array should be regarded as significant; values must be in the range [1,BitDepth].  For indexed images: a three-element vector. For grayscale images: a scalar. For grayscale images with an alpha channel: a two-element vector. For truecolor images: a three-element vector. For truecolor images with an alpha channel: a four-element vector.
'Software'	A string
'Source'	A string

Parameter	Values
'Transparency'	<p>This value is used to indicate transparency information only when no alpha channel is used. Set to the value that indicates which pixels should be considered transparent. (If the image uses a colormap, this value represents an index number to the colormap.)</p> <p>For indexed images: a <math>Q</math>-element vector in the range <math>[0,1]</math>, where <math>Q</math> is no larger than the colormap length and each value indicates the transparency associated with the corresponding colormap entry. In most cases, <math>Q = 1</math>.</p> <p>For grayscale images: a scalar in the range <math>[0,1]</math>. The value indicates the grayscale color to be considered transparent.</p> <p>For truecolor images: a three-element vector in the range <math>[0,1]</math>. The value indicates the truecolor color to be considered transparent.</p> <hr/> <p><b>Note</b> You cannot specify 'Transparency' and 'Alpha' at the same time.</p> <hr/>
'Warning'	A string
'XResolution'	A scalar indicating the number of pixels/unit in the horizontal direction
'YResolution'	A scalar indicating the number of pixels/unit in the vertical direction

### RAS-Specific Parameters

This table describes the available parameters for RAS files.

Parameter	Values	Default
'Alpha'	A matrix specifying the transparency of each pixel individually; the row and column dimensions must be the same as the data array; can be <code>uint8</code> , <code>uint16</code> , or <code>double</code> . Can only be used with truecolor images.	Empty matrix ([])
'Type'	One of these strings: 'standard' (uncompressed, b-g-r color order with truecolor images) 'rgb' (like 'standard', but uses r-g-b color order for truecolor images) 'rle' (run-length encoding of 1-bit and 8-bit images)	'standard'

### TIFF-Specific Parameters

This table describes the available parameters for TIFF files.

Parameter	Values
'ColorSpace'	<p>Specifies the color space used to represent the color data. 'rgb' 'cielab' 'icclab' <sup>1</sup> (default is 'rgb').</p> <p>Note: To use the CMYK color space in a TIFF file, do not use the 'ColorSpace' parameter. It is sufficient to specify an M-by-N-by-4 input array.</p>

# imwrite

Parameter	Values
'Compression'	'none' 'packbits' (default for non-binary images) 'lzw' 'deflate' 'jpeg' 'ccitt' (binary images only; default) 'fax3' (binary images only) 'fax4' (binary images only)  Note: 'jpeg' is a lossy compression scheme; other compression modes are lossless. Also, if you specify 'jpeg' compression, you must specify the 'RowsPerStrip' parameter and the value must be a multiple of 8.
'Description'	Any string; fills in the ImageDescription field returned by <code>imfinfo</code> . By default, the field is empty.
'Resolution'	A two-element vector containing the XResolution and YResolution, or a scalar indicating both resolutions. The default value is 72.
'RowsPerStrip'	A scalar value specifying the number of rows to include in each strip. The default will be such that each strip is about 8K bytes.  Note: You must specify the 'RowsPerStrip' parameter if you specify 'jpeg' compression. The value must be a multiple of 8.
'WriteMode'	'overwrite' (default)  'append'

`imwrite` can write color image data that uses the  $L^*a^*b^*$  color space to TIFF files. The 1976 CIE  $L^*a^*b^*$  specification defines numeric values that represent luminance ( $L^*$ ) and chrominance ( $a^*$  and  $b^*$ ) information. To store  $L^*a^*b^*$  color data in a TIFF file, the values must be encoded to fit into either 8-bit or 16-bit storage. `imwrite` can store  $L^*a^*b^*$  color data in a TIFF file using 8-bit and 16-bit encodings defined by the TIFF specification, called the CIELAB encodings, or using 8-bit and 16-bit encodings defined by the International Color Consortium, called ICCLAB encodings.

The output class and encoding used by `imwrite` depends on the class of the input array and the value of the TIFF-specific `ColorSpace` parameter. The following table explains these options. (The 8-bit and 16-bit CIELAB encodings cannot be input arrays because they use a mixture of signed and unsigned values and cannot be represented as a single MATLAB array.)

<b>Input Class and Encoding</b>	<b>ColorSpace Parameter Value</b>	<b>Output Class and Encoding</b>
8-bit ICCLAB  Represents values as integers in the range [0 255]. $L^*$ values are multiplied by 255/100. 128 is added to both the $a^*$ and $b^*$ values.	'icclab'	8-bit ICCLAB
	'cielab'	8-bit CIELAB

<b>Input Class and Encoding</b>	<b>ColorSpace Parameter Value</b>	<b>Output Class and Encoding</b>
16-bit ICCLAB  Represents the values as integers in the range [0, 65280]. $L^*$ values are multiplied by 65280/100. 32768 is added to both the $a^*$ and $b^*$ values, which are represented as integers in the range [0,65535].	'icclab'	16-bit ICCLAB
	'cielab'	16-bit CIELAB
Double-precision 1976 CIE $L^*a^*b^*$ values  $L^*$ is in the dynamic range [0, 100]. $a^*$ and $b^*$ can take any value. Setting $a^*$ and $b^*$ to 0 (zero) produces a neutral color (gray).	'icclab'	8-bit ICCLAB
	'cielab'	8-bit CIELAB

## Example

This example appends an indexed image `X` and its colormap `map` to an existing uncompressed multipage HDF4 file.

```
imwrite(X,map,'your_hdf_file.hdf','Compression','none',...  
        'WriteMode','append')
```

## See Also

`fwrite`, `getframe`, `imfinfo`, `imformats`, `imread`

“Bit-Mapped Images” on page 1-96 for related functions

# ind2rgb

---

<b>Purpose</b>	Convert indexed image to RGB image
<b>Syntax</b>	<code>RGB = ind2rgb(X,map)</code>
<b>Description</b>	<code>RGB = ind2rgb(X,map)</code> converts the matrix <code>X</code> and corresponding colormap <code>map</code> to RGB (truecolor) format.
<b>Class Support</b>	<code>X</code> can be of class <code>uint8</code> , <code>uint16</code> , or <code>double</code> . <code>RGB</code> is an <code>m-by-n-by-3</code> array of class <code>double</code> .
<b>See Also</b>	<code>image</code> “Bit-Mapped Images” on page 1-96 for related functions



**Purpose**

Subscripts from linear index

**Syntax**

```
[I,J] = ind2sub(siz,IND)
[I1,I2,I3,...,In] = ind2sub(siz,IND)
```

**Description**

The `ind2sub` command determines the equivalent subscript values corresponding to a single index into an array.

`[I,J] = ind2sub(siz,IND)` returns the matrices `I` and `J` containing the equivalent row and column subscripts corresponding to each linear index in the matrix `IND` for a matrix of size `siz`. `siz` is a vector with `ndim(A)` elements (in this case, 2), where `siz(1)` is the number of rows and `siz(2)` is the number of columns.

---

**Note** For matrices, `[I,J] = ind2sub(size(A),find(A>5))` returns the same values as `[I,J] = find(A>5)`.

---

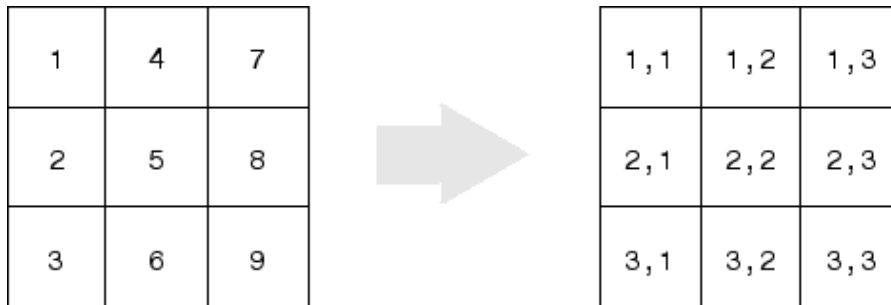
`[I1,I2,I3,...,In] = ind2sub(siz,IND)` returns `n` subscript arrays `I1,I2,...,In` containing the equivalent multidimensional array subscripts equivalent to `IND` for an array of size `siz`. `siz` is an `n`-element vector that specifies the size of each array dimension.

**Examples****Example 1 – Two-Dimensional Matrices**

The mapping from linear indexes to subscript equivalents for a 3-by-3 matrix is

# ind2sub

---



This code determines the row and column subscripts in a 3-by-3 matrix, of elements with linear indices 3, 4, 5, 6.

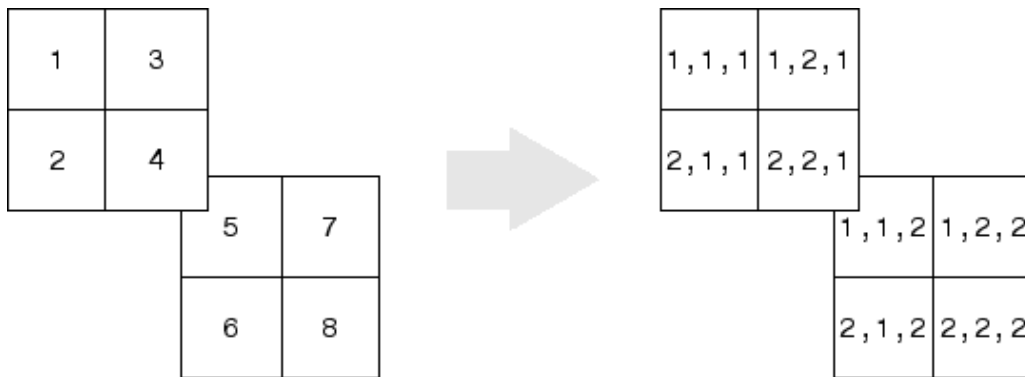
```
IND = [3 4 5 6]
s = [3,3];
[I,J] = ind2sub(s,IND)
```

```
I =
     3     1     2     3
```

```
J =
     1     2     2     2
```

## Example 2 – Three-Dimensional Matrices

The mapping from linear indexes to subscript equivalents for a 2-by-2-by-2 array is



This code determines the subscript equivalents in a 2-by-2-by-2 array, of elements whose linear indices 3, 4, 5, 6 are specified in the IND matrix.

```
IND = [3 4;5 6];
s = [2,2,2];
[I,J,K] = ind2sub(s,IND)
```

```
I =
     1     2
     1     2
```

```
J =
     2     2
     1     1
```

```
K =
     1     1
     2     2
```

### Example 3 – Effects of Returning Fewer Outputs

When calling `ind2sub` for an N-dimensional matrix, you would typically supply N output arguments in the call: one for each dimension of the matrix. This example shows what happens when you return three, two, and one output when calling `ind2sub` on a 3-dimensional matrix.

# ind2sub

---

The matrix is 2-by-2-by-2 and the linear indices are 1 through 8:

```
dims = [2 2 2];  
indices = [1 2 3 4 5 6 7 8];
```

The 3-output call to `ind2sub` returns the expected subscripts for the 2-by-2-by-2 matrix:

```
[rowsub colsub pagsub] = ind2sub(dims, indices)  
rowsub =  
     1     2     1     2     1     2     1     2  
colsub =  
     1     1     2     2     1     1     2     2  
pagsub =  
     1     1     1     1     2     2     2     2
```

If you specify only two outputs (row and column), `ind2sub` still returns a subscript for each specified index, but drops the third dimension from the matrix, returning subscripts for a 2-dimensional, 2-by-4 matrix instead:

```
[rowsub colsub] = ind2sub(dims, indices)  
rowsub =  
     1     2     1     2     1     2     1     2  
colsub =  
     1     1     2     2     3     3     4     4
```

If you specify one output (row), `ind2sub` drops both the second and third dimensions from the matrix, and returns subscripts for a 1-dimensional, 1-by-8 matrix instead:

```
[rowsub] = ind2sub(dims, indices)  
rowsub =  
     1     2     3     4     5     6     7     8
```

## See Also

`find`, `size`, `sub2ind`

**Purpose**

Infinity

**Syntax**

```
Inf
Inf('double')
Inf('single')
Inf(n)
Inf(m,n)
Inf(m,n,p,...)
Inf(...,classname)
```

**Description**

Inf returns the IEEE arithmetic representation for positive infinity. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values.

Inf('double') is the same as Inf with no inputs.

Inf('single') is the single precision representation of Inf.

Inf(n) is an n-by-n matrix of Infs.

Inf(m,n) or inf([m,n]) is an m-by-n matrix of Infs.

Inf(m,n,p,...) or Inf([m,n,p,...]) is an m-by-n-by-p-by-... array of Infs.

---

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

---

Inf(...,classname) is an array of Infs of class specified by classname. classname must be either 'single' or 'double'.

**Examples**

1/0, 1.e1000, 2^2000, and exp(1000) all produce Inf.

log(0) produces -Inf.

Inf - Inf and Inf/Inf both produce NaN (Not-a-Number).

# Inf

---

## See Also

`isinf`, `NaN`

---

<b>Purpose</b>	Specify inferior class relationship
<b>Syntax</b>	<code>inferiorto('class1','class2',...)</code>
<b>Description</b>	The <code>inferiorto</code> function establishes a precedence that determines which object method is called.

---

**Note** You can use this function only from a constructor that calls the `class` function to create an object, which was the only way to create MATLAB classes prior to MATLAB Version 7.6.

---

`inferiorto('class1','class2',...)` invoked within a class constructor method, establishes that class as having lower precedence than the classes in the function argument list for purposes of function dispatching (i.e., which method or function is called in any given situation).

## Remarks

Suppose `a` is an object of class `'class_a'`, `b` is an object of class `'class_b'` and `c` is an object of class `'class_c'`. Also suppose the constructor method of `class_c.m` contains the statement:

```
inferiorto('class_a')
```

This establishes `'class_a'` as taking precedence over `'class_c'` for function dispatching. Therefore, either of the following two statements:

```
e = fun(a,c)
e = fun(c,a)
```

Invoke `class_a/fun`.

If a function is called with two objects having an unspecified relationship, the two objects have equal precedence, and the left-most object's method is called. So `fun(b, c)` calls `class_b/fun`, while `fun(c, b)` calls `class_c/fun`.

# inferiorto

---

## **See Also**

See *Object-Oriented Programming* for information on the creating MATLAB classes.

See “Object Precedence in Expressions Using Operators”

`superiorto`



<b>Purpose</b>	Information about contacting The MathWorks
<b>Syntax</b>	<code>info</code>
<b>Description</b>	<code>info</code> displays in the Command Window, information about contacting The MathWorks.
<b>See Also</b>	<code>help</code> , <code>version</code>

# inline

---

**Purpose** Construct inline object

**Syntax**  
`inline(expr)`  
`inline(expr, arg1, arg2, ...)`  
`inline(expr, n)`

**Description** `inline(expr)` constructs an inline function object from the MATLAB expression contained in the string `expr`. The input argument to the inline function is automatically determined by searching `expr` for an isolated lower case alphabetic character, other than `i` or `j`, that is not part of a word formed from several alphabetic characters. If no such character exists, `x` is used. If the character is not unique, the one closest to `x` is used. If two characters are found, the one later in the alphabet is chosen.

`inline(expr, arg1, arg2, ...)` constructs an inline function whose input arguments are specified by the strings `arg1`, `arg2`, ... . Multicharacter symbol names may be used.

`inline(expr, n)` where `n` is a scalar, constructs an inline function whose input arguments are `x`, `P1`, `P2`, ... .

**Remarks** Three commands related to `inline` allow you to examine an inline function object and determine how it was created.

`char(fun)` converts the inline function into a character array. This is identical to `formula(fun)`.

`argnames(fun)` returns the names of the input arguments of the inline object `fun` as a cell array of strings.

`formula(fun)` returns the formula for the inline object `fun`.

A fourth command `vectorize(fun)` inserts a `.` before any `^`, `*` or `/'` in the formula for `fun`. The result is a vectorized version of the inline function.

**Examples** **Example 1**

This example creates a simple inline function to square a number.

```
g = inline('t^2')
g =

    Inline function:
    g(t) = t^2
```

You can convert the result to a string using the `char` function.

```
char(g)

ans =

    t^2
```

## Example 2

This example creates an inline function to represent the formula  $f = 3 \sin(2x^2)$ . The resulting inline function can be evaluated with the `argnames` and `formula` functions.

```
f = inline('3*sin(2*x.^2)')

f =

    Inline function:
    f(x) = 3*sin(2*x.^2)

argnames(f)

ans =

    'x'

formula(f)
ans =

    3*sin(2*x.^2)
```

### Example 3

This call to `inline` defines the function `f` to be dependent on two variables, `alpha` and `x`:

```
f = inline('sin(alpha*x)')  
  
f =  
    Inline function:  
    f(alpha,x) = sin(alpha*x)
```

If `inline` does not return the desired function variables or if the function variables are in the wrong order, you can specify the desired variables explicitly with the `inline` argument list.

```
g = inline('sin(alpha*x)', 'x', 'alpha')  
  
g =  
    Inline function:  
    g(x,alpha) = sin(alpha*x)
```

<b>Purpose</b>	Names of M-files, MEX-files, Sun Java classes in memory
<b>Syntax</b>	<pre>M = inmem [M, X] = inmem [M, X, J] = inmem [...] = inmem('-completenames')</pre>
<b>Description</b>	<p><code>M = inmem</code> returns a cell array of strings containing the names of the M-files that are currently loaded.</p> <p><code>[M, X] = inmem</code> returns an additional cell array <code>X</code> containing the names of the MEX-files that are currently loaded.</p> <p><code>[M, X, J] = inmem</code> also returns a cell array <code>J</code> containing the names of the Java classes that are currently loaded.</p> <p><code>[...] = inmem('-completenames')</code> returns not only the names of the currently loaded M- and MEX-files, but the path and filename extension for each as well. No additional information is returned for loaded Java classes.</p>
<b>Examples</b>	<p>Example 1</p> <p>This example lists the M-files that are required to run <code>erf</code>.</p> <pre>clear all;           % Clear the workspace erf(0.5);  M = inmem M =     'erf'</pre> <p><b>Example 2</b></p> <p>Generate a plot, and then find the M- and MEX-files that had been loaded to perform this operation:</p> <pre>clear all surf(peaks)</pre>

# inmem

---

```
[m x] = inmem('-completenames');

m(1:5)
ans =
    'F:\matlab\toolbox\matlab\ops\ismember.m'
    'F:\matlab\toolbox\matlab\datatypes\@opaque\double.m'
    'F:\matlab\toolbox\matlab\datatypes\isfield.m'
    'F:\matlab\toolbox\matlab\graphics\gcf.m'
    'F:\matlab\toolbox\matlab\elmat\meshgrid.m'

x(1:end)
ans =
    'F:\matlab\toolbox\matlab\graph2d\private\lineseriesmex.dll'
```

## See Also

`clear`

**Purpose** Points inside polygonal region

**Syntax** `IN = inpolygon(X,Y,xv,yv)`  
`[IN ON] = inpolygon(X,Y,xv,yv)`

**Description** `IN = inpolygon(X,Y,xv,yv)` returns a matrix `IN` the same size as `X` and `Y`. Each element of `IN` is assigned the value 1 or 0 depending on whether the point  $(X(p,q), Y(p,q))$  is inside the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

`IN(p,q) = 1` If  $(X(p,q), Y(p,q))$  is inside the polygonal region or on the polygon boundary

`IN(p,q) = 0` If  $(X(p,q), Y(p,q))$  is outside the polygonal region

`[IN ON] = inpolygon(X,Y,xv,yv)` returns a second matrix `ON` the same size as `X` and `Y`. Each element of `ON` is assigned the value 1 or 0 depending on whether the point  $(X(p,q), Y(p,q))$  is on the boundary of the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

`ON(p,q) = 1` If  $(X(p,q), Y(p,q))$  is on the polygon boundary

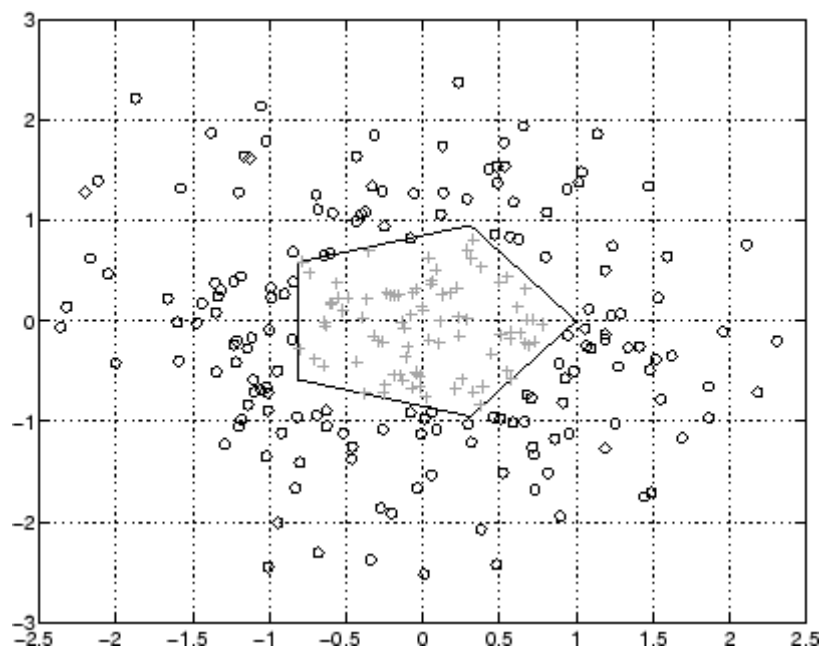
`ON(p,q) = 0` If  $(X(p,q), Y(p,q))$  is inside or outside the polygon boundary

**Examples**

```
L = linspace(0,2.*pi,6); xv = cos(L)';yv = sin(L)';
xv = [xv ; xv(1)]; yv = [yv ; yv(1)];
x = randn(250,1); y = randn(250,1);
in = inpolygon(x,y,xv,yv);
plot(xv,yv,x(in),y(in), 'r+',x(~in),y(~in), 'bo')
```

# inpolygon

---





---

<b>Purpose</b>	Request user input
<b>Syntax</b>	<pre>user_entry = input('prompt') user_entry = input('prompt', 's')</pre>
<b>Description</b>	<p>The response to the <code>input</code> prompt can be any MATLAB expression, which is evaluated using the variables in the current workspace.</p> <p><code>user_entry = input('prompt')</code> displays <i>prompt</i> as a prompt on the screen, waits for input from the keyboard, and returns the value entered in <code>user_entry</code>.</p> <p><code>user_entry = input('prompt', 's')</code> returns the entered string as a text variable rather than as a variable name or numerical value.</p>
<b>Remarks</b>	<p>If you press the <b>Return</b> key without entering anything, <code>input</code> returns an empty matrix.</p> <p>The text string for the prompt can contain one or more <code>'\n'</code> characters. The <code>'\n'</code> means to skip to the next line. This allows the prompt string to span several lines. To display just a backslash, use <code>'\\'</code>.</p> <p>If you enter an invalid expression at the prompt, MATLAB displays the relevant error message and then prompts you again to enter input.</p>
<b>Examples</b>	<p>Press <b>Return</b> to select a default value by detecting an empty matrix:</p> <pre>reply = input('Do you want more? Y/N [Y]: ', 's'); if isempty(reply)     reply = 'Y'; end</pre>
<b>See Also</b>	<code>keyboard</code> , <code>menu</code> , <code>ginput</code> , <code>uicontrol</code>

# inputdlg

---

**Purpose** Create and open input dialog box

**Syntax**

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,dlg_title)
answer = inputdlg(prompt,dlg_title,num_lines)
answer = inputdlg(prompt,dlg_title,num_lines,defAns)
answer = inputdlg(prompt,dlg_title,num_lines,defAns,options)
```

**Description** `answer = inputdlg(prompt)` creates a modal dialog box and returns user input for multiple prompts in the cell array. `prompt` is a cell array containing prompt strings.

---

**Note** A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowState` in the MATLAB Figure Properties.

---

---

**Note** `inputdlg` uses the `uiwait` function to suspend execution until the user responds.

---

`answer = inputdlg(prompt,dlg_title)` `dlg_title` specifies a title for the dialog box.

`answer = inputdlg(prompt,dlg_title,num_lines)` `num_lines` specifies the number of lines for each user-entered value. `num_lines` can be a scalar, column vector, or matrix.

- If `num_lines` is a scalar, it applies to all prompts.
- If `num_lines` is a column vector, each element specifies the number of lines of input for a prompt.
- If `num_lines` is a matrix, it should be size `m-by-2`, where `m` is the number of prompts on the dialog box. Each row refers to a prompt.

The first column specifies the number of lines of input for a prompt. The second column specifies the width of the field in characters.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns)` `defAns` specifies the default value to display for each prompt. `defAns` must contain the same number of elements as `prompt` and all elements must be strings.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns,options)` If `options` is the string `'on'`, the dialog is made resizable in the horizontal direction. If `options` is a structure, the fields shown in the following table are recognized:

Field	Description
Resize	Can be <code>'on'</code> or <code>'off'</code> (default). If <code>'on'</code> , the window is resizable horizontally.
WindowStyle	Can be either <code>'normal'</code> or <code>'modal'</code> (default).
Interpreter	Can be either <code>'none'</code> (default) or <code>'tex'</code> . If the value is <code>'tex'</code> , the prompt strings are rendered using LaTeX.

If the user clicks the **Cancel** button to close an `inputdlg` box, the dialog returns an empty cell array:

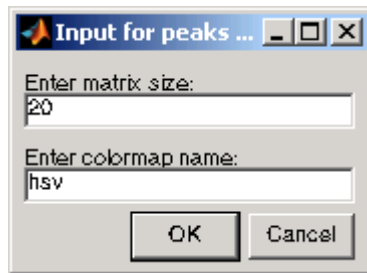
```
answer =
    {}
```

## Example

### Example 1

Create a dialog box to input an integer and colormap name. Allow one line for each value.

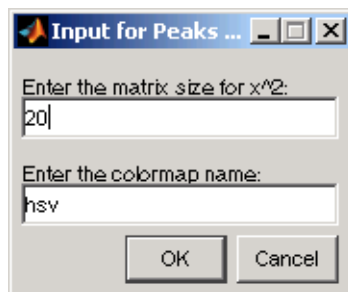
```
prompt = {'Enter matrix size:', 'Enter colormap name:'};
dlg_title = 'Input for peaks function';
num_lines = 1;
def = {'20', 'hsv'};
answer = inputdlg(prompt,dlg_title,num_lines,def);
```



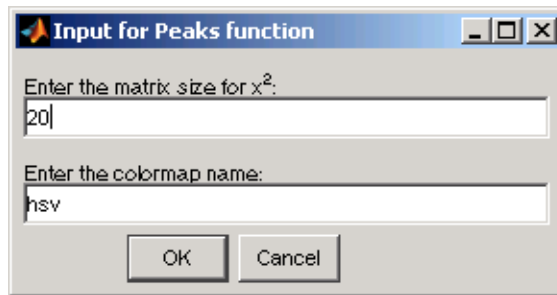
## Example 2

Create a dialog box using the default options. Then use the options to make it resizable and not modal, and to interpret the text using LaTeX.

```
prompt={'Enter the matrix size for x^2:',...  
        'Enter the colormap name:'};  
name='Input for Peaks function';  
numlines=1;  
defaultanswer={'20','hsv'};  
answer=inputdlg(prompt,name,numlines,defaultanswer);
```



```
options.Resize='on';  
options.WindowStyle='normal';  
options.Interpreter='tex';  
  
answer=inputdlg(prompt,name,numlines,defaultanswer,options);
```

**See Also**

dialog, errordlg, helpdlg, listdlg, msgbox, questdlg, warndlg  
figure, uiwait, uiresume  
“Predefined Dialog Boxes” on page 1-108 for related functions

# inputname

---

**Purpose** Variable name of function input

**Syntax** `inputname(argnum)`

**Description** This command can be used only inside the body of a function. `inputname(argnum)` returns the workspace variable name corresponding to the argument number *argnum*. If the input argument has no name (for example, if it is an expression instead of a variable), the `inputname` command returns the empty string ('').

**Examples** Suppose the function `myfun.m` is defined as

```
function c = myfun(a,b)
    disp(sprintf('First calling variable is "%s".', inputname(1)))
```

Then

```
x = 5; y = 3; myfun(x,y)
```

produces

```
First calling variable is "x".
```

But

```
myfun(pi+1, pi-1)
```

produces

```
First calling variable is "".
```

**See Also** `nargin`, `nargout`, `nargchk`

**Purpose** Construct input parser object

**Syntax** `p = inputParser`

**Description** `p = inputParser` constructs an empty `inputParser` object. Use this utility object to parse and validate input arguments to the functions that you develop. The input parser object follows handle semantics; that is, methods called on it affect the original object, not a copy of it.

The MATLAB software configures `inputParser` objects to recognize an input schema. Use any of the following methods to create the schema for parsing a particular function.

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

## Methods

Method	Description
<code>addOptional</code>	Add an optional argument to the schema
<code>addParamValue</code>	Add a parameter-value pair argument to the schema
<code>addRequired</code>	Add a required argument to the schema
<code>createCopy</code>	Create a copy of the <code>inputParser</code> object
<code>parse</code>	Parse and validate the named inputs

## Properties

Property	Description
<code>CaseSensitive</code>	Enable or disable case-sensitive matching of argument names
<code>FunctionName</code>	Function name to be included in error messages

# inputParser

---

Property	Description
KeepUnmatched	Enable or disable errors on unmatched arguments
Parameters	Names of arguments defined in inputParser schema
Results	Names and values of arguments passed in function call that are in the schema for this function
StructExpand	Enable or disable passing arguments in a structure
Unmatched	Names and values of arguments passed in function call that are not in the schema for this function
UsingDefaults	Names of arguments not passed in function call that are given default values

## Property Descriptions

Properties of the `inputParser` class are described below.

### CaseSensitive

Purpose — Enable or disable case sensitive matching of argument names

`p.CaseSensitive = TF` enables or disables case-sensitivity when matching entries in the argument list with argument names in the schema. Set `CaseSensitive` to logical 1 (true) to enable case-sensitive matching, or to logical 0 (false) to disable it. By default, case-sensitive matching is disabled.

### FunctionName

Purpose — Function name to be included in error messages

`p.FunctionName = name` stores a function name that is to be included in error messages that might be thrown in the process of validating input arguments to the function. The `name` input is a string containing



the name of the function for which you are parsing inputs with `inputParser`.

## KeepUnmatched

Purpose — Enable or disable errors on unmatched arguments

`p.KeepUnmatched = TF` controls whether MATLAB throws an error when the function being called is passed an argument that has not been defined in the `inputParser` schema for this file. When this property is set to logical 1 (true), MATLAB does not throw an error, but instead stores the names and values of unmatched arguments in the `Unmatched` property of object `p`. When `KeepUnmatched` is set to logical 0 (false), MATLAB does throw an error whenever this condition is encountered and the `Unmatched` property is not affected.

## Parameters

Purpose — Names of arguments defined in `inputParser` schema

`c = p.Parameters` is a cell array of strings containing the names of those arguments currently defined in the schema for the object. Each row of the `Parameters` cell array is a string containing the full name of a known argument.

## Results

Purpose — Names and values of arguments passed in function call that are in the schema for this function

`arglist = p.Results` is a structure containing the results of the most recent parse of the input argument list. Each argument passed to the function is represented by a field in the `Results` structure, and the value of that argument is represented by the value of that field.

## StructExpand

Purpose — Enable or disable passing arguments in a structure

`p.StructExpand = TF`, when set to logical 1 (true), tells MATLAB to accept a structure as an input in place of individual parameter-value arguments. If `StructExpand` is set to logical 0 (false), a structure is treated as a regular, single input.

## Unmatched

Purpose — Names and values of arguments passed in function call that are not in the schema for this function

`c = p.Unmatched` is a structure array containing the names and values of all arguments passed in a call to the function that are not included in the schema for the function. `Unmatched` only contains this list of the `KeepUnmatched` property is set to true. If `KeepUnmatched` is set to false, MATLAB throws an error when unmatched arguments are passed in the function call. The `Unmatched` structure has the same format as the `Results` property of the `inputParser` class.

## UsingDefaults

Purpose — Names of arguments not passed in function call that are given default values

`defaults = p.UsingDefaults` is a cell array of strings containing the names of those arguments that were not passed in the call to this function and consequently are set to their default values.

## Examples

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

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

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

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

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

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

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

```
p.Results.argname
```

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

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

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

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

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

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

```
The maximum height is 300.
```

```
List of all arguments:
    format: 'ppt'
```

# inputParser

---

```
maxHeight: 300
maxWidth: 500
outputDir: 'C:/matlab/test'
script: 'ipscript.m'
```

## See Also

`addRequired(inputParser)`, `addOptional(inputParser)`,  
`addParamValue(inputParser)`, `parse(inputParser)`,  
`createCopy(inputParser)`, `validateattributes`, `validatestring`,  
`varargin`, `nargchk`, `nargin`

**Purpose** Open Property Inspector

**Syntax** `inspect`  
`inspect(h)`  
`inspect([h1,h2,...])`

**Description** `inspect` creates a separate Property Inspector window to enable the display and modification of the properties of any object you select in the figure window or Layout Editor. If no object is selected, the Property Inspector is blank.

`inspect(h)` creates a Property Inspector window for the object whose handle is `h`.


`inspect([h1,h2,...])` displays properties that objects `h1` and `h2` have in common, or a blank window if there are no such properties; any number of objects can be inspected and edited in this way (for example, handles returned by the `bar` command).




The Property Inspector has the following behaviors:

- Only one Property Inspector window is active at any given time; when you inspect a new object, its properties replace those of the object last inspected.
- When the Property Inspector is open and plot edit mode is on, clicking any object in the figure window displays the properties of that object (or set of objects) in the Property Inspector.
- When you select and inspect two or more objects of different types, the Property Inspector only shows the properties that all objects have in common.
- To change the value of any property, click on the property name shown at the left side of the window, and then enter the new value in the field at the right.

The Property Inspector provides two different views:

- List view — properties are ordered alphabetically (default); this is the only view available for annotation objects.
- Group view — properties are grouped under classified headings (Handle Graphics objects only)

To view alphabetically, click the “AZ” Icon  in the Property Inspector toolbar. To see properties in groups, click

the “++” icon . When properties are grouped, the “-” and “+” icons are enabled; click  to expand all categories and click  to collapse all categories. You can also expand and collapse individual categories by clicking on the “+” next to the category name. Some properties expand and collapse

---

**Notes** To see a complete description of any property, right-click on its name or value and select **What’s This**; a help window opens that displays the reference page entry for it.

The Property Inspector displays most, but not all, properties of Handle Graphics objects. For example, the `parent` and `children` of HG objects are not shown.

`inspect h` displays a Property Inspector window that enables modification of the string 'h', not the object whose handle is h.

If you modify properties at the MATLAB command line, you must refresh the Property Inspector window to see the change reflected there. Refresh the Property Inspector by reinvoking `inspect` on the object.

---

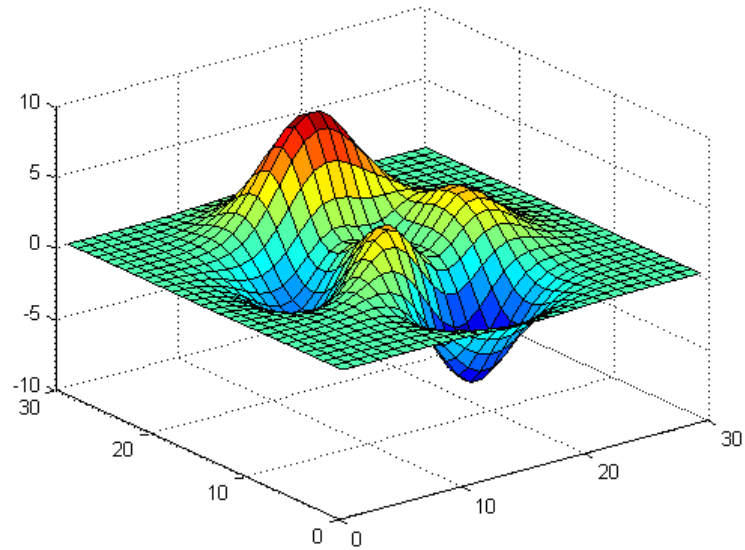
## Examples

### Example 1

Create a surface mesh plot and view its properties with the Property Inspector:

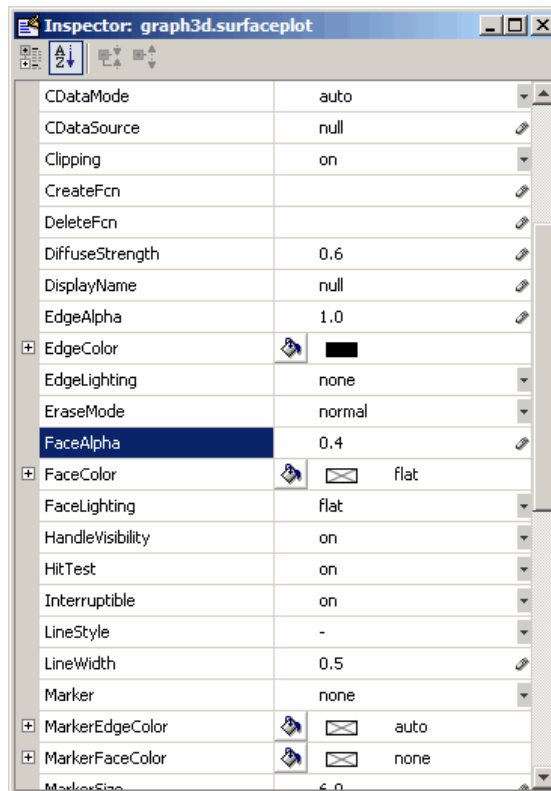
```
Z = peaks(30);  
h = surf(Z)
```

```
inspect(h)
```



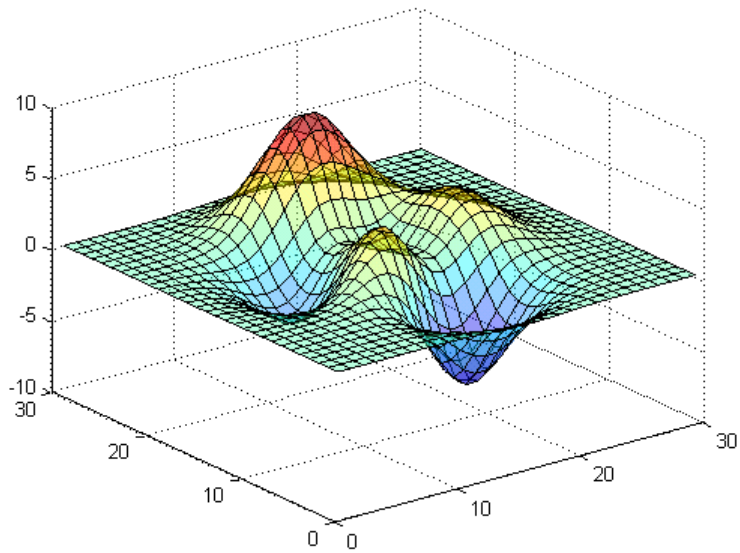
Use the Property Inspector to change the `FaceAlpha` property from 1.0 to 0.4 (equivalent to the command `set(h, 'FaceAlpha', 0.4)`). `FaceAlpha` controls the transparency of patch faces.

# inspect



When you press **Enter** or click a different field, the FaceAlpha property of the surface object is updated:

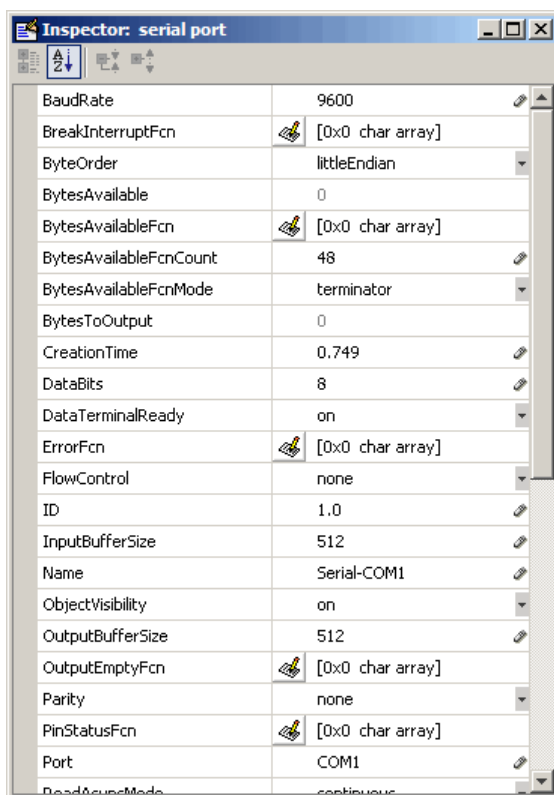




### Example 2

Create a serial port object for COM1 and use the Property Inspector to peruse its properties:

```
s = serial('COM1');  
inspect(s)
```



Because COM objects do not define property groupings, only the alphabetical list view of their properties is available.

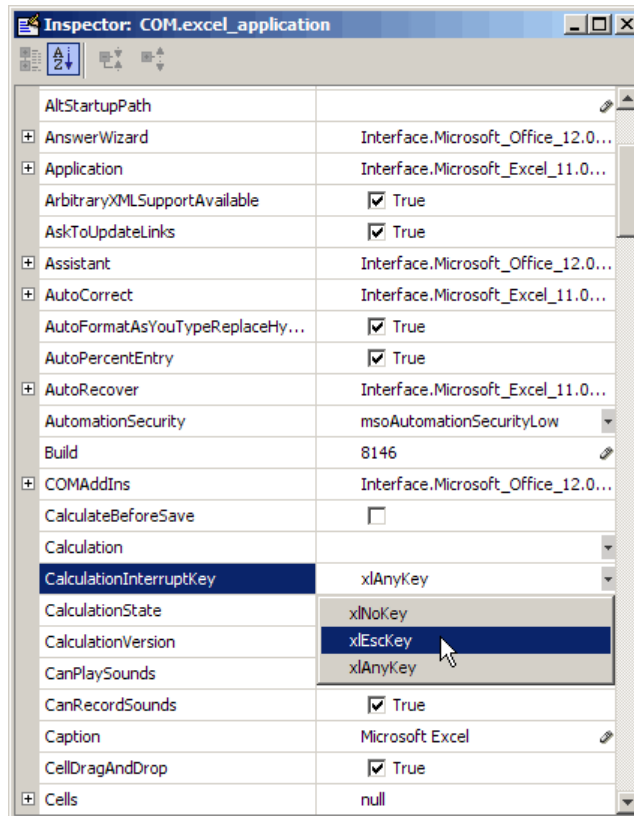
### Example 3

Create a COM Excel server and open a Property Inspector window with inspect:

```
h = actxserver('excel.application');  
inspect(h)
```

Scroll down until you see the CalculationInterruptKey property, which by default is x1AnyKey. Click on the down-arrow in the right

margin of the property inspector and select `xlEscKey` from the drop-down menu, as shown below:



Check this field in the MATLAB command window using `get` to confirm that it has changed:

```
get(h, 'CalculationInterruptKey')
```

```
ans =  
xlEscKey
```

# inspect

---

## **See Also**

get, set, isprop, guide, addproperty, deleteproperty

<b>Purpose</b>	Event information when event occurs
<b>Syntax</b>	<code>instrcallback(obj,event)</code>
<b>Description</b>	<p><code>instrcallback(obj,event)</code> displays a message that contains the event type, event, the time the event occurred, and the name of the serial port object, <code>obj</code>, that caused the event to occur.</p> <p>For error events, the error message is also displayed. For pin status events, the pin that changed value and its value are also displayed.</p>
<b>Remarks</b>	You should use <code>instrcallback</code> as a template from which you create callback functions that suit your specific application needs.
<b>Example</b>	<p>The following example creates the serial port objects <code>s</code>, and configures <code>s</code> to execute <code>instrcallback</code> when an output-empty event occurs. The event occurs after the <code>*IDN?</code> command is written to the instrument.</p>

```
s = serial('COM1');
set(s,'OutputEmptyFcn',@instrcallback)
fopen(s)
fprintf(s,'*IDN?', 'async')
```

The resulting display from `instrcallback` is shown below.

```
OutputEmpty event occurred at 08:37:49 for the object:
Serial-COM1.
```

Read the identification information from the input buffer and end the serial port session.

```
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

# instrfind

---

**Purpose** Read serial port objects from memory to MATLAB workspace

**Syntax**

```
out = instrfind
out = instrfind('PropertyName',PropertyValue,...)
out = instrfind(S)
out = instrfind(obj,'PropertyName',PropertyValue,...)
```

**Description**

`out = instrfind` returns all valid serial port objects as an array to `out`.

`out = instrfind('PropertyName',PropertyValue,...)` returns an array of serial port objects whose property names and property values match those specified.

`out = instrfind(S)` returns an array of serial port objects whose property names and property values match those defined in the structure `S`. The field names of `S` are the property names, while the field values are the associated property values.

`out = instrfind(obj,'PropertyName',PropertyValue,...)` restricts the search for matching property name/property value pairs to the serial port objects listed in `obj`.

**Remarks** Refer to “Displaying Property Names and Property Values” for a list of serial port object properties that you can use with `instrfind`.

You must specify property values using the same format as the `get` function returns. For example, if `get` returns the `Name` property value as `MyObject`, `instrfind` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of string values. For example, `instrfind` will find an object with a `Parity` property value of `Even` or `even`.

You can use property name/property value string pairs, structures, and cell array pairs in the same call to `instrfind`.

**Example** Suppose you create the following two serial port objects.

```
s1 = serial('COM1');
s2 = serial('COM2');
```

```
set(s2,'BaudRate',4800)
fopen([s1 s2])
```

You can use `instrfind` to return serial port objects based on property values.

```
out1 = instrfind('Port','COM1');
out2 = instrfind({'Port','BaudRate'},{'COM2',4800});
```

You can also use `instrfind` to return cleared serial port objects to the MATLAB workspace.

```
clear s1 s2
newobjs = instrfind
```

```
Instrument Object Array
Index:  Type:      Status:      Name:
1       serial    open        Serial-COM1
2       serial    open        Serial-COM2
```

To close both `s1` and `s2`

```
fclose(newobjs)
```

## See Also

## Functions

`clear`, `get`

# instrfindall

---

**Purpose** Find visible and hidden serial port objects

**Syntax**

```
out = instrfindall
out = instrfindall('P1',V1,...)
out = instrfindall(s)
out = instrfindall(objs,'P1',V1,...)
```

**Description** `out = instrfindall` finds all serial port objects, regardless of the value of the objects' `ObjectVisibility` property. The object or objects are returned to `out`.

`out = instrfindall('P1',V1,...)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified as arguments.

`out = instrfindall(s)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified in the structure `s`, where the field names correspond to property names and the field values correspond to the current value of the respective property.

`out = instrfindall(objs,'P1',V1,...)` restricts the search for objects with matching property name/value pairs to the serial port objects listed in `objs`.

Note that you can use string property name/property value pairs, structures, and cell array property name/property value pairs in the same call to `instrfindall`.

**Remarks** `instrfindall` differs from `instrfind` in that it finds objects whose `ObjectVisibility` property is set to `off`.

Property values are case sensitive. You must specify property values using the same format as that returned by the `get` function. For example, if `get` returns the `Name` property value as `'MyObject'`, `instrfindall` will not find an object with a `Name` property value of `'myobject'`. However, this is not the case for properties that have a finite set of string values. For example, `instrfindall` will find an object with a `Parity` property value of `'Even'` or `'even'`.



**Examples**

Suppose you create the following serial port objects:

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s2,'ObjectVisibility','off')
```

Because object `s2` has its `ObjectVisibility` set to 'off', it is not visible to commands like `instrfind`:

```
instrfind

Serial Port Object : Serial-COM1
```

However, `instrfindall` finds all objects regardless of the value of `ObjectVisibility`:

```
instrfindall

Instrument Object Array
Index:   Type:      Status:   Name:
1        serial     closed   Serial-COM1
2        serial     closed   Serial-COM2
```

The following statements use `instrfindall` to return objects with specific property settings, which are passed as cell arrays:

```
props = {'PrimaryAddress','SecondaryAddress'};
vals = {2,0};
obj = instrfindall(props,vals);
```

You can use `instrfindall` as an argument when you want to apply the command to all objects, visible and invisible. For example, the following statement makes all objects visible:

```
set(instrfindall,'ObjectVisibility','on')
```

**See Also****Functions**

`get`, `instrfind`

## Properties

ObjectVisibility

**Purpose** Convert integer to string

**Syntax** `str = int2str(N)`

**Description** `str = int2str(N)` converts an integer to a string with integer format. The input `N` can be a single integer or a vector or matrix of integers. Noninteger inputs are rounded before conversion.

**Examples** `int2str(2+3)` is the string '5'.

One way to label a plot is

```
title(['case number ' int2str(n)])
```

For matrix or vector inputs, `int2str` returns a string matrix:

```
int2str(eye(3))
```

```
ans =
```

```
1 0 0
0 1 0
0 0 1
```

**See Also** `fprintf`, `num2str`, `sprintf`

# int8, int16, int32, int64

**Purpose** Convert to signed integer

**Syntax**

```
I = int8(X)
I = int16(X)
I = int32(X)
I = int64(X)
```

**Description** `I = int*(X)` converts the elements of array `X` into signed integers. `X` can be any numeric object (such as a double). The results of an `int*` operation are shown in the next table.

Operation	Output Range	Output Type	Bytes per Element	Output Class
int8	-128 to 127	Signed 8-bit integer	1	int8
int16	-32,768 to 32,767	Signed 16-bit integer	2	int16
int32	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	4	int32
int64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	8	int64

double and single values are rounded to the nearest `int*` value on conversion. A value of `X` that is above or below the range for an integer class is mapped to one of the endpoints of the range. For example,

```
int16(40000)
ans =
    32767
```

If  $X$  is already a signed integer of the same class, then `int*` has no effect.

You can define or overload your own methods for `int*` (as you can for any object) by placing the appropriately named method in an `@int*` directory within a directory on your path. Type `help datatypes` for the names of the methods you can overload.

## Remarks

Most operations that manipulate arrays without changing their elements are defined for integer values. Examples are `reshape`, `size`, the logical and relational operators, subscripted assignment, and subscripted reference.

Some arithmetic operations are defined for integer arrays on interaction with other integer arrays of the same class (e.g., where both operands are `int16`). Examples of these operations are `+`, `-`, `.*`, `./`, `.\` and `.^`. If at least one operand is scalar, then `*`, `/`, `\`, and `^` are also defined. Integer arrays may also interact with scalar `double` variables, including constants, and the result of the operation is an integer array of the same class. Integer arrays saturate on overflow in arithmetic.

---

**Note** Only the lower order integer data types support math operations. Math operations are not supported for `int64` and `uint64`.

---

A particularly efficient way to initialize a large array is by specifying the data type (i.e., class name) for the array in the `zeros`, `ones`, or `eye` function. For example, to create a 100-by-100 `int64` array initialized to zero, type

```
I = zeros(100, 100, 'int64');
```

An easy way to find the range for any MATLAB integer type is to use the `intmin` and `intmax` functions as shown here for `int32`:

```
intmin('int32')          intmax('int32')
ans =                    ans =
    -2147483648          2147483647
```

## **int8, int16, int32, int64**

---

### **See Also**

double, single, uint8, uint16, uint32, uint64, intmax, intmin

**Purpose** List custom interfaces to COM server

**Syntax**  
`C = h.interfaces`  
`C = interfaces(h)`

**Description** `C = h.interfaces` returns cell array of strings `C` listing all custom interfaces implemented by the component in a specific COM server. The server is designated by input argument, `h`, which is the handle returned by the `actxcontrol` or `actxserver` function when creating that server. `C = interfaces(h)` is an alternate syntax.

---

**Note** `interfaces` only lists the custom interfaces; it does not return any interfaces. Use the `invoke` function to return a handle to a specific custom interface.

---

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

**Examples** Once you have created a COM server, you can query the server component to see if any custom interfaces are implemented. Use the `interfaces` function to return a list of all available custom interfaces:

```
h = actxserver('mytestenv.calculator')
h =
    COM.mytestenv.calculator

customlist = h.interfaces
customlist =
    ICalc1
    ICalc2
    ICalc3
```

To get a handle to the custom interface you want, use the `invoke` function, specifying the handle returned by `actxcontrol` or `actxserver` and also the name of the custom interface:

# interfaces

---

```
c1 = h.invoke('ICalc1')
c1 =
    Interface.Calc_1.0_Type_Library.ICalc_Interface
```

You can now use this handle with most of the COM client functions to access the properties and methods of the object through the selected custom interface. For example, to list the properties available through the ICalc1 interface, use

```
c1.get
    background: 'Blue'
    height: 10
    width: 0
```

To list the methods, use

```
c1.invoke
    Add = double Add(handle, double, double)
    Divide = double Divide(handle, double, double)
    Multiply = double Multiply(handle, double, double)
    Subtract = double Subtract(handle, double, double)
```

Add and multiply numbers using the Add and Multiply methods of the custom object c1:

```
sum = c1.Add(4, 7)
sum =
    11

prod = c1.Multiply(4, 7)
prod =
    28
```

## See Also

actxcontrol, actxserver, invoke, get (COM)



**Purpose** 1-D data interpolation (table lookup)

**Syntax**

```
yi = interp1(x,Y,xi)
yi = interp1(Y,xi)
yi = interp1(x,Y,xi,method)
yi = interp1(x,Y,xi,method,'extrap')
yi = interp1(x,Y,xi,method,extrapval)
pp = interp1(x,Y,method,'pp')
```

**Description** `yi = interp1(x,Y,xi)` interpolates to find `yi`, the values of the underlying function `Y` at the points in the vector or array `xi`. `x` must be a vector. `Y` can be a scalar, a vector, or an array of any dimension, subject to the following conditions:

- If `Y` is a vector, it must have the same length as `x`. A scalar value for `Y` is expanded to have the same length as `x`. `xi` can be a scalar, a vector, or a multidimensional array, and `yi` has the same size as `xi`.
- If `Y` is an array that is not a vector, the size of `Y` must have the form `[n,d1,d2,...,dk]`, where `n` is the length of `x`. The interpolation is performed for each `d1-by-d2-by-...-dk` value in `Y`. The sizes of `xi` and `yi` are related as follows:
  - If `xi` is a scalar or vector, `size(yi)` equals `[length(xi), d1, d2, ..., dk]`.
  - If `xi` is an array of size `[m1,m2,...,mj]`, `yi` has size `[m1,m2,...,mj,d1,d2,...,dk]`.

`yi = interp1(Y,xi)` assumes that `x = 1:N`, where `N` is the length of `Y` for vector `Y`, or `size(Y,1)` for matrix `Y`.

`yi = interp1(x,Y,xi,method)` interpolates using alternative methods:

'nearest'	Nearest neighbor interpolation
'linear'	Linear interpolation (default)

# interp1

---

'spline'	Cubic spline interpolation
'pchip'	Piecewise cubic Hermite interpolation
'cubic'	(Same as 'pchip')
'v5cubic'	Cubic interpolation used in MATLAB 5. This method does not extrapolate. Also, if x is not equally spaced, 'spline' is used/

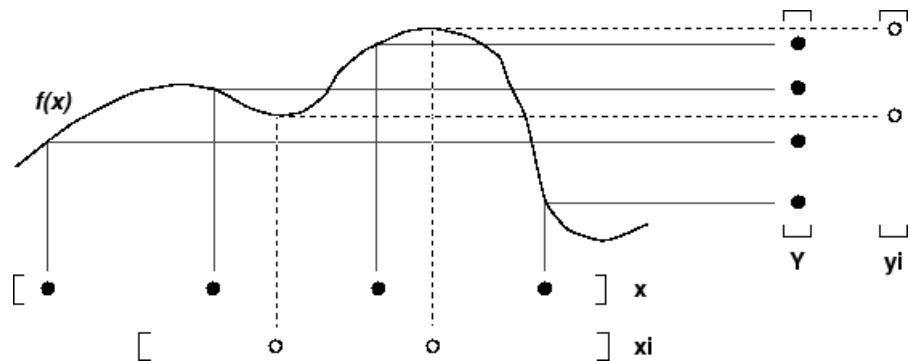
For the 'nearest', 'linear', and 'v5cubic' methods, `interp1(x,Y,xi,method)` returns NaN for any element of `xi` that is outside the interval spanned by `x`. For all other methods, `interp1` performs extrapolation for out of range values.

`yi = interp1(x,Y,xi,method,'extrap')` uses the specified method to perform extrapolation for out of range values.

`yi = interp1(x,Y,xi,method,extrapval)` returns the scalar `extrapval` for out of range values. NaN and 0 are often used for `extrapval`.

`pp = interp1(x,Y,method,'pp')` uses the specified method to generate the piecewise polynomial form (ppform) of `Y`. You can use any of the methods in the preceding table, except for 'v5cubic'. `pp` can then be evaluated via `ppval`. `ppval(pp,xi)` is the same as `interp1(x,Y,xi,method,'extrap')`.

The `interp1` command interpolates between data points. It finds values at intermediate points, of a one-dimensional function  $f(x)$  that underlies the data. This function is shown below, along with the relationship between vectors `x`, `Y`, `xi`, and `yi`.



Interpolation is the same operation as *table lookup*. Described in table lookup terms, the *table* is  $[x, Y]$  and `interp1` *looks up* the elements of  $xi$  in  $x$ , and, based upon their locations, returns values  $yi$  interpolated within the elements of  $Y$ .

---

**Note** `interp1q` is quicker than `interp1` on non-uniformly spaced data because it does no input checking. For `interp1q` to work properly,  $x$  must be a monotonically increasing column vector and  $Y$  must be a column vector or matrix with `length(X)` rows. Type `help interp1q` at the command line for more information.

---

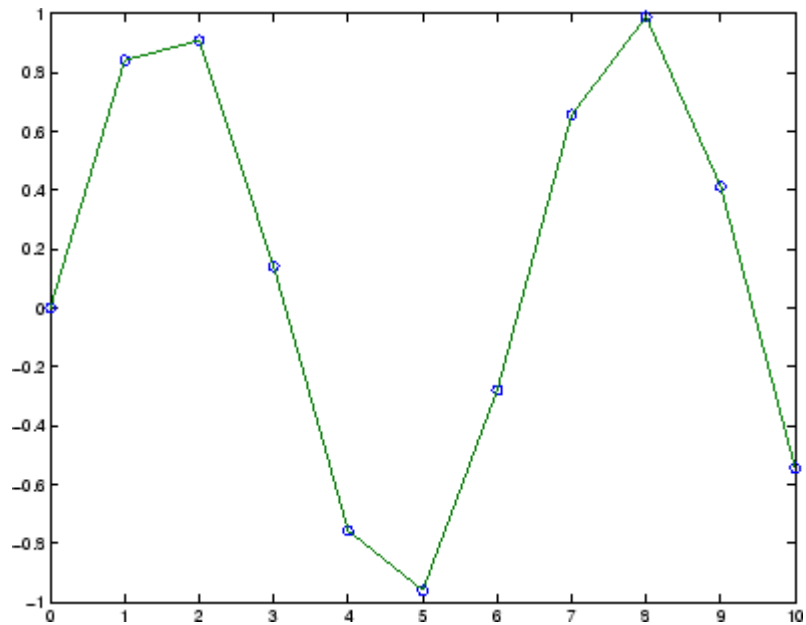
## Examples

### Example 1

Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = 0:10;
y = sin(x);
xi = 0:.25:10;
yi = interp1(x,y,xi);
plot(x,y,'o',xi,yi)
```

# interp1



## Example 2

The following multidimensional example creates 2-by-2 matrices of interpolated function values, one matrix for each of the three functions  $x^2$ ,  $x^3$ , and  $x^4$ .

```
x = [1:10]'; y = [ x.^2, x.^3, x.^4 ];  
xi = [1.5, 1.75; 7.5, 7.75];  
yi = interp1(x,y,xi);
```

The result `yi` has size 2-by-2-by-3.

```
size(yi)
```

```
ans =
```

```
2 2 3
```

### Example 3

Here are two vectors representing the census years from 1900 to 1990 and the corresponding United States population in millions of people.

```
t = 1900:10:1990;  
p = [75.995 91.972 105.711 123.203 131.669...  
     150.697 179.323 203.212 226.505 249.633];
```

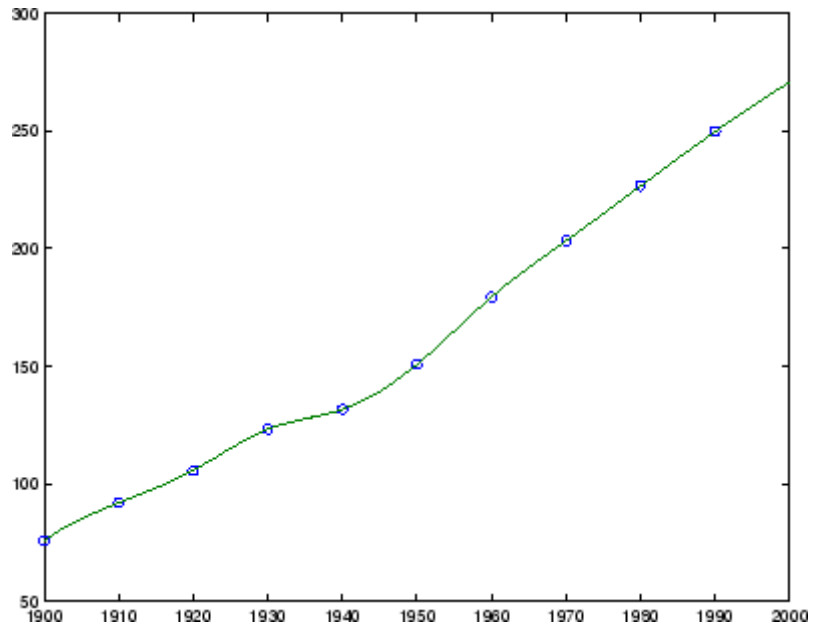
The expression `interp1(t,p,1975)` interpolates within the census data to estimate the population in 1975. The result is

```
ans =  
    214.8585
```

Now interpolate within the data at every year from 1900 to 2000, and plot the result.

```
x = 1900:1:2000;  
y = interp1(t,p,x,'spline');  
plot(t,p,'o',x,y)
```

# interp1



Sometimes it is more convenient to think of interpolation in table lookup terms, where the data are stored in a single table. If a portion of the census data is stored in a single 5-by-2 table,

```
tab =  
    1950    150.697  
    1960    179.323  
    1970    203.212  
    1980    226.505  
    1990    249.633
```

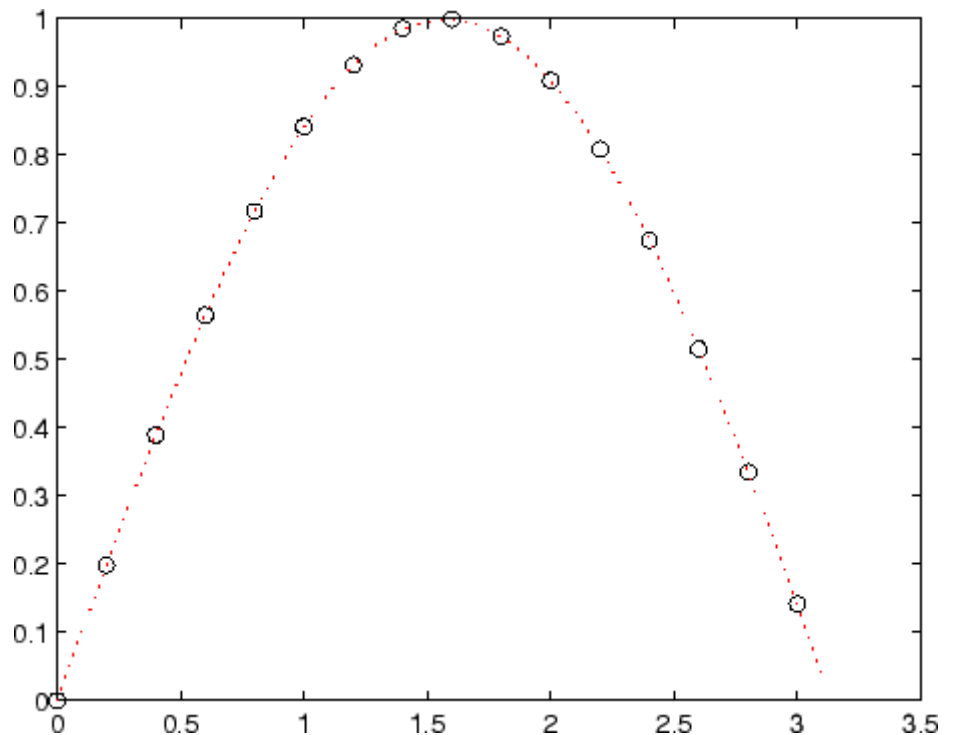
then the population in 1975, obtained by table lookup within the matrix `tab`, is

```
p = interp1(tab(:,1),tab(:,2),1975)  
p =  
    214.8585
```

**Example 4**

The following example uses the 'cubic' method to generate the piecewise polynomial form (ppform) of  $Y$ , and then evaluates the result using `ppval`.

```
x = 0:.2:pi; y = sin(x);  
pp = interp1(x,y,'cubic','pp');  
xi = 0:.1:pi;  
yi = ppval(pp,xi);  
plot(x,y,'ko'), hold on, plot(xi,yi,'r:'), hold off
```

**Algorithm**

The `interp1` command is a MATLAB M-file. The 'nearest' and 'linear' methods have straightforward implementations.

For the 'spline' method, `interp1` calls a function `spline` that uses the functions `ppval`, `mkpp`, and `unmkpp`. These routines form a small suite of functions for working with piecewise polynomials. `spline` uses them to perform the cubic spline interpolation. For access to more advanced features, see the `spline` reference page, the M-file help for these functions, and the Spline Toolbox™.

For the 'pchip' and 'cubic' methods, `interp1` calls a function `pchip` that performs piecewise cubic interpolation within the vectors `x` and `y`. This method preserves monotonicity and the shape of the data. See the `pchip` reference page for more information.

## Interpolating Complex Data

**For Real `x` and Complex `Y`.** For `interp1(x,Y,...)` where `x` is real and `Y` is complex, you can use any `interp1` method except for 'pchip'. The shape-preserving aspect of the 'pchip' algorithm involves the signs of the slopes between the data points. Because there is no notion of sign with complex data, it is impossible to talk about whether a function is increasing or decreasing. Consequently, the 'pchip' algorithm does not generalize to complex data.

The 'spline' method is often a good choice because piecewise cubic splines are derived purely from smoothness conditions. The second derivative of the interpolant must be continuous across the interpolating points. This does not involve any notion of sign or shape and so generalizes to complex data.

**For Complex `x`.** For `interp1(x,Y,...)` where `x` is complex and `Y` is either real or complex, use the two-dimensional interpolation routine `interp2(REAL(x),IMAG(x),Y,...)` instead.

## See Also

`interp1q`, `interpft`, `interp2`, `interp3`, `interpN`, `pchip`, `spline`

## References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.



**Purpose** Quick 1-D linear interpolation

**Syntax** `yi = interp1q(x,Y,xi)`

**Description** `yi = interp1q(x,Y,xi)` returns the value of the 1-D function `Y` at the points of column vector `xi` using linear interpolation. The vector `x` specifies the coordinates of the underlying interval. The length of output `yi` is equal to the length of `xi`.

`interp1q` is quicker than `interp1` on non-uniformly spaced data because it does no input checking.

For `interp1q` to work properly,

- `x` must be a monotonically increasing column vector.
- `Y` must be a column vector or matrix with `length(x)` rows.
- `xi` must be a column vector

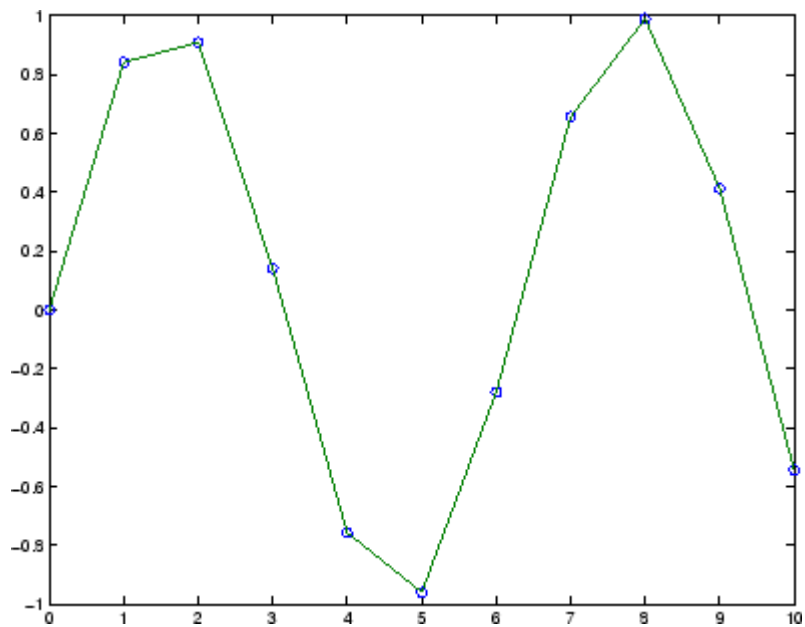
`interp1q` returns NaN for any values of `xi` that lie outside the coordinates in `x`. If `Y` is a matrix, then the interpolation is performed for each column of `Y`, in which case `yi` is `length(xi)-by-size(Y,2)`.

**Example** Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = (0:10)';  
y = sin(x);  
xi = (0:.25:10)';  
yi = interp1q(x,y,xi);  
plot(x,y,'o',xi,yi)
```

# interp1q

---



## See Also

interp1, interp2, interp3, interpn

**Purpose** 2-D data interpolation (table lookup)

**Syntax**

```

ZI = interp2(X,Y,Z,XI,YI)
ZI = interp2(Z,XI,YI)
ZI = interp2(Z,ntimes)
ZI = interp2(X,Y,Z,XI,YI,method)
ZI = interp2(...,method, extrapval)

```

**Description** `ZI = interp2(X,Y,Z,XI,YI)` returns matrix `ZI` containing elements corresponding to the elements of `XI` and `YI` and determined by interpolation within the two-dimensional function specified by matrices `X`, `Y`, and `Z`. `X` and `Y` must be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. Matrices `X` and `Y` specify the points at which the data `Z` is given. Out of range values are returned as NaNs.

`XI` and `YI` can be matrices, in which case `interp2` returns the values of `Z` corresponding to the points  $(XI(i, j), YI(i, j))$ . Alternatively, you can pass in the row and column vectors `xi` and `yi`, respectively. In this case, `interp2` interprets these vectors as if you issued the command `meshgrid(xi,yi)`.

`ZI = interp2(Z,XI,YI)` assumes that `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`.

`ZI = interp2(Z,ntimes)` expands `Z` by interleaving interpolates between every element, working recursively for `ntimes`. `interp2(Z)` is the same as `interp2(Z,1)`.

`ZI = interp2(X,Y,Z,XI,YI,method)` specifies an alternative interpolation method:

'nearest'	Nearest neighbor interpolation
'linear'	Linear interpolation (default)

# interp2

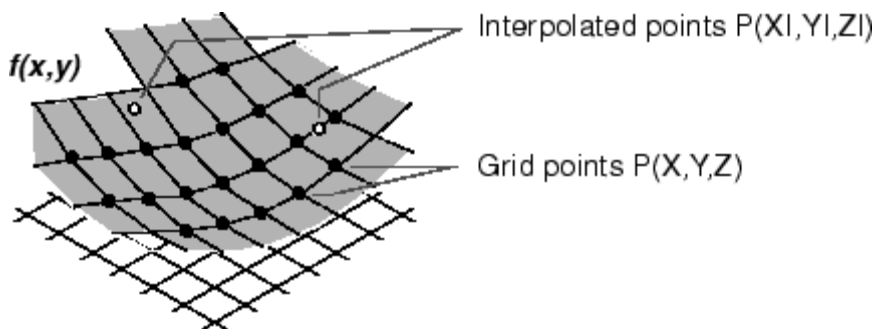
'spline'	Cubic spline interpolation
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. Otherwise, this method is the same as 'spline'.

All interpolation methods require that X and Y be monotonic, and have the same format (“plaid”) as if they were produced by meshgrid. If you provide two monotonic vectors, interp2 changes them to a plaid internally. Variable spacing is handled by mapping the given values in X, Y, XI, and YI to an equally spaced domain before interpolating. For faster interpolation when X and Y are equally spaced and monotonic, use the methods '\*linear', '\*cubic', '\*spline', or '\*nearest'.

ZI = interp2(...,method, extrapval) specifies a method and a scalar value for ZI outside of the domain created by X and Y. Thus, ZI equals extrapval for any value of YI or XI that is not spanned by Y or X respectively. A method must be specified to use extrapval. The default method is 'linear'.

## Remarks

The interp2 command interpolates between data points. It finds values of a two-dimensional function  $f(x, y)$  underlying the data at intermediate points.



Interpolation is the same operation as table lookup. Described in table lookup terms, the table is `tab = [NaN, Y; X, Z]` and interp2 looks up

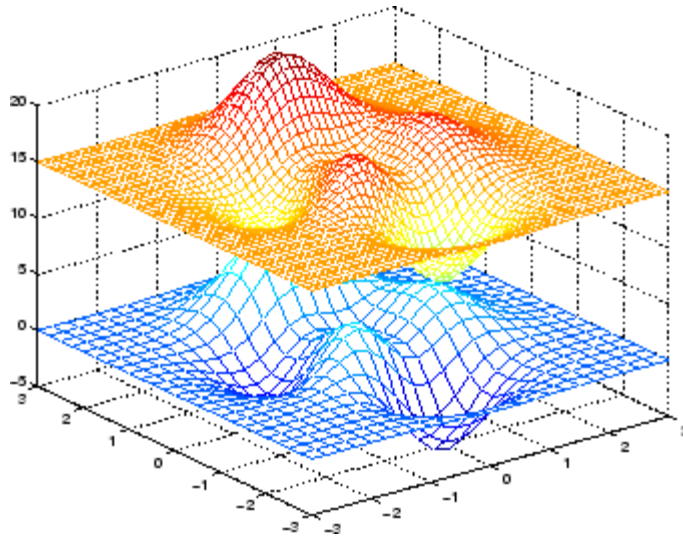
the elements of XI in X, YI in Y, and, based upon their location, returns values ZI interpolated within the elements of Z.

## Examples

### Example 1

Interpolate the `peaks` function over a finer grid.

```
[X,Y] = meshgrid(-3:.25:3);
Z = peaks(X,Y);
[XI,YI] = meshgrid(-3:.125:3);
ZI = interp2(X,Y,Z,XI,YI);
mesh(X,Y,Z), hold, mesh(XI,YI,ZI+15)
hold off
axis([-3 3 -3 3 -5 20])
```



### Example 2

Given this set of employee data,

```
years = 1950:10:1990;
service = 10:10:30;
```

## interp2

---

```
wage = [150.697 199.592 187.625  
179.323 195.072 250.287  
203.212 179.092 322.767  
226.505 153.706 426.730  
249.633 120.281 598.243];
```

it is possible to interpolate to find the wage earned in 1975 by an employee with 15 years' service:

```
w = interp2(service,years,wage,15,1975)  
w =  
190.6287
```

### See Also

griddata, interp1, interp1q, interp3, interpn, meshgrid

**Purpose** 3-D data interpolation (table lookup)

**Syntax**

```

VI = interp3(X,Y,Z,V,XI,YI,ZI)
VI = interp3(V,XI,YI,ZI)
VI = interp3(V,ntimes)
VI = interp3(...,method)
VI = interp3(...,method,extrapval)

```

**Description** `VI = interp3(X,Y,Z,V,XI,YI,ZI)` interpolates to find `VI`, the values of the underlying three-dimensional function `V` at the points in arrays `XI`, `YI` and `ZI`. `XI`, `YI`, `ZI` must be arrays of the same size, or vectors. Vector arguments that are not the same size, and have mixed orientations (i.e. with both row and column vectors) are passed through `meshgrid` to create the `Y1`, `Y2`, `Y3` arrays. Arrays `X`, `Y`, and `Z` specify the points at which the data `V` is given. Out of range values are returned as `NaN`.

`VI = interp3(V,XI,YI,ZI)` assumes `X=1:N`, `Y=1:M`, `Z=1:P` where `[M,N,P]=size(V)`.

`VI = interp3(V,ntimes)` expands `V` by interleaving interpolates between every element, working recursively for `ntimes` iterations. The command `interp3(V)` is the same as `interp3(V,1)`.

`VI = interp3(...,method)` specifies alternative methods:

'nearest'	Nearest neighbor interpolation
'linear'	Linear interpolation (default)
'spline'	Cubic spline interpolation
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. Otherwise, this method is the same as 'spline'.

`VI = interp3(...,method,extrapval)` specifies a method and a value for `VI` outside of the domain created by `X`, `Y` and `Z`. Thus, `VI` equals `extrapval` for any value of `XI`, `YI` or `ZI` that is not spanned by `X`, `Y`, and `Z`, respectively. You must specify a method to use `extrapval`. The default method is 'linear'.

# interp3

---

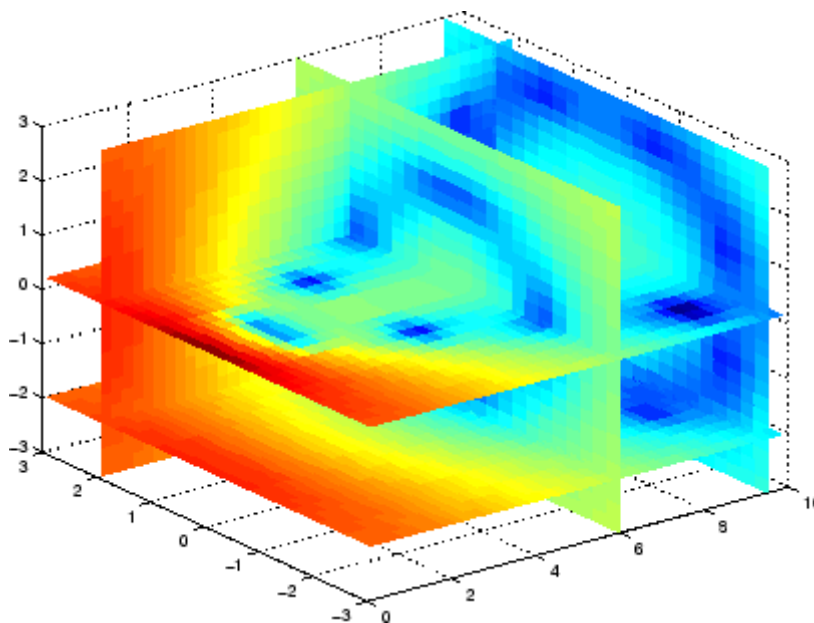
## Discussion

All the interpolation methods require that X,Y and Z be monotonic and have the same format (“plaid”) as if they were created using meshgrid. X, Y, and Z can be non-uniformly spaced. For faster interpolation when X, Y, and Z are equally spaced and monotonic, use the methods `*linear`, `*cubic`, or `*nearest`.

## Examples

To generate a coarse approximation of flow and interpolate over a finer mesh:

```
[x,y,z,v] = flow(10);  
[xi,yi,zi] = meshgrid(.1:.25:10, -3:.25:3, -3:.25:3);  
vi = interp3(x,y,z,v,xi,yi,zi); % vi is 25-by-40-by-25  
slice(xi,yi,zi,vi,[6 9.5],2,[-2 .2]), shading flat
```



## See Also

`interp1`, `interp1q`, `interp2`, `interp3`, `meshgrid`



<b>Purpose</b>	1-D interpolation using FFT method
<b>Syntax</b>	<pre>y = interpft(x,n) y = interpft(x,n,dim)</pre>
<b>Description</b>	<p><code>y = interpft(x,n)</code> returns the vector <code>y</code> that contains the value of the periodic function <code>x</code> resampled to <code>n</code> equally spaced points.</p> <p>If <code>length(x) = m</code>, and <code>x</code> has sample interval <code>dx</code>, then the new sample interval for <code>y</code> is <code>dy = dx*m/n</code>. Note that <code>n</code> cannot be smaller than <code>m</code>.</p> <p>If <code>X</code> is a matrix, <code>interpft</code> operates on the columns of <code>X</code>, returning a matrix <code>Y</code> with the same number of columns as <code>X</code>, but with <code>n</code> rows.</p> <p><code>y = interpft(x,n,dim)</code> operates along the specified dimension.</p>
<b>Algorithm</b>	The <code>interpft</code> command uses the FFT method. The original vector <code>x</code> is transformed to the Fourier domain using <code>fft</code> and then transformed back with more points.
<b>Examples</b>	<p>Interpolate a triangle-like signal using an interpolation factor of 5. First, set up signal to be interpolated:</p> <pre>y = [0 .5 1 1.5 2 1.5 1 .5 0 -.5 -1 -1.5 -2 -1.5 -1 -.5 0]; N = length(y);</pre> <p>Perform the interpolation:</p> <pre>L = 5; M = N*L; x = 0:L:L*N-1; xi = 0:M-1; yi = interpft(y,M); plot(x,y,'o',xi,yi,'*') legend('Original data','Interpolated data')</pre>
<b>See Also</b>	<code>interp1</code>

# interp

---

**Purpose** N-D data interpolation (table lookup)

**Syntax**

```
VI = interp(X1,X2,X3,...,V,Y1,Y2,Y3,...)
VI = interp(V,Y1,Y2,Y3,...)
VI = interp(V,ntimes)
VI = interp(...,method)
VI = interp(...,method,extrapval)
```

**Description** `VI = interp(X1,X2,X3,...,V,Y1,Y2,Y3,...)` interpolates to find `VI`, the values of the underlying multidimensional function `V` at the points in the arrays `Y1`, `Y2`, `Y3`, etc. For an `n`-dimensional array `V`, `interp` is called with `2*N+1` arguments. Arrays `X1`, `X2`, `X3`, etc. specify the points at which the data `V` is given. Out of range values are returned as NaNs. `Y1`, `Y2`, `Y3`, etc. must be arrays of the same size, or vectors. Vector arguments that are not the same size, and have mixed orientations (i.e. with both row and column vectors) are passed through `ndgrid` to create the `Y1`, `Y2`, `Y3`, etc. arrays. `interp` works for all `n`-dimensional arrays with 2 or more dimensions.

`VI = interp(V,Y1,Y2,Y3,...)` interpolates as above, assuming `X1 = 1:size(V,1)`, `X2 = 1:size(V,2)`, `X3 = 1:size(V,3)`, etc.

`VI = interp(V,ntimes)` expands `V` by interleaving interpolates between each element, working recursively for `ntimes` iterations. `interp(V)` is the same as `interp(V,1)`.

`VI = interp(...,method)` specifies alternative methods:

'nearest'	Nearest neighbor interpolation
'linear'	Linear interpolation (default)
'spline'	Cubic spline interpolation
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. Otherwise, this method is the same as 'spline'.

`VI = interp(...,method,extrapval)` specifies a method and a value for `VI` outside of the domain created by `X1`, `X2`, ... Thus, `VI` equals

`extrapval` for any value of  $Y_1, Y_2, \dots$  that is not spanned by  $X_1, X_2, \dots$  respectively. You must specify a method to use `extrapval`. The default method is `'linear'`.

`interp` requires that  $X_1, X_2, X_3, \dots$  be monotonic and plaid (as if they were created using `ndgrid`).  $X_1, X_2, X_3$ , and so on can be non-uniformly spaced.

## Discussion

All the interpolation methods require that  $X_1, X_2, X_3 \dots$  be monotonic and have the same format ("plaid") as if they were created using `ndgrid`.  $X_1, X_2, X_3, \dots$  and  $Y_1, Y_2, Y_3, \dots$  can be non-uniformly spaced. For faster interpolation when  $X_1, X_2, X_3, \dots$  are equally spaced and monotonic, use the methods `'*linear'`, `'*cubic'`, or `'*nearest'`.

## Examples

Start by defining an anonymous function to compute  $f = te^{-x^2 - y^2 - z^2}$ :

```
f = @(x,y,z,t) t.*exp(-x.^2 - y.^2 - z.^2);
```

Build the lookup table by evaluating the function `f` on a grid constructed by `ndgrid`:

```
[x,y,z,t] = ndgrid(-1:0.2:1,-1:0.2:1,-1:0.2:1,0:2:10);
v = f(x,y,z,t);
```

Now construct a finer grid:

```
[xi,yi,zi,ti] = ndgrid(-1:0.05:1,-1:0.08:1,-1:0.05:1, ...
                        0:0.5:10);
```

Compute the spline interpolation at `xi, yi, zi`, and `ti`:

```
vi = interp(x,y,z,t,v,xi,yi,zi,ti,'spline');
```

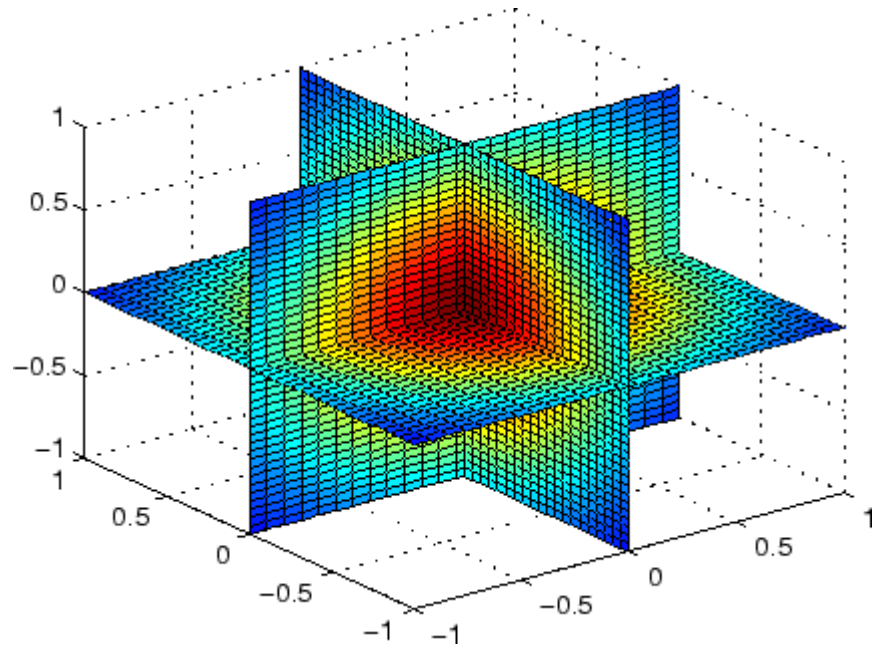
Plot the interpolated function, and then create a movie from the plot:

```
nframes = size(ti, 4);
for j = 1:nframes
    slice(yi(:,:,j), xi(:,:,j), zi(:,:,j), ...
```

# interp

---

```
        vi(:,:,:,j),0,0,0);  
    caxis([0 10]);  
    M(j) = getframe;  
end  
movie(M);
```



## See Also

interp1, interp2, interp3, ndgrid

**Purpose** Interpolate stream-line vertices from flow speed

**Syntax**

```
interpstreamspeed(X,Y,Z,U,V,W,vertices)
interpstreamspeed(U,V,W,vertices)
interpstreamspeed(X,Y,Z,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(X,Y,U,V,vertices)
interpstreamspeed(U,V,vertices)
interpstreamspeed(X,Y,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(...,sf)
vertsout = interpstreamspeed(...)
```

**Description** `interpstreamspeed(X,Y,Z,U,V,W,vertices)` interpolates streamline vertices based on the magnitude of the vector data U, V, W. The arrays X, Y, Z define the coordinates for U, V, W and must be monotonic and 3-D plaid (as if produced by `meshgrid`).

`interpstreamspeed(U,V,W,vertices)` assumes X, Y, and Z are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(U)`.

`interpstreamspeed(X,Y,Z,speed,vertices)` uses the 3-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes X, Y, and Z are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p]=size(speed)`.

`interpstreamspeed(X,Y,U,V,vertices)` interpolates streamline vertices based on the magnitude of the vector data U, V. The arrays X, Y define the coordinates for U, V and must be monotonic and 2-D plaid (as if produced by `meshgrid`).

# interpstreamspeed

---

`interpstreamspeed(U,V,vertices)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[M N]=size(U)`.

`interpstreamspeed(X,Y,speed,vertices)` uses the 2-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[M,N]= size(speed)`.

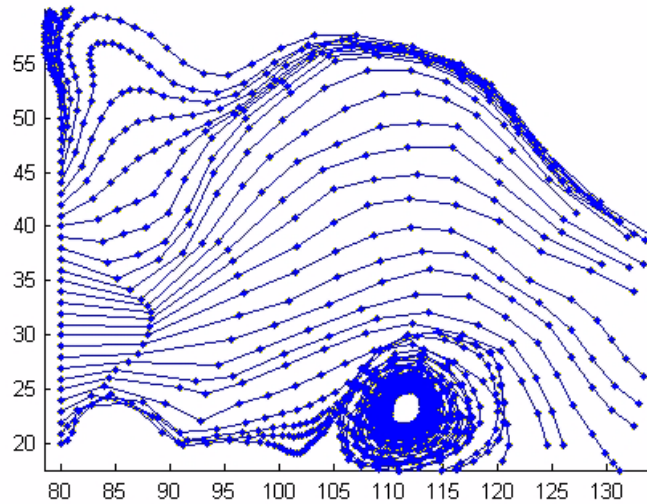
`interpstreamspeed(...,sf)` uses `sf` to scale the magnitude of the vector data and therefore controls the number of interpolated vertices. For example, if `sf` is 3, then `interpstreamspeed` creates only one-third of the vertices.

`vertsout = interpstreamspeed(...)` returns a cell array of vertex arrays.

## Examples

This example draws streamlines using the vertices returned by `interpstreamspeed`. Dot markers indicate the location of each vertex. This example enables you to visualize the relative speeds of the flow data. Streamlines having widely spaced vertices indicate faster flow; those with closely spaced vertices indicate slower flow.

```
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.2);
sl = streamline(iverts);
set(sl,'Marker','.')
axis tight; view(2); daspect([1 1 1])
```

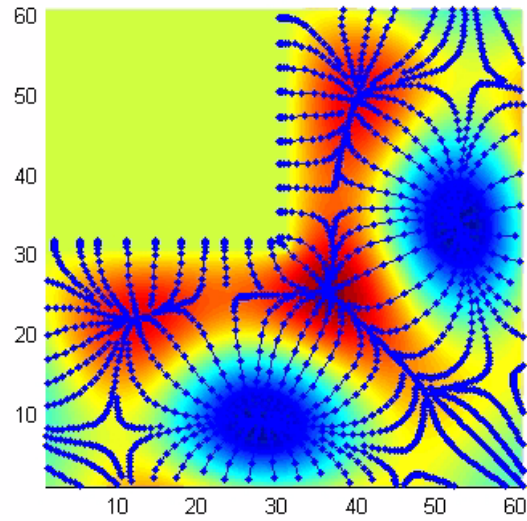


This example plots streamlines whose vertex spacing indicates the value of the gradient along the streamline.

```
z = membrane(6,30);  
[u v] = gradient(z);  
[verts averts] = streamslice(u,v);  
iverts = interpstreamspeed(u,v,verts,15);  
sl = streamline(iverts);  
set(sl,'Marker','.')  
hold on; pcolor(z); shading interp  
axis tight; view(2); daspect([1 1 1])
```

# interpstreamspeed

---



## See Also

`stream2`, `stream3`, `streamline`, `streamslice`, `streamparticles`  
“Volume Visualization” on page 1-106 for related functions



**Purpose**

Find set intersection of two vectors

**Syntax**

```
c = intersect(A, B)
c = intersect(A, B, 'rows')
[c, ia, ib] = intersect(a, b)
```

**Description**

`c = intersect(A, B)` returns the values common to both A and B. In set theoretic terms, this is `A[[INTERSECT]] B`. Inputs A and B can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted in ascending order.

`c = intersect(A, B, 'rows')` when A and B are matrices with the same number of columns returns the rows common to both A and B. `MATLAB ignores the rows flag for all cell arrays.

`[c, ia, ib] = intersect(a, b)` also returns column index vectors `ia` and `ib` such that `c = a(ia)` and `c = b(ib)` (or `c = a(ia,:)` and `c = b(ib,:)`).

**Remarks**

Because NaN is considered to be not equal to itself, it is never included in the result `c`.

**Examples**

```
A = [1 2 3 6]; B = [1 2 3 4 6 10 20];
[c, ia, ib] = intersect(A, B);
disp([c; ia; ib])
     1     2     3     6
     1     2     3     4
     1     2     3     5
```

**See Also**

`ismember`, `issorted`, `setdiff`, `setxor`, `union`, `unique`

# intmax

---

**Purpose** Largest value of specified integer type

**Syntax**  
`v = intmax`  
`v = intmax('classname')`

**Description** `v = intmax` is the largest positive value that can be represented in the MATLAB software with a 32-bit integer. Any value larger than the value returned by `intmax` saturates to the `intmax` value when cast to a 32-bit integer.

`v = intmax('classname')` is the largest positive value in the integer class `classname`. Valid values for the string `classname` are

'int8'	'int16'	'int32'	'int64'
'uint8'	'uint16'	'uint32'	'uint64'

`intmax('int32')` is the same as `intmax` with no arguments.

## Examples

Find the maximum value for a 64-bit signed integer:

```
v = intmax('int64')
v =
    9223372036854775807
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
    2147483647
```

Compare the result with the default value returned by `intmax`:

```
isequal(x, intmax)
ans =
     1
```

## See Also

`intmin`, `realmax`, `realmin`, `int8`, `uint8`, `isa`, `class`

**Purpose** Smallest value of specified integer type

**Syntax**

```
v = intmin
v = intmin('classname')
```

**Description** `v = intmin` is the smallest value that can be represented in the MATLAB software with a 32-bit integer. Any value smaller than the value returned by `intmin` saturates to the `intmin` value when cast to a 32-bit integer.

`v = intmin('classname')` is the smallest positive value in the integer class `classname`. Valid values for the string `classname` are

'int8'	'int16'	'int32'	'int64'
'uint8'	'uint16'	'uint32'	'uint64'

`intmin('int32')` is the same as `intmin` with no arguments.

## Examples

Find the minimum value for a 64-bit signed integer:

```
v = intmin('int64')
v =
-9223372036854775808
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
2147483647
```

Compare the result with the default value returned by `intmin`:

```
isequal(x, intmin)
ans =
1
```

## See Also

`intmax`, `realmin`, `realmax`, `int8`, `uint8`, `isa`, `class`

# intwarning

---

**Purpose** Control state of integer warnings

**Syntax**

```
intwarning('action')
s = intwarning('action')
intwarning(s)
sOld = intwarning(sNew)
```

**Description** The MATLAB software offers four types of warnings on invalid operations that involve integers. The `intwarning` function enables, disables, or returns information on these warnings:

- `MATLAB:intConvertNaN` — Warning on an attempt to convert NaN (Not a Number) to an integer. The result of the operation is zero.
- `MATLAB:intConvertNonIntVal` — Warning on an attempt to convert a non-integer value to an integer. The result is that the input value is rounded to the nearest integer for that class.
- `MATLAB:intConvertOverflow` — Warning on overflow when attempting to convert from a numeric class to an integer class. The result is the maximum value for the target class.
- `MATLAB:intMathOverflow` — Warning on overflow when attempting an integer arithmetic operation. The result is the maximum value for the class of the input value. MATLAB also issues this warning when NaN is computed (e.g., `int8(0)/0`).

`intwarning('action')` sets or displays the state of integer warnings in MATLAB according to the string, *action*. There are three possible actions, as shown here. The default state is 'off'.

Action	Description
off	Disable the display of integer warnings
on	Enable the display of integer warnings
query	Display the state of all integer warnings

`s = intwarning('action')` sets the state of integer warnings in MATLAB according to the string *action*, and then returns the previous state in a 4-by-1 structure array, `s`. The return structure array has two fields: `identifier` and `state`.

`intwarning(s)` sets the state of integer warnings in MATLAB according to the `identifier` and `state` fields in structure array `s`.

`sOld = intwarning(sNew)` sets the state of integer warnings in MATLAB according to `sNew`, and then returns the previous state in `sOld`.

## Remarks

---

**Caution** Enabling the `MATLAB:intMathOverflow` warning slows down integer arithmetic. It is recommended that you enable this particular warning only when you need to diagnose unusual behavior in your code, and disable it during normal program operation. The other integer warnings listed here do not affect program performance.

---

## Examples

### General Usage

Examples of the four types of integer warnings are shown here:

- **MATLAB:intConvertNaN**

Attempt to convert NaN (Not a Number) to an unsigned integer:

```
uint8(NaN);  
Warning: NaN converted to uint8(0).
```

- **MATLAB:intConvertNonIntVal**

Attempt to convert a floating point number to an unsigned integer:

```
uint8(2.7);  
Warning: Conversion rounded non-integer floating point  
value to nearest uint8 value.
```

- **MATLAB:intConvertOverflow**

Attempt to convert a large unsigned integer to a signed integer, where the operation overflows:

```
int8(uint8(200));  
Warning: Out of range value converted to intmin('int8')  
        or intmax('int8').
```

- **MATLAB:intMathOverflow**

Attempt an integer arithmetic operation that overflows:

```
intmax('uint8') + 5;  
Warning: Out of range value or NaN computed in  
integer arithmetic.
```

## Example 1

Check the initial state of integer warnings:

```
intwarning('query')  
The state of warning 'MATLAB:intConvertNaN' is 'off'.  
The state of warning 'MATLAB:intConvertNonIntVal' is 'off'.  
The state of warning 'MATLAB:intConvertOverflow' is 'off'.  
The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

Convert a floating point value to an 8-bit unsigned integer. MATLAB does the conversion, but that requires rounding the resulting value. Because all integer warnings have been disabled, no warning is displayed:

```
uint8(2.7)  
ans =  
    3
```

Store this state in structure array iwState:

```
iwState = intwarning('query');
```

Change the state of the `ConvertNonIntVal` warning to 'on' by first setting the state to 'on' in the `iwState` structure array, and then loading `iwState` back into the internal integer warning settings for your MATLAB session:

```
maxintwarn = 4;

for k = 1:maxintwarn
    if strcmp(iwState(k).identifier, ...
             'MATLAB:intConvertNonIntVal')
        iwState(k).state = 'on';
        intwarning(iwState);
    end
end
```

Verify that the state of `ConvertNonIntVal` has changed:

```
intwarning('query')
The state of warning 'MATLAB:intConvertNaN' is 'off'.
The state of warning 'MATLAB:intConvertNonIntVal' is 'on'.
The state of warning 'MATLAB:intConvertOverflow' is 'off'.
The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

Now repeat the conversion from floating point to integer. This time MATLAB displays the warning:

```
uint8(2.7)
Warning: Conversion rounded non-integer floating point
value to nearest uint8 value.
ans =
    3
```

## See Also

`warning`, `lastwarn`

# inv

---

**Purpose** Matrix inverse

**Syntax** `Y = inv(X)`

**Description** `Y = inv(X)` returns the inverse of the square matrix `X`. A warning message is printed if `X` is badly scaled or nearly singular.

In practice, it is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises when solving the system of linear equations  $Ax = b$ . One way to solve this is with `x = inv(A)*b`. A better way, from both an execution time and numerical accuracy standpoint, is to use the matrix division operator `x = A\b`. This produces the solution using Gaussian elimination, without forming the inverse. See `\` and `/` for further information.

## Examples

Here is an example demonstrating the difference between solving a linear system by inverting the matrix with `inv(A)*b` and solving it directly with `A\b`. A random matrix `A` of order 500 is constructed so that its condition number, `cond(A)`, is  $1.e10$ , and its norm, `norm(A)`, is 1. The exact solution `x` is a random vector of length 500 and the right-hand side is `b = A*x`. Thus the system of linear equations is badly conditioned, but consistent.

On a 300 MHz, laptop computer the statements

```
n = 500;
Q = orth(randn(n,n));
d = logspace(0, -10,n);
A = Q*diag(d)*Q';
x = randn(n,1);
b = A*x;
tic, y = inv(A)*b; toc
err = norm(y-x)
res = norm(A*y-b)
```

produce

```
elapsed_time =
```



```

    1.4320
err =
    7.3260e-006
res =
    4.7511e-007

```

while the statements

```

tic, z = A\b, toc
err = norm(z-x)
res = norm(A*z-b)

```

produce

```

elapsed_time =
    0.6410
err =
    7.1209e-006
res =
    4.4509e-015

```

It takes almost two and one half times as long to compute the solution with  $y = \text{inv}(A)*b$  as with  $z = A\b$ . Both produce computed solutions with about the same error,  $1.e-6$ , reflecting the condition number of the matrix. But the size of the residuals, obtained by plugging the computed solution back into the original equations, differs by several orders of magnitude. The direct solution produces residuals on the order of the machine accuracy, even though the system is badly conditioned.

The behavior of this example is typical. Using  $A\b$  instead of  $\text{inv}(A)*b$  is two to three times as fast and produces residuals on the order of machine accuracy, relative to the magnitude of the data.

## Algorithm

### Inputs of Type Double

For inputs of type double, `inv` uses the following LAPACK routines to compute the matrix inverse:

Matrix	Routine
Real	DLANGE, DGETRF, DGECON, DGETRI
Complex	ZLANGE, ZGETRF, ZGECON, ZGETRI

## Inputs of Type Single

For inputs of type `single`, `inv` uses the following LAPACK routines to compute the matrix inverse:

Matrix	Routine
Real	SLANGE, SGETRF, SGECON, SGETRI
Complex	CLANGE, CGETRF, CGECON, CGETRI

## See Also

`det`, `lu`, `rref`

The arithmetic operators `\`, `/`

## References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* ([http://www.netlib.org/lapack/lug/lapack\\_lug.html](http://www.netlib.org/lapack/lug/lapack_lug.html)), Third Edition, SIAM, Philadelphia, 1999.

---

<b>Purpose</b>	Inverse of Hilbert matrix																
<b>Syntax</b>	$H = \text{invhilb}(n)$																
<b>Description</b>	$H = \text{invhilb}(n)$ generates the exact inverse of the exact Hilbert matrix for $n$ less than about 15. For larger $n$ , $\text{invhilb}(n)$ generates an approximation to the inverse Hilbert matrix.																
<b>Limitations</b>	<p>The exact inverse of the exact Hilbert matrix is a matrix whose elements are large integers. These integers may be represented as floating-point numbers without roundoff error as long as the order of the matrix, <math>n</math>, is less than 15.</p> <p>Comparing <math>\text{invhilb}(n)</math> with <math>\text{inv}(\text{hilb}(n))</math> involves the effects of two or three sets of roundoff errors:</p> <ul style="list-style-type: none"><li>• The errors caused by representing <math>\text{hilb}(n)</math></li><li>• The errors in the matrix inversion process</li><li>• The errors, if any, in representing <math>\text{invhilb}(n)</math></li></ul> <p>It turns out that the first of these, which involves representing fractions like <math>1/3</math> and <math>1/5</math> in floating-point, is the most significant.</p>																
<b>Examples</b>	<p><math>\text{invhilb}(4)</math> is</p> <table><tr><td>16</td><td>-120</td><td>240</td><td>-140</td></tr><tr><td>-120</td><td>1200</td><td>-2700</td><td>1680</td></tr><tr><td>240</td><td>-2700</td><td>6480</td><td>-4200</td></tr><tr><td>-140</td><td>1680</td><td>-4200</td><td>2800</td></tr></table>	16	-120	240	-140	-120	1200	-2700	1680	240	-2700	6480	-4200	-140	1680	-4200	2800
16	-120	240	-140														
-120	1200	-2700	1680														
240	-2700	6480	-4200														
-140	1680	-4200	2800														
<b>See Also</b>	<code>hilb</code>																
<b>References</b>	[1] Forsythe, G. E. and C. B. Moler, <i>Computer Solution of Linear Algebraic Systems</i> , Prentice-Hall, 1967, Chapter 19.																

# invoke

---

**Purpose** Invoke method on COM object or interface, or display methods

**Syntax**

```
S = h.invoke
S = h.invoke('methodname')
S = h.invoke('methodname', arg1, arg2, ...)
S = h.invoke('custominterfacename')
S = invoke(h, ...)
```

**Description** `S = h.invoke` returns structure array `S` containing a list of all methods supported by the object or interface, `h`, along with the prototypes for these methods.

If `S` is empty, either there are no properties or methods in the object, or the MATLAB software cannot read the object's type library. Refer to the COM vendor's documentation. For Automation objects, if the vendor provides documentation for specific properties or methods, use the `S = invoke(h, ...)` syntax to call them.

`S = h.invoke('methodname')` invokes the method specified in the string `methodname`, and returns an output value, if any, in `S`. The data type of the return value is dependent upon the specific method being invoked and is determined by the specific control or server.

`S = h.invoke('methodname', arg1, arg2, ...)` invokes the method specified in the string `methodname` with input arguments `arg1`, `arg2`, etc.

`S = h.invoke('custominterfacename')` returns an `Interface` object that serves as a handle to a custom interface implemented by the COM component. The `h` argument is a handle to the COM object. The `custominterfacename` argument is a quoted string returned by the `interfaces` function.

`S = invoke(h, ...)` is an alternate syntax for the same operation.

**Remarks** If the method returns a COM interface, then `invoke` returns a new MATLAB COM object that represents the interface returned. See "Handling COM Data in MATLAB Software" in the External Interfaces

documentation for a description of how MATLAB converts COM data types.

COM functions are available on Microsoft Windows systems only.

## Examples

### Example 1 – Invoking a Method

Create an `mwsamp` control and invoke its `Redraw` method:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.1', [0 0 200 200], f);

h.Radius = 100;
h.invoke('Redraw');
```

Alternatively, call the method directly, passing the handle and any arguments:

```
h.Redraw;
```

Call `invoke` with only the handle argument to display a list of all `mwsamp` methods:

```
h.invoke
```

MATLAB displays:

```
ans =
  AboutBox = void AboutBox(handle)
  Beep = void Beep(handle)
  FireClickEvent = void FireClickEvent(handle)
  .
  .
```

### Example 2 – Getting a Custom Interface

Once you have created a COM server, you can query the server component to see if any custom interfaces are implemented. Use the `interfaces` function to return a list of all available custom interfaces:

# invoke

---

```
h = actxserver('mytestenv.calculator');  
customlist = h.interfaces
```

MATLAB displays:

```
customlist =  
    ICalc1  
    ICalc2  
    ICalc3
```

To get a handle to the custom interface you want, use the `invoke` function, specifying the handle returned by `actxcontrol` or `actxserver` and also the name of the custom interface:

```
c1 = h.invoke('ICalc1')
```

MATLAB displays:

```
c1 =  
    Interface.Calc_1.0_Type_Library.ICalc_Interface
```

You can now use this handle with most of the COM client functions to access the properties and methods of the object through the selected custom interface.

## See Also

`methods`, `ismethod`, `interfaces`

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

**Syntax** `A = ipermute(B,order)`

**Description** `A = ipermute(B,order)` is the inverse of `permute`. `ipermute` rearranges the dimensions of `B` so that `permute(A,order)` will produce `B`. `B` has the same values as `A` but the order of the subscripts needed to access any particular element are rearranged as specified by `order`. All the elements of `order` must be unique.

**Remarks** `permute` and `ipermute` are a generalization of transpose (`.`' ) for multidimensional arrays.

**Examples** Consider the 2-by-2-by-3 array `a`:

```
a = cat(3,eye(2),2*eye(2),3*eye(2))
```

```
a(:,:,1) =          a(:,:,2) =
    1     0          2     0
    0     1          0     2
```

```
a(:,:,3) =
    3     0
    0     3
```

Permuting and inverse permuting `a` in the same fashion restores the array to its original form:

```
B = permute(a,[3 2 1]);
C = ipermute(B,[3 2 1]);
isequal(a,C)
ans=
```

```
1
```

**See Also** `permute`

# iqr (timeseries)

---

**Purpose** Interquartile range of timeseries data

**Syntax** `ts_iqr = iqr(ts)`  
`iqr(ts, 'PropertyName1', PropertyValue1, ...)`

**Description** `ts_iqr = iqr(ts)` returns the interquartile range of `ts.Data`. When `ts.Data` is a vector, `ts_iqr` is the difference between the 75th and the 25th percentiles of the `ts.Data` values. When `ts.Data` is a matrix, `ts_iqr` is a row vector containing the interquartile range of each column of `ts.Data` (when `IsTimeFirst` is true and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `iqr` always operates along the first nonsingleton dimension of `ts.Data`.

`iqr(ts, 'PropertyName1', PropertyValue1, ...)` specifies the following optional input arguments:

- 'MissingData' property has two possible values, 'remove' (default) or 'interpolate', indicating how to treat missing data during the calculation.
- 'Quality' values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- 'Weighting' property has two possible values, 'none' (default) or 'time'.  
When you specify 'time', larger time values correspond to larger weights.

**Examples** Create a time series with a missing value, represented by NaN.

```
ts = timeseries([3.0 NaN 5 6.1 8], 1:5);
```

Calculate the interquartile range of `ts.Data` after removing the missing value from the calculation.

```
iqr(ts, 'MissingData', 'remove')
```



```
ans =  
3.0500
```

**See Also** [timeseries](#)

**Purpose** Detect state

**Description** These functions detect the state of MATLAB entities:

isa	Detect object of given MATLAB class or Java class
isappdata	Determine if object has specific application-defined data
iscell	Determine if input is cell array
iscellstr	Determine if input is cell array of strings
ischar	Determine if input is character array
iscom	Determine if input is Component Object Model (COM) object
isdir	Determine if input is directory
isempty	Determine if input is empty array
isequal	Determine if arrays are numerically equal
isequalwithequalnans	Determine if arrays are numerically equal, treating NaNs as equal
isevent	Determine if input is object event
isfield	Determine if input is MATLAB structure array field
isfinite	Detect finite elements of array
isfloat	Determine if input is floating-point array
isglobal	Determine if input is global variable
ishandle	Detect valid graphics object handles
ishold	Determine if graphics hold state is on
isinf	Detect infinite elements of array
isinteger	Determine if input is integer array
isinterface	Determine if input is Component Object Model (COM) interface

isjava	Determine if input is Java object
iskeyword	Determine if input is MATLAB keyword
isletter	Detect elements that are alphabetic letters
islogical	Determine if input is logical array
ismac	Determine if running MATLAB for Macintosh OS X platform
ismember	Detect members of specific set
ismethod	Determine if input is object method
isnan	Detect elements of array that are not a number (NaN)
isnumeric	Determine if input is numeric array
isobject	Determine if input is MATLAB object
ispc	Determine if running MATLAB for PC (Windows) platform
isprime	Detect prime elements of array
isprop	Determine if input is object property
isreal	Determine if all array elements are real numbers
isscalar	Determine if input is scalar
issorted	Determine if set elements are in sorted order
isspace	Detect space characters in array
issparse	Determine if input is sparse array
isstrprop	Determine if string is of specified category
isstruct	Determine if input is MATLAB structure array
isstudent	Determine if Student Version of MATLAB

## is\*

---

isunix	Determine if running MATLAB for UNIX <sup>15</sup> platform.
isvarname	Determine if input is valid variable name
isvector	Determine if input is vector

### See Also

isa

15. UNIX is a registered trademark of The Open Group in the United States and other countries.

**Purpose** Determine whether input is object of given class

**Syntax** `K = isa(obj, 'class_name')`

**Description** `K = isa(obj, 'class_name')` returns logical 1 (true) if `obj` is of class (or a subclass of) `class_name`, and logical 0 (false) otherwise.

The argument `obj` is a MATLAB object or an object of the Java programming language. The argument `class_name` is the name of a MATLAB (predefined or user-defined) or a Java class. Predefined MATLAB classes include

<code>logical</code>	Logical array of true and false values
<code>char</code>	Characters array
<code>numeric</code>	Integer or floating-point array
<code>integer</code>	Signed or unsigned integer array
<code>int8</code>	8-bit signed integer array
<code>uint8</code>	8-bit unsigned integer array
<code>int16</code>	16-bit signed integer array
<code>uint16</code>	16-bit unsigned integer array
<code>int32</code>	32-bit signed integer array
<code>uint32</code>	32-bit unsigned integer array
<code>int64</code>	64-bit signed integer array
<code>uint64</code>	64-bit unsigned integer array
<code>float</code>	Single- or double-precision floating-point array
<code>single</code>	Single-precision floating-point array
<code>double</code>	Double-precision floating-point array
<code>cell</code>	Cell array
<code>struct</code>	Structure array

function\_handle    Function handle  
'class\_name'        MATLAB class or Java class

To check for a sparse array, use `issparse`. To check for a complex array, use `~isreal`.

## Examples

```
isa(rand(3,4), 'double')  
ans =  
    1
```

The following example creates an instance of the user-defined MATLAB class named `polynom`. The `isa` function identifies the object as being of the `polynom` class.

```
polynom_obj = polynom([1 0 -2 -5]);  
isa(polynom_obj, 'polynom')  
ans =  
    1
```

## See Also

`class`, `is*`

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

**Syntax** `isappdata(h,name)`

**Description** `isappdata(h,name)` returns 1 if application-defined data with the specified name exists on the object specified by handle `h`, and returns 0 otherwise.

**Remarks** Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property.

**See Also** `getappdata`, `rmapdata`, `setappdata`

# iscell

---

**Purpose** Determine whether input is cell array

**Syntax** `tf = iscell(A)`

**Description** `tf = iscell(A)` returns logical 1 (true) if A is a cell array and logical 0 (false) otherwise.

**Examples**

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];  
A{1,2} = 'Anne Smith';  
A{2,1} = 3+7i;  
A{2,2} = -pi:pi/10:pi;
```

```
iscell(A)
```

```
ans =
```

```
1
```

**See Also** `cell`, `iscellstr`, `isstruct`, `isnumeric`, `islogical`, `isobject`, `isa`, `is*`



**Purpose** Determine whether input is cell array of strings

**Syntax** `tf = iscellstr(A)`

**Description** `tf = iscellstr(A)` returns logical 1 (true) if A is a cell array of strings (or an empty cell array), and logical 0 (false) otherwise. A cell array of strings is a cell array where every element is a character array.

**Examples**

```
A{1,1} = 'Thomas Lee';  
A{1,2} = 'Marketing';  
A{2,1} = 'Allison Jones';  
A{2,2} = 'Development';
```

```
iscellstr(A)
```

```
ans =
```

```
1
```

**See Also** `cellstr`, `iscell`, `isstrprop`, `strings`, `char`, `isstruct`, `isa`, `is*`

# ischar

---

**Purpose** Determine whether item is character array

**Syntax** `tf = ischar(A)`

**Description** `tf = ischar(A)` returns logical 1 (true) if A is a character array and logical 0 (false) otherwise.

**Examples** Given the following cell array,

```
C{1,1} = magic(3);           % double array
C{1,2} = 'John Doe';       % char array
C{1,3} = 2 + 4i            % complex double
```

```
C =
```

```
    [3x3 double]    'John Doe'    [2.0000+ 4.0000i]
```

`ischar` shows that only `C{1,2}` is a character array.

```
for k = 1:3
    x(k) = ischar(C{1,k});
end
```

```
x
```

```
x =
```

```
    0    1    0
```

**See Also** `char`, `strings`, `isletter`, `isspace`, `isstrprop`, `iscellstr`, `isnumeric`, `isa`, `is*`

<b>Purpose</b>	Is input COM object
<b>Syntax</b>	<pre>tf = h.iscom tf = iscom(h)</pre>
<b>Description</b>	<p><code>tf = h.iscom</code> returns logical 1 (true) if the input handle, <code>h</code>, is a COM or Microsoft ActiveX object. Otherwise, <code>iscom</code> returns logical 0 (false).</p> <p><code>tf = iscom(h)</code> is an alternate syntax for the same operation.</p>
<b>Remarks</b>	COM functions are available on Microsoft Windows systems only.
<b>Examples</b>	<p>Create a COM server running Microsoft Excel software. The <code>actxserver</code> function returns a handle <code>h</code> to the server object. Testing this handle with <code>iscom</code> returns true:</p> <pre>h = actxserver('Excel.application'); h.iscom</pre> <p>MATLAB software displays:</p> <pre>ans =      1</pre> <p>Create an interface to workbooks, returning handle <code>w</code>. Testing this handle with <code>iscom</code> returns false:</p> <pre>w = h.get('workbooks'); w.iscom</pre> <p>MATLAB displays:</p> <pre>ans =      0</pre>
<b>See Also</b>	<code>isinterface</code>

# isdir

---

**Purpose** Determine whether input is a directory

**Syntax** `tf = isdir('A')`

**Description** `tf = isdir('A')` returns logical 1 (true) if A is a directory. Otherwise, it returns logical 0 (false).

**Examples** Type

```
tf=isdir('myfiles/results')
```

and the MATLAB software returns

```
tf =  
    1
```

indicating that `myfiles/results` is a directory.

**See Also** `dir`, `is*`

**Purpose** Determine whether array is empty

**Syntax** `TF = isempty(A)`

**Description** `TF = isempty(A)` returns logical 1 (true) if `A` is an empty array and logical 0 (false) otherwise. An empty array has at least one dimension of size zero, for example, 0-by-0 or 0-by-5.

**Examples**

```
B = rand(2,2,2);  
B(:,:,:) = [];  
  
isempty(B)  
  
ans = 1
```

**See Also** `is*`

# isempty (timeseries)

---

**Purpose** Determine whether `timeseries` object is empty

**Syntax** `isempty(ts)`

**Description** `isempty(ts)` returns a logical value for `timeseries` object `ts`, as follows:

- 1 — When `ts` contains no data samples or `ts.Data` is empty.
- 0 — When `ts` contains data samples

**See Also** `length (timeseries)`, `size (timeseries)`, `timeseries`, `tsprops`

**Purpose**

Determine whether `tscollection` object is empty

**Syntax**

`isempty(tsc)`

**Description**

`isempty(tsc)` returns a logical value for `tscollection` object `tsc`, as follows:

- 1 — When `tsc` contains neither `timeseries` members nor a time vector
- 0 — When `tsc` contains either `timeseries` members or a time vector

**See Also**

`length (tscollection)`, `size (tscollection)`, `timeseries`, `tscollection`

# isequal

---

**Purpose** Test arrays for equality

**Syntax** `tf = isequal(A, B, ...)`

**Description** `tf = isequal(A, B, ...)` returns logical 1 (true) if the input arrays have the same contents, and logical 0 (false) otherwise. Nonempty arrays must be of the same data type and size.

**Remarks** When comparing structures, the order in which the fields of the structures were created is not important. As long as the structures contain the same fields, with corresponding fields set to equal values, `isequal` considers the structures to be equal. See Example 2, below.

When comparing numeric values, `isequal` does not consider the data type used to store the values in determining whether they are equal. See Example 3, below. This is also true when comparing numeric values with certain nonnumeric values, such as logical true and 1, or the character A and its numeric equivalent, 65.

NaNs (Not a Number), by definition, are not equal. Therefore, arrays that contain NaN elements are not equal, and `isequal` returns zero when comparing such arrays. See Example 4, below. Use the `isequalwithequalnans` function when you want to test for equality with NaNs treated as equal.

`isequal` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequal` returns logical 1.

## Examples

### Example 1

Given

A =		B =		C =	
	1    0		1    0		1    0
	0    1		0    1		0    0

`isequal(A,B,C)` returns 0, and `isequal(A,B)` returns 1.



## Example 2

When comparing structures with `isequal`, the order in which the fields of the structures were created is not important:

```
A.f1 = 25;    A.f2 = 50
A =
    f1: 25
    f2: 50

B.f2 = 50;    B.f1 = 25
B =
    f2: 50
    f1: 25

isequal(A, B)
ans =
    1
```

## Example 3

When comparing numeric values, the data types used to store the values are not important:

```
A = [25 50];    B = [int8(25) int8(50)];

isequal(A, B)
ans =
    1
```

## Example 4

Arrays that contain NaN (Not a Number) elements cannot be equal, since NaNs, by definition, are not equal:

```
A = [32 8 -29 NaN 0 5.7];
B = A;

isequal(A, B)
ans =
```

# isequal

---

0

**See Also**      `isequalwithequalnans`, `strcmp`, `isa`, `is*`, relational operators

<b>Purpose</b>	Compare MException objects for equality
<b>Syntax</b>	TF = isequal(eObj1, eObj2)
<b>Description</b>	TF = isequal(eObj1, eObj2) tests MException objects eObj1 and eObj2 for equality, returning logical 1 (true) if the two objects are identical, otherwise returning logical 0 (false).
<b>See Also</b>	try, catch, error, assert, MException, eq(MException), ne(MException), getReport(MException), disp(MException), throw(MException), rethrow(MException), throwAsCaller(MException), addCause(MException), last(MException),

# isequalwithequalnans

---

**Purpose** Test arrays for equality, treating NaNs as equal

**Syntax** `tf = isequalwithequalnans(A, B, ...)`

**Description** `tf = isequalwithequalnans(A, B, ...)` returns logical 1 (true) if the input arrays are the same type and size and hold the same contents, and logical 0 (false) otherwise. NaN (Not a Number) values are considered to be equal to each other. Numeric data types and structure field order do not have to match.

**Remarks** `isequalwithequalnans` is the same as `isequal`, except `isequalwithequalnans` considers NaN (Not a Number) values to be equal, and `isequal` does not.

`isequalwithequalnans` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequalwithequalnans` returns logical 1.

**Examples** Arrays containing NaNs are handled differently by `isequal` and `isequalwithequalnans`. `isequal` does not consider NaNs to be equal, while `isequalwithequalnans` does.

```
A = [32 8 -29 NaN 0 5.7];
B = A;
isequal(A, B)
ans =
    0
```

```
isequalwithequalnans(A, B)
ans =
    1
```

The position of NaN elements in the array does matter. If they are not in the same position in the arrays being compared, then `isequalwithequalnans` returns zero.

```
A = [2 4 6 NaN 8]; B = [2 4 NaN 6 8];
```

```
isequalwithequalnans(A, B)  
ans =  
    0
```

**See Also**

isequal, strcmp, isa, is\*, relational operators

# isevent

---

**Purpose** True if COM object event

**Syntax**  
`tf = h.isevent('name')`  
`tf = isevent(h, 'name')`

**Description** `tf = h.isevent('name')` returns logical 1 (true) if the specified name is an event that can be recognized and responded to by COM object `h`. Otherwise, `isevent` returns logical 0 (false).

`tf = isevent(h, 'name')` is an alternate syntax for the same operation.

**Remarks** The string specified in the name argument is not case sensitive.  
For COM control objects, `isevent` returns the same value regardless of whether the specified event is registered with the control or not. In order for the control to respond to the event, you must first register the event using either `activexcontrol` or `registerevent`.

COM functions are available on Microsoft Windows systems only.

## Examples **Test the Sample Control Example**

Create an `mwsamp` control and test to see if `Db1Click` is an event recognized by the control.

```
f = figure ('position', [100 200 200 200]);  
h = axescontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);  
h.isevent('Db1Click')
```

The MATLAB software displays `ans = 1 (true)`, showing that `Db1Click` is an event.

Try the same test on `Redraw`, which is one of the control's methods.

```
h.isevent('Redraw')
```

MATLAB displays `ans = 0 (false)`, showing that `Redraw` is not an event.

## Test a Workbook Example

Create a Microsoft Excel Workbook object.

```
myApp = actxserver('Excel.Application');  
wbs = myApp.Workbooks;  
wb = wbs.Add;
```

Test the Activate event:

```
wb.isevent('Activate')
```

MATLAB displays `ans = 1 (true)`, showing that Activate is an event.

Test Save:

```
wb.isevent('Save')
```

MATLAB displays `ans = 0 (false)`, showing that Save is not an event; it is a method.

## See Also

events (COM), eventlisteners, registerevent, unregisterevent, unregisterallevnts

# isfield

---

**Purpose** Determine whether input is structure array field

**Syntax**

```
tf = isfield(S, 'fieldname')  
tf = isfield(S, C)
```

**Description**

`tf = isfield(S, 'fieldname')` examines structure `S` to see if it includes the field specified by the quoted string `'fieldname'`. Output `tf` is set to logical 1 (true) if `S` contains the field, or logical 0 (false) if not. If `S` is not a structure array, `isfield` returns false.

`tf = isfield(S, C)` examines structure `S` for multiple fieldnames as specified in cell array of strings `C`, and returns an array of logical values to indicate which of these fields are part of the structure. Elements of output array `tf` are set to a logical 1 (true) if the corresponding element of `C` holds a fieldname that belongs to structure `S`. Otherwise, logical 0 (false) is returned in that element. In other words, if structure `S` contains the field specified in `C{m,n}`, `isfield` returns a logical 1 (true) in `tf(m,n)`.

---

**Note** `isfield` returns false if the field or fieldnames input is empty.

---

## Examples

### Example 1 – Single Fieldname Syntax

Given the following MATLAB structure,

```
patient.name = 'John Doe';  
patient.billing = 127.00;  
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

`isfield` identifies `billing` as a field of that structure.

```
isfield(patient,'billing')  
ans =  
    1
```



### Example 2 – Multiple Fieldname Syntax

Check structure S for any of four possible fieldnames. Only the first is found, so the first element of the return value is set to true:

```
S = struct('one', 1, 'two', 2);

fields = isfield(S, {'two', 'pi', 'One', 3.14})
fields =
     1     0     0     0
```

### See Also

fieldnames, setfield, getfield, orderfields, rmfield, struct, isstruct, iscell, isa, is\*, dynamic field names

# isfinite

---

**Purpose** Array elements that are finite

**Syntax** TF = isfinite(A)

**Description** TF = isfinite(A) returns an array the same size as A containing logical 1 (true) where the elements of the array A are finite and logical 0 (false) where they are infinite or NaN. For a complex number z, isfinite(z) returns 1 if both the real and imaginary parts of z are finite, and 0 if either the real or the imaginary part is infinite or NaN. For any real A, exactly one of the three quantities isfinite(A), isinf(A), and isnan(A) is equal to one.

**Examples**

```
a = [-2 -1 0 1 2];
isfinite(1./a)
ans =
     1     1     0     1     1
isfinite(0./a)
ans =
     1     1     0     1     1
```

**See Also** isinf, isnan, is\*

<b>Purpose</b>	Determine whether input is floating-point array
<b>Syntax</b>	<code>isfloat(A)</code>
<b>Description</b>	<code>isfloat(A)</code> returns a logical 1 (true) if A is a floating-point array and a logical 0 (false) otherwise. The only floating-point data types in the MATLAB programming language are <code>single</code> and <code>double</code> .
<b>See Also</b>	<code>isa</code> , <code>isinteger</code> , <code>double</code> , <code>single</code> , <code>isnumeric</code>

# isglobal

---

**Purpose** Determine whether input is global variable

---

**Note** Support for the `isglobal` function will be removed in a future release of the MATLAB software. See Remarks below.

---

**Syntax** `tf = isglobal(A)`

**Description** `tf = isglobal(A)` returns logical 1 (true) if `A` has been declared to be a global variable in the context from which `isglobal` is called, and logical 0 (false) otherwise.

**Remarks** `isglobal` is most commonly used in conjunction with conditional global declaration. An alternate approach is to use a pair of variables, one local and one declared global.

Instead of using

```
if condition
    global x
end

x = some_value

if isglobal(x)
    do_something
end
```

You can use

```
global gx
if condition
    gx = some_value
else
    x = some_value
end
```

```
if condition
    do_something
end
```

If no other workaround is possible, you can replace the command

```
isglobal(variable)
```

with

```
~isempty(whos('global','variable'))
```

## See Also

global, isvarname, isa, is\*

# ishandle

---

**Purpose** Determine whether input is valid Handle Graphics handle

**Syntax** `ishandle(H)`

**Description** `ishandle(H)` returns an array whose elements are 1 where the elements of `H` are valid graphics or Sun Java object handles, and 0 where they are not.

You should use the `isa` function to determine the class and validity of MATLAB objects.

**See Also** `findobj`, `gca`, `gcf`, `gco`, `isa` set

“Accessing Object Handles” for more information.

<b>Purpose</b>	Current hold state
<b>Syntax</b>	<code>ishold</code>
<b>Description</b>	<p><code>ishold</code> returns 1 if <code>hold</code> is on, and 0 if it is off. When <code>hold</code> is on, the current plot and most axis properties are held so that subsequent graphing commands add to the existing graph.</p> <p>A state of <code>hold</code> on implies that both figure and axes <code>NextPlot</code> properties are set to <code>add</code>.</p>
<b>See Also</b>	<p><code>hold</code>, <code>newplot</code></p> <p>“Controlling Graphics Output” for related information</p> <p>“Axes Operations” on page 1-100 for related functions</p>

# isinf

---

**Purpose** Array elements that are infinite

**Syntax** TF = isinf(A)

**Description** TF = isinf(A) returns an array the same size as A containing logical 1 (true) where the elements of A are +Inf or -Inf and logical 0 (false) where they are not. For a complex number z, isinf(z) returns 1 if either the real or imaginary part of z is infinite, and 0 if both the real and imaginary parts are finite or NaN.

For any real A, exactly one of the three quantities isfinite(A), isinf(A), and isnan(A) is equal to one.

## Examples

```
a = [-2 -1 0 1 2]

isinf(1./a)
Warning: Divide by zero.

ans =
     0     0     1     0     0

isinf(0./a)
Warning: Divide by zero.

ans =
     0     0     0     0     0
```

**See Also** isfinite, isnan, is\*



**Purpose** Determine whether input is integer array

**Syntax**

**Description** `isinteger(A)` returns a logical 1 (true) if the array `A` has integer data type and a logical 0 (false) otherwise. The integer data types in the MATLAB language are

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `int64`
- `uint64`

**See Also** `isa`, `isnumeric`, `isfloat`

# isinterface

---

**Purpose** Is input COM interface

**Syntax** `tf = h.isinterface`  
`tf = isinterface(h)`

**Description** `tf = h.isinterface` returns logical 1 (true) if the input handle, `h`, is a COM interface. Otherwise, `isinterface` returns logical 0 (false).  
`tf = isinterface(h)` is an alternate syntax for the same operation.

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

**Examples** Create a COM server running Microsoft Excel application. The `actxserver` function returns a handle `h` to the server object. Testing this handle with `isinterface` returns false:

```
h = actxserver('Excel.application');  
h.isinterface
```

MATLAB software displays:

```
ans =  
    0
```

Create an interface to workbooks, returning handle `w`. Testing this handle with `isinterface` returns true:

```
w = h.get('workbooks');  
w.isinterface
```

MATLAB displays:

```
ans =  
    1
```

**See Also** `iscom`, `interfaces`, `get (COM)`

<b>Purpose</b>	Determine whether input is Sun Java object
<b>Syntax</b>	<code>tf = isjava(A)</code>
<b>Description</b>	<code>tf = isjava(A)</code> returns logical 1 (true) if <i>A</i> is a Java object, and logical 0 (false) otherwise.
<b>Examples</b>	<p>Create an instance of the Java Date class and <code>isjava</code> indicates that it is a Java object.</p> <pre>myDate = java.util.Date; isjava(myDate)</pre> <p>The MATLAB software displays:</p> <pre>ans =  1</pre> <p>Note that <code>isobject</code>, which tests for MATLAB objects, returns logical 0 (false). Type:</p> <pre>isobject(myDate)</pre> <p>MATLAB displays:</p> <pre>ans =  0</pre>
<b>See Also</b>	<code>isobject</code> , <code>javaArray</code> , <code>javaMethod</code> , <code>javaObject</code> , <code>isa</code> , <code>is*</code>

# isKey (Map)

---

**Purpose** Check if containers.Map contains key

**Syntax** `tf = isKey(M, keys)`

**Description** `tf = isKey(M, keys)` looks for the specified keys in the Map instance `M`, and returns logical 1 (true) for those elements that it finds, and logical 0 (false) for those it does not. `keys` is a scalar key or cell array of keys. If `keys` is nonscalar, then return value `tf` is a nonscalar logical array that has the same dimensions and size as `keys`.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

**Examples** Construct a Map object where the keys are states in the United States and the value associated with each key is that state's capital city:

```
US_Capitals = containers.Map( ...
    {'Arizona', 'Nebraska', 'Nevada', 'New York', ...
     'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...
    {'Phoenix', 'Lincoln', 'Carson City', 'Albany', ...
     'Atlanta', 'Juneau', 'Montpelier', 'Salem'})
```

Check three states to see if they are in the map:

```
isKey(US_Capitals, {'Georgia', 'Alaska', 'Wyoming'})
ans =
     1     1     0
```

Check two states and a capital to see if they are all keys in the map:

```
isKey(US_Capitals, {'Georgia'; 'Montpelier'; 'Alaska'})
ans =
     1     0     1
```

Identify the capital city of a specific state, but only attempt this if you know that this state is in the map:

```
S = 'Nebraska';
```

```
if isKey(US_Capitals, S)
  sprintf('  The capital of %s is %s', S, US_Capitals(S))
else error('The state of %s is not in the map', S)
end
ans =
    The capital of Nebraska is Lincoln

S = Montana';
if isKey(US_Capitals, S)
  sprintf('  The capital of %s is %s', S, US_Capitals(S))
else error('The state of %s is not in the map', S)
end

??? The state of Montana is not in the map
```

### See Also

containers.Map, keys(Map), values(Map), size(Map), length(Map),  
remove(Map), handle

# iskeyword

---

**Purpose** Determine whether input is MATLAB keyword

**Syntax** `tf = iskeyword('str')`  
`iskeyword str`  
`iskeyword`

**Description** `tf = iskeyword('str')` returns logical 1 (true) if the string `str` is a keyword in the MATLAB language and logical 0 (false) otherwise.

`iskeyword str` uses the MATLAB command format.

`iskeyword` returns a list of all MATLAB keywords.

**Examples** To test if the word `while` is a MATLAB keyword,

```
iskeyword while
ans =
     1
```

To obtain a list of all MATLAB keywords,

```
iskeyword
'break'
'case'
'catch'
'classdef'
'continue'
'else'
'elseif'
'end'
'for'
'function'
'global'
'if'
'otherwise'
'parfor'
'persistent'
'return'
```

```
'switch'  
'try'  
'while'
```

**See Also** [isvarname](#), [genvarname](#), [is\\*](#)

# isletter

---

**Purpose** Array elements that are alphabetic letters

**Syntax** `tf = isletter('str')`

**Description** `tf = isletter('str')` returns an array the same size as `str` containing logical 1 (true) where the elements of `str` are letters of the alphabet and logical 0 (false) where they are not.

**Examples** Find the letters in character array `s`.

```
s = 'A1,B2,C3';  
  
isletter(s)  
ans =  
     1     0     0     1     0     0     1     0
```

**See Also** `ischar`, `isspace`, `isstrprop`, `iscellstr`, `isnumeric`, `char`, `strings`, `isa`, `is*`



**Purpose** Determine whether input is logical array

**Syntax** `tf = islogical(A)`

**Description** `tf = islogical(A)` returns logical 1 (true) if A is a logical array and logical 0 (false) otherwise.

**Examples** Given the following cell array,

```
C{1,1} = pi;           % double
C{1,2} = 1;           % double
C{1,3} = ispc;       % logical
C{1,4} = magic(3)    % double array
```

```
C =
    [3.1416]    [1]    [1]    [3x3 double]
```

`islogical` shows that only `C{1,3}` is a logical array.

```
for k = 1:4
    x(k) = islogical(C{1,k});
end
```

```
x
x =
    0    0    1    0
```

**See Also** `logical`, `isnumeric`, `ischar`, `isreal`, `,` logical operators (elementwise and short-circuit), `isa`, `is*`

# ismac

---

**Purpose** Determine if version is for Mac OS X platform

**Syntax** `tf = ismac`

**Description** `tf = ismac` returns logical 1 (true) if the version of MATLAB software is for the Apple Mac OS X platform, and returns logical 0 (false) otherwise.

**See Also** `isunix, ispc, isstudent, is*`

**Purpose** Array elements that are members of set

**Syntax**

```
tf = ismember(A, S)
tf = ismember(A, S, 'rows')
[tf, loc] = ismember(A, S, ...)
```

**Description** `tf = ismember(A, S)` returns a vector the same length as `A`, containing logical 1 (true) where the elements of `A` are in the set `S`, and logical 0 (false) elsewhere. In set theory terms, `k` is 1 where  $A \in S$ . Inputs `A` and `S` can be numeric or character arrays or cell arrays of strings.

`tf = ismember(A, S, 'rows')`, when `A` and `S` are matrices with the same number of columns, returns a vector containing 1 where the rows of `A` are also rows of `S` and 0 otherwise. You cannot use this syntax if `A` or `S` is a cell array of strings.

`[tf, loc] = ismember(A, S, ...)` returns an array `loc` containing the highest index in `S` for each element in `A` that is a member of `S`. For those elements of `A` that do not occur in `S`, `ismember` returns 0.

**Remarks** Because NaN is considered to be not equal to anything, it is never a member of any set.

**Examples**

```
set = [0 2 4 6 8 10 12 14 16 18 20];
a = reshape(1:5, [5 1])
```

```
a =
     1
     2
     3
     4
     5

ismember(a, set)
ans =
     0
     1
```

# ismember

---

```
0
1
0
set = [5 2 4 2 8 10 12 2 16 18 20 3];
[tf, index] = ismember(a, set);

index
index =
    0
     8
    12
     3
     1
```

## See Also

issorted, intersect, setdiff, setxor, union, unique, is\*

**Purpose** Determine whether input is COM object method

**Syntax** `ismethod(h, 'name')`

**Description** `ismethod(h, 'name')` returns a logical 1 (true) if the specified name is a method that you can call on COM object `h`. Otherwise, `ismethod` returns logical 0 (false).

**Examples** Create a Microsoft Excel application and test to see if `SaveWorkspace` is a method of the object. `ismethod` returns true:

```
h = actxserver ('Excel.Application');  
  
ismethod(h, 'SaveWorkspace')
```

MATLAB software displays:

```
ans =  
    1
```

Try the same test on `UsableWidth`, which is a property. `ismethod` returns false:

```
ismethod(h, 'UsableWidth')
```

MATLAB displays:

```
ans =  
    0
```

**See Also** `methods`, `methodsviw`, `isprop`, `isevent`, `isobject`, `class`, `invoke`

# isnan

---

**Purpose** Array elements that are NaN

**Syntax** TF = isnan(A)

**Description** TF = isnan(A) returns an array the same size as A containing logical 1 (true) where the elements of A are NaNs and logical 0 (false) where they are not. For a complex number z, isnan(z) returns 1 if either the real or imaginary part of z is NaN, and 0 if both the real and imaginary parts are finite or Inf.

For any real A, exactly one of the three quantities isfinite(A), isinf(A), and isnan(A) is equal to one.

## Examples

```
a = [-2 -1 0 1 2]
```

```
isnan(1./a)
```

```
Warning: Divide by zero.
```

```
ans =
```

```
0 0 0 0 0
```

```
isnan(0./a)
```

```
Warning: Divide by zero.
```

```
ans =
```

```
0 0 1 0 0
```

## See Also

isfinite, isinf, is\*

**Purpose** Determine whether input is numeric array

**Syntax** `tf = isnumeric(A)`

**Description** `tf = isnumeric(A)` returns logical 1 (true) if `A` is a numeric array and logical 0 (false) otherwise. For example, sparse arrays and double-precision arrays are numeric, while strings, cell arrays, and structure arrays and logicals are not.

**Examples** Given the following cell array,

```
C{1,1} = pi; % double
C{1,2} = 'John Doe'; % char array
C{1,3} = 2 + 4i; % complex double
C{1,4} = ispc; % logical
C{1,5} = magic(3) % double array

C =
    [3.1416] 'John Doe' [2.0000+ 4.0000i] [1][3x3 double]
```

`isnumeric` shows that all but `C{1,2}` and `C{1,4}` are numeric arrays.

```
for k = 1:5
    x(k) = isnumeric(C{1,k});
end

x
x =
     1     0     1     0     1
```

**See Also** `isstrprop`, `isnan`, `isreal`, `isprime`, `isfinite`, `isinf`, `isa`, `is*`

# isobject

---

**Purpose** Determine if input is MATLAB object

**Syntax** `tf = isobject(A)`

**Description** `tf = isobject(A)` returns logical 1 (true) if A is a MATLAB object and logical 0 (false) otherwise. Note that Handle Graphics objects return false. Use `ishandle` to test for Handle Graphics

**Examples** Suppose you define the following MATLAB class:

```
classdef button < handle
    properties
        UiHandle
    end
    methods
        function obj = button(pos)
            obj.UiHandle = uicontrol('Position',pos,...
                'Style','pushbutton');
        end
    end
end
```

You can use `isobject` to determine what objects are instances of MATLAB classes. For example:

```
h = button([20 20 60 60]);
isobject(h)
ans =
    1
isobject(h.UiHandle)
ans =
    0
```

Note that `isjava`, which tests for Sun Java objects in MATLAB, returns false for MATLAB objects:

```
isjava(h)
```



```
ans =  
    0
```

For more information on MATLAB classes and objects, see: “MATLAB Classes Overview”.

## See Also

`isjava`, `isstruct`, `iscell`, `ischar`, `isnumeric`, `islogical`, `methods`, `class`, `isa`, `is*`

# isocaps

---

**Purpose** Compute isosurface end-cap geometry

**Syntax**

```
fvc = isocaps(X,Y,Z,V,isovalue)
fvc = isocaps(V,isovalue)
fvc = isocaps(...,'enclose')
fvc = isocaps(...,'whichplane')
[f,v,c] = isocaps(...)
isocaps(...)
```

**Description** `fvc = isocaps(X,Y,Z,V,isovalue)` computes isosurface end-cap geometry for the volume data `V` at isosurface value `isovalue`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`.

The struct `fvc` contains the face, vertex, and color data for the end-caps and can be passed directly to the `patch` command.

`fvc = isocaps(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isocaps(...,'enclose')` specifies whether the end-caps enclose data values above or below the value specified in `isovalue`. The string `enclose` can be either `above` (default) or `below`.

`fvc = isocaps(...,'whichplane')` specifies on which planes to draw the end-caps. Possible values for `whichplane` are `all` (default), `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, or `zmax`.

`[f,v,c] = isocaps(...)` returns the face, vertex, and color data for the end-caps in three arrays instead of the struct `fvc`.

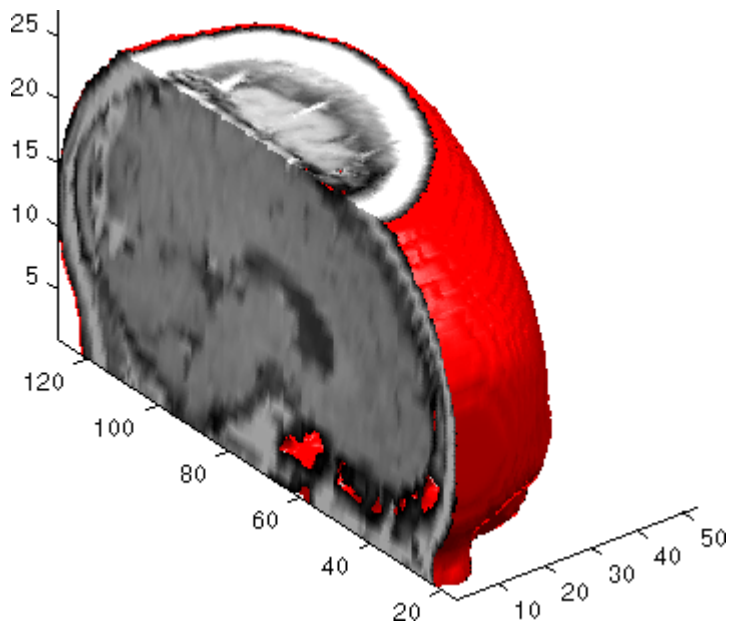
`isocaps(...)` without output arguments draws a patch with the computed faces, vertices, and colors.

**Examples** This example uses a data set that is a collection of MRI slices of a human skull. It illustrates the use of `isocaps` to draw the end-caps on this cutaway volume.

The red isosurface shows the outline of the volume (skull) and the end-caps show what is inside of the volume.

The patch created from the end-cap data (p2) uses interpolated face coloring, which means the gray colormap and the light sources determine how it is colored. The isosurface patch (p1) used a flat red face color, which is affected by the lights, but does not use the colormap.

```
load mri
D = squeeze(D);
D(:,1:60,:) = [];
p1 = patch(isosurface(D, 5), 'FaceColor', 'red', ...
    'EdgeColor', 'none');
p2 = patch(isocaps(D, 5), 'FaceColor', 'interp', ...
    'EdgeColor', 'none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight left; camlight; lighting gouraud
isonormals(D,p1)
```



# isocaps

---

## **See Also**

isosurface, isonormals, smooth3, subvolume, reducevolume,  
reducepatch

“Isocaps Add Context to Visualizations” for more illustrations of isocaps

“Volume Visualization” on page 1-106 for related functions

**Purpose**

Calculate isosurface and patch colors

**Syntax**

```
nc = isocolors(X,Y,Z,C,vertices)
nc = isocolors(X,Y,Z,R,G,B,vertices)
nc = isocolors(C,vertices)
nc = isocolors(R,G,B,vertices)
nc = isocolors(...,PatchHandle)
isocolors(...,PatchHandle)
```

**Description**

`nc = isocolors(X,Y,Z,C,vertices)` computes the colors of isosurface (patch object) vertices (`vertices`) using color values `C`. Arrays `X`, `Y`, `Z` define the coordinates for the color data in `C` and must be monotonic vectors or 3-D plaid arrays (as if produced by `meshgrid`). The colors are returned in `nc`. `C` must be 3-D (index colors).

`nc = isocolors(X,Y,Z,R,G,B,vertices)` uses `R`, `G`, `B` as the red, green, and blue color arrays (true color).

`nc = isocolors(C,vertices)`, and `nc = isocolors(R,G,B,vertices)` assume `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(C)`.

`nc = isocolors(...,PatchHandle)` uses the vertices from the patch identified by `PatchHandle`.

`isocolors(...,PatchHandle)` sets the `FaceVertexCData` property of the patch specified by `PatchHandle` to the computed colors.

**Examples****Indexed Color Data**

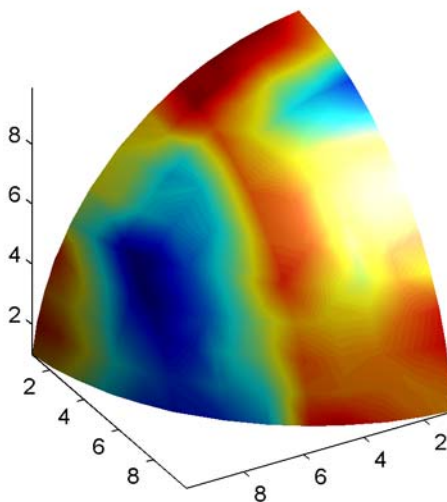
This example displays an isosurface and colors it with random data using indexed color. (See “Interpolating in Indexed Color Versus Truecolor” for information on how patch objects interpret color data.)

```
[x y z] = meshgrid(1:20,1:20,1:20);
```

# isocolors

---

```
data = sqrt(x.^2 + y.^2 + z.^2);
cdata = smooth3(rand(size(data)), 'box', 7);
p = patch(isosurface(x,y,z,data,10));
isonormals(x,y,z,data,p);
isocolors(x,y,z,cdata,p);
set(p, 'FaceColor', 'interp', 'EdgeColor', 'none')
view(150,30); daspect([1 1 1]); axis tight
camlight; lighting phong;
```

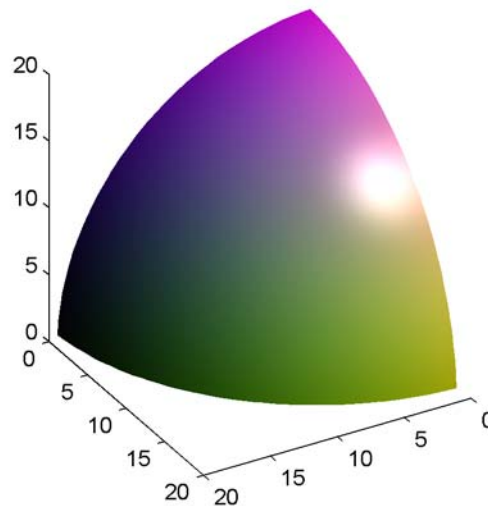


## True Color Data

This example displays an isosurface and colors it with true color (RGB) data.

```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(x,y,z,data,20));
isonormals(x,y,z,data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
```

```
isocolors(x,y,z,r/20,g/20,b/20,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);
camlight; lighting phong;
```



### Modified True Color Data

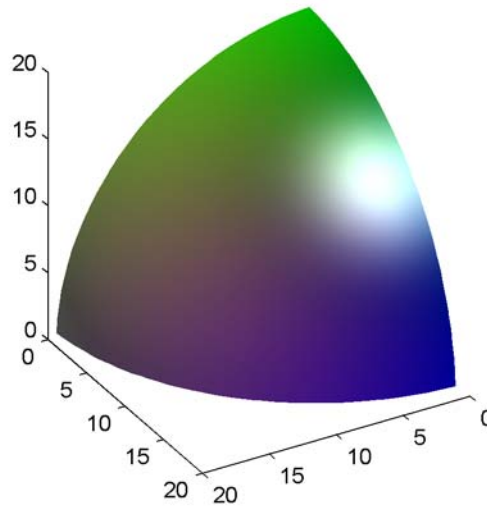
This example uses `isocolors` to calculate the true color data using the isosurface's (patch object's) vertices, but then returns the color data in a variable (`c`) in order to modify the values. It then explicitly sets the isosurface's `FaceVertexCData` to the new data (`1-c`).

```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(data,20));
isonormals(data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
c = isocolors(r/20,g/20,b/20,p);
set(p,'FaceVertexCData',1-c)
```

# isocolors

---

```
set(p,'FaceColor','interp','EdgeColor','none')  
view(150,30); daspect([1 1 1]);  
camlight; lighting phong;
```



## See Also

isosurface, isocaps, smooth3, subvolume, reducevolume,  
reducepatch, isonormals

“Volume Visualization” on page 1-106 for related functions



**Purpose**

Compute normals of isosurface vertices

**Syntax**

```
n = isonormals(X,Y,Z,V,vertices)
n = isonormals(V,vertices)
n = isonormals(V,p) and n = isonormals(X,Y,Z,V,p)
n = isonormals(...,'negate')
isonormals(V,p) and isonormals(X,Y,Z,V,p)
```

**Description**

`n = isonormals(X,Y,Z,V,vertices)` computes the normals of the isosurface vertices from the vertex list, `vertices`, using the gradient of the data `V`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The computed normals are returned in `n`.

`n = isonormals(V,vertices)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`n = isonormals(V,p)` and `n = isonormals(X,Y,Z,V,p)` compute normals from the vertices of the patch identified by the handle `p`.

`n = isonormals(...,'negate')` negates (reverses the direction of) the normals.

`isonormals(V,p)` and `isonormals(X,Y,Z,V,p)` set the `VertexNormals` property of the patch identified by the handle `p` to the computed normals rather than returning the values.

**Examples**

This example compares the effect of different surface normals on the visual appearance of lit isosurfaces. In one case, the triangles used to draw the isosurface define the normals. In the other, the `isonormals` function uses the volume data to calculate the vertex normals based on the gradient of the data points. The latter approach generally produces a smoother-appearing isosurface.

Define a 3-D array of volume data (`cat`, `interp3`):

```
data = cat(3, [0 .2 0; 0 .3 0; 0 0 0], ...
             [.1 .2 0; 0 1 0; .2 .7 0], ...
             [0 .4 .2; .2 .4 0;.1 .1 0]);
```

# isonormals

---

```
data = interp3(data,3,'cubic');
```

Draw an isosurface from the volume data and add lights. This isosurface uses triangle normals (patch, isosurface, view, daspect, axis, camlight, lighting, title):

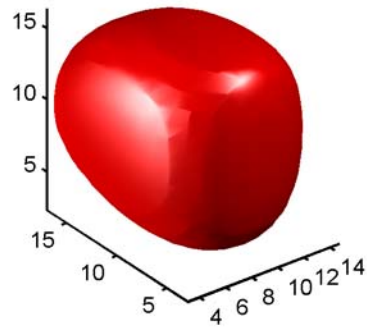
```
subplot(1,2,1)
p1 = patch(isosurface(data,.5),...
'FaceColor','red','EdgeColor','none');
view(3); daspect([1,1,1]); axis tight
camlight; camlight(-80,-10); lighting phong;
title('Triangle Normals')
```

Draw the same lit isosurface using normals calculated from the volume data:

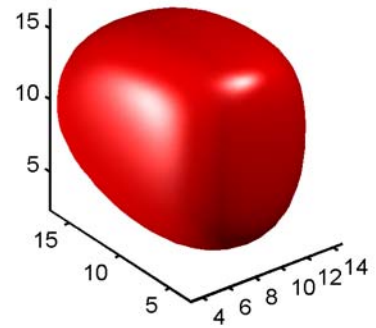
```
subplot(1,2,2)
p2 = patch(isosurface(data,.5),...
'FaceColor','red','EdgeColor','none');
isonormals(data,p2)
view(3); daspect([1 1 1]); axis tight
camlight; camlight(-80,-10); lighting phong;
title('Data Normals')
```

These isosurfaces illustrate the difference between triangle and data normals:

Triangle Normals



Data Normals



## See Also

`interp3`, `isosurface`, `isocaps`, `smooth3`, `subvolume`, `reducevolume`, `reducepatch`

“Volume Visualization” on page 1-106 for related functions

# isosurface

---

**Purpose** Extract isosurface data from volume data

**Syntax**

```
fv = isosurface(X,Y,Z,V,isovalue)
fv = isosurface(V,isovalue)
fvc = isosurface(...,colors)
fv = isosurface(...,'noshare')
fv = isosurface(...,'verbose')
[f,v] = isosurface(...)
[f,v,c] = isosurface(...)
isosurface(...)
```

**Description** `fv = isosurface(X,Y,Z,V,isovalue)` computes isosurface data from the volume data `V` at the isosurface value specified in `isovalue`. That is, the isosurface connects points that have the specified value much the way contour lines connect points of equal elevation.

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The structure `fv` contains the faces and vertices of the isosurface, which you can pass directly to the `patch` command.

`fv = isosurface(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isosurface(...,colors)` interpolates the array `colors` onto the scalar field and returns the interpolated values in the `facevertexcdata` field of the `fvc` structure. The size of the `colors` array must be the same as `V`. The `colors` argument enables you to control the color mapping of the isosurface with data different from that used to calculate the isosurface (e.g., temperature data superimposed on a wind current isosurface).

`fv = isosurface(...,'noshare')` does not create shared vertices. This is faster, but produces a larger set of vertices.

`fv = isosurface(...,'verbose')` prints progress messages to the command window as the computation progresses.

`[f,v] = isosurface(...)` or `[f,v,c] = isosurface(...)` returns the faces and vertices (and `faceVertexcCData`) in separate arrays instead of a struct.

`isosurface(...)` with no output arguments, creates a patch in the current axes with the computed faces and vertices. If no current axes exists, a new axes is created with a 3-D view and appropriate lighting.

## Special Case Behavior – isosurface Called with No Output Arguments

If there is no current axes and you call `isosurface` without assigning output arguments, MATLAB creates a new axes, sets it to a 3-D view, and adds lighting to the isosurface graph.

## Remarks

You can pass the `fv` structure created by `isosurface` directly to the `patch` command, but you cannot pass the individual faces and vertices arrays (`f`, `v`) to `patch` without specifying property names. For example,

```
patch(isosurface(X,Y,Z,V,isovalue))
```

or

```
[f,v] = isosurface(X,Y,Z,V,isovalue);  
patch('Faces',f,'Vertices',v)
```

## Examples

### Example 1

This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type `help flow` for more information). The isosurface is drawn at the data value of -3. The statements that follow the `patch` command prepare the isosurface for lighting by

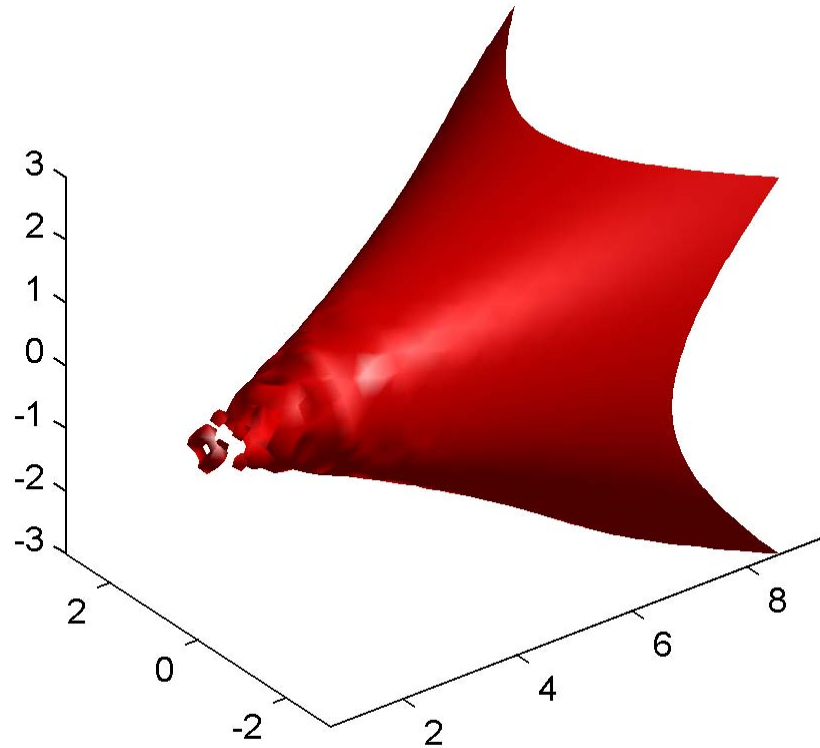
- Recalculating the isosurface normals based on the volume data (`isonormals`)
- Setting the face and edge color (`set`, `FaceColor`, `EdgeColor`)
- Specifying the view (`daspect`, `view`)

# isosurface

---

- Adding lights (camlight, lighting)

```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p)
set(p,'FaceColor','red','EdgeColor','none');
daspect([1 1 1])
view(3); axis tight
camlight
lighting gouraud
```



## Example 2

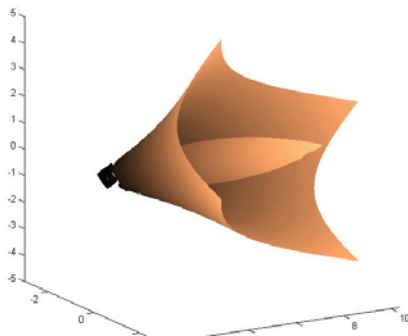
Visualize the same flow data as above, but color-code the surface to indicate magnitude along the X-axis. Use a sixth argument to `isosurface`, which provides a means to overlay another data set by coloring the resulting isosurface. The `colors` variable is a vector containing a scalar value for each vertex in the isosurface, to be portrayed with the current color map. In this case, it is one of the

# isosurface

---

variables that define the surface, but it could be entirely independent. You can apply a different color scheme by changing the current figure color map.

```
[x,y,z,v] = flow;  
[faces,verts,colors] = isosurface(x,y,z,v,-3,x);  
patch('Vertices', verts, 'Faces', faces, ...  
      'FaceVertexCData', colors, ...  
      'FaceColor','interp', ...  
      'edgecolor', 'interp');  
view(30,-15);  
axis vis3d;  
colormap copper
```



## See Also

`isonormals`, `shrinkfaces`, `smooth3`, `subvolume`

“Connecting Equal Values with Isosurfaces” for more examples

“Volume Visualization” on page 1-106 for related functions



**Purpose** Determine if version is for Windows (PC) platform

**Syntax** `tf = ispc`

**Description** `tf = ispc` returns logical 1 (true) if the version of MATLAB software is for the Microsoft Windows platform, and returns logical 0 (false) otherwise.

**See Also** `isunix`, `ismac`, `isstudent`, `is*`

# ispref

---

**Purpose** Test for existence of preference

**Syntax**

```
ispref('group','pref')  
ispref('group')  
ispref('group',{'pref1','pref2',... 'prefn'})
```

**Description**

`ispref('group','pref')` returns 1 if the preference specified by `group` and `pref` exists, and 0 otherwise.

`ispref('group')` returns 1 if the `GROUP` exists, and 0 otherwise.

`ispref('group',{'pref1','pref2',... 'prefn'})` returns a logical array the same length as the cell array of preference names, containing 1 where each preference exists, and 0 elsewhere.

**Examples**

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

ans =  
1.0

**See Also** `addpref`, `getpref`, `rmpref`, `setpref`, `uigetpref`, `uisetpref`

**Purpose** Array elements that are prime numbers

**Syntax** TF = isprime(A)

**Description** TF = isprime(A) returns an array the same size as A containing logical 1 (true) for the elements of A which are prime, and logical 0 (false) otherwise. A must contain only positive integers.

**Examples**

```
c = [2 3 0 6 10]

c =
     2     3     0     6    10

isprime(c)

ans =
     1     1     0     0     0
```

**See Also** is\*

# isprop

---

**Purpose** Determine whether input is COM object property

**Syntax** `isprop(h, 'name')`

**Description** `isprop(h, 'name')` returns logical 1 (true) if the specified name is a property you can use with COM object h. Otherwise, `isprop` returns logical 0 (false).

**Examples** Create a Microsoft Excel application and test to see if `UsableWidth` is a property of the object. `isprop` returns true:

```
h = actxserver ('Excel.Application');  
  
isprop(h, 'UsableWidth')
```

MATLAB software displays:

```
ans =  
    1
```

Try the same test on `SaveWorkspace`, which is a method, and `isprop` returns false:

```
isprop(h, 'SaveWorkspace')
```

MATLAB displays:

```
ans =  
    0
```

**See Also** `get(COM)`, `inspect`, `addproperty`, `deleteproperty`, `ismethod`, `isevent`, `isobject`, `methods`, `class`

---

<b>Purpose</b>	Check if input is real array
<b>Syntax</b>	TF = isreal(A)
<b>Description</b>	TF = isreal(A) returns logical 1 (true) if A does not have an imaginary part. It returns logical 0 (false) otherwise. If A has a stored imaginary part of value 0, isreal(A) returns logical 0 (false).

---

**Note** For logical and char data classes, isreal always returns true. For numeric data types, if A does not have an imaginary part isreal returns true; if A does have an imaginary part isreal returns false. For cell, struct, function\_handle, and object data types, isreal always returns false.

---

`~isreal(x)` returns true for arrays that have at least one element with an imaginary component. The value of that component can be 0.

<b>Remarks</b>	<p>If A is real, <code>complex(A)</code> returns a complex number whose imaginary component is 0, and <code>isreal(complex(A))</code> returns false. In contrast, the addition <code>A + 0i</code> returns the real value A, and <code>isreal(A + 0i)</code> returns true.</p> <p>If B is real and <code>A = complex(B)</code>, then A is a complex matrix and <code>isreal(A)</code> returns false, while <code>A(m:n)</code> returns a real matrix and <code>isreal(A(m:n))</code> returns true.</p> <p>Because MATLAB software supports complex arithmetic, certain of its functions can introduce significant imaginary components during the course of calculations that appear to be limited to real numbers. Thus, you should use <code>isreal</code> with discretion.</p>
----------------	---

<b>Examples</b>	<b>Example 1</b> <p>If a computation results in a zero-value imaginary component, <code>isreal</code> returns true.</p>
-----------------	---

# isreal

---

```
x=3+4i;  
y=5-4i;  
isreal(x+y)
```

```
ans =
```

```
1
```

## Example 2

These examples use `isreal` to detect the presence or absence of imaginary numbers in an array. Let

```
x = magic(3);  
y = complex(x);
```

`isreal(x)` returns `true` because no element of `x` has an imaginary component.

```
isreal(x)  
ans =  
1
```

`isreal(y)` returns `false`, because every element of `x` has an imaginary component, even though the value of the imaginary components is 0.

```
isreal(y)  
ans =  
0
```

This expression detects strictly real arrays, i.e., elements with 0-valued imaginary components are treated as real.

```
~any(imag(y(:)))  
ans =  
1
```

## Example 3

Given the following cell array,

```
C{1} = pi; % double
C{2} = 'John Doe'; % char array
C{3} = 2 + 4i; % complex double
C{4} = ispc; % logical
C{5} = magic(3); % double array
C{6} = complex(5,0) % complex double

C =
    [3.1416] 'John Doe' [2.0000+ 4.0000i] [1] [3x3 double] [5]
```

isreal shows that all but C{1,3} and C{1,6} are real arrays.

```
for k = 1:6
    x(k) = isreal(C{k});
end

x
x =
     1     1     0     1     1     0
```

## See Also

complex, isnumeric, isnan, isprime, isfinite, isinf, isa, is\*

# isscalar

---

**Purpose** Determine whether input is scalar

**Syntax** TF = isscalar(A)

**Description** TF = isscalar(A) returns logical 1 (true) if A is a 1-by-1 matrix, and logical 0 (false) otherwise.

The A argument can be a structure or cell array. It also be a MATLAB object, as described in *Object-Oriented Programming*, as long as that object overloads the size function.

**Examples** Test matrix A and one element of the matrix:

```
A = rand(5);  
  
isscalar(A)  
ans =  
    0  
  
isscalar(A(3,2))  
ans =  
    1
```

**See Also** isvector, isempty, isnumeric, islogical, ischar, isa, is\*



**Purpose** Determine whether set elements are in sorted order

**Syntax** TF = issorted(A)  
TF = issorted(A, 'rows')

**Description** TF = issorted(A) returns logical 1 (true) if the elements of A are in sorted order, and logical 0 (false) otherwise. Input A can be a vector or an N-by-1 or 1-by-N cell array of strings. A is considered to be sorted if A and the output of sort(A) are equal.

TF = issorted(A, 'rows') returns logical 1 (true) if the rows of two-dimensional matrix A are in sorted order, and logical 0 (false) otherwise. Matrix A is considered to be sorted if A and the output of sortrows(A) are equal.

**Note** Only the issorted(A) syntax supports A as a cell array of strings.

**Remarks** For character arrays, issorted uses ASCII, rather than alphabetical, order.

You cannot use issorted on arrays of greater than two dimensions.

**Examples** **Example 1 – Using issorted on a vector**

```
A = [5 12 33 39 78 90 95 107 128 131];
```

```
issorted(A)
ans =
     1
```

**Example 2 – Using issorted on a matrix**

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
```

# issorted

---

```
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

```
issorted(A, 'rows')
ans =
     0
```

```
B = sortrows(A)
B =
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
    17    24     1     8    15
    23     5     7    14    16
```

```
issorted(B)
ans =
     1
```

## Example 3 – Using issorted on a cell array

```
x = {'one'; 'two'; 'three'; 'four'; 'five'};
issorted(x)
ans =
     0
```

```
y = sort(x)
y =
    'five'
    'four'
    'one'
    'three'
    'two'
```

```
issorted(y)
```

**See Also**

sort, sortrows, ismember, unique, intersect, union, setdiff, setxor, is\*

# isspace

---

**Purpose** Array elements that are space characters

**Syntax** `tf = isspace('str')`

**Description** `tf = isspace('str')` returns an array the same size as 'str' containing logical 1 (true) where the elements of str are ASCII white spaces and logical 0 (false) where they are not. White spaces in ASCII are space, newline, carriage return, tab, vertical tab, or formfeed characters.

**Examples**

```
isspace(' Find spa ces ')
Columns 1 through 13
    1    1    0    0    0    0    1    0    0    0    1    0    0
Columns 14 through 15
    0    1
```

**See Also** `isletter`, `isstrprop`, `ischar`, `strings`, `isa`, `is*`

**Purpose** Determine whether input is sparse

**Syntax** TF = issparse(S)

**Description** TF = issparse(S) returns logical 1 (true) if the storage class of S is sparse and logical 0 (false) otherwise.

**See Also** is\*, sparse, full

# isstr

---

**Purpose** Determine whether input is character array

---

**Note** Use the `ischar` function in place of `isstr`. The `isstr` function will be removed in a future version of MATLAB.

---

**See Also** `ischar`, `isa`, `is*`

**Purpose** Determine whether string is of specified category

**Syntax** `tf = isstrprop('str', 'category')`

**Description** `tf = isstrprop('str', 'category')` returns a logical array the same size as `str` containing logical 1 (true) where the elements of `str` belong to the specified `category`, and logical 0 (false) where they do not.

The `str` input can be a character array, cell array, or any MATLAB numeric type. If `str` is a cell array, then the return value is a cell array of the same shape as `str`.

The `category` input can be any of the strings shown in the left column below:

Category	Description
alpha	True for those elements of <code>str</code> that are alphabetic
alphanum	True for those elements of <code>str</code> that are alphanumeric
cntrl	True for those elements of <code>str</code> that are control characters (for example, <code>char(0:20)</code> )
digit	True for those elements of <code>str</code> that are numeric digits
graphic	True for those elements of <code>str</code> that are graphic characters. These are all values that represent any characters except for the following:  unassigned, space, line separator, paragraph separator, control characters, Unicode format control characters, private user-defined characters, Unicode surrogate characters, Unicode other characters
lower	True for those elements of <code>str</code> that are lowercase letters
print	True for those elements of <code>str</code> that are graphic characters, plus <code>char(32)</code>

Category	Description
punct	True for those elements of <code>str</code> that are punctuation characters
wspace	True for those elements of <code>str</code> that are white-space characters. This range includes the ANSI C definition of white space, {' ', '\t', '\n', '\r', '\v', '\f'}.
upper	True for those elements of <code>str</code> that are uppercase letters
xdigit	True for those elements of <code>str</code> that are valid hexadecimal digits

## Remarks

Numbers of type `double` are converted to `int32` according to MATLAB rules of double-to-integer conversion. Numbers of type `int64` and `uint64` bigger than `int32(inf)` saturate to `int32(inf)`.

MATLAB classifies the elements of the `str` input according to the Unicode definition of the specified category. If the numeric value of an element in the input array falls within the range that defines a Unicode character category, then this element is classified as being of that category. The set of Unicode character codes includes the set of ASCII character codes, but also covers a large number of languages beyond the scope of the ASCII set. The classification of characters is dependent on the global location of the platform on which MATLAB is installed.

## Examples

Test for alphabetic characters in a string:

```
A = isstrprop('abc123def', 'alpha')
A =
    1  1  1  0  0  0  1  1  1
```

Test for numeric digits in a string:

```
A = isstrprop('abc123def', 'digit')
A =
    0  0  0  1  1  1  0  0  0
```



Test for hexadecimal digits in a string:

```
A = isstrprop('abcd1234efgh', 'xdigit')
A =
    1 1 1 1 1 1 1 1 1 1 0 0
```

Test for numeric digits in a character array:

```
A = isstrprop(char([97 98 99 49 50 51 101 102 103]), ...
               'digit')
A =
    0 0 0 1 1 1 0 0 0
```

Test for alphabetic characters in a two-dimensional cell array:

```
A = isstrprop({'abc123def'; '456ghi789'}, 'alpha')
A =
    [1x9 logical]
    [1x9 logical]

A{:,:}
ans =
    1 1 1 0 0 0 1 1 1
    0 0 0 1 1 1 0 0 0
```

Test for white-space characters in a string:

```
A = isstrprop(sprintf('a bc\n'), 'wspace')
A =
    0 1 0 0 1
```

## See Also

strings, ischar, isletter, isspace, iscellstr, isnumeric, isa, is\*

# isstruct

---

**Purpose** Determine whether input is structure array

**Syntax** `tf = isstruct(A)`

**Description** `tf = isstruct(A)` returns logical 1 (true) if A is a MATLAB structure and logical 0 (false) otherwise.

**Examples**

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];

isstruct(patient)

ans =

     1
```

**See Also** `struct`, `isfield`, `iscell`, `ischar`, `isobject`, `isnumeric`, `islogical`, `isa`, `is*`, dynamic field names

**Purpose** Determine if version is Student Version

**Syntax** `tf = isstudent`

**Description** `tf = isstudent` returns logical 1 (true) if the version of MATLAB software is the Student Version, and returns logical 0 (false) for commercial versions.

**See Also** `ver`, `version`, `license`, `ispc`, `isunix`, `is*`

# isunix

---

**Purpose** Determine if version is for UNIX platform.

**Syntax** `tf = isunix`

**Description** `tf = isunix` returns logical 1 (true) if the version of MATLAB software is for the UNIX<sup>16</sup> platform, and returns logical 0 (false) otherwise.

**See Also** `ispc`, `ismac`, `isstudent`, `is*`

16. UNIX is a registered trademark of The Open Group in the United States and other countries

**Purpose** Is object valid handle object

**Syntax** H1 = isvalid(Hobj)

**Description** H1 = isvalid(Hobj) returns a logical array (or scalar if Hobj is scalar) in which each element is true if the corresponding element in Hobj is a valid handle.

**See Also** delete (handle)

# isvalid (serial)

---

**Purpose** Determine whether serial port objects are valid

**Syntax** `out = isvalid(obj)`

**Description** `out = isvalid(obj)` returns the logical array `out`, which contains a 0 where the elements of the serial port object, `obj` are invalid serial port objects and a 1 where the elements of `obj` are valid serial port objects.

**Remarks** `obj` becomes invalid after it is removed from memory with the `delete` function. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the `clear` command.

**Example** Suppose you create the following two serial port objects.

```
s1 = serial('COM1');  
s2 = serial('COM1');
```

`s2` becomes invalid after it is deleted.

```
delete(s2)
```

`isvalid` verifies that `s1` is valid and `s2` is invalid.

```
sarray = [s1 s2];  
isvalid(sarray)  
ans =  
     1     0
```

**See Also** **Functions**

`clear`, `delete`

**Purpose** Determine whether timer object is valid

**Syntax** `out = isvalid(obj)`

**Description** `out = isvalid(obj)` returns a logical array, `out`, that contains a 0 where the elements of `obj` are invalid timer objects and a 1 where the elements of `obj` are valid timer objects.

An invalid timer object is an object that has been deleted and cannot be reused. Use the `clear` command to remove an invalid timer object from the workspace.

**Examples** Create a valid timer object.

```
t = timer;  
out = isvalid(t)  
out =  
  
1
```

Delete the timer object, making it invalid.

```
delete(t)  
out1 = isvalid(t)  
out1 =  
  
0
```

**See Also** `timer`, `delete(timer)`

# isvarname

---

**Purpose** Determine whether input is valid variable name

**Syntax** `tf = isvarname('str')\`  
`isvarname str`

**Description** `tf = isvarname('str')\` returns logical 1 (true) if the string `str` is a valid MATLAB variable name and logical 0 (false) otherwise. A valid variable name is a character string of letters, digits, and underscores, totaling not more than `namelengthmax` characters and beginning with a letter.

MATLAB keywords are not valid variable names. Type the command `iskeyword` with no input arguments to see a list of MATLAB keywords.

`isvarname str` uses the MATLAB command format.

## Examples

This variable name is valid:

```
isvarname foo
ans =
     1
```

This one is not because it starts with a number:

```
isvarname 8th_column
ans =
     0
```

If you are building strings from various pieces, place the construction in parentheses.

```
d = date;

isvarname(['Monday_', d(1:2)])
ans =
     1
```

## See Also

`genvarname`, `isglobal`, `iskeyword`, `namelengthmax`, `is*`



**Purpose** Determine whether input is vector

**Syntax** `TF = isvector(A)`

**Description** `TF = isvector(A)` returns logical 1 (true) if A is a 1-by-N or N-by-1 vector where  $N \geq 0$ , and logical 0 (false) otherwise.

The A argument can also be a MATLAB object, as described in *Object-Oriented Programming*, as long as that object overloads the `size` function.

**Examples** Test matrix A and its row and column vectors:

```
A = rand(5);

isvector(A)
ans =
     0

isvector(A(3, :))
ans =
     1

isvector(A(:, 2))
ans =
     1
```

**See Also** `isscalar`, `isempty`, `isnumeric`, `islogical`, `ischar`, `isa`, `is*`

**Purpose**            Imaginary unit

**Syntax**            j  
                      x+yj  
                      x+j\*y

**Description**      Use the character j in place of the character i, if desired, as the imaginary unit.

As the basic imaginary unit  $\sqrt{-1}$ , j is used to enter complex numbers. Since j is a function, it can be overridden and used as a variable. This permits you to use j as an index in for loops, etc.

It is possible to use the character j without a multiplication sign as a suffix in forming a numerical constant.

**Examples**            Z = 2+3j  
                          Z = x+j\*y  
                          Z = r\*exp(j\*theta)

**See Also**            conj, i, imag, real

**Purpose** Add entries to dynamic Sun Java class path

**Syntax**  
`javaaddpath('dpath')`  
`javaaddpath('dpath', '-end')`

**Description**  
`javaaddpath('dpath')` adds one or more directories or JAR files to the beginning of the current dynamic Java class path. `dpath` is a string or cell array of strings containing the directory or JAR file. (See the Remarks section for a description of static and dynamic Java paths.)  
`javaaddpath('dpath', '-end')` adds one or more directories or files to the end of the current dynamic Java path.

**Remarks**  
The Java path consists of two segments: a static path (read only at startup) and a dynamic path. The MATLAB software always searches the static path (defined in `classpath.txt`) before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path. Use `javaclasspath` to see the current static and dynamic Java paths.

MATLAB calls the `clear java` command whenever you change the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the file <code>classpath.txt</code> . The static Java path offers better Java class loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file <code>classpath.txt</code> and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the <code>javaclasspath</code> function. You can define the dynamic path (using <code>javaclasspath</code> ), modify the path (using <code>javaaddpath</code> and <code>javarmpath</code> ), and refresh the Java class definitions for all classes on the dynamic path (using <code>clear java</code> ) without restarting MATLAB.

# javaaddpath

---

## Examples

Create function to set initial dynamic Java class path:

```
function setdynpath
javaclasspath({
    'C:\Work\Java\ClassFiles', ...
    'C:\Work\JavaTest\curvefit.jar', ...
    'C:\Work\JavaTest\timer.jar', ...
    'C:\Work\JavaTest\patch.jar'});
% end of file
```

Call this function to set up your dynamic class path. Then, use the javaclasspath function with no arguments to display all current static and dynamic paths:

```
setdynpath;
```

```
javaclasspath
```

### STATIC JAVA PATH

```
D:\Sys0\Java\util.jar
D:\Sys0\Java\widgets.jar
D:\Sys0\Java\beans.jar
.
.
```

### DYNAMIC JAVA PATH

```
C:\Work\Java\ClassFiles
C:\Work\JavaTest\curvefit.jar
C:\Work\JavaTest\timer.jar
C:\Work\JavaTest\patch.jar
```

At some later time, add the following two entries to the dynamic path. (Calling javaaddpath clears all variables from the workspace). One entry specifies a directory and the other a Java Archive (JAR) file.

When you add a directory to the path, MATLAB includes all files in that directory as part of the path:

```
javaaddpath({
    'C:\Work\Java\Curvefit\Test', ...
    'C:\Work\Java\mywidgets.jar'});
```

Use `javaclasspath` with just an output argument to return the dynamic path alone:

```
p = javaclasspath
p =
    'C:\Work\Java\ClassFiles'
    'C:\Work\JavaTest\curvefit.jar'
    'C:\Work\JavaTest\timer.jar'
    'C:\Work\JavaTest\patch.jar'
    'C:\Work\Java\Curvefit\Test'
    'C:\Work\Java\mywidgets.jar'
```

Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

If you modify one or more classes that are defined on the dynamic path, you need to clear the former definition for those classes from MATLAB memory. You can clear all dynamic Java class definitions from memory using:

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

Use `javarmpath` to remove a file or directory from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

# javaaddpath

---

## Other Examples

Add a JAR file from an internet URL to your dynamic Java path:

```
javaaddpath http://www.example.com/my.jar
```

Add the current directory with the following statement:

```
javaaddpath(pwd)
```

## See Also

`javaclasspath`, `javarmpath`, `clear`

See “Bringing Java Classes and Methods into MATLAB Workspace” for more information.

**Purpose** Construct Sun Java array

**Syntax** `javaArray('package_name.class_name',x1,...,xn)`

**Description** `javaArray('package_name.class_name',x1,...,xn)` constructs an empty Java array capable of storing objects of Java class, '*class\_name*'. The dimensions of the array are *x1* by ... by *xn*. You must include the package name when specifying the class.

The array that you create with `javaArray` is equivalent to the array that you would create with the Java code

```
A = new class_name[x1]...[xn];
```

**Examples** The following example constructs and populates a 4-by-5 array of `java.lang.Double` objects.

```
dblArray = javaArray ('java.lang.Double', 4, 5);
for m = 1:4
    for n = 1:5
        dblArray(m,n) = java.lang.Double((m*10) + n);
    end
end
```

```
dblArray
```

```
dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

**See Also** `javaObject`, `javaMethod`, `class`, `methodsview`, `isjava`

# javachk

---

**Purpose** Generate error message based on Sun Java feature support

**Syntax**  
`javachk(feature)`  
`javachk(feature, component)`

**Description** `javachk(feature)` returns a generic error message if the specified Java feature is not available in the current MATLAB session. If it is available, `javachk` returns an empty matrix. Possible feature arguments are shown in the following table.

Feature	Description
'awt'	Abstract Window Toolkit components <sup>1</sup> are available.
'desktop'	The MATLAB interactive desktop is running.
'jvm'	The Java Virtual Machine software(JVM™) is running.
'swing'	Swing components <sup>2</sup> are available.

1. Java GUI components in the Abstract Window Toolkit
2. Java lightweight GUI components in the Java Foundation Classes

`javachk(feature, component)` works the same as the above syntax, except that the specified component is also named in the error message. (See the example below.)

## Examples

The following M-file displays an error with the message "CreateFrame is not supported on this platform." when run in a MATLAB session in which the AWT's GUI components are not available. The second argument to `javachk` specifies the name of the M-file, which is then included in the error message generated by MATLAB.



```
javamsg = javachk('awt', mfilename);  
if isempty(javamsg)  
    myFrame = java.awt.Frame;  
    myFrame.setVisible(1);  
else  
    error(javamsg);  
end
```

**See Also**      usejava

# javaclasspath

---

**Purpose** Set and get dynamic Sun Java class path

**Syntax**

```
javaclasspath
javaclasspath(dpath)
dpath = javaclasspath
spath = javaclasspath('-static')
jpath = javaclasspath('-all')
javaclasspath(statusmsg)
```

**Description** javaclasspath displays the static and dynamic segments of the Java path. (See the Remarks section, below, for a description of static and dynamic Java paths.)

javaclasspath(dpath) sets the dynamic Java path to one or more directory or file specifications given in dpath, where dpath can be a string or cell array of strings. Uses the clear java command to refresh the classes defined on the dynamic Java path.

dpath = javaclasspath returns the dynamic segment of the Java path in cell array, dpath. If no dynamic paths are defined, javaclasspath returns an empty cell array.

spath = javaclasspath('-static') returns the static segment of the Java path in cell array, spath. No path information is displayed unless you specify an output variable. If no static paths are defined, javaclasspath returns an empty cell array.

jpath = javaclasspath('-all') returns the entire Java path in cell array, jpath. The returned cell array contains first the static segment of the path, and then the dynamic segment. No path information is displayed unless you specify an output variable. If no dynamic paths are defined, javaclasspath returns an empty cell array.

javaclasspath(*statusmsg*) enables or disables the display of status messages from the javaclasspath, javaaddpath, and javarmpath functions. Values for the *statusmsg* argument are shown in the following table:

statusmsg	Description
' -v1 '	Display status messages while loading the Java path from the file system.
' -v0 '	Do not display status messages. This is the default.

## Remarks

The Java path consists of two segments: a static path (read only at startup) and a dynamic path. The MATLAB software always searches the static path (defined in `classpath.txt`) before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path. Use `javaclasspath` to see the current static and dynamic Java paths.

MATLAB calls the `clear java` command whenever you change the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the file <code>classpath.txt</code> . The static Java path offers better Java class loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file <code>classpath.txt</code> and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the <code>javaclasspath</code> function. You can define the dynamic path (using <code>javaclasspath</code> ), modify the path (using <code>javaaddpath</code> and <code>javarmpath</code> ), and refresh the Java class definitions for all classes on the dynamic path (using <code>clear java</code> ) without restarting MATLAB.

## Examples

Create a function to set your initial dynamic Java class path:

```
function setdynpath
javaclasspath({
    'C:\Work\Java\ClassFiles', ...
    'C:\Work\JavaTest\curvefit.jar', ...
```

# javaclasspath

---

```
        'C:\Work\JavaTest\timer.jar', ...  
        'C:\Work\JavaTest\patch.jar'});  
%                end of file
```

Call this function to set up your dynamic class path. Then, use the `javaclasspath` function with no arguments to display all current static and dynamic paths:

```
setdynpath;  
  
javaclasspath  
  
        STATIC JAVA PATH  
  
        D:\Sys0\Java\util.jar  
        D:\Sys0\Java\widgets.jar  
        D:\Sys0\Java\beans.jar  
        .  
        .  
  
        DYNAMIC JAVA PATH  
  
        C:\Work\Java\ClassFiles  
        C:\Work\JavaTest\curvefit.jar  
        C:\Work\JavaTest\timer.jar  
        C:\Work\JavaTest\patch.jar
```

At some later time, add the following two entries to the dynamic path. One entry specifies a directory and the other a Java Archive (JAR) file. When you add a directory to the path, The MATLAB software includes all files in that directory as part of the path:

```
javaaddpath(  
    'C:\Work\Java\Curvefit\Test', ...  
    'C:\Work\Java\mywidgets.jar');
```

Use `javaclasspath` with just an output argument to return the dynamic path alone:

```
p = javaclasspath
p =
'C:\Work\Java\ClassFiles'
'C:\Work\JavaTest\curvefit.jar'
'C:\Work\JavaTest\timer.jar'
'C:\Work\JavaTest\patch.jar'
'C:\Work\Java\Curvefit\Test'
'C:\Work\Java\mywidgets.jar'
```

Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

If, at some time, you modify one or more classes that are defined on the dynamic path, you will need to clear the former definition for those classes from MATLAB memory. You can clear all dynamic Java class definitions from memory using:

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

Use `javarmpath` to remove a file or directory from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

## See Also

`javaaddpath`, `javarmpath`, `clear`

# javaMethod

---

## Purpose

Invoke Sun Java method

## Syntax

```
javaMethod('MethodName','ClassName',x1,...,xn)
javaMethod('MethodName',J,x1,...,xn)
```

## Description

`javaMethod('MethodName','ClassName',x1,...,xn)` invokes the static method `MethodName` in the class `ClassName`, with the argument list `x1,...,xn`.

`javaMethod('MethodName',J,x1,...,xn)` invokes the nonstatic method `MethodName` on the object `J`, with the argument list `x1,...,xn`.

## Remarks

Use the `javaMethod` function to:

- Use methods having names longer than 31 characters.
- Specify the method you want to invoke at run-time, for example, as input from an application user.

The `javaMethod` function enables you to use methods having names longer than 31 characters. This is the only way you can invoke such a method in the MATLAB software. For example:

```
javaMethod('DataDefinitionAndDataManipulationTransactions', T);
```

With `javaMethod`, you can also specify the method to be invoked at run-time. In this situation, your code calls `javaMethod` with a string variable in place of the method name argument. When you use `javaMethod` to invoke a static method, you can also use a string variable in place of the class name argument.

---

**Note** Typically, you do not need to use `javaMethod`. Use the default MATLAB syntax for invoking a Java methods instead. Use `javaMethod` for the cases described above.

---

## Examples

To invoke the static Java method `isNaN` on class, `java.lang.Double`, type:

```
javaMethod('isNaN', 'java.lang.Double', 2.2)
```

The following example invokes the nonstatic method `setMonth`, where `myDate` is a `java.util.Date` object.

```
myDate = java.util.Date;  
javaMethod('setMonth', myDate, 3);
```

## See Also

[javaArray](#), [javaObject](#), [import](#), [methods](#), [isjava](#)

# javaObject

---

**Purpose** Construct Sun Java object

**Syntax** `javaObject('ClassName',x1,...,xn)`

**Description** `javaObject('ClassName',x1,...,xn)` invokes the Java constructor for class 'ClassName' with the argument list that matches `x1,...,xn`, to return a new object.

If there is no constructor that matches the class name and argument list passed to `javaObject`, an error occurs.

**Remarks** Use the `javaObject` function to:

- Use classes having names with more than 31 consecutive characters.
- Specify the class for an object at run-time, for example, as input from an application user.

The default MATLAB constructor syntax requires that no segment of the input class name be longer than 31 characters. (A *name segment*, is any portion of the class name before, between, or after a period. For example, there are three segments in class, `java.lang.String`.) Any class name segment that exceeds 31 characters is truncated by MATLAB. In the rare case where you need to use a class name of this length, you must use `javaObject` to instantiate the class.

The `javaObject` function also allows you to specify the Java class for the object being constructed at run-time. In this situation, you call `javaObject` with a string variable in place of the class name argument.

```
class = 'java.lang.String';
text = 'hello';
strObj = javaObject(class, text);
```

In the usual case, when the class to instantiate is known at development time, it is more convenient to use the MATLAB constructor syntax. For example, to create a `java.lang.String` object, type:



```
strObj = java.lang.String('hello');
```

---

**Note** Typically, you do not need to use `javaObject`. Use the default MATLAB syntax for instantiating a Java class instead. Use `javaObject` for the cases described above.

---

## Examples

The following example constructs and returns a Java object of class `java.lang.String`:

```
strObj = javaObject('java.lang.String','hello')
```

## See Also

`javaArray`, `javaMethod`, `import`, `methods`, `fieldnames`, `isjava`

# javarmpath

---

**Purpose** Remove entries from dynamic Sun Java class path

**Syntax**

```
javarmpath('dpath')
javarmpath dpath1 dpath2 ... dpathN
javarmpath(v1, v2, ..., vN)
```

**Description** `javarmpath('dpath')` removes a directory or file from the current dynamic Java path. `dpath` is a string containing the directory or file specification. (See the Remarks section, below, for a description of static and dynamic Java paths.)

`javarmpath dpath1 dpath2 ... dpathN` removes those directories and files specified by `dpath1`, `dpath2`, ..., `dpathN` from the dynamic Java path. Each input argument is a string containing a directory or file specification.

`javarmpath(v1, v2, ..., vN)` removes those directories and files specified by `v1`, `v2`, ..., `vN` from the dynamic Java path. Each input argument is a variable to which a directory or file specification is assigned.

**Remarks** The Java path consists of two segments: a static path (read only at startup) and a dynamic path. The MATLAB software always searches the static path (defined in `classpath.txt`) before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path. Use `javaclasspath` to see the current static and dynamic Java paths.

MATLAB calls the `clear java` command whenever you change the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the file <code>classpath.txt</code> . The static Java path offers better Java class loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file <code>classpath.txt</code> and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the <code>javaclasspath</code> function. You can define the dynamic path (using <code>javaclasspath</code> ), modify the path (using <code>javaaddpath</code> and <code>javarmpath</code> ), and refresh the Java class definitions for all classes on the dynamic path (using <code>clear java</code> ) without restarting MATLAB.

## Examples

Create a function to set your initial dynamic Java class path:

```
function setdynpath
javaclasspath({
    'C:\Work\Java\ClassFiles', ...
    'C:\Work\JavaTest\curvefit.jar', ...
    'C:\Work\JavaTest\timer.jar', ...
    'C:\Work\JavaTest\patch.jar'});
% end of file
```

Call this function to set up your dynamic class path. Then, use the `javaclasspath` function with no arguments to display all current static and dynamic paths:

```
setdynpath;

javaclasspath

    STATIC JAVA PATH

D:\Sys0\Java\util.jar
D:\Sys0\Java\widgets.jar
D:\Sys0\Java\beans.jar
```

```
·  
·
```

## DYNAMIC JAVA PATH

```
C:\Work\Java\ClassFiles  
C:\Work\JavaTest\curvefit.jar  
C:\Work\JavaTest\timer.jar  
C:\Work\JavaTest\patch.jar
```

At some later time, add the following two entries to the dynamic path. One entry specifies a directory and the other a Java Archive (JAR) file. When you add a directory to the path, MATLAB includes all files in that directory as part of the path:

```
javaaddpath(  
    'C:\Work\Java\Curvefit\Test', ...  
    'C:\Work\Java\mywidgets.jar' });
```

Use `javaclasspath` with just an output argument to return the dynamic path alone:

```
p = javaclasspath  
p =  
    'C:\Work\Java\ClassFiles'  
    'C:\Work\JavaTest\curvefit.jar'  
    'C:\Work\JavaTest\timer.jar'  
    'C:\Work\JavaTest\patch.jar'  
    'C:\Work\Java\Curvefit\Test'  
    'C:\Work\Java\mywidgets.jar'
```

Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

If, at some time, you modify one or more classes that are defined on the dynamic path, you will need to clear the former definition for those classes from MATLAB memory. You can clear all dynamic Java class definitions from memory using:

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

Use `javarmpath` to remove a file or directory from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

## See Also

`javaclasspath`, `javaaddpath`, `clear`

# keyboard

---

**Purpose** Input from keyboard

**Syntax** keyboard

**Description** keyboard , when placed in an M-file, stops execution of the file and gives control to the keyboard. The special status is indicated by a K appearing before the prompt. You can examine or change variables; all MATLAB commands are valid. This keyboard mode is useful for debugging your M-files..

To terminate the keyboard mode, type the command

```
return
```

then press the **Return** key.

**See Also** dbstop, input, quit, pause, return

**Purpose** Return all keys of containers.Map object

**Syntax** `k = keys(M)`

**Description** `k = keys(M)` returns cell array `k` that contains all of the keys stored in Map object `M`.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

**Examples** Construct a Map object that relates states in the United States to their capital cities:

```
US_Capitals = containers.Map( ...
    {'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...
    {'Atlanta', 'Juneau', 'Montpelier', 'Salem'})
```

Use the keys and values methods to list all keys and values in the map:

```
keys(US_Capitals)
ans =
    'Arizona'    'Nebraska'    'New York'    'Oregon'

values(US_Capitals)
ans =
    'Phoenix'    'Lincoln'    'Albany'    'Salem'
```

Use the map to look up a capital when given a specific state:

```
sprintf('    The capital of %s is %s', ...
    'Alaska', US_Capitals('Alaska'))
ans =
    The capital of Alaska is Juneau
```

**See Also** `containers.Map`, `values(Map)`, `size(Map)`, `length(Map)`, `isKey(Map)`, `remove(Map)`, `handle`

# kron

---

**Purpose** Kronecker tensor product

**Syntax** `K = kron(X,Y)`

**Description** `K = kron(X,Y)` returns the Kronecker tensor product of `X` and `Y`. The result is a large array formed by taking all possible products between the elements of `X` and those of `Y`. If `X` is `m`-by-`n` and `Y` is `p`-by-`q`, then `kron(X,Y)` is `m*p`-by-`n*q`.

**Examples** If `X` is 2-by-3, then `kron(X,Y)` is

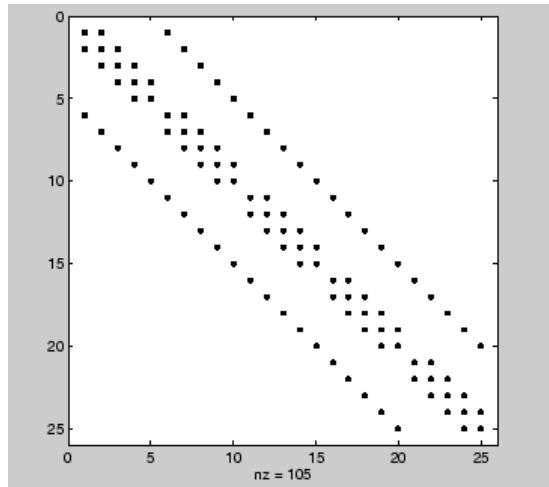
```
[ X(1,1)*Y X(1,2)*Y X(1,3)*Y
  X(2,1)*Y X(2,2)*Y X(2,3)*Y ]
```

The matrix representation of the discrete Laplacian operator on a two-dimensional, `n`-by-`n` grid is a `n^2`-by-`n^2` sparse matrix. There are at most five nonzero elements in each row or column. The matrix can be generated as the Kronecker product of one-dimensional difference operators with these statements:

```
I = speye(n,n);
E = sparse(2:n,1:n-1,1,n,n);
D = E+E' -2*I;
A = kron(D,I)+kron(I,D);
```

Plotting this with the `spy` function for `n = 5` yields:





**See Also** hankel, toeplitz

# last (MException)

---

**Purpose** Last uncaught exception

**Syntax** ME = MException.last  
MException.last('reset')

**Description** ME = MException.last displays the contents of the MException object representing your most recent uncaught error. This is a static method of the MException class; it is not a method of an MException class object. Use this method from the MATLAB command line only, and not within an M-file.

MException.last('reset') sets the identifier and message properties of the most recent exception to the empty string, the stack property to a 0-by-1 structure, and cause property to an empty cell array.

last is not set in a try-catch statement.

**Examples** This example displays the last error that was caught during this MATLAB session:

```
A = 25;  
A(2)  
??? Index exceeds matrix dimensions.
```

```
MException.last  
ans =
```

MException object with properties:

```
    identifier: 'MATLAB:badsubscript'  
    message: 'Index exceeds matrix dimensions.'  
    stack: [0x1 struct]  
    cause: {}
```

**See Also** try, catch, error, assert, MException, throw(MException), rethrow(MException), throwAsCaller(MException),

```
addCause(MException), getReport(MException), disp(MException),  
isequal(MException), eq(MException), ne(MException)
```

# lasterr

---

## Purpose

Last error message

---

**Note** `lasterr` has been replaced by `lasterror`, but will be maintained for backward compatibility.

---

## Syntax

```
msgstr = lasterr
[msgstr, msgid] = lasterr
lasterr('new_msgstr')
lasterr('new_msgstr', 'new_msgid')
[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')
```

## Description

`msgstr = lasterr` returns the last error message generated by the MATLAB software.

`[msgstr, msgid] = lasterr` returns the last error in `msgstr` and its message identifier in `msgid`. If the error was not defined with an identifier, `lasterr` returns an empty string for `msgid`. See “Message Identifiers” in the MATLAB Programming Fundamentals documentation for more information on the `msgid` argument and how to use it.

`lasterr('new_msgstr')` sets the last error message to a new string, `new_msgstr`, so that subsequent invocations of `lasterr` return the new error message string. You can also set the last error to an empty string with `lasterr('')`.

`lasterr('new_msgstr', 'new_msgid')` sets the last error message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lasterr` return the new error message and message identifier.

`[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')` returns the last error message and its identifier, also changing these values so that subsequent invocations of `lasterr` return the message and identifier strings specified by `new_msgstr` and `new_msgid` respectively.

## Examples

### Example 1

Here is a function that examines the `lasterr` string and displays its own message based on the error that last occurred. This example deals with two cases, each of which is an error that can result from a matrix multiply:

```
function matrix_multiply(A, B)
try
    A * B
catch
    errmsg = lasterr;
    if(strfind(errmsg, 'Inner matrix dimensions'))
        disp('** Wrong dimensions for matrix multiply')
    else
        if(strfind(errmsg, 'not defined for variables of class'))
            disp('** Both arguments must be double matrices')
        end
    end
end
end
```

If you call this function with matrices that are incompatible for matrix multiplication (e.g., the column dimension of A is not equal to the row dimension of B), MATLAB catches the error and uses `lasterr` to determine its source:

```
A = [1 2 3; 6 7 2; 0 -1 5];
B = [9 5 6; 0 4 9];

matrix_multiply(A, B)
** Wrong dimensions for matrix multiply
```

### Example 2

Specify a message identifier and error message string with error:

```
error('MyToolbox:angleTooLarge', ...
    'The angle specified must be less than 90 degrees.');
```

# lasterr

---

In your error handling code, use `lasterr` to determine the message identifier and error message string for the failing operation:

```
[errmsg, msgid] = lasterr
errmsg =
    The angle specified must be less than 90 degrees.
msgid =
    MyToolbox:angleTooLarge
```

## See Also

`error`, `lasterror`, `rethrow`, `warning`, `lastwarn`

**Purpose** Last error message and related information

**Syntax**

```
s = lasterror
s = lasterror(err)
s = lasterror('reset')
```

**Description** `s = lasterror` returns a structure `s` containing information about the most recent error issued by the MATLAB software. The return structure contains the following fields:

Fieldname	Description
message	Character array containing the text of the error message.
identifier	Character array containing the message identifier of the error message. If the last error issued by MATLAB had no message identifier, then the <code>identifier</code> field is an empty character array.
stack	Structure providing information on the location of the error. The structure has fields <code>file</code> , <code>name</code> , and <code>line</code> , and is the same as the structure returned by the <code>dbstack</code> function. If <code>lasterror</code> returns no stack information, <code>stack</code> is a 0-by-1 structure having the same three fields.

---

**Note** The `lasterror` return structure might contain additional fields in future versions of MATLAB.

---

The fields of the structure returned in `stack` are

# lasterror

---

Fieldname	Description
file	Name of the file in which the function generating the error appears. This field is the empty string if there is no file.
name	Name of the function in which the error occurred. If this is the primary function of the M-file, and the function name differs from the M-file name, name is set to the M-file name.
line	M-file line number where the error occurred.

See “Message Identifiers” in the MATLAB Programming Fundamentals documentation for more information on the syntax and usage of message identifiers.

`s = lasterror(err)` sets the last error information to the error message and identifier specified in the structure `err`. Subsequent invocations of `lasterror` return this new error information. The optional return structure `s` contains information on the previous error.

`s = lasterror('reset')` sets the last error information to the default state. In this state, the `message` and `identifier` fields of the return structure are empty strings, and the `stack` field is a 0-by-1 structure.

## Remarks

The MathWorks is gradually transitioning MATLAB error handling to an object-oriented scheme that is based on the `MException` class. Although support for `lasterror` is expected to continue, using the static `last` method of `MException` is preferable.

## Warning

**`lasterror` and `MException.last` are not guaranteed to always return identical results. For example, `MException.last` updates its error status only on uncaught errors, where `lasterror` can update its error status on any error, whether it is caught or not.**



## Examples

### Example 1

Save the following MATLAB code in an M-file called `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector elements.
% Nonvector input results in an error.
check_inputs(x)
y = sum(x)/length(x);    % The actual computation

function check_inputs(x)
[m,n] = size(x);
if ~(m == 1 || (n == 1) || (m == 1 && n == 1))
    error('AVG:NotAVector', 'Input must be a vector.')
end
```

Now run the function. Because this function requires vector input, passing a scalar value to it forces an error. The error occurs in subroutine `check_inputs`:

```
average(200)
??? Error using ==> average>check_inputs
Input must be a vector.

Error in ==> average at 5
check_inputs(x)
```

Get the three fields from `lasterror`:

```
err = lasterror
err =
    message: [1x61 char]
  identifier: 'AVG:NotAVector'
         stack: [2x1 struct]
```

Display the text of the error message:

```
msg = err.message
```

# lasterror

---

```
msg =  
Error using ==> average>check_inputs  
Input must be a vector.
```

Display the fields containing the stack information. `err.stack` is a 2-by-1 structure because it provides information on the failing subroutine `check_inputs` and also the outer, primary function `average`:

```
st1 = err.stack(1,1)  
st1 =  
file: 'd:\matlab_test\average.m'  
name: 'check_inputs'  
line: 11  
  
st2 = err.stack(2,1)  
st2 =  
file: 'd:\matlab_test\average.m'  
name: 'average'  
line: 5
```

---

**Note** As a rule, the name of your primary function should be the same as the name of the M-file containing that function. If these names differ, MATLAB uses the M-file name in the `name` field of the `stack` structure.

---

## Example 2

`lasterror` is often used in conjunction with the `rethrow` function in try-catch statements. For example,

```
try  
do_something  
catch  
do_cleanup  
rethrow(lasterror)  
end
```

**See Also**

last (MException), MException, try, catch, error, assert, rethrow, lastwarn, dbstack

# lastwarn

---

**Purpose** Last warning message

**Syntax**

```
msgstr = lastwarn
[msgstr, msgid] = lastwarn
lastwarn('new_msgstr')
lastwarn('new_msgstr', 'new_msgid')
[msgstr, msgid] = lastwarn('new_msgstr', 'new_msgid')
```

**Description** `msgstr = lastwarn` returns the last warning message generated by the MATLAB software.

`[msgstr, msgid] = lastwarn` returns the last warning in `msgstr` and its message identifier in `msgid`. If the warning was not defined with an identifier, `lastwarn` returns an empty string for `msgid`. See “Message Identifiers” and “Warning Control” in the MATLAB Programming Fundamentals documentation for more information on the `msgid` argument and how to use it.

`lastwarn('new_msgstr')` sets the last warning message to a new string, `new_msgstr`, so that subsequent invocations of `lastwarn` return the new warning message string. You can also set the last warning to an empty string with `lastwarn('')`.

`lastwarn('new_msgstr', 'new_msgid')` sets the last warning message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lastwarn` return the new warning message and message identifier.

`[msgstr, msgid] = lastwarn('new_msgstr', 'new_msgid')` returns the last warning message and its identifier, also changing these values so that subsequent invocations of `lastwarn` return the message and identifier strings specified by `new_msgstr` and `new_msgid`, respectively.

**Remarks** `lastwarn` does not return warnings that are reported during the parsing of MATLAB commands. (Warning messages that include the failing file name and line number are parse-time warnings.)

## Examples

Specify a message identifier and warning message string with `warning`:

```
warning('MATLAB:divideByZero', 'Divide by zero');
```

Use `lastwarn` to determine the message identifier and error message string for the operation:

```
[warnmsg, msgid] = lastwarn
warnmsg =
    Divide by zero
msgid =
    MATLAB:divideByZero
```

## See Also

`warning`, `error`, `lasterr`, `lasterror`

# lcm

---

**Purpose** Least common multiple

**Syntax** `L = lcm(A,B)`

**Description** `L = lcm(A,B)` returns the least common multiple of corresponding elements of arrays A and B. Inputs A and B must contain positive integer elements and must be the same size (or either can be scalar).

**Examples**

```
lcm(8,40)

ans =

    40

lcm(pascal(3),magic(3))

ans =

     8     1     6
     3    10    21
     4     9     6
```

**See Also** `gcd`

**Purpose**

Block LDL' factorization for Hermitian indefinite matrices

**Syntax**

```
L = ldl(A)
[L,D] = ldl(A)
[L,D,P] = ldl(A)
[L,D,p] = ldl(A, 'vector')
[U,D,P] = ldl(A, 'upper')
[U,D,p] = ldl(A, 'upper', 'vector')
[L,D,P,S] = ldl(A)
[L,D,P,S] = LDL(A, THRESH)
[U,D,p,S] = LDL(A, THRESH, 'upper', 'vector')
```

**Description**

`L = ldl(A)` returns only the "psychologically lower triangular matrix" `L` as in the two-output form. The permutation information is lost, as is the block diagonal factor `D`. By default, `ldl` references only the diagonal and lower triangle of `A`, and assumes that the upper triangle is the complex conjugate transpose of the lower triangle. Therefore `[L,D,P] = ldl(TRIL(A))` and `[L,D,P] = ldl(A)` both return the exact same factors. Note, this syntax is not valid for sparse `A`.

`[L,D] = ldl(A)` stores a block diagonal matrix `D` and a "psychologically lower triangular matrix" (i.e a product of unit lower triangular and permutation matrices) in `L` such that  $A = L*D*L'$ . The block diagonal matrix `D` has 1-by-1 and 2-by-2 blocks on its diagonal. Note, this syntax is not valid for sparse `A`.

`[L,D,P] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, and permutation matrix `P` such that  $P'*A*P = L*D*L'$ . This is equivalent to `[L,D,P] = ldl(A, 'matrix')`.

`[L,D,p] = ldl(A, 'vector')` returns the permutation information as a vector, `p`, instead of a matrix. The `p` output is a row vector such that  $A(p,p) = L*D*L'$ .

`[U,D,P] = ldl(A, 'upper')` references only the diagonal and upper triangle of `A` and assumes that the lower triangle is the complex conjugate transpose of the upper triangle. This syntax returns a unit upper triangular matrix `U` such that  $P'*A*P = U'*D*U$  (assuming that

A is Hermitian, and not just upper triangular). Similarly,  $[L,D,P] = \text{ldl}(A, 'lower')$  gives the default behavior.

$[U,D,p] = \text{ldl}(A, 'upper', 'vector')$  returns the permutation information as a vector,  $p$ , as does  $[L,D,p] = \text{ldl}(A, 'lower', 'vector')$ .  $A$  must be a full matrix.

$[L,D,P,S] = \text{ldl}(A)$  returns unit lower triangular matrix  $L$ , block diagonal  $D$ , permutation matrix  $P$ , and scaling matrix  $S$  such that  $P' * S * A * S * P = L * D * L'$ . This syntax is only available for real sparse matrices, and only the lower triangle of  $A$  is referenced. `ldl` uses MA57 for sparse real symmetric  $A$ .

$[L,D,P,S] = \text{LDL}(A, \text{THRESH})$  uses `THRESH` as the pivot tolerance in MA57. `THRESH` must be a double scalar lying in the interval  $[0, 0.5]$ . The default value for `THRESH` is 0.01. Using smaller values of `THRESH` may give faster factorization times and fewer entries, but may also result in a less stable factorization. This syntax is available only for real sparse matrices.

$[U,D,p,S] = \text{LDL}(A, \text{THRESH}, 'upper', 'vector')$  sets the pivot tolerance and returns upper triangular  $U$  and permutation vector  $p$  as described above.

## Examples

These examples illustrate the use of the various forms of the `ldl` function, including the one-, two-, and three-output form, and the use of the `vector` and `upper` options. The topics covered are:

- “Example 1 — Two-Output Form of `ldl`” on page 2-1987
- “Example 2 — Three Output Form of `ldl`” on page 2-1987
- “Example 3 — The Structure of  $D$ ” on page 2-1988
- “Example 4 — Using the `'vector'` Option” on page 2-1988
- “Example 5 — Using the `'upper'` Option” on page 2-1989
- “Example 6 — `linsolve` and the Hermitian indefinite solver” on page 2-1990



Before running any of these examples, you will need to generate the following positive definite and indefinite Hermitian matrices:

```
A = full(delsq(numgrid('L', 10)));
rand('state', 0);
B = rand(10);
M = [eye(10) B; B' zeros(10)];
```

The structure of M here is very common in optimization and fluid-flow problems, and M is in fact indefinite. Note that the positive definite matrix A must be full, as ldl does not accept sparse arguments.

### Example 1 – Two-Output Form of ldl

The two-output form of ldl returns L and D such that  $A - (L^*D^*L')$  is small, L is "psychologically unit lower triangular" (i.e., a permuted unit lower triangular matrix), and D is a block 2-by-2 diagonal. Note also that, because A is positive definite, the diagonal of D is all positive:

```
[LA,DA] = ldl(A);
fprintf(1, ...
'The factorization error ||A - LA*DA*LA' || is %g\n', ...
norm(A - LA*DA*LA'));
neginds = find(diag(DA) < 0)
```

Given a b, solve  $Ax=b$  using LA, DA:

```
bA = sum(A,2);
x = LA' \ (DA \ (LA \ bA));
fprintf(...
'The absolute error norm ||x - ones(size(bA)) || is %g\n', ...
norm(x - ones(size(bA))));
```

### Example 2 – Three Output Form of ldl

The three output form returns the permutation matrix as well, so that L is in fact unit lower triangular:

```
[Lm, Dm, Pm] = ldl(M);
fprintf(1, ...
```

```
'The error norm ||Pm' * M * Pm - Lm * Dm * Lm' || is %g\n', ...
norm(Pm' * M * Pm - Lm * Dm * Lm');
fprintf(1, ...
'The difference between Lm and tril(Lm) is %g\n', ...
norm(Lm - tril(Lm)));
```

Given  $b$ , solve  $Mx=b$  using  $Lm$ ,  $Dm$ , and  $Pm$ :

```
bM = sum(M,2);
x = Pm*(Lm' \ (Dm \ (Lm \ (Pm' * bM))));
fprintf(...
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

### Example 3 – The Structure of D

$D$  is a block diagonal matrix with 1-by-1 blocks and 2-by-2 blocks. That makes it a special case of a tridiagonal matrix. When the input matrix is positive definite,  $D$  is almost always diagonal (depending on how definite the matrix is). When the matrix is indefinite however,  $D$  may be diagonal or it may express the block structure. For example, with  $A$  as above,  $DA$  is diagonal. But if you shift  $A$  just a bit, you end up with an indefinite matrix, and then you can compute a  $D$  that has the block structure.

```
figure; spy(DA); title('Structure of D from ldl(A)');
[Las, Das] = ldl(A - 4*eye(size(A)));
figure; spy(Das);
title('Structure of D from ldl(A - 4*eye(size(A)))');
```

### Example 4 – Using the 'vector' Option

Like the `lu` function, `ldl` accepts an argument that determines whether the function returns a permutation vector or permutation matrix. `ldl` returns the latter by default. When you select 'vector', the function executes faster and uses less memory. For this reason, specifying the 'vector' option is recommended. Another thing to note is that indexing is typically faster than multiplying for this kind of operation:

```

[Lm, Dm, pm] = ldl(M, 'vector');
fprintf(1, 'The error norm ||M(pm,pm) - Lm*Dm*Lm'|| is %g\n', ...
        norm(M(pm,pm) - Lm*Dm*Lm'));

% Solve a system with this kind of factorization.
clear x;
x(pm,:) = Lm\'(Dm\'(Lm\'(bM(pm,:))));
fprintf('The absolute error norm ||x - ones(size(b))|| is %g\n', ...
        norm(x - ones(size(bM))));

```

### Example 5 – Using the 'upper' Option

Like the `chol` function, `ldl` accepts an argument that determines which triangle of the input matrix is referenced, and also whether `ldl` returns a lower (L) or upper (L') triangular factor. For dense matrices, there are no real savings with using the upper triangular version instead of the lower triangular version:

```

M1 = tril(M);
[Lm1, Dm1, Pm1] = ldl(M1, 'lower'); % 'lower' is default behavior.
fprintf(1, ...
        'The difference between Lm1 and Lm is %g\n', norm(Lm1 - Lm));
[Umu, Dmu, pmu] = ldl(triu(M), 'upper', 'vector');
fprintf(1, ...
        'The difference between Umu and Lm' is %g\n', norm(Umu - Lm));

% Solve a system using this factorization.
clear x;
x(pm,:) = Umu\'(Dmu\'(Umu\'(bM(pmu,:))));
fprintf(...
        'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
        norm(x - ones(size(bM))));

```

When specifying both the 'upper' and 'vector' options, 'upper' must precede 'vector' in the argument list.

## Example 6 – linsolve and the Hermitian indefinite solver

When using the `linsolve` function, you may experience better performance by exploiting the knowledge that a system has a symmetric matrix. The matrices used in the examples above are a bit small to see this so, for this example, generate a larger matrix. The matrix here is symmetric positive definite, and below we will see that with each bit of knowledge about the matrix, there is a corresponding speedup. That is, the symmetric solver is faster than the general solver while the symmetric positive definite solver is faster than the symmetric solver:

```
Abig = full(delsq(numgrid('L', 30)));
bbig = sum(Abig, 2);
LSopts.POSDEF = false;
LSopts.SYM = false;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.SYM = true;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.POSDEF = true;
tic; linsolve(Abig, bbig, LSopts); toc;
```

### Algorithm

`ldl` uses the MA57 routines in the Harwell Subroutine Library (HSL) for real sparse matrices.

### References

- [1] Ashcraft, C., R.G. Grimes, and J.G. Lewis. "Accurate Symmetric Indefinite Linear Equations Solvers." *SIAM J. Matrix Anal. Appl.* Vol. 20. Number 2, 1998, pp. 513–561.
- [2] Duff, I. S. "MA57 — A new code for the solution of sparse symmetric definite and indefinite systems." Technical Report RAL-TR-2002-024, Rutherford Appleton Laboratory, 2002.

### See Also

`chol`, `lu`, `qr`

**Purpose** Left or right array division

**Syntax** `ldivide(A,B)`  
`A.\B`  
`rdivide(A,B)`  
`A./B`

**Description** `ldivide(A,B)` and the equivalent `A.\B` divides each entry of `B` by the corresponding entry of `A`. `A` and `B` must be arrays of the same size. A scalar value for either `A` or `B` is expanded to an array of the same size as the other.

`rdivide(A,B)` and the equivalent `A./B` divides each entry of `A` by the corresponding entry of `B`. `A` and `B` must be arrays of the same size. A scalar value for either `A` or `B` is expanded to an array of the same size as the other.

**Example**

```
A = [1 2 3;4 5 6];  
B = ones(2, 3);  
A.\B
```

```
ans =
```

```
1.0000    0.5000    0.3333  
0.2500    0.2000    0.1667
```

**See Also** Arithmetic Operators, `mldivide`, `mrdivide`

**Purpose** Test for less than or equal to

**Syntax** A <= B  
le(A, B)

**Description** A <= B compares each element of array A with the corresponding element of array B, and returns an array with elements set to logical 1 (true) where A is less than or equal to B, or set to logical 0 (false) where A is greater than B. 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.

le(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 less than or equal to the corresponding elements of B:

```
A = magic(6);  
B = repmat(3*magic(3), 2, 2);
```

```
A <= B  
ans =  
     0     1     1     0     0     0  
     1     0     1     0     0     0  
     0     1     1     0     1     0  
     1     0     0     1     0     1
```

---

0	1	0	0	1	1
1	0	0	0	1	0

**See Also**

lt, eq, ge, gt, ne, Relational Operators

# legend

---

## Purpose

Graph legend for lines and patches

## GUI Alternatives

Add a legend to a selected axes on a graph with the **Insert Legend** tool



on the figure toolbar, or use **Insert** → **Legend** from the figure menu. Use the Property Editor to modify the position, font, and other properties of a legend. For details, see Using Plot Edit Mode in the MATLAB Graphics documentation.

## Syntax

```
legend('string1','string2',...)  
legend(h,'string1','string2',...)  
legend(M)  
legend(h,M)  
legend(M,'parameter_name','parameter_value',...)  
legend(h,M,'parameter_name','parameter_value',...)  
legend(axes_handle,...)  
legend('off'), legend(axes_handle,'off')  
legend('toggle'), legend(axes_handle,'toggle')  
legend('hide'), legend(axes_handle,'hide')  
legend('show'), legend(axes_handle,'show')  
legend('boxoff'), legend(axes_handle,'boxoff')  
legend('boxon'), legend(axes_handle,'boxon')  
legend_handle = legend(...)  
legend(...,'Location',location)  
legend(...,'Orientation','orientation')  
[legend_h,object_h,plot_h,text_strings] = legend(...)  
legend(li_object,string1,string2,string3)  
legend(li_objects,M)  
legend('v6',M,...)  
legend('v6',AX)
```

## Description

`legend` places a legend on various types of graphs (line plots, bar graphs, pie charts, etc.). For each line plotted, the legend shows a sample of the line type, marker symbol, and color beside the text label you specify. When plotting filled areas (patch or surface objects), the legend contains a sample of the face color next to the text label.



The font size and font name for the legend strings match the axes `FontSize` and `FontName` properties.

`legend('string1','string2',...)` displays a legend in the current axes using the specified strings to label each set of data.

`legend(h,'string1','string2',...)` displays a legend on the plot containing the objects identified by the handles in the vector `h` and uses the specified strings to label the corresponding graphics object (line, barseries, etc.).

`legend(M)` adds a legend containing the rows of the matrix or cell array of strings `M` as labels. For matrices, this is the same as `legend(M(1,:),M(2,:),...)`.

`legend(h,M)` associates each row of the matrix or cell array of strings `M` with the corresponding graphics object (patch or line) in the vector of handles `h`.

`legend(M,'parameter_name','parameter_value',...)` and `legend(h,M,'parameter_name','parameter_value',...)` allow parameter/value pairs to be set when creating a legend (you can also assign them with `set` or with the Property Editor or Property Inspector). `M` must be a cell array of names. Legends inherit the properties of axes, although not all of them are relevant to legend objects.

`legend(axes_handle,...)` displays the legend for the axes specified by `axes_handle`.

`legend('off')`, `legend(axes_handle,'off')` removes the legend in the current axes or the axes specified by `axes_handle`.

`legend('toggle')`, `legend(axes_handle,'toggle')` toggles the legend on or off. If no legend exists for the current axes, one is created using default strings.

The *default string* for an object is the value of the object's `DisplayName` property, if you have defined a value for `DisplayName` (which you can do using the Property Editor or calling `set`). Otherwise, `legend` constructs a string of the form `data1, data2, etc.` Setting display names is useful

# legend

---

when you are experimenting with legends and might forget how objects in a lineseries, for example, are ordered.

When you specify legend strings in a legend command, their respective `DisplayNames` are set to these strings. If you delete a legend and then create a new legend without specifying labels for it, the values of `DisplayName` are (re)used as label names. Naturally, the associated plot objects must have a `DisplayName` property for this to happen: all `_series` and `_group` plot objects have a `DisplayName` property; Handle Graphics primitives, such as `line` and `patch`, do not.

`legend('hide')`, `legend(axes_handle, 'hide')` makes the legend in the current axes or the axes specified by `axes_handle` invisible.

`legend('show')`, `legend(axes_handle, 'show')` makes the legend in the current axes or the axes specified by `axes_handle` visible. A legend is created if one did not exist previously. Legends created automatically are limited to depict only the first 20 lines in the plot; if you need more legend entries, you can manually create a legend for them all with `legend('string1', 'string2', ...)` syntax.

`legend('boxoff')`, `legend(axes_handle, 'boxoff')` removes the box from the legend in the current axes or the axes specified by `axes_handle`, and makes its background transparent.

`legend('boxon')`, `legend(axes_handle, 'boxon')` adds a box with an opaque background to the legend in the current axes or the axes specified by `axes_handle`.

You can also type the above six commands using the syntax

`legend keyword`

If the keyword is not recognized, it is used as legend text, creating a legend or replacing the current legend.

`legend_handle = legend(...)` returns the handle to the legend on the current axes, or `[]` if no legend exists.

`legend(..., 'Location', location)` uses *location* to determine where to place the legend. *location* can be either a 1-by-4 position vector ([left bottom width height]) or one of the following strings.

Specifier	Location in Axes
North	Inside plot box near top
South	Inside bottom
East	Inside right
West	Inside left
NorthEast	Inside top right (default)
NorthWest	Inside top left
SouthEast	Inside bottom right
SouthWest	Inside bottom left
NorthOutside	Outside plot box near top
SouthOutside	Outside bottom
EastOutside	Outside right
WestOutside	Outside left
NorthEastOutside	Outside top right
NorthWestOutside	Outside top left
SouthEastOutside	Outside bottom right
SouthWestOutside	Outside bottom left
Best	Least conflict with data in plot
BestOutside	Least unused space outside plot

If the legend text does not fit in the 1-by-4 position vector, the position vector is resized around the midpoint to fit the legend text given its font and size, making the legend taller or wider. The *location* string can be all lowercase and can be abbreviated by sentinel letter (e.g., N, NE, NEO, etc.). Using one of the ...Outside values for *location*

# legend

---

ensures that the legend does not overlap the plot, whereas overlaps can occur when you specify any of the other cardinal values. The *location* property applies to colorbars and legends, but not to axes.

## Obsolete Location Values

The first column of the following table shows the now-obsolete specifiers for legend locations that were in use prior to Version 7, along with a description of the locations and their current equivalent syntaxes:

Obsolete Specifier	Location in Axes	Current Specifier
-1	Outside axes on right side	NorthEastOutside
0	Inside axes	Best
1	Upper right corner of axes	NorthEast
2	Upper left corner of axes	NorthWest
3	Lower left corner of axes	SouthWest
4	Lower right corner of axes	SouthEast

`legend(..., 'Orientation', 'orientation')` creates a legend with the legend items arranged in the specified orientation. *orientation* can be `vertical` (the default) or `horizontal`.

`[legend_h, object_h, plot_h, text_strings] = legend(...)` returns

- `legend_h` — Handle of the legend axes
- `object_h` — Handles of the line, patch, and text graphics objects used in the legend
- `plot_h` — Handles of the lines and other objects used in the plot
- `text_strings` — Cell array of the text strings used in the legend

These handles enable you to modify the properties of the respective objects.

`legend(li_object,string1,string2,string3)` creates a legend for legendinfo objects `li_objects` with strings `string1`, etc.

`legend(li_objects,M)` creates a legend of legendinfo objects `li_objects`, where `M` is a string matrix or cell array of strings corresponding to the legendinfo objects.

## Backward Compatibility

`legend('v6',M,...)`, for a cell array of strings `M`, creates a legend compatible with MATLAB 6.5 from the strings in `M` and any additional inputs.

`legend('v6',AX)`, for an axes handle `AX`, updates any Version 6 legends and returns the legend handle.

The following calls to `legend` are passed to the Version 6 legend mechanism to maintain backward compatibility:

```
legend('DeleteLegend')
legend('EditLegend',h)
legend('ShowLegendPlot',h)
legend('ResizeLegend')
legend('RestoreSize',hLegend)
legend('RecordSize',hPlot)
```

---

**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 MATLAB version.

---

## Remarks

`legend` associates strings with the objects in the axes in the same order that they are listed in the axes `Children` property. By default, the legend annotates the current axes.

You can only display one legend per axes. `legend` positions the legend based on a variety of factors, such as what objects the legend obscures.

The properties that legends do not share with axes are

# legend

---

- Location
- Orientation
- EdgeColor
- TextColor
- Interpreter
- String

Legends for graphs that contain groups of objects such as lineseries, barseries, contourgroups, etc. created by high-level plotting commands such as `plot`, `bar`, `contour`, etc., by default display a single legend entry for the entire group, regardless of how many member objects it contains. However, you can customize such legends to show individual entries for all or selected member objects and assign a unique `DisplayName` to any of them. You control how groups appear in the legend by setting values for their `Annotation` and `DisplayName` properties with M-code. For information and examples about customizing legends in this manner, see “Controlling Legends” in the MATLAB Graphics documentation.

You can specify `EdgeColor` and `TextColor` as RGB triplets or as `ColorSpecs`. You cannot set these colors to 'none'. To hide the box surrounding a legend, set the `Box` property to 'off'. To allow the background to show through the legend box, set the legend's `Color` property to 'none', for example,

```
set(legend_handle, 'Box', 'off')
set(legend_handle, 'Color', 'none')
```

This is similar to the effect of the command `legend boxoff`, except that `boxoff` also hides the legend's border.

You can use a legend's handle to set text properties for all the strings in a legend at once with a cell array of strings, rather than looping through each of them. See the last line of the example below, which demonstrates setting a legend's `Interpreter` property. In that example, you could reset the `String` property of the legend as follows:

```
set(h,'String',{'cos(x)', 'sin(x)'});
```

See the documentation for Text Properties for additional details.

legend installs a figure `ResizeFcn` if there is not already a user-defined `ResizeFcn` assigned to the figure. This `ResizeFcn` attempts to keep the legend the same size.

## Moving the Legend

Move the legend by pressing the left mouse button while the cursor is over the legend and dragging the legend to a new location.

Double-clicking a label allows you to edit the label.

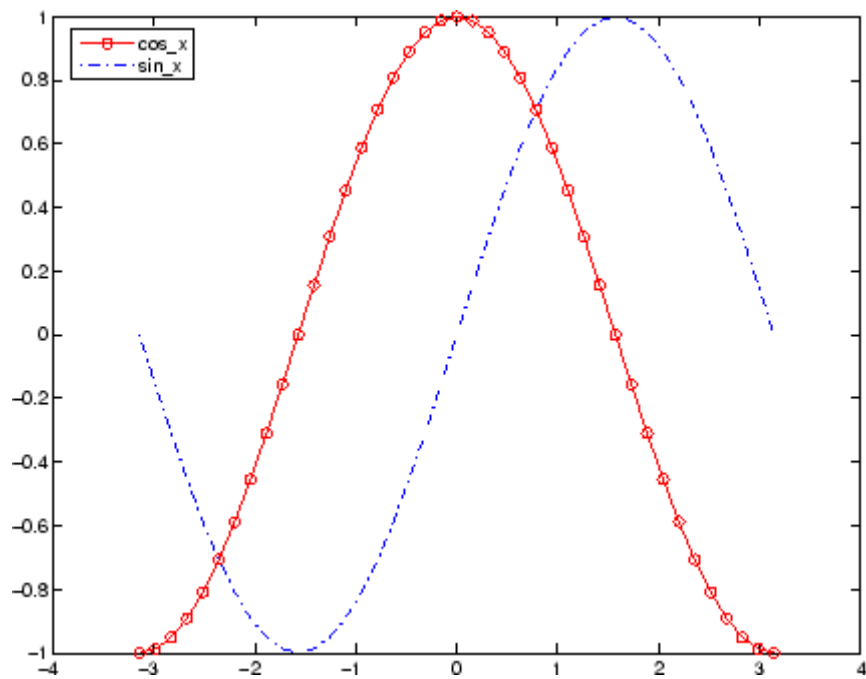
## Example

Add a legend to a graph showing a sine and cosine function:

```
x = -pi:pi/20:pi;  
plot(x,cos(x),'-ro',x,sin(x),'-.b')  
h = legend('cos_x','sin_x',2);  
set(h,'Interpreter','none')
```

# legend

---



In this example, the `plot` command specifies a solid, red line (`'-r'`) for the cosine function and a dash-dot, blue line (`'-.b'`) for the sine function.

## See Also

`LineStyle`, `plot`

“Adding a Legend to a Graph” for more information on using legends

“Annotating Plots” on page 1-91 for related functions



**Purpose** Associated Legendre functions

**Syntax**  
 $P = \text{legendre}(n, X)$   
 $S = \text{legendre}(n, X, 'sch')$   
 $N = \text{legendre}(n, X, 'norm')$

## Definitions **Associated Legendre Functions**

The Legendre functions are defined by

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x)$$

where

$$P_n(x)$$

is the Legendre polynomial of degree  $n$ .

$$P_n(x) = \frac{1}{2^n n!} \left[ \frac{d^n}{dx^n} (x^2 - 1)^n \right]$$

## **Schmidt Seminormalized Associated Legendre Functions**

The Schmidt seminormalized associated Legendre functions are related to the nonnormalized associated Legendre functions  $P_n^m(x)$  by

$$P_n^m(x) \text{ for } m = 0$$

$$S_n^m(x) = (-1)^m \sqrt{\frac{2(n-m)!}{(n+m)!}} P_n^m(x) \text{ for } m > 0.$$

## **Fully Normalized Associated Legendre Functions**

The fully normalized associated Legendre functions are normalized such that

# legendre

---

$$\int_{-1}^1 (N_n^m(x))^2 dx = 1$$

and are related to the unnormalized associated Legendre functions  $P_n^m(x)$  by

$$N_n^m(x) = (-1)^m \sqrt{\frac{\left(n + \frac{1}{2}\right)(n - m)!}{(n + m)!}} P_n^m(x)$$

## Description

`P = legendre(n,X)` computes the associated Legendre functions  $P_n^m(x)$  of degree  $n$  and order  $m = 0, 1, \dots, n$ , evaluated for each element of  $X$ . Argument  $n$  must be a scalar integer, and  $X$  must contain real values in the domain  $-1 \leq x \leq 1$ .

If  $X$  is a vector, then  $P$  is an  $(n+1)$ -by- $q$  matrix, where  $q = \text{length}(X)$ . Each element  $P(m+1, i)$  corresponds to the associated Legendre function of degree  $n$  and order  $m$  evaluated at  $X(i)$ .

In general, the returned array  $P$  has one more dimension than  $X$ , and each element  $P(m+1, i, j, k, \dots)$  contains the associated Legendre function of degree  $n$  and order  $m$  evaluated at  $X(i, j, k, \dots)$ . Note that the first row of  $P$  is the Legendre polynomial evaluated at  $X$ , i.e., the case where  $m = 0$ .

`S = legendre(n,X, 'sch')` computes the Schmidt seminormalized associated Legendre functions  $S_n^m(x)$ .

`N = legendre(n,X, 'norm')` computes the fully normalized associated Legendre functions  $N_n^m(x)$ .

## Examples

### Example 1

The statement `legendre(2,0:0.1:0.2)` returns the matrix

	<b>x = 0</b>	<b>x = 0.1</b>	<b>x = 0.2</b>
m = 0	-0.5000	-0.4850	-0.4400
m = 1	0	-0.2985	-0.5879
m = 2	3.0000	2.9700	2.8800

## Example 2

Given,

```
X = rand(2,4,5);
n = 2;
P = legendre(n,X)
```

then

```
size(P)
ans =
     3     2     4     5
```

and

```
P(:,1,2,3)
ans =
    -0.2475
    -1.1225
     2.4950
```

is the same as

```
legendre(n,X(1,2,3))
ans =
    -0.2475
    -1.1225
     2.4950
```

## Algorithm

legendre uses a three-term backward recursion relationship in  $m$ . This recursion is on a version of the Schmidt seminormalized associated

# Legendre

---

Legendre functions  $Q_n^m(x)$ , which are complex spherical harmonics. These functions are related to the standard Abramowitz and Stegun [1] functions  $P_n^m(x)$  by

$$P_n^m(x) = \sqrt{\frac{(n+m)!}{(n-m)!}} Q_n^m(x)$$

They are related to the Schmidt form given previously by

$$S_n^m(x) = Q_n^0(x) \text{ for } m = 0$$

$$S_n^m(x) = (-1)^m \sqrt{2} Q_n^m(x) \text{ for } m > 0$$

## References

- [1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Ch.8.
- [2] Jacobs, J. A., *Geomagnetism*, Academic Press, 1987, Ch.4.

**Purpose** Length of vector

**Syntax** `n = length(X)`

**Description** The statement `length(X)` is equivalent to `max(size(X))` for nonempty arrays and 0 for empty arrays.

`n = length(X)` returns the size of the longest dimension of X. If X is a vector, this is the same as its length.

**Examples**

```
x = ones(1,8);  
n = length(x)  
  
n =  
    8  
x = rand(2,10,3);  
n = length(x)  
  
n =  
    10
```

**See Also** `ndims`, `size`

# length (Map)

---

**Purpose** Length of containers.Map object.

**Syntax** L = length(M)

**Description** L = length(M) returns the number of pairs in the map M. The number returned by this method is equivalent to `size(M,1)`.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

**Examples** Create a Map object containing the names of several US states and the capital city of each:

```
US_Capitals = containers.Map( ...
    {'Arizona', 'Nebraska', 'Nevada', 'New York', ...
     'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...
    {'Phoenix', 'Lincoln', 'Carson City', 'Albany', ...
     'Atlanta', 'Juneau', 'Montpelier', 'Salem'})
```

Find out how many keys are in the map:

```
length(US_Capitals)
ans =
     8
```

This should be equal to the Count property for the object:

```
length(US_Capitals) == US_Capitals.Count
ans =
     1
```

Use the map to find the capital of one of these states:

```
state = 'Georgia';
sprintf(' The capital of %s is %s', ...
    state, US_Capitals(state))
ans =
```

The capital of Georgia is Atlanta

### **See Also**

`containers.Map`, `keys(Map)`, `values(Map)`, `size(Map)`, `isKey(Map)`,  
`remove(Map)`, `handle`

# length (serial)

---

**Purpose** Length of serial port object array

**Syntax** length(obj)

**Description** length(obj) returns the length of the serial port object, obj. It is equivalent to the command max(size(obj)).

**See Also** **Functions**  
size



<b>Purpose</b>	Length of time vector
<b>Syntax</b>	<code>length(ts)</code>
<b>Description</b>	<code>length(ts)</code> returns an integer that represents the length of the time vector for the <code>timeseries</code> object <code>ts</code> . It returns 0 if <code>ts</code> is empty.
<b>See Also</b>	<code>isempty (timeseries)</code> , <code>size (timeseries)</code>

# length (tscollection)

---

**Purpose** Length of time vector

**Syntax** `length(tsc)`

**Description** `length(tsc)` returns an integer that represents the length of the time vector for the `tscollection` object `tsc`.

**See Also** `isempty (tscollection)`, `size (tscollection)`, `tscollection`

**Purpose** Return information on functions in shared library

**Syntax**

```
m = libfunctions('libname')  
m = libfunctions('libname', '-full')  
libfunctions libname -full
```

**Description**

`m = libfunctions('libname')` returns the names of all functions defined in the external shared library, `libname`, that has been loaded into the MATLAB software with the `loadlibrary` function. The return value, `m`, is a cell array of strings.

`m = libfunctions('libname', '-full')` returns a full description of the functions in the library, including function signatures. This includes duplicate function names with different signatures. The return value, `m`, is a cell array of strings.

`libfunctions libname -full` is the command format for this function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

**Examples**

To list the functions in the MATLAB `libmx` library, see “Viewing Functions in the Command Window”.

**See Also** `loadlibrary`, `libfunctionsview`, `calllib`, `unloadlibrary`

# libfunctionsview

---

**Purpose** View functions in a shared library

**Syntax** `libfunctionsview('libname')`  
`libfunctionsview libname`

**Description** `libfunctionsview('libname')` displays the names of the functions in the external shared library, `libname`, that has been loaded into the MATLAB software with the `loadlibrary` function.

`libfunctionsview libname` is the command format for this function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

MATLAB creates a new window in response to the `libfunctionsview` command. This window displays all of the functions defined in the specified library. For each of these functions, the following information is supplied:

- Type returned by the function
- Name of the function
- Arguments passed to the function

An additional column entitled “Inherited From” is displayed at the far right of the window. The information in this column is not useful for external libraries.

**Examples** To open a window showing functions in the `libmx` library, see “Viewing Functions in a GUI”.

**See Also** `loadlibrary`, `libfunctions`, `calllib`, `unloadlibrary`

**Purpose** Determine if shared library is loaded

**Syntax** `libisloaded('libname')`  
`libisloaded libname`

**Description** `libisloaded('libname')` returns logical 1 (true) if the shared library `libname` is loaded and logical 0 (false) otherwise.

`libisloaded libname` is the command format for this function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

## Examples

### Example 1

Load the `shrlibsample` library and check to see if the load was successful before calling one of its functions:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h

if libisloaded('shrlibsample')
    x = calllib('shrlibsample', 'addDoubleRef', 1.78, 5.42, 13.3)
end
```

Since the library is successfully loaded, the call to `addDoubleRef` works as expected and returns

```
x =
    20.5000
```

```
unloadlibrary shrlibsample
```

### Example 2

Load the same library, this time giving it an alias. If you use `libisloaded` with the library name, `shrlibsample`, it now returns `false`. Since you loaded the library using an alias, all further references to the library must also use that alias:

# libisloaded

---

```
addpath([matlabroot '\extern\examples\shrlib'])  
loadlibrary shrlibsampl shrlibsampl.h alias lib
```

```
libisloaded shrlibsampl  
ans =  
    0
```

```
libisloaded lib  
ans =  
    1
```

```
unloadlibrary lib
```

## See Also

loadlibrary, unloadlibrary

**Purpose** Create pointer object for use with shared libraries

**Syntax**

```
p = libpointer
p = libpointer('type')
p = libpointer('type',value)
```

**Description** `p = libpointer` returns an empty (void) pointer.

`p = libpointer('type')` returns an empty pointer that contains a reference to the specified type. This type can be any MATLAB numeric type, or a structure or enumerated type defined in an external library that has been loaded into MATLAB with the `loadlibrary` function. For valid types, see the table under “C and MATLAB Equivalent Types” in the MATLAB External Interfaces documentation.

---

**Note** Using this syntax, `p` is a NULL pointer. You, therefore, must ensure that any library function to which you pass `p` must be able to accept a NULL pointer as an argument.

---

`p = libpointer('type',value)` returns a pointer to the specified data type and initialized to the value supplied.

**Remarks** MATLAB automatically converts data passed to and from external library functions to the data type expected by the external function. The `libpointer` function enables you to convert your argument data manually. This is an advanced feature available to experienced C programmers. For more information about using pointer objects, see “Working with Pointers” in the MATLAB External Interfaces documentation. Additional examples for using `libpointer` can be found in “Reference Pointers” in the same documentation.

**Examples** This example passes an `int16` pointer to a function that multiplies each value in a matrix by its index. The function `multiplyShort` is defined in the MATLAB sample shared library, `shrlibsample`.

# libpointer

---

Here is the C function:

```
void multiplyShort(short *x, int size)
{
    int i;
    for (i = 0; i < size; i++)
        *x++ *= i;
}
```

Load the `shrlibsample` library. Create the matrix, `v`, and also a pointer to it, `pv`:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h

v = [4 6 8; 7 5 3];

pv = libpointer('int16Ptr', v);
get(pv, 'Value')
ans =
     4     6     8
     7     5     3
```

Now call the C function in the library, passing the pointer to `v`. If you were to pass a *copy* of `v`, the results would be lost once the function terminates. Passing a pointer to `v` enables you to get back the results:

```
calllib('shrlibsample', 'multiplyShort', pv, 6);
get(pv, 'Value')
ans =
     0    12    32
     7    15    15

unloadlibrary shrlibsample
```

## See Also

`loadlibrary`, `libstruct`



**Purpose** Create structure pointer for use with shared libraries

**Syntax**

```
s = libstruct('structtype')
s = libstruct('structtype',mlstruct)
```

**Description** `s = libstruct('structtype')` returns a `libstruct` object `s` that is a MATLAB object designed to resemble a C structure of type specified by `structtype`. The structure type, `structtype`, is defined in an external library that must be loaded into MATLAB using the `loadlibrary` function.

---

**Note** Using this syntax, `s` is a NULL pointer. You, therefore, must ensure that any library function to which you pass `s` must be able to accept a NULL pointer as an argument.

---

`s = libstruct('structtype',mlstruct)` returns a `libstruct` object `s` with its fields initialized from MATLAB structure, `mlstruct`.

The `libstruct` function creates a C-style structure that you can pass to functions in an external library. You handle this structure in MATLAB as you would a true MATLAB structure.

### What Types Are Available

To determine which MATLAB types to use when passing arguments to library functions, see the output of `libfunctionsview` or `libfunctions -full`. These functions list all of the functions found in a particular library along with a specification of the types required for each argument.

**Examples** This example performs a simple addition of the fields of a structure. The function `addStructFields` is defined in the MATLAB sample shared library, `shrlibsample`.

Here is the C function:

```
double addStructFields(struct c_struct st)
```

# libstruct

---

```
{
    double t = st.p1 + st.p2 + st.p3;
    return t;
}
```

Start by loading the `shrlibsample` library and creating the structure, `sm`:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h

sm.p1 = 476;    sm.p2 = -299;    sm.p3 = 1000;
```

Construct a `libstruct` object `sc` that uses the `c_struct` template:

```
sc = libstruct('c_struct', sm);

get(sc)
    p1: 476
    p2: -299
    p3: 1000
```

Now call the function, passing the `libstruct` object, `sc`:

```
calllib('shrlibsample', 'addStructFields', sc)
ans =
    1177
```

You must clear the `libstruct` object before unloading the library:

```
clear sc
unloadlibrary shrlibsample
```

---

**Note** In most cases, you can pass a MATLAB structure and MATLAB automatically converts the argument to a C structure. See “Working with Structures” in the MATLAB External Interfaces documentation for more information.

---

**See Also**      `loadlibrary`, `libpointer`

# license

---

**Purpose** Return license number or perform licensing task

**Syntax**

```
license
license('inuse')
S = license('inuse')
S = license('inuse', feature)
license('test', feature)
license('test', feature, toggle)
result = license('checkout', feature)
```

**Description** `license` returns the license number for this MATLAB product. The return value is always a string but is not guaranteed to be a number. The following table lists text strings that `license` can return.

String	Description
'demo'	MATLAB is a demonstration version
'student'	MATLAB is the student version
'unknown'	License number cannot be determined

`license('inuse')` returns a list of licenses checked out in the current MATLAB session. In the list, products are listed alphabetically by their license feature names, i.e., the text string used to identify products in the INCREMENT lines in a License File (`license.dat`). Note that the feature names returned in the list contain only lower-case characters.

`S = license('inuse')` returns an array of structures, where each structure represents a checked-out license. The structures contains two fields: `feature` and `user`. The `feature` field contains the license feature name. The `user` field contains the username of the person who has the license checked out.

`S = license('inuse', feature)` checks if the product specified by the text string `feature` is checked out in the current MATLAB session. If the product is checked out, the `license` function returns the product name and the username of the person who has it checked out in the

structure `S`. If the product is not currently checked out, the fields in the structure are empty.

The `feature` string must be a license feature name, spelled exactly as it appears in the `INCREMENT` lines in a License File. For example, the string `'Identification_Toolbox'` is the feature name for the System Identification Toolbox™. The `feature` string is not case-sensitive and must not exceed 27 characters.

`license('test', feature)` tests if a license exists for the product specified by the text string `feature`. The `license` command returns 1 if the license exists and 0 if the license does not exist. The `feature` string identifies a product, as described in the previous syntax.

---

**Note** Testing for a license only confirms that the license exists. It does not confirm that the license can be checked out. For example, `license` will return 1 if a license exists, even if the license has expired or if a system administrator has excluded you from using the product in an options file.

---

`license('test', feature, toggle)` enables or disables testing of the product specified by the text string `feature`, depending on the value of `toggle`. The parameter `toggle` can have either of two values:

`'enable'` The syntax `license('test', feature)` returns 1 if the product license exists and 0 if the product license does not exist.

`'disable'` The syntax `license('test', feature)` always returns 0 (product license does not exist) for the specified product.

---

**Note** Disabling a test for a particular product can impact other tests for the existence of the license, not just tests performed using the `license` command.

---

# license

---

`result = license('checkout', feature)` checks out a license for the product identified by the text string `feature`. The `license` command returns 1 if it could check out a license for the product and 0 if it could not check out a license for the product.

## Examples

Get the license number for this MATLAB.

```
license
```

Get a list of licenses currently being used. Note that the products appear in alphabetical order by their license feature name in the list returned.

```
license('inuse')
```

```
image_toolbox  
map_toolbox  
matlab
```

Get a list of licenses in use with information about who is using the license.

```
S = license('inuse');  
  
S(1)  
  
ans =  
  
    feature: 'image_toolbox'  
    user: 'juser'
```

Determine if the license for MATLAB is currently in use.

```
S = license('inuse', 'MATLAB')  
  
S =  
  
    feature: 'matlab'  
    user: 'jsmith'
```

Determine if a license exists for the Mapping Toolbox™.

```
license('test','map_toolbox')
```

```
ans =
```

```
1
```

Check out a license for the Control System Toolbox.

```
license('checkout','control_toolbox')
```

```
ans =
```

```
1
```

Determine if the license for the Control System Toolbox is checked out.

```
license('inuse')
```

```
control_toolbox  
image_toolbox  
map_toolbox  
matlab
```

## See Also

isstudent

# light

---

## Purpose

Create light object

## Syntax

```
light('PropertyName',propertyvalue,...)
handle = light(...)
```

## Description

`light` creates a light object in the current axes. Lights affect only patch and surface objects.

`light('PropertyName',propertyvalue,...)` creates a light object using the specified values for the named properties. The MATLAB software parents the light to the current axes unless you specify another axes with the `Parent` property.

`handle = light(...)` returns the handle of the light object created.

## Remarks

You cannot see a light object *per se*, but you can see the effects of the light source on patch and surface objects. You can also specify an axes-wide ambient light color that illuminates these objects. However, ambient light is visible only when at least one light object is present and visible in the axes.

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

See also the patch and surface `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, `SpecularColorReflectance`, and `VertexNormals` properties. Also see the lighting and material commands.

## Examples

Light the peaks surface plot with a light source located at infinity and oriented along the direction defined by the vector `[1 0 0]`, that is, along the  $x$ -axis.

```
h = surf(peaks);
set(h,'FaceLighting','phong','FaceColor','interp',...
    'AmbientStrength',0.5)
light('Position',[1 0 0],'Style','infinite');
```



## Setting Default Properties

You can set default light properties on the axes, figure, and root objectlevels:

```
set(0, 'DefaultLightProperty',PropertyValue...)  
set(gcf, 'DefaultLightProperty',PropertyValue...)  
set(gca, 'DefaultLightProperty',PropertyValue...)
```

where *Property* is the name of the light property and *PropertyValue* is the value you are specifying. Use `set` and `get` to access light properties.

## See Also

lighting, material, patch, surface

“Lighting as a Visualization Tool” for more information about lighting

“Lighting” on page 1-105 for related functions

Light Properties for property descriptions

# Light Properties

---

## Purpose

Light properties

## Modifying Properties

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

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

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

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

## Light Property Descriptions

This section lists property names along with the type of values each accepts.

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. The MATLAB software sets the `BeingDeleted` property to on when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to on while the delete function executes, after which the object no longer exists.

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

BusyAction  
cancel | {queue}

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

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

`ButtonDownFcn`  
function handle

This property is not used on lights.

`Children`  
handles

The empty matrix; light objects have no children.

`Clipping`  
`on` | `off`

Clipping has no effect on light objects.

`Color`  
`ColorSpec`

*Color of light.* This property defines the color of the light emanating from the light object. Define it as a three-element RGB vector or one of the MATLAB predefined names. See the `ColorSpec` reference page for more information.

# Light 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 a light object. You must define this property as a default value for lights or in a call to the `light` function to create a new light object. For example, the following statement:

```
set(0,'DefaultLightCreateFcn',@light_create)
```

defines a default value for the line `CreateFcn` property on the root level that sets the current figure colormap to `gray` and uses a reddish light color whenever you create a light object.

```
function light_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
set(src,'Color',[.9 .2 .2])
set(gcf,'Colormap',gray)
end
```

MATLAB executes this function after setting all light properties. Setting this property on an existing light object has no effect. The function must define at least two input arguments (handle of light object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

## DeleteFcn

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

*Delete light callback function.* A callback function that executes when you delete the light object (e.g., when you issue a `delete` command or clear the axes `cla` or figure `clf`). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src, evnt)
% src - the object that is the source of the event
% evnt - empty for this property
    obj_tp = get(src, 'Type');
    disp([obj_tp, ' object deleted'])
    disp('Its user data is:')
    disp(get(src, 'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

## 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

# Light Properties

---

its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is on.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

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

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

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

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

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

## HitTest

{on} | off

This property is not used by light objects.

## Interruptible

{on} | off

*Callback routine interruption mode.* Light object callback routines defined for the `DeleteFcn` property are not affected by the `Interruptible` property.

## Parent

handle of parent axes

*Parent of light object.* This property contains the handle of the light object's parent. The parent of a light object is the axes object that contains it.

Note that light objects cannot be parented to `hggroup` or `hgtransform` objects.

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

## Position

[x,y,z] in axes data units

*Location of light object.* This property specifies a vector defining the location of the light object. The vector is defined from the origin to the specified *x*-, *y*-, and *z*-coordinates. The placement of the light depends on the setting of the `Style` property:

# Light Properties

---

- If the `Style` property is set to `local`, `Position` specifies the actual location of the light (which is then a point source that radiates from the location in all directions).
- If the `Style` property is set to `infinite`, `Position` specifies the direction from which the light shines in parallel rays.

## Selected

on | off

This property is not used by light objects.

## SelectionHighlight

{on} | off

This property is not used by light objects.

## Style

{infinite} | local

*Parallel or divergent light source.* This property determines whether MATLAB places the light object at infinity, in which case the light rays are parallel, or at the location specified by the `Position` property, in which case the light rays diverge in all directions. See the `Position` property.

## Tag

string

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

## Type

string (read only)



*Type of graphics object.* This property contains a string that identifies the class of graphics object. For light objects, Type is always 'light'.

## UIContextMenu

handle of a uicontextmenu object

This property is not used by light objects.

## UserData

matrix

*User-specified data.* This property can be any data you want to associate with the light object. The light does not use this property, but you can access it using `set` and `get`.

## Visible

{on} | off

*Light visibility.* While light objects themselves are not visible, you can see the light on patch and surface objects. When you set `Visible` to `off`, the light emanating from the source is not visible. There must be at least one light object in the axes whose `Visible` property is `on` for any lighting features to be enabled (including the axes `AmbientLightColor` and patch and surface `AmbientStrength`).

# lightangle

---

**Purpose** Create or position `light` object in spherical coordinates

**Syntax**

```
lightangle(az,e1)
light_handle = lightangle(az,e1)
lightangle(light_handle,az,e1)
[az,e1] = lightangle(light_handle)
```

**Description** `lightangle(az,e1)` creates a light at the position specified by azimuth and elevation. `az` is the azimuthal (horizontal) rotation and `e1` is the vertical elevation (both in degrees). The interpretation of azimuth and elevation is the same as that of the `view` command.

`light_handle = lightangle(az,e1)` creates a light and returns the handle of the light in `light_handle`.

`lightangle(light_handle,az,e1)` sets the position of the light specified by `light_handle`.

`[az,e1] = lightangle(light_handle)` returns the azimuth and elevation of the light specified by `light_handle`.

**Remarks** By default, when a light is created, its style is `infinite`. If the light handle passed in to `lightangle` refers to a local light, the distance between the light and the camera target is preserved as the position is changed.

**Examples**

```
surf(peaks)
axis vis3d
h = light;
for az = -50:10:50
    lightangle(h,az,30)
drawnow
end
```

**See Also** `light`, `camlight`, `view`

“Lighting as a Visualization Tool” for more information about lighting

“Lighting” on page 1-105 for related functions

<b>Purpose</b>	Specify lighting algorithm
<b>Syntax</b>	<code>lighting flat</code> <code>lighting gouraud</code> <code>lighting phong</code> <code>lighting none</code>
<b>Description</b>	<p><code>lighting</code> selects the algorithm used to calculate the effects of light objects on all surface and patch objects in the current axes. In order for the <code>lighting</code> command to have any effects, however, you must create a lighting object by using the <code>light</code> function.</p> <p><code>lighting flat</code> produces uniform lighting across each of the faces of the object. Select this method to view faceted objects.</p> <p><code>lighting gouraud</code> calculates the vertex normals and interpolates linearly across the faces. Select this method to view curved surfaces.</p> <p><code>lighting phong</code> interpolates the vertex normals across each face and calculates the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but it takes longer to render.</p> <p><code>lighting none</code> turns off lighting.</p>
<b>Remarks</b>	The <code>surf</code> , <code>mesh</code> , <code>pcolor</code> , <code>fill</code> , <code>fill3</code> , <code>surface</code> , and <code>patch</code> functions create graphics objects that are affected by light sources. The <code>lighting</code> command sets the <code>FaceLighting</code> and <code>EdgeLighting</code> properties of surfaces and patches appropriately for the graphics object.
<b>See Also</b>	<code>fill</code> , <code>fill3</code> , <code>light</code> , <code>material</code> , <code>mesh</code> , <code>patch</code> , <code>pcolor</code> , <code>shading</code> , <code>surface</code> “Lighting as a Visualization Tool” for more information about lighting “Lighting” on page 1-105 for related functions

# lin2mu

---

**Purpose** Convert linear audio signal to mu-law

**Syntax** `mu = lin2mu(y)`

**Description** `mu = lin2mu(y)` converts linear audio signal amplitudes in the range  $-1 \leq Y \leq 1$  to mu-law encoded “flints” in the range  $0 \leq u \leq 255$ .

**See Also** `auwrite`, `mu2lin`

**Purpose**

Create line object

**Syntax**

```
line(X,Y)
line(X,Y,Z)
line(X,Y,Z,'PropertyName',propertyvalue,...)
line('XData',x,'YData',y,'ZData',z,...)
h = line(...)
```

**Description**

`line` creates a line object in the current axes. You can specify the color, width, line style, and marker type, as well as other characteristics.

The `line` function has two forms:

- Automatic color and line style cycling. When you specify matrix coordinate data using the informal syntax (i.e., the first three arguments are interpreted as the coordinates),

```
line(X,Y,Z)
```

MATLAB cycles through the axes `ColorOrder` and `LineStyleOrder` property values the way the `plot` function does. However, unlike `plot`, `line` does not call the `newplot` function.

- Purely low-level behavior. When you call `line` with only property name/property value pairs,

```
line('XData',x,'YData',y,'ZData',z)
```

MATLAB draws a line object in the current axes using the default line color (see the `colordef` function for information on color defaults). Note that you cannot specify matrix coordinate data with the low-level form of the `line` function.

`line(X,Y)` adds the line defined in vectors `X` and `Y` to the current axes. If `X` and `Y` are matrices of the same size, `line` draws one line per column.

`line(X,Y,Z)` creates lines in three-dimensional coordinates.

# line

---

`line(X,Y,Z, 'PropertyName', propertyvalue, ...)` creates a line using the values for the property name/property value pairs specified and default values for all other properties.

See the `LineStyle` and `Marker` properties for a list of supported values.

`line('XData',x, 'YData',y, 'ZData',z, ...)` creates a line in the current axes using the property values defined as arguments. This is the low-level form of the `line` function, which does not accept matrix coordinate data as the other informal forms described above.

`h = line(...)` returns a column vector of handles corresponding to each line object the function creates.

## Remarks

In its informal form, the `line` function interprets the first three arguments (two for 2-D) as the X, Y, and Z coordinate data, allowing you to omit the property names. You must specify all other properties as name/value pairs. For example,

```
line(X,Y,Z, 'Color', 'r', 'LineWidth', 4)
```

The low-level form of the `line` function can have arguments that are only property name/property value pairs. For example,

```
line('XData',x, 'YData',y, 'ZData',z, 'Color', 'r', 'LineWidth', 4)
```

Line properties control various aspects of the line object and are described in the "Line Properties" section. You can also set and query property values after creating the line using `set` and `get`.

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

Unlike high-level functions such as `plot`, `line` does not respect the settings of the figure and axes `NextPlot` properties. It simply adds line objects to the current axes. However, axes properties that are under automatic control, such as the axis limits, can change to accommodate the line within the current axes.

## Connecting the dots

The coordinate data is interpreted as vectors of corresponding x, y, and z values:

```
X = [x(1) x(2) x(3) ...x(n)]
Y = [y(1) x(2) y(3) ...y(n)]
Z = [z(1) z(2) x(3) ...z(n)]
```

where a point is determined by the corresponding vector elements:

```
p1(x(i),y(i),z(i))
```

For example, to draw a line from the point located at  $x = .3$  and  $y = .4$  and  $z = 1$  to the point located at  $x = .7$  and  $y = .9$  and  $z = 1$ , use the following data:

```
axis([0 1 0 1])
line([.3 .7],[.4 .9],[1 1], 'Marker', '.', 'LineStyle', '-')
```

## Examples

This example uses the line function to add a shadow to plotted data. First, plot some data and save the line's handle:

```
t = 0:pi/20:2*pi;
hline1 = plot(t,sin(t), 'k');
```

Next, add a shadow by offsetting the x-coordinates. Make the shadow line light gray and wider than the default LineWidth:

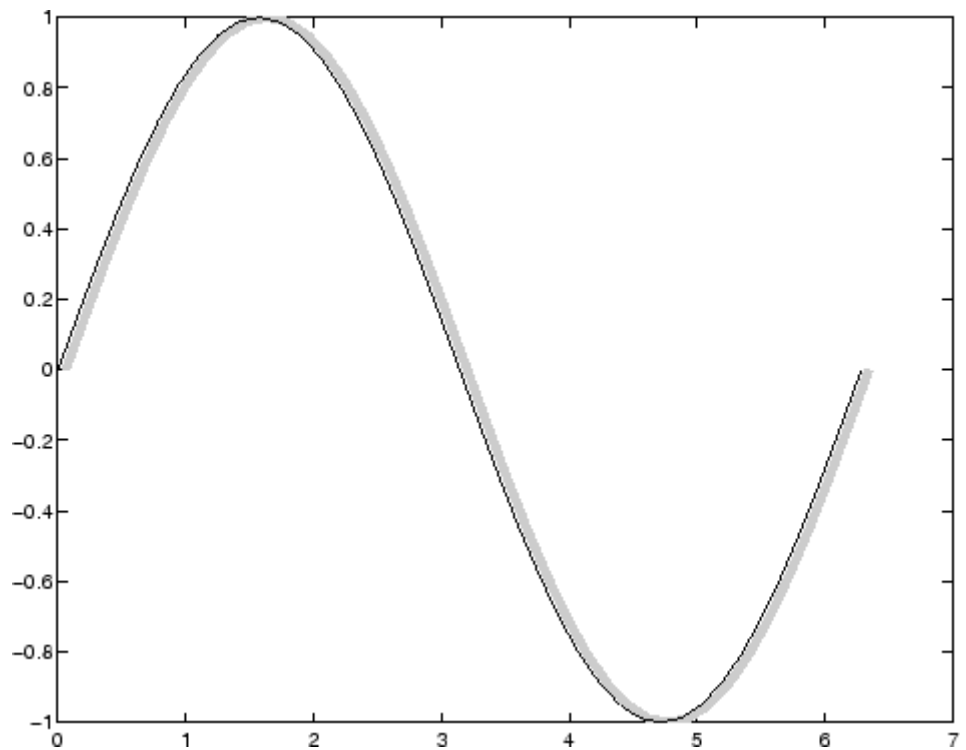
```
hline2 = line(t+.06,sin(t), 'LineWidth',4, 'Color', [.8 .8 .8]);
```

Finally, pull the first line to the front:

```
set(gca, 'Children', [hline1 hline2])
```

# line

---



## Drawing Lines Interactively

You can use the `ginput` function to select points from a figure. For example:

```
axis([0 1 0 1])
for n = 1:5
    [x(n),y(n)] = ginput(1);
end
line(x,y)
```

The for loop enables you to select five points and build the `x` and `y` arrays. Because `line` requires arrays of corresponding `x` and `y` coordinates, you can just pass these arrays to the `line` function.



### Drawing with mouse motion

You can use the axes `CurrentPoint` property and the figure `WindowButtonDownFcn` and `WindowButtonMotionFcn` properties to select a point with a mouse click and draw a line to another point by dragging the mouse, like a simple drawing program. The following example illustrates a few useful techniques for doing this type of interactive drawing.

Click to view in editor — This example enables you to click and drag the cursor to draw lines.

Click to run example — Click the left mouse button in the axes and move the cursor, left-click to define the line end point, right-click to end drawing mode.

### Input Argument Dimensions – Informal Form

This statement reuses the one-column matrix specified for `ZData` to produce two lines, each having four points.

```
line(rand(4,2),rand(4,2),rand(4,1))
```

If all the data has the same number of columns and one row each, MATLAB transposes the matrices to produce data for plotting. For example,

```
line(rand(1,4),rand(1,4),rand(1,4))
```

is changed to

```
line(rand(4,1),rand(4,1),rand(4,1))
```

This also applies to the case when just one or two matrices have one row. For example, the statement

```
line(rand(2,4),rand(2,4),rand(1,4))
```

is equivalent to

```
line(rand(4,2),rand(4,2),rand(4,1))
```

# line

---

## Setting Default Properties

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

```
set(0, 'DefaultLinePropertyName', PropertyValue, ...)  
set(gcf, 'DefaultLinePropertyName', PropertyValue, ...)  
set(gca, 'DefaultLinePropertyName', PropertyValue, ...)
```

Where *PropertyName* is the name of the line property and *PropertyValue* is the value you are specifying. Use `set` and `get` to access line properties.

## See Also

`annotationaxes`, `newplot`, `plot`, `plot3`

“Object Creation” on page 1-98 for related functions

Line Properties for property descriptions

## Purpose

Line properties

## Modifying Properties

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

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

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

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

## Line Property Descriptions

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

### Annotation

`hg.Annotation` object Read Only

*Control the display of line objects in legends.* The `Annotation` property enables you to specify whether this line 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 line object is displayed in a figure legend:

# Line Properties

---

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
on	Represent this line object in a legend (default)
off	Do not include this line object in a legend
children	Same as on because line objects do not have children

## Setting the **IconDisplayStyle** property

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

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

## Using the **IconDisplayStyle** property

See “Controlling Legends” for more information and examples.

## BeingDeleted

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. The MATLAB software sets the **BeingDeleted** property to `on` when the object’s delete function callback is called (see the **DeleteFcn** property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted

and, therefore, can check the object's `BeingDeleted` property before acting.

### `BusyAction`

`cancel` | `{queue}`

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

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

### `ButtonDownFcn`

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

*Button press callback function.* A callback function that executes whenever you press a mouse button while the pointer is over the line object.

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

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of line associated with the button down event and an event structure, which is empty for this property)

# Line Properties

---

The following example shows how to access the callback object's handle as well as the handle of the figure that contains the object from the callback function.

```
function button_down(src, evnt)
% src - the object that is the source of the event
% evnt - empty for this property
    sel_typ = get(gcf, 'SelectionType')
    switch sel_typ
        case 'normal'
            disp('User clicked left-mouse button')
            set(src, 'Selected', 'on')
        case 'extend'
            disp('User did a shift-click')
            set(src, 'Selected', 'on')
        case 'alt'
            disp('User did a control-click')
            set(src, 'Selected', 'on')
            set(src, 'SelectionHighlight', 'off')
    end
end
```

Suppose `h` is the handle of a line object and that the `button_down` function is on your MATLAB path. The following statement assigns the function above to the `ButtonDownFcn`:

```
set(h, 'ButtonDownFcn', @button_down)
```

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

**Children**  
vector of handles

The empty matrix; line objects have no children.

**Clipping**  
{on} | off

*Clipping mode.* MATLAB clips lines to the axes plot box by default. If you set `Clipping` to `off`, lines are displayed outside the axes plot box. This can occur if you create a line, set `hold` to `on`, freeze axis scaling (set `axis` to `manual`), and then create a longer line.

## Color

`ColorSpec`

*Line color.* A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

## CreateFcn

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 a line object. You must define this property as a default value for lines or in a call to the `line` function to create a new line object. For example, the statement

```
set(0,'DefaultLineCreateFcn',@line_create)
```

defines a default value for the line `CreateFcn` property on the root level that sets the axes `LineStyleOrder` whenever you create a line object. The callback function must be on your MATLAB path when you execute the above statement.

```
function line_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
axh = get(src,'Parent');
set(axh,'LineStyleOrder','-.-|--')
end
```

MATLAB executes this function after setting all line properties. Setting this property on an existing line object has no effect. The

# Line Properties

---

function must define at least two input arguments (handle of line object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

## DeleteFcn

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

*Delete line callback function.* A callback function that executes when you delete the line object (e.g., when you issue a `delete` command or clear the axes `cla` or figure `clf`). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
    obj_tp = get(src,'Type');
    disp([obj_tp, ' object deleted'])
    disp('Its user data is:')
    disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object’s properties so these values are available to the callback function. The function must define at least two input arguments (handle of line object being deleted and an event structure, which is empty for this property)



The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

## `DisplayName`

string (default is empty string)

*String used by legend for this line object.* The `legend` function uses the string defined by the `DisplayName` property to label this line object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this line 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.

The following code shows how to use the `DisplayName` property from the command line or in an M-file.

```
t = 0:.1:2*pi;  
a(:,1)=sin(t); a(:,2)=cos(t);
```

# Line Properties

---

```
h = plot(a);  
set(h,{'DisplayName'},{'Sine','Cosine'})  
legend show
```

## EraseMode

{normal} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase line objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** (the default) — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- **none** — Do not erase the line when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it, because MATLAB stores no information about its former location.
- **xor** — Draw and erase the line by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the line. However, the line's color depends on the color of whatever is beneath it on the display.
- **background** — Erase the line by drawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`. This damages objects that are behind the erased line, but lines are always properly colored.

## Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR 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.

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

`HitTest`  
{on} | off

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

`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

# Line Properties

---

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

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

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

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

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

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

`Interruptible`  
{on} | off

*Callback routine interruption mode.* The `Interruptible` property controls whether a line callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible`

property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine.

## LineStyle

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

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

Symbol	Line Style
' - '	Solid line (default)
' - - '	Dashed line
' : '	Dotted line
' . - '	Dash-dot line
' none '	No line

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

## LineWidth

scalar

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

## Marker

character (see table)

*Marker symbol.* The `Marker` property specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the table.

# Line Properties

---

Marker Specifier	Description
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No marker (default)

## MarkerEdgeColor

ColorSpec | none | {auto}

*Marker edge color.* The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none specifies no color, which makes nonfilled markers invisible. auto sets MarkerEdgeColor to the same color as the line's Color property.

## MarkerFaceColor

ColorSpec | {none} | auto

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

four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or the figure color, if the axes `Color` property is set to `none` (which is the factory default for axes).

**MarkerSize**  
size in points

*Marker size.* A scalar specifying the size of the marker, in points. The default value for `MarkerSize` is six points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

**Parent**  
handle of axes, `hggroup`, or `hgtransform`

*Parent of line object.* This property contains the handle of the line object's parent. The parent of a line object is the axes that contains it. You can reparent line objects to other axes, `hggroup`, or `hgtransform` objects.

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

**Selected**  
on | off

*Is object selected?* When this property is on, MATLAB displays selection handles if the `SelectionHighlight` property is also on. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

**SelectionHighlight**  
{on} | off

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

# Line Properties

---

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

## Tag

string

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

## Type

string (read only)

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

## UIContextMenu

handle of a `uicontextmenu` object

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

## UserData

matrix

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

## Visible

{on} | off



*Line visibility.* By default, all lines are visible. When set to off, the line is not visible, but still exists, and you can get and set its properties.

## XData

vector of coordinates

*X-coordinates.* A vector of  $x$ -coordinates defining the line. YData and ZData must be the same length and have the same number of rows. (See “Examples” on page 2-2041.)

## YData

vector of coordinates

*Y-coordinates.* A vector of  $y$ -coordinates defining the line. XData and ZData must be the same length and have the same number of rows.

## ZData

vector of coordinates

*Z-coordinates.* A vector of  $z$ -coordinates defining the line. XData and YData must have the same number of rows.

# Lineseries Properties

---

**Purpose** Define lineseries properties

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

See “Plot Objects” for more information on lineseries objects.

Note that you cannot define default properties for lineseries objects.

## Lineseries Property Descriptions

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

Annotation

`hg.Annotation` object Read Only

*Control the display of lineseries objects in legends.* The `Annotation` property enables you to specify whether this lineseries 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 lineseries object is displayed in a figure legend:

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
<code>on</code>	Include the lineseries object in a legend as one entry, but not its children objects
<code>off</code>	Do not include the lineseries or its children in a legend (default)
<code>children</code>	Include only the children of the lineseries as separate entries in the legend

## 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.

# Lineseries 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 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`

vector of handles

The empty matrix; line objects have no children.

## Clipping

{on} | off

*Clipping mode.* MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

## Color

ColorSpec

*Color of the object.* A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

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

## CreateFcn

string or function handle

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

```
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.

# Lineseries Properties

---

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

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

## `DeleteFcn`

string or function handle

*Callback executed during object deletion.* A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

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

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

See the `BeingDeleted` property for related information.

## `DisplayName`

string (default is empty string)

*String used by legend for this lineseries object.* The `legend` function uses the string defined by the `DisplayName` property to label this lineseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this lineseries object’s corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where  $n$  is the number assigned to the object

based on its location in the list of legend entries. However, legend does not set `DisplayName` to this string.

- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

## `EraseMode`

`{normal} | none | xor | background`

*Erase mode.* This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

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

# Lineseries Properties

---

the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to none). That is, it isn't erased correctly if there are objects behind it.

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

## Printing with Nonnormal Erase Modes

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

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

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

`HandleVisibility`  
{on} | callback | off

*Control access to object's handle by command-line users and GUIs.* This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.



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

## Functions Affected by Handle Visibility

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

## Properties Affected by Handle Visibility

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

## Overriding Handle Visibility

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

# Lineseries Properties

---

## Handle Validity

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

---

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

---

## HitTest

{on} | off

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

## Interruptible

{on} | off

*Callback routine interruption mode.* The Interruptible property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to on allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

## LineStyle

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

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

Specifier String	Line Style
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line
none	No line

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

# Lineseries Properties

---

Marker property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
+	Plus sign
o	Circle
*	Asterisk
.	Point
x	Cross
s	Square
d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Five-pointed star (pentagram)
h	Six-pointed star (hexagram)
none	No marker (default)

`MarkerEdgeColor`  
`ColorSpec | none | {auto}`

*Marker edge color.* The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `Color` property.

`MarkerFaceColor`  
`ColorSpec | {none} | auto`

*Marker face color.* The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

**MarkerSize**  
size in points

*Marker size.* A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

**Parent**  
handle of parent axes, `hggroup`, or `hgtransform`

*Parent of this object.* This property contains the handle of the object's parent. The parent is normally the axes, `hggroup`, or `hgtransform` object that contains the object.

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

**Selected**  
on | {off}

*Is object selected?* When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

**SelectionHighlight**  
{on} | off

# Lineseries Properties

---

*Objects are highlighted when selected.* When the `Selected` property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

## Tag

string

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

For example, you might create an `areaseries` object and set the `Tag` property.

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

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

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

## Type

string (read only)

*Class of graphics object.* For `lineseries` objects, `Type` is always the string `line`.

## 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

# Lineseries Properties

---

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 of coordinates



*Y-coordinates.* A vector of *y*-coordinates defining the values along the *y*-axis for the graph. *XData* and *ZData* must be the same length and have the same number of rows.

**YDataSource**  
string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

---

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

---

**ZData**  
vector of coordinates

*Z-coordinates.* A vector defining the *z*-coordinates for the graph. *XData* and *YData* must be the same length and have the same number of rows.

**ZDataSource**  
string (MATLAB variable)

# Lineseries Properties

---

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

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

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


See the `refreshdata` reference page for more information.

---

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

---

**Purpose** Line specification string syntax

**GUI Alternative** To modify the style, width, and color of lines on a graph, use the Property Editor, one of the plotting tools . For details, see The Property Editor in the MATLAB Graphics documentation.

**Description** This page describes how to specify the properties of lines used for plotting. MATLAB graphics give you control over these visual characteristics:

- Line style
- Line width
- Color
- Marker type
- Marker size
- Marker face and edge coloring (for filled markers)

You indicate the line styles, marker types, and colors you want to display using *string specifiers*, detailed in the following tables:

## Line Style Specifiers

Specifier	Line Style
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line

## Marker Specifiers

Specifier	Marker Type
+	Plus sign

## LineSpec (Line Specification)

---

Specifier	Marker Type
o	Circle
*	Asterisk
.	Point (see note below)
x	Cross
'square' or s	Square
'diamond' or d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
'pentagram' or p	Five-pointed star (pentagram)
'hexagram' or h	Six-pointed star (hexagram)

---

**Note** The point (.) marker type does not change size when the specified value is less than 5.

---

### Color Specifiers

Specifier	Color
r	Red
g	Green
b	Blue
c	Cyan
m	Magenta
y	Yellow

# LineStyle (Line Specification)

Specifier	Color
k	Black
w	White

All high-level plotting functions (except for the `ez...` family of function-plotting functions) accept a `LineStyle` argument that defines three components used to specify lines:

- Line style
- Marker symbol
- Color

For example:

```
plot(x,y, '-.or')
```

plots `y` versus `x` using a dash-dot line (`-.`), places circular markers (`o`) at the data points, and colors both line and marker red (`r`). Specify the components (in any order) as a quoted string after the data arguments. Note that linespecs are single strings, not property-value pairs.

## Plotting Data Points with No Line

If you specify a marker, but not a line style, only the markers are plotted. For example:

```
plot(x,y, 'd')
```

## Related Properties

When using the `plot` and `plot3` functions, you can also specify other characteristics of lines using graphics properties:

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

# LineStyle (Line Specification)

---

- `MarkerFaceColor` — Specifies the color of the face of filled markers.
- `MarkerSize` — Specifies the size of the marker in points (must be greater than 0).

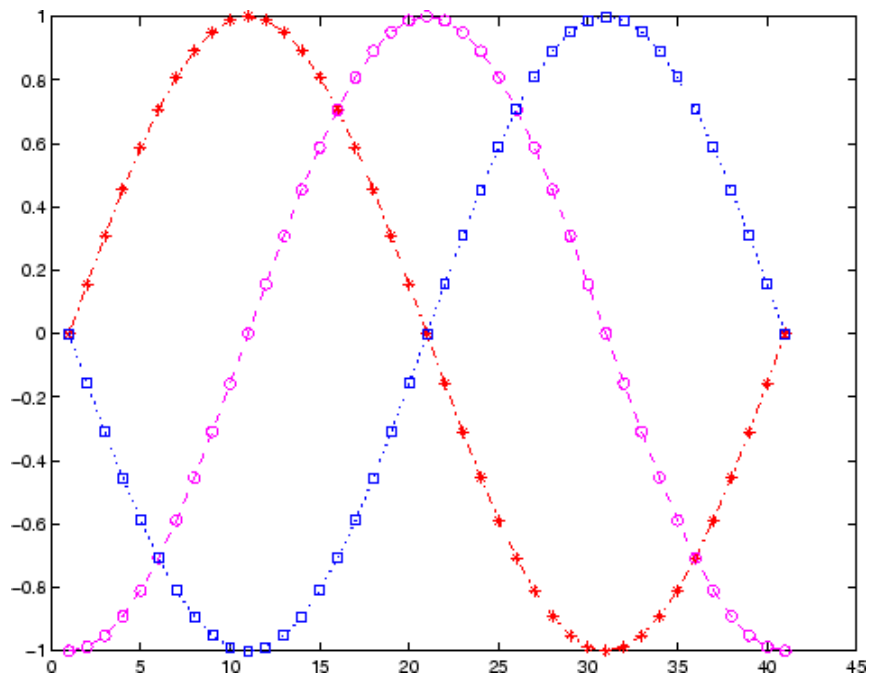
In addition, you can specify the `LineStyle`, `Color`, and `Marker` properties instead of using the symbol string. This is useful if you want to specify a color that is not in the list by using RGB values. See [Line Properties](#) for details on these properties and [ColorSpec](#) for more information on color.

## Examples

Plot the sine function over three different ranges using different line styles, colors, and markers.

```
t = 0:pi/20:2*pi;
plot(t,sin(t),'-.r*')
hold on
plot(t,sin(t-pi/2),'--mo')
plot(t,sin(t-pi),' :bs')
hold off
```

# LineSpec (Line Specification)

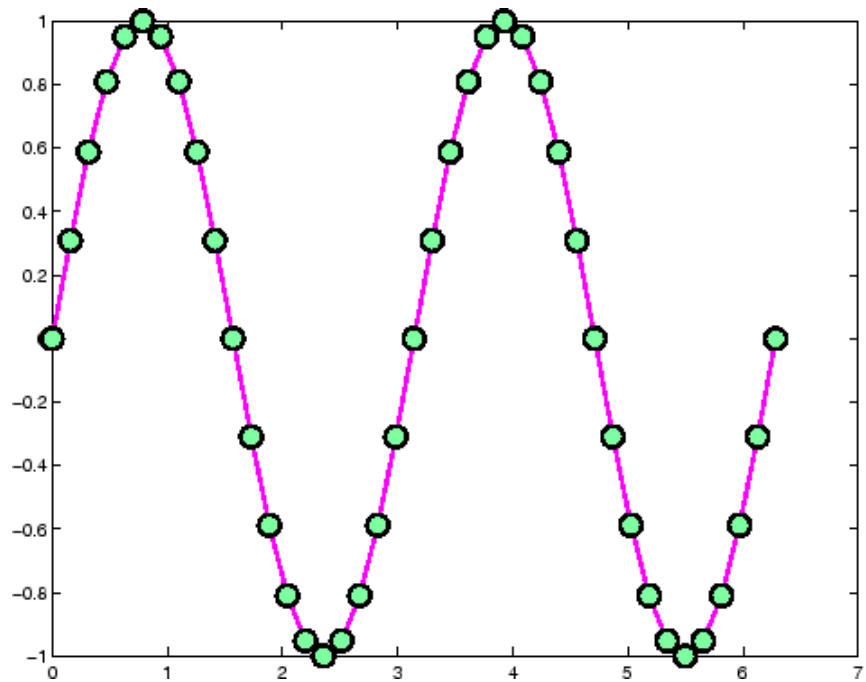


Create a plot illustrating how to set line properties:

```
plot(t,sin(2*t),'-mO',...  
      'LineWidth',2,...  
      'MarkerEdgeColor','k',...  
      'MarkerFaceColor',[.49 1 .63],...  
      'MarkerSize',12)
```

## LineStyle (Line Specification)

---



### See Also

[axes](#), [line](#), [plot](#), [patch](#), [set](#), [surface](#), [Line Properties](#), [ColorSpec](#)

“[Line Styles Used for Plotting — LineStyleOrder](#)” for information about defining an order for applying linestyles

“[Types of MATLAB Plots](#)” for functions that use linespecs

“[Basic Plots and Graphs](#)” on page 1-90 for related functions



**Purpose** Synchronize limits of specified 2-D axes

**Syntax** `linkaxes(axes_handles)`  
`linkaxes(axes_handles, 'option')`

**Description** Use `linkaxes` to synchronize the individual axis limits across several figures or subplots within a figure. Calling `linkaxes` will make all input axes have identical limits. Linking axes is most useful when you want to zoom or pan in one subplot and display the same range of data in another subplot.

`linkaxes(axes_handles)` links the *x*- and *y*-axis limits of the axes specified in the vector `axes_handles`. You can link any number of existing plots or subplots. The `axes_handles` input should be a vector of the handles for each plot or subplot. Entering an array of values results in an error message.

`linkaxes(axes_handles, 'option')` links the axes' `axes_handles` according to the specified option. The *option* argument can be one of the following strings:

<code>x</code>	Link <i>x</i> -axis only
<code>y</code>	Link <i>y</i> -axis only
<code>xy</code>	Link <i>x</i> -axis and <i>y</i> -axis
<code>off</code>	Remove linking

See the `linkprop` function for more advanced capabilities that allow linking object properties on any graphics object.

**Remarks** The first axes provided to `linkaxes` determines the *x*-limits and *y*-limits for all axes linked. This can cause plots to partly or entirely disappear if their limits or scaling are very different. To override this behavior, after calling `linkaxes` specify the limits of the axes that you wish to control with the `set` command, as shown in Example 3, below.

## Examples

You can use interactive zooming or panning (selected from the figure toolbar) to see the effect of axes linking. For example, pan in one graph and notice how the  $x$ -axis also changes in the other. The axes will respond in the same way to zoom and pan directives typed in the Command Window.

### Example 1

This example creates two subplots and links the  $x$ -axis limits of the two axes:

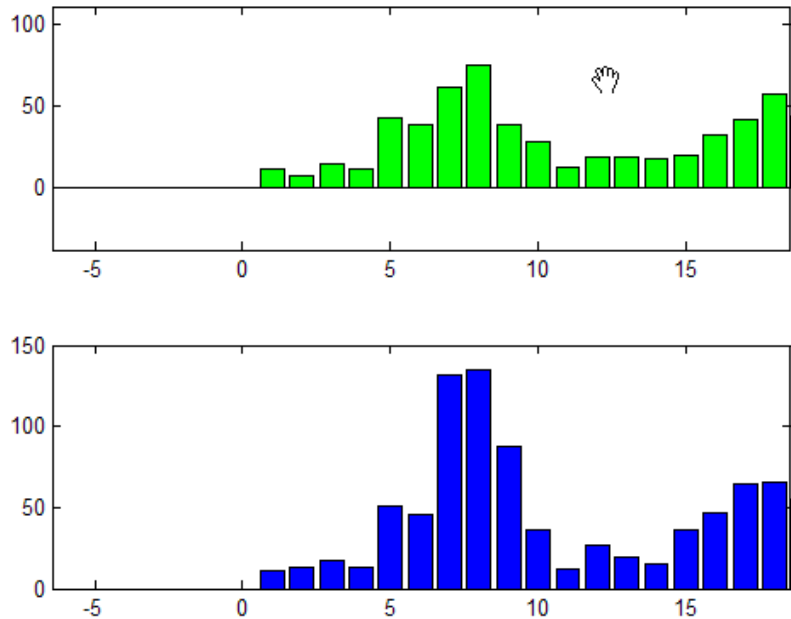
```
ax(1) = subplot(2,2,1);
plot(rand(1,10)*10, 'Parent', ax(1));
ax(2) = subplot(2,2,2);
plot(rand(1,10)*100, 'Parent', ax(2));
linkaxes(ax, 'x');
```

### Example 2

This example creates two figures and links the  $x$ -axis limits of the two axes. The illustration shows the effect of manually panning the top subplot:

```
load count.dat
figure; ax(1) = subplot(2,1,1);
h(1) = bar(ax(1), count(:,1), 'g');
ax(2) = subplot(2,1,2);
h(2) = bar(ax(2), count(:,2), 'b');
linkaxes(ax, 'x');
```

Choose the Pan tool (**Tools > Pan**) and drag the top axes. Both axes will pan in step in  $x$ , but only the top one pans in  $y$ .



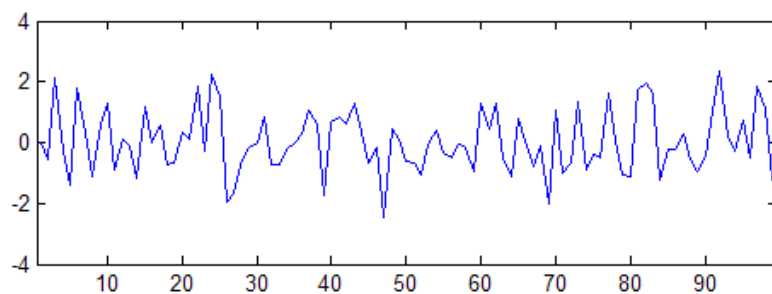
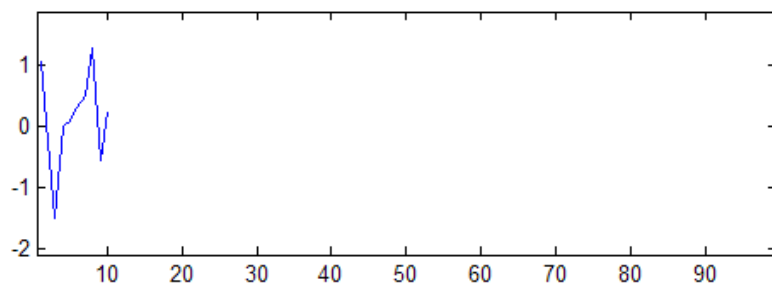
### Example 3

Create two subplots containing data having different ranges. The first axes handle passed to `linkaxes` determines the data range for all other linked axes. In this example, calling `set` for the lower axes overrides the `x`-limits established by the call to `linkaxes`:

```
a1 = subplot(2,1,1);
plot(randn(10,1));      % Plot 10 numbers on top
a2 = subplot(2,1,2);
plot(a2,randn(100,1))  % Plot 100 numbers below
linkaxes([a1 a2], 'x'); % Link the axes; subplot 2 now out of range
set(a2,'xlimmode','auto'); % Now both axes run from 1-100 in x
                           % You could also set(a2,'xlim',[1 100])
```

# linkaxes


---



**See Also** [linkdata](#), [linkprop](#), [pan](#), [zoom](#)

**Purpose** Automatically update graphs when variables change

**GUI Alternatives**

To turn data linking on or off, click the Data Linking tool  in the figure toolbar. When on, an information bar appears below the figure's toolbar to identify and specify data sources for graphs.



For details, see “Making Graphs Responsive with Data Linking” in the MATLAB Data Analysis documentation.

**Syntax**

```
linkdata on
linkdata off
linkdata
linkdata(figure_handle,...)
linkobj = linkdata(figure_handle)
```

**Description**

`linkdata on` turns on data linking for the current figure.

`linkdata off` turns data linking off.

`linkdata` by itself toggles the state of data linking.

`linkdata(figure_handle,...)` applies the function to the specified figure handle.

`linkobj = linkdata(figure_handle)` returns a *linkdata object* for the specified figure. The object has one read-only property, `Enable`, the string 'on' or 'off', depending on the linked state of the figure.

Data linking connects graphs in figure windows to variables in the base or a function's workspace via their `XDataSource`, `YDataSource`, and `ZDataSource` properties. When you turn on data linking for a figure, variables in the current (base or caller) workspace are compared to the `XData`, `YData`, and `ZData` properties of graphs in the affected figure to try to match them. When a match is found, the appropriate `XDataSource`, `YDataSource` and/or `ZDataSource` for the graph are set to strings that name the matching variables.

Any subsequent changes to linked variables are then reflected in graphs that use them as data sources and in the Variable Editor, if the linked variables are displayed there. Conversely, any changes to plotted data values made at the command line, in the Variable Editor, or with the Brush tool (such as deleting or replacing data points), are immediately reflected in the workspace variables linked to the data points.

When a figure containing graphs is linked and any variable identified as `XDataSource`, `YDataSource`, and/or `ZDataSource` changes its values in the workspace, all graphs displaying it in that and other linked figures automatically update. This operation is equivalent to automatically calling the `refreshdata` function on the corresponding figure when a variable changes.

Linked figure windows identify themselves by the appearance of the Linked Plot information bar at the top of the window. When `linkdata` is off for a figure, the Linked Plot information bar is removed. If `linkdata` cannot unambiguously identify data sources for a graph in a linked figure, it reports this via the Linked Plot information bar, which gives the user an opportunity to identify data sources. The information bar displays a warning icon and a message, **Graphics have no data sources** and also prompts **Click here to fix it**. Clicking the word **here** opens the Specify Data Sources dialog box for specifying names and ranges of data sources for the figure.

## Remarks

- “Types of Variables You Can Link” on page 2-2088
- “Breaking and Restoring Links” on page 2-2089
- “Linking Brushed Graphs” on page 2-2089

## Types of Variables You Can Link

You can use `linkdata` to connect a graph with scalar, vector and matrix numeric variables of any class (including `complex`, if the graphing function can plot it) — essentially any data for which `isnumeric` equals `true`. See “Example 3” on page 2-2091 for instructions on linking complex variables. You can also link plots to numeric fields within structures. You can specify MATLAB expressions as data sources, for example, `sqrt(y)+1`.

## Breaking and Restoring Links

Refreshing data on a linked plot fails if the strings in the `XDataSource`, `YDataSource`, or `ZDataSource` properties, when evaluated, are incompatible with what is in the current workspace, such that the corresponding `XData`, `YData`, or `ZData` are unable to respond. The visual appearance of the object in the graph is not affected by such failures, so graphic objects show no indication of broken links. Instead, a warning icon and the message **Failing links** appear on the Linked Plot information bar along with an **Edit** button that opens the Specify Data Sources dialog box.

`linkdata` buffers updates to data and dispatches them to plots at roughly half-second intervals. This makes data linking not suitable for smoothly animating changes in data values unless they are updated in loops that are forced to execute two times per second or less.

## Linking Brushed Graphs

If you link data sources to graphs that have been brushed, their brushing marks can change or vanish. This is because the workspace variables in those graphs now dictate which, if any, observations are brushed, superseding any brushing annotations that were applied to their graphical data (`YData`, etc.). For more details, see “How Data Linking Affects Data Brushing” on page 2-434 in the brush reference page.

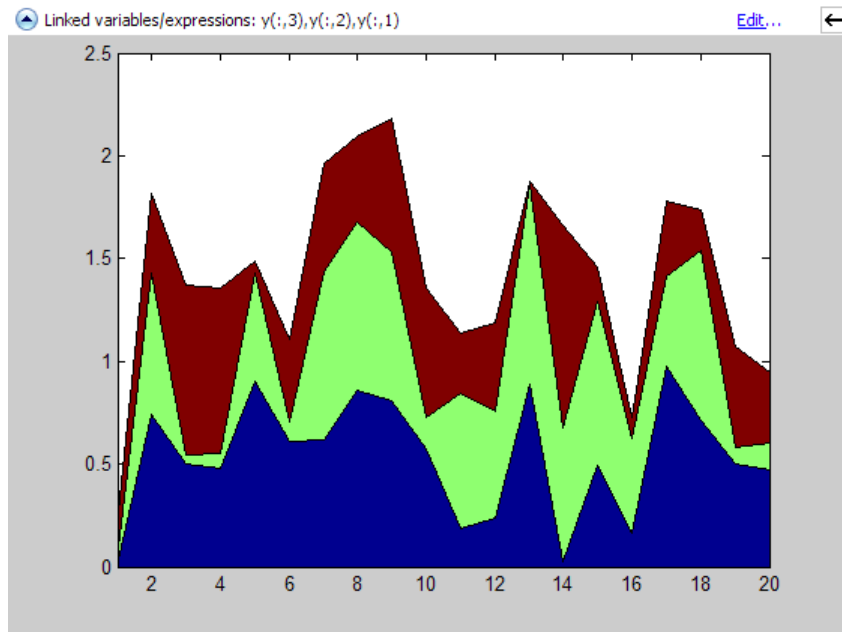
## Examples

### Example 1

Create two variables, graph them as areaseries, and link the plot to them:

```
x = [1:20];  
y = rand(20,3);  
area(x,y)  
linkdata on
```

# linkdata

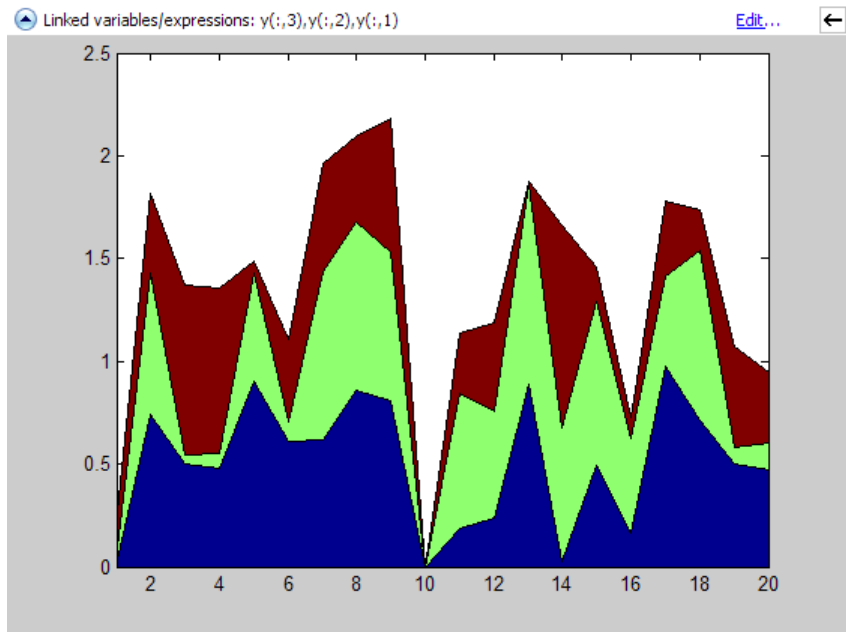


Change values for linked variable  $y$  in the workspace:

```
y(10,:) = 0;
```

The area graph immediately updates.





### Example 2

Delete a figure if it is not linked, based on a returned linkdata object:

```
ld = linkdata(fig)

ld =
    graphics.linkdata

if strcmp(ld.Enable,'off')
    delete(fig)
end
```

### Example 3

If a graphing function can display a complex variable, you can link such plots. To do so, you need to describe the data sources as expressions to separate the real and imaginary parts of the variable. For example,

# linkdata

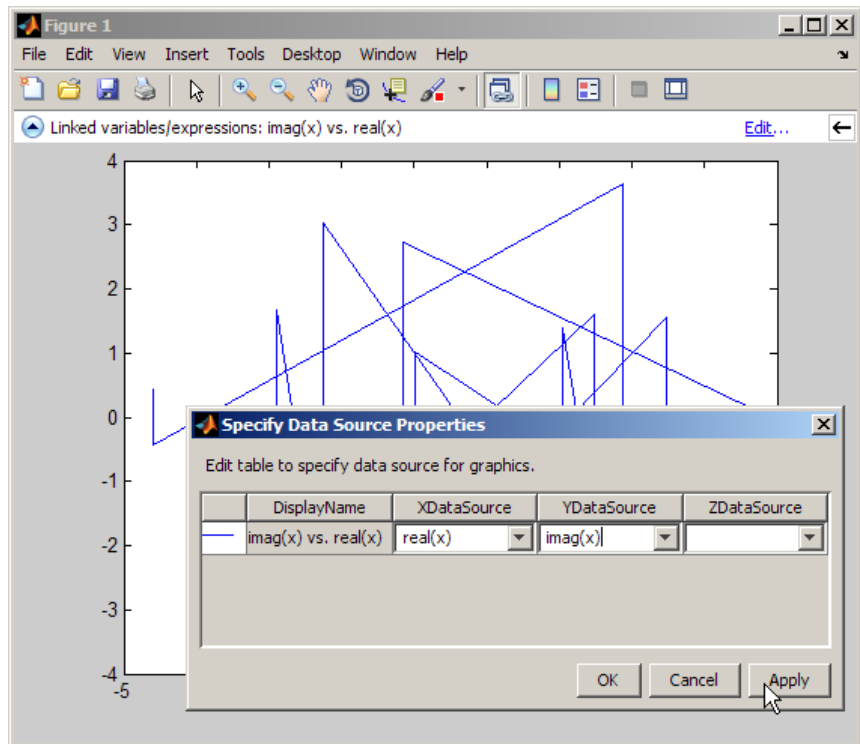
---

```
x = eig(randn(20,20))
whos
  Name      Size      Bytes  Class  Attributes
  x         20x1         320  double complex
```

yields a complex vector. You can use `plot` to display the real portion as  $x$  and the imaginary portion as  $y$ , then link the graph to the variable:

```
plot(x)
linkdata
```

However, `linkdata` cannot unambiguously identify the graph's data sources, and you must tell it by typing `real(x)` and `imag(x)` into the Specify Data Source Properties dialog box that displays when you click **fix it** in the Linked Plot information bar.



To avoid having to type the data source names in the dialog box, you can specify them when you plot:

```
plot(x, 'XDataSource', 'real(x)', 'YDataSource', 'imag(x)')
```

If you subsequently change values of  $x$  programmatically or manually, the plot updates accordingly.

# linkdata

---

---

**Note** Although you can use data brushing on linked plots of complex data, your brush marks only appear in the plot you are brushing, not in other plots or in the Variable Editor. This is because function calls, such as `real(x)` and `imag(x)`, that you specify as data sources are not interpreted when brushing graphed data.

---

## See Also

`brush`, `linkaxes`, `linkprop`, `refreshdata`

---

<b>Purpose</b>	Keep same value for corresponding properties
<b>Syntax</b>	<pre>hlink = linkprop(obj_handles, 'PropertyName') hlink = linkprop(obj_handles, {'PropertyName1', 'PropertyName2', ...})</pre>
<b>Description</b>	<p>Use linkprop to maintain the same values for the corresponding properties of different objects.</p> <p><code>hlink = linkprop(obj_handles, 'PropertyName')</code> maintains the same value for the property <i>PropertyName</i> on all objects whose handles appear in <code>obj_handles</code>. <code>linkprop</code> returns the link object in <code>hlink</code>. See “Link Object” on page 2-2095 for more information.</p> <p><code>hlink = linkprop(obj_handles, {'PropertyName1', 'PropertyName2', ...})</code> maintains the same respective values for all properties passed as a cell array on all objects whose handles appear in <code>obj_handles</code>.</p> <p>Note that the linked properties of all linked objects are updated immediately when <code>linkprop</code> is called. The first object in the list (<code>obj_handles</code>) determines the property values for the rest of the objects.</p>
<b>Link Object</b>	<p>The mechanism to link the properties of different graphics objects is stored in the link object, which is returned by <code>linkprop</code>. Therefore, the link object must exist within the context where you want property linking to occur (such as in the base workspace if users are to interact with the objects from the command line or figure tools).</p> <p>The following list describes ways to maintain a reference to the link object.</p> <ul style="list-style-type: none"><li>• Return the link object as an output argument from a function and keep it in the base workspace while interacting with the linked objects.</li><li>• Make the <code>hlink</code> variable global.</li></ul>

- Store the `hlink` variable in an object's `UserData` property or in application data. See the “Examples” on page 2-2096 section for an example that uses application data.

## Modifying Link Object

If you want to change either the graphics objects or the properties that are linked, you need to use the link object methods designed for that purpose. These methods are functions that operate only on link objects. To use them, you must first create a link object using `linkprop`.

Method	Purpose
<code>addtarget</code>	Add specified graphics object to the link object's targets.
<code>removetarget</code>	Remove specified graphics object from the link object's targets.
<code>addprop</code>	Add specified property to the linked properties.
<code>removeprop</code>	Remove specified property from the linked properties.

### Method Syntax

```
addtarget(hlink, obj_handles)
removetarget(hlink, obj_handles)
addprop(hlink, 'PropertyName')
removeprop(hlink, 'PropertyName')
```

### Arguments

- `hlink` — Link object returned by `linkprop`
- `obj_handles` — One or more graphic object handles
- `PropertyName` — Name of a property common to all target objects

## Examples

This example creates four isosurface graphs of fluid flow data, each displaying a different isovalue. The `CameraPosition` and `CameraUpVector` properties of each subplot axes are linked so that the user can rotate all subplots in unison.

After running the example, select **Rotate 3D** from the figure **Tools** menu and observe how all subplots rotate together.

---

**Note** If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

---

The property linking code is in step 3.

- 1 Define the data using the `flow` M-file and specify property values for the isosurface (which is a patch object).

```
function linkprop_example
[x y z v] = flow;
isoval = [-3 -1 0 1];
props.FaceColor = [0 0 .5];
props.EdgeColor = 'none';
props.AmbientStrength = 1;
props.FaceLighting = 'gouraud';
```

- 2 Create four subplot axes and add an isosurface graph to each one. Add a title and set viewing and lighting parameters using a local function (`set_view`). (`subplot`, `patch`, `isosurface`, `title`, `num2str`)

```
for k = 1:4
    h(k) = subplot(2,2,k);
    patch(isosurface(x,y,z,v,isoval(k)),props)
    title(h(k),['Isovalue = ',num2str(k)])
    set_view(h(k))
end
```

- 3 Link the `CameraPosition` and `CameraTarget` properties of all subplot axes. Since this example function will have completed execution when the user is rotating the subplots, the link object is stored in the first subplot axes application data. See `setappdata` for more information on using application data.

# linkprop

---

```
hlink = linkprop(h,{'CameraPosition','CameraUpVector'});  
key = 'graphics_linkprop';  
% Store link object on first subplot axes  
setappdata(h(1),key,hlink);
```

- 4 The following local function contains viewing and lighting commands issued on each axes. It is called with the creation of each subplot (view, axis, camlight).

```
function set_view(ax)  
% Set the view and add lighting  
view(ax,3); axis(ax,'tight','equal')  
camlight left; camlight right  
% Make axes invisible and title visible  
axis(ax,'off')  
set(get(ax,'title'),'Visible','on')
```

## Linking an Additional Property

Suppose you want to add the axes `PlotBoxAspectRatio` to the linked properties in the previous example. You can do this by modifying the link object that is stored in the first subplot axes' application data.

- 1 First click the first subplot axes to make it the current axes (since its handle was saved only within the creating function). Then get the link object's handle from application data (`getappdata`).

```
hlink = getappdata(gca,'graphics_linkprop');
```

- 2 Use the `addprop` method to add a new property to the link object.

```
addprop(hlink,'PlotBoxAspectRatio')
```

Since `hlink` is a reference to the link object (i.e., not a copy), `addprop` can change the object that is stored in application data.

## See Also

`getappdata`, `linkaxes`, `linkdata`, `setappdata`



**Purpose** Solve linear system of equations

**Syntax**  
`X = linsolve(A,B)`  
`X = linsolve(A,B,opts)`

**Description** `X = linsolve(A,B)` solves the linear system  $A*X = B$  using LU factorization with partial pivoting when  $A$  is square and QR factorization with column pivoting otherwise. The number of rows of  $A$  must equal the number of rows of  $B$ . If  $A$  is  $m$ -by- $n$  and  $B$  is  $m$ -by- $k$ , then  $X$  is  $n$ -by- $k$ . `linsolve` returns a warning if  $A$  is square and ill conditioned or if it is not square and rank deficient.

`[X, R] = linsolve(A,B)` suppresses these warnings and returns  $R$ , which is the reciprocal of the condition number of  $A$  if  $A$  is square, or the rank of  $A$  if  $A$  is not square.

`X = linsolve(A,B,opts)` solves the linear system  $A*X = B$  or  $A'*X = B$ , using the solver that is most appropriate given the properties of the matrix  $A$ , which you specify in `opts`. For example, if  $A$  is upper triangular, you can set `opts.UT = true` to make `linsolve` use a solver designed for upper triangular matrices. If  $A$  has the properties in `opts`, `linsolve` is faster than `mldivide`, because `linsolve` does not perform any tests to verify that  $A$  has the specified properties.

---

**Notes** If  $A$  does not have the properties that you specify in `opts`, `linsolve` returns incorrect results and does not return an error message. If you are not sure whether  $A$  has the specified properties, use `mldivide` instead.

For small problems, there is no speed benefit in using `linsolve` on triangular matrices as opposed to using the `mldivide` function.

---

The `TRANSA` field of the `opts` structure specifies the form of the linear system you want to solve:

# linsolve

- If you set `opts.TRANSA = false`, `linsolve(A,B,opts)` solves  $A*X = B$ .
- If you set `opts.TRANSA = true`, `linsolve(A,B,opts)` solves  $A'*X = B$ .

The following table lists all the field of `opts` and their corresponding matrix properties. The values of the fields of `opts` must be logical and the default value for all fields is `false`.

Field Name	Matrix Property
LT	Lower triangular
UT	Upper triangular
UHESS	Upper Hessenberg
SYM	Real symmetric or complex Hermitian
POSDEF	Positive definite
RECT	General rectangular
TRANSA	Conjugate transpose — specifies whether the function solves $A*X = B$ or $A'*X = B$

The following table lists all combinations of field values in `opts` that are valid for `linsolve`. A `true/false` entry indicates that `linsolve` accepts either true or false.

LT	UT	UHESS	SYM	POSDEF	RECT	TRANSA
true	false	false	false	false	true/false	true/false
false	true	false	false	false	true/false	true/false
false	false	true	false	false	false	true/false
false	false	false	true	true/false	false	true/false
false	false	false	false	false	true/false	true/false

**Example**

The following code solves the system  $A'x = b$  for an upper triangular matrix  $A$  using both `mldivide` and `linsolve`.

```
A = triu(rand(5,3)); x = [1 1 1 0 0]'; b = A'*x;  
y1 = (A')\b  
opts.UT = true; opts.TRANS_A = true;  
y2 = linsolve(A,b,opts)
```

y1 =

```
1.0000  
1.0000  
1.0000  
0  
0
```

y2 =

```
1.0000  
1.0000  
1.0000  
0  
0
```

---

**Note** If you are working with matrices having different properties, it is useful to create an options structure for each type of matrix, such as `opts_sym`. This way you do not need to change the fields whenever you solve a system with a different type of matrix  $A$ .

---

**See Also**

`mldivide`

# linspace

---

**Purpose** Generate linearly spaced vectors

**Syntax** `y = linspace(a,b)`  
`y = linspace(a,b,n)`

**Description** The `linspace` function generates linearly spaced vectors. It is similar to the colon operator `:`, but gives direct control over the number of points.

`y = linspace(a,b)` generates a row vector `y` of 100 points linearly spaced between and including `a` and `b`.

`y = linspace(a,b,n)` generates a row vector `y` of `n` points linearly spaced between and including `a` and `b`.

**See Also** `logspace`

The colon operator `:`

**Purpose** Random number generator algorithms

**Class** @RandStream

**Syntax** RandStream.list

**Description** RandStream.list lists all the generator algorithms that may be used when creating a random number stream with RandStream or RandStream.create. The available generator algorithms and their properties are given in the following table.

<b>Keyword</b>	<b>Generator</b>	<b>Multiple Stream and Substream Support</b>	<b>Approximate Period In Full Precision</b>
mt19937ar	Mersenne twister (used by default stream at MATLAB startup)	No	$2^{19936} - 1$
mcg16807	Multiplicative congruential generator	No	$2^{31} - 2$
mlfg6331_64	Multiplicative lagged Fibonacci generator	Yes	$2^{124}$
mrg32k3a	Combined multiple recursive generator	Yes	$2^{127}$

## list (RandStream)

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
shr3cong	Shift-register generator summed with linear congruential generator	No	$2^{64}$
swb2712	Modified subtract with borrow generator	No	$2^{1492}$

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

All of the generator and transformation algorithms that were available in MATLAB versions 7.6 and earlier are available in the current version. To create random streams that are equivalent to the legacy generators without entering into legacy mode, use the following syntaxes:

Legacy mode	RandStream syntax
<code>rand('seed',0)</code>	<code>(RandStream('mcg16807', 'Seed',0))</code>
<code>rand('state'0)</code>	<code>(RandStream('swb2712', 'Seed',0))</code>
<code>rand('twister',5489)</code>	<code>(RandStream('mt19937ar', 'Seed', 5489))</code>
<code>randn('seed',0)</code>	<code>(RandStream('mcg16807', 'Seed',0))</code>
<code>randn('state',0)</code>	<code>(RandStream('shr3cong'))</code>

For more information on compatibility issues with MATLAB versions 7.6 and earlier, see “Legacy Mode” in the *MATLAB Mathematics* documentation.

# listdlg

---

**Purpose** Create and open list-selection dialog box

**Syntax** [Selection,ok] = listdlg('ListString',S)

**Description** [Selection,ok] = listdlg('ListString',S) creates a modal dialog box that enables you to select one or more items from a list. **Selection** is a vector of indices of the selected strings (in single selection mode, its length is 1). **Selection** is [] when **ok** is 0. **ok** is 1 if you click the **OK** button, or 0 if you click the **Cancel** button or close the dialog box. Double-clicking on an item or pressing **Return** when multiple items are selected has the same effect as clicking the **OK** button. The dialog box has a **Select all** button (when in multiple selection mode) that enables you to select all list items.

Inputs are in parameter/value pairs:

Parameter	Description
'ListString'	Cell array of strings that specify the list box items.
'SelectionMode'	String indicating whether one or many items can be selected: 'single' or 'multiple' (the default).
'ListSize'	List box size in pixels, specified as a two-element vector [width height]. Default is [160 300].
'InitialValue'	Vector of indices of the list box items that are initially selected. Default is 1, the first item.
'Name'	String for the dialog box's title. Default is ''.
'PromptString'	String matrix or cell array of strings that appears as text above the list box. Default is {}.
'OKString'	String for the OK button. Default is 'OK'.
'CancelString'	String for the Cancel button. Default is 'Cancel'.
'uh'	Uicontrol button height, in pixels. Default is 18.



---

Parameter	Description
'fus'	Frame/uicontrol spacing, in pixels. Default is 8.
'ffs'	Frame/figure spacing, in pixels. Default is 8.

---

**Note** A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowState` in the MATLAB Figure Properties.

---

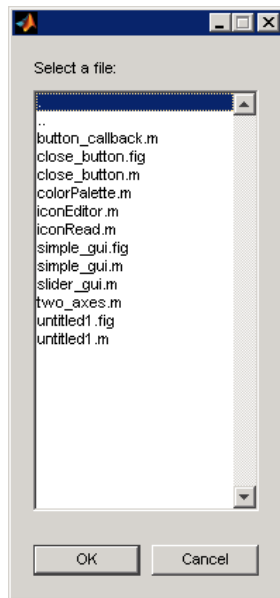
## Example

This example displays a dialog box that enables the user to select a file from the current directory. The function returns a vector. Its first element is the index to the selected file; its second element is 0 if no selection is made, or 1 if a selection is made.

```
d = dir;
str = {d.name};
[s,v] = listdlg('PromptString','Select a file:',...
               'SelectionMode','single',...
               'ListString',str)
```

# listdlg

---



## See Also

dialog, errordlg, helpdlg, inputdlg, msgbox, questdlg, warndlg  
dir, figure, uiwait, uiresume

“Predefined Dialog Boxes” on page 1-108 for related functions

---

<b>Purpose</b>	List available system fonts
<b>Syntax</b>	<code>c = listfonts</code> <code>c = listfonts(h)</code>
<b>Description</b>	<code>c = listfonts</code> returns sorted list of available system fonts. <code>c = listfonts(h)</code> returns sorted list of available system fonts and includes the <code>FontName</code> property of the object with handle <code>h</code> .
<b>Remarks</b>	Calling <code>listfonts</code> returns a list of all fonts residing on your system, possibly including fonts that cannot be used because they are bitmapped. You can instead use the <code>uifont</code> utility (a GUI) to preview fonts you might want to use; it only displays fonts that can be rendered in MATLAB figures and GUIs. Like <code>uifont</code> , the <b>Custom Fonts</b> pane of MATLAB Preferences also previews available fonts and only shows those that can be rendered.
<b>Examples</b>	<b>Example 1</b> This example returns a list of available system fonts similar in format to the one shown. <pre>list = listfonts  list =     'Agency FB'     'Algerian'     'Arial'     ...     'ZapfChancery'     'ZapfDingbats'     'ZWAdobeF'</pre> <b>Example 2</b> This example returns a list of available system fonts with the value of the <code>FontName</code> property, for the object with handle <code>h</code> , sorted into the list.

# listfonts

---

```
h = uicontrol('Style','text','String','My Font','FontName','MyFont');  
list = listfonts(h)
```

```
list =  
    'Agency FB'  
    'Algerian'  
    'Arial'  
    ...  
    'MyFont'  
    ...  
    'ZapfChancery'  
    'ZapfDingbats'  
    'ZWAdobeF'
```

**See Also**      `uisetfont`

**Purpose**

Load workspace variables from disk

**Syntax**

```
load
load filename
load filename X Y Z ...
load filename -regexp expr1 expr2 ...
load -ascii filename
load -mat filename
S = load('arg1', 'arg2', 'arg3', ...)
```

**Description**

`load` loads all the variables from the MAT-file `matlab.mat`, if it exists, or returns an error if the file doesn't exist.

`load filename` loads all the variables from the file specified by `filename`. `filename` is an unquoted string specifying a file name, and can also include a file extension and a full or partial path name. If `filename` has no extension, `load` looks for a file named `filename.mat` and treats it as a binary MAT-file. If `filename` has an extension other than `.mat`, `load` treats the file as ASCII data.

`load filename X Y Z ...` loads just the specified variables `X`, `Y`, `Z`, etc. from the MAT-file. The wildcard `'*'` loads variables that match a pattern (MAT-file only).

`load filename -regexp expr1 expr2 ...` loads those variables that match any of the "Regular Expressions" given by `expr1`, `expr1`, etc.

`load -ascii filename` forces `load` to treat the file as an ASCII file, regardless of file extension. If the file is not numeric text, `load` returns an error. Use `load -ascii` only on files that have been created with the `save -ascii` command.

`load -mat filename` forces `load` to treat the file as a MAT-file, regardless of file extension. If the file is not a MAT-file, `load` returns an error.

`S = load('arg1', 'arg2', 'arg3', ...)` calls `load` using MATLAB *function syntax*, (as opposed to the MATLAB *command syntax* that has been shown thus far). You can use function syntax with any form

of the `load` command shown above, replacing `arg1`, `arg2`, etc. with the arguments shown. For example,

```
S = load('myfile.mat', -regexp , '^Mon', '^Tue')
```

To specify a command line option, such as `-mat`, with the functional form, specify the option as a string argument, and include the hyphen. For example,

```
load('myfile.dat', '-mat')
```

Function syntax enables you to assign values returned by `load` to an output variable. You can also use function syntax when loading from a file having a name that contains space characters, or a filename that is stored in a variable.

If the file you are loading from is a MAT-file, then output `S` is a structure containing fields that match the variables retrieved. If the file contains ASCII data, then `S` is a double-precision array.

## Remarks

For information on any of the following topics related to saving to MAT-files, see “Importing Data From MAT-Files” in the MATLAB Programming Fundamentals documentation:

- Previewing MAT-file contents
- Loading binary data
- Loading ASCII data

You can also use the Current Directory browser to view the contents of a MAT-file without loading it — see “Performing MATLAB Operations in the Current Directory Browser”.

MATLAB saves numeric data in MAT-files in the native byte format. The header of the MAT-file contains a 2-byte Endian Indicator that MATLAB uses to determine the byte format when loading the MAT-file. When MATLAB reads a MAT-file, it determines whether byte-swapping needs to be performed by the state of this indicator.

## Examples

### Example 1 – Loading From a Binary MAT-file

To see what is in the MAT-file prior to loading it, use `whos -file`:

```
whos -file mydata.mat
```

Name	Size	Bytes	Class
javArray	10x1		java.lang.Double[][]
spArray	5x5	84	double array (sparse)
strArray	2x5	678	cell array
x	3x2x2	96	double array
y	4x5	1230	cell array

Clear the workspace and load it from MAT-file `mydata.mat`:

```
clear
load mydata
```

```
whos
```

Name	Size	Bytes	Class
javArray	10x1		java.lang.Double[][]
spArray	5x5	84	double array (sparse)
strArray	2x5	678	cell array
x	3x2x2	96	double array
y	4x5	1230	cell array

### Example 2 – Loading a List of Variables

You can use a comma-separated list to pass the names of those variables you want to load from a file. This example generates a comma-separated list from a cell array

In this example, the file name is stored in a variable, `saved_file`. You must call `load` using the function syntax of the command if you intend to reference the file name through a variable:

```
saved_file = 'myfile.mat';
saved_file = 'ptarray.mat';
whos('-file', saved_file)
```

# load

---

Name	Size	Bytes	Class
AName	1x24	48	char array
AVal	1x1	8	double array
BName	1x24	48	char array
BVal	1x1	8	double array
CVal	5x5	84	double array (sparse)
DArr	2x5	678	cell array

```
filevariables = {'AName', 'BVal', 'DArr'};  
load(saved_file, filevariables{:});
```

The second part of this example generates a comma-separated list from the name field of a structure array, and loads the first ten variables from the specified file:

```
saved_file = 'myfile.mat';  
vars = whos('-file', saved_file);  
load(saved_file, vars(1:10).name);
```

### Example 3 – Loading From an ASCII File

Create several 4-column matrices and save them to an ASCII file:

```
a = magic(4); b = ones(2, 4) * -5.7; c = [8 6 4 2];  
save -ascii mydata.dat
```

Clear the workspace and load it from the file `mydata.dat`. If the filename has an extension other than `.mat`, MATLAB assumes that it is ASCII:

```
clear  
load mydata.dat
```

MATLAB loads all data from the ASCII file, merges it into a single matrix, and assigns the matrix to a variable named after the filename:

```
mydata
```



```
mydata =
    16.0000    2.0000    3.0000   13.0000
     5.0000   11.0000   10.0000    8.0000
     9.0000    7.0000    6.0000   12.0000
     4.0000   14.0000   15.0000    1.0000
    -5.7000   -5.7000   -5.7000   -5.7000
    -5.7000   -5.7000   -5.7000   -5.7000
     8.0000    6.0000    4.0000    2.0000
```

### Example 4 – Using Regular Expressions

Using regular expressions, load from MAT-file `mydata.mat` those variables with names that begin with `Mon`, `Tue`, or `Wed`:

```
load('mydata', '-regexp', '^Mon|^Tue|^Wed');
```

Here is another way of doing the same thing. In this case, there are three separate expression arguments:

```
load('mydata', '-regexp', '^Mon', '^Tue', '^Wed');
```

### See Also

`clear`, `fprintf`, `fscanf`, `partialpath`, `save`, `spconvert`, `who`

# load (COM)

---

**Purpose** Initialize control object from file

**Syntax** `h.load('filename')`  
`load(h, 'filename')`

**Description** `h.load('filename')` initializes the COM object associated with the interface represented by the MATLAB COM object `h` from file specified in the string `filename`. The file must have been created previously by serializing an instance of the same control.

`load(h, 'filename')` is an alternate syntax for the same operation.

---

**Note** The COM load function is only supported for controls at this time.

---

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

**Examples** Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl1.2', [0 0 200 200], f);
h.save('mwsample')
```

Now, alter the figure by changing its label and the radius of the circle:

```
h.Label = 'Circle';
h.Radius = 50;
h.Redraw;
```

Using the `load` function, you can restore the control to its original state:

```
h.load('mwsample');
h.get
```

MATLAB displays the original values:

```
ans =
```

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

**See Also**

save (COM), actxcontrol, actxserver, release, delete (COM)

# load (serial)

---

**Purpose** Load serial port objects and variables into MATLAB workspace

**Syntax**  
`load filename`  
`load filename obj1 obj2...`

**Description**  
`load filename` returns all variables from the MAT-file specified by `filename` into the MATLAB workspace.  
`load filename obj1 obj2...` returns the serial port objects specified by `obj1 obj2 ...` from the MAT-file `filename` into the MATLAB workspace.  
`out = load('filename','obj1','obj2',...)` returns the specified serial port objects from the MAT-file `filename` as a structure to `out` instead of directly loading them into the workspace. The field names in `out` match the names of the loaded serial port objects.

**Remarks** Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

**Example** Suppose you create the serial port objects `s1` and `s2`, configure a few properties for `s1`, and connect both objects to their instruments:

```
s1 = serial('COM1');  
s2 = serial('COM2');  
set(s1,'Parity','mark','DataBits',7);  
fopen(s1);  
fopen(s2);
```

Save `s1` and `s2` to the file `MyObject.mat`, and then load the objects back into the workspace:

```
save MyObject s1 s2;  
load MyObject s1;  
load MyObject s2;  
  
get(s1, {'Parity', 'DataBits'})
```

```
ans =  
    'mark'      [7]  
get(s2, {'Parity', 'DataBits'})  
ans =  
    'none'      [8]
```

## See Also

### Functions

save

### Properties

Status

# loadlibrary

---

**Purpose** Load shared library into MATLAB software

**Syntax**

```
loadlibrary('shrlib', 'hfile')
loadlibrary('shrlib', @protofile)
loadlibrary('shrlib', ..., 'options')
loadlibrary shrlib hfile options
[notfound, warnings] = loadlibrary('shrlib', 'hfile')
```

**Description** `loadlibrary('shrlib', 'hfile')` loads the functions defined in header file `hfile` and found in shared library `shrlib` into MATLAB.

The `hfile` and `shrlib` file names are case sensitive. The name you use in `loadlibrary` must use the same case as the file on your system.

On Microsoft Windows systems, `shrlib` refers to the name of a dynamic link library (`.dll`) file. On Linux systems, it refers to the name of a shared object (`.so`) file. On Apple Macintosh systems, it refers to a dynamic shared library (`.dylib`). See “File Extensions for Libraries” on page 2-2120 for more information.

`loadlibrary('shrlib', @protofile)` uses the prototype M-file `protofile` in place of a header file in loading the library `shrlib`. The string `@protofile` specifies a function handle to the prototype M-file. (See the description of “Prototype M-Files” on page 2-2123 below).

---

**Note** The MATLAB Generic Shared Library interface does not support library functions that have function pointer inputs.

---

## File Extensions for Libraries

If you do not include a file extension with the `shrlib` argument, `loadlibrary` attempts to find the library with either the appropriate platform MEX-file extension or the appropriate platform library extension (usually `.dll`, `.so`, or `.dylib`). For a list of file extensions, see “Binary MEX-File Extensions”.

If you do not include a file extension with the second argument, and this argument is not a function handle, `loadlibrary` uses `.h` for the extension.

`loadlibrary('shrlib', ..., 'options')` loads the library `shrlib` with one or more of the following *options*.

Option	Description
<b>addheader</b> hfileN	<p>Loads the functions defined in the additional header file, <code>hfileN</code>. Note that each file specified by <code>addheader</code> must be referenced by a corresponding <code>#include</code> statement in the base header file.</p> <p>Specify the string <code>hfileN</code> as a file name without a file extension. MATLAB does not verify the existence of the header files and ignores any that are not needed.</p> <p>You can specify additional header files using the syntax:</p> <pre>loadlibrary shrlib hfile ...     addheader hfile1 ...     addheader hfile2 ...      % and so on</pre>
<b>alias</b> name	Associates the specified alias name with the library. All subsequent calls to MATLAB functions that reference this library must use this alias until the library is unloaded.
<b>includepath</b> path	Specifies an additional path in which to look for included header files.

# loadlibrary

---

Option	Description
<b>mfilename</b> <i>mfile</i>	Generates a prototype M-file <i>mfile</i> in the current directory. You can use this file in place of a header file when loading the library. (See the following description of “Prototype M-Files” on page 2-2123).
<b>thunkfilename</b> <i>tfile</i>	Overrides the default thunk file name with <i>tfile</i> . For more information, see “Using loadlibrary on 64-Bit Platforms” on page 2-2124.

Only the **alias** option is available when loading using a prototype M-file.

If you have more than one library file of the same name, load the first using the library file name, and load the additional libraries using the **alias** option.

`loadlibrary shrlib hfile options` is the command format for this function.

`[notfound, warnings] = loadlibrary('shrlib', 'hfile')` returns warning information from the `shrlib` library file. `notfound` is a cell array of the names of functions found in the header file `hfile`, or any header added with the `addheader` option, but not found in the `shrlib` library. `warnings` contains a single character array of warnings produced while processing the header file `hfile`.

## Remarks

### How to Use the `addheader` Option

The `addheader` option enables you to add functions for MATLAB to load from those listed in header files included in the base header file (with a `#include` statement). For example, if your library header file contains the statement:

```
#include header2.h
```

then to load the functions in `header2.h`, you need to use `addheader` in the call to `loadlibrary`:



```
loadlibrary libname libname.h addheader header2.h
```

You can use the `addheader` option with a header file that lists function prototypes for only the functions that are needed by your library, and thereby avoid loading functions that you do not define in your library. To do this, you might need to create a header file that contains a subset of the functions listed in large header file.

## addheader Syntax

When using `addheader` to specify which functions to load, ensure that there are `#include` statements in the base header file for each additional header file in the `loadlibrary` call. For example, to use the following statement:

```
loadlibrary mylib mylib.h addheader header2.h
```

the file `mylib.h` must contain this statement:

```
#include header2.h
```

## Prototype M-Files

When you use the `mfilename` option with `loadlibrary`, MATLAB generates an M-file called a *prototype file*. Use this file on subsequent calls to `loadlibrary` in place of a header file.

Like a header file, the prototype file supplies MATLAB with function prototype information for the library. You can make changes to the prototypes by editing this file and reloading the library.

Here are some reasons for using a prototype file, along with the changes you would need to make to the file:

- You want to make temporary changes to signatures of the library functions.

Edit the prototype file, changing the `fcns.LHS` or `fcns.RHS` field for that function. This changes the types of arguments on the left hand side or right hand side, respectively.

- You want to rename some of the library functions.

# loadlibrary

---

Edit the prototype file, defining the `fcns.alias` field for that function.

- You expect to use only a small percentage of the functions in the library you are loading.

Edit the prototype file, commenting out the unused functions. This reduces the amount of memory required for the library.

- You need to specify a number of include files when loading a particular library.

Specify the full list of include files (plus the `mfilename` option) in the first call to `loadlibrary`. This puts all the information from the include files into the prototype file. After that, specify just the prototype file.

## Using loadlibrary on 64-Bit Platforms

You must install a C compiler to use `loadlibrary` on a 64-bit platform and Perl must be available. The supported compilers are shown in the following table.

64-bit Platform	Required Compiler
Windows	Microsoft® Visual C++® 2005 SP1 Version 8.0 Professional Edition
Linux	gcc / g++ Version 4.1.1
Sun Solaris SPARC	Sun Studio 12 cc / CC Version 5.9

MATLAB generates a *thunk file*, which is a compatibility layer to your 64-bit library. The name of the thunk file is:

```
BASENAME_thunk_COMPUTER.c
```

where *BASENAME* is either the name of the shared library or the *mfilename*, if specified. *COMPUTER* is the string returned by the `computer` function.

MATLAB compiles this file and creates the file:

```
BASENAME_thunk_COMPUTER.LIBEXT
```

where *LIBEXT* is the platform-dependent default shared library extension, for example, *dll* on Windows.

## Examples

### Load `shrlibsample` Example

Use `loadlibrary` to load the MATLAB sample shared library, `shrlibsample`:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h
```

### Using `alias` Example

Load sample library `shrlibsample`, giving it an alias name of `lib`. Once you have set an alias, you need to use this name in all further interactions with the library for this session:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h alias lib

libfunctionsview lib

str = 'This was a Mixed Case string';
calllib('lib', 'stringToUpper', str)
ans =
    THIS WAS A MIXED CASE STRING
unloadlibrary lib
```

### Using `addpath` Example

Load the library, specifying an additional path in which to search for included header files:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary('shrlibsample','shrlibsample.h','includepath', ...
    fullfile(matlabroot , 'extern', 'include'));
```

### Using `Prototype` Example

Load the `libmx` library and generate a prototype M-file containing the prototypes defined in header file `matrix.h`:

# loadlibrary

---

```
hfile = [matlabroot '\extern\include\matrix.h'];  
loadlibrary('libmx', hfile, 'mfilename', 'mxproto')
```

```
dir mxproto.m  
    mxproto.m
```

Edit the generated file `mxproto.m` and locate the function `mxGetNumberOfDimensions`. Give it an alias of `mxGetDims` by adding this text to the line before `fcnNum` is incremented:

```
fcns.alias{fcnNum}='mxGetDims';
```

Here is the new function prototype. The change is shown in bold:

```
fcns.name{fcnNum}='mxGetNumberOfDimensions';  
fcns.calltype{fcnNum}='cdecl';  
fcns.LHS{fcnNum}='int32';  
fcns.RHS{fcnNum}={'MATLAB array'};  
fcns.alias{fcnNum}= mxGetDims ; % Alias defined  
fcnNum=fcnNum+1; % Increment fcnNum
```

Unload the library and then reload it using the prototype M-file.

```
unloadlibrary libmx  
  
loadlibrary('libmx', @mxproto)
```

Now call `mxGetNumberOfDimensions` using the alias function name:

```
y = rand(4, 7, 2);  
  
calllib('libmx', 'mxGetDims', y)  
ans =  
    3  
  
unloadlibrary libmx
```

## See Also

`unloadlibrary`, `libisloaded`, `libfunctions`,

<b>Purpose</b>	User-defined class method called by load function
<b>Syntax</b>	<code>b = loadobj (a)</code>
<b>Description</b>	<p><code>b = loadobj (a)</code> is called by the load function for user objects. When an object is loaded from a MAT-file, the load function calls the <code>loadobj</code> method for the object's class, if one is defined.</p> <p>You must define the <code>loadobj</code> method as a static method so it can accept as an argument whatever object or structure was saved in the MAT-file, as determined by the output of the <code>saveobj</code> method. Note that if you define a <code>saveobj</code> method that modifies the form of the objects saved by the <code>save</code> command, then you must implement a <code>loadobj</code> method that can reconvert the data saved to a proper object.</p> <p>The input argument <code>a</code> can be the object as loaded from the MAT-file, a structure created by <code>load</code> if the object cannot be resolved, or some other variable that was output by the <code>saveobj</code> method. The output argument <code>b</code> is the object that the load function loads into the workspace.</p> <p>See "Saving and Loading Objects" for more information.</p>
<b>Remarks</b>	<p><code>loadobj</code> can be overloaded only for user objects. <code>load</code> does not call <code>loadobj</code> for built-in data types (such as <code>double</code>).</p> <p><code>loadobj</code> is invoked separately for each object in the MAT-file. The load function recursively descends cell arrays and structures, applying the <code>loadobj</code> method to each object encountered.</p> <p>A child object inherits the <code>loadobj</code> method of its parent class. First the child object's <code>loadobj</code> method is called, then the parents <code>loadobj</code> is called. Note that this behavior is different from that of the <code>saveobj</code> method, which is not inherited from its parent.</p>
<b>See Also</b>	<code>load</code> , <code>save</code> , <code>saveobj</code>

# log

---

**Purpose** Natural logarithm

**Syntax**  $Y = \log(X)$

**Description** The log function operates element-wise on arrays. Its domain includes complex and negative numbers, which may lead to unexpected results if used unintentionally.

$Y = \log(X)$  returns the natural logarithm of the elements of  $X$ . For complex or negative  $z$ , where  $z = x + y*i$ , the complex logarithm is returned.

$$\log(z) = \log(\text{abs}(z)) + i*\text{atan2}(y,x)$$

**Examples** The statement `abs(log(-1))` is a clever way to generate  $\pi$ .

```
ans =
```

```
3.1416
```

**See Also** `exp`, `log10`, `log2`, `logm`, `reallog`

**Purpose** Common (base 10) logarithm

**Syntax**  $Y = \log_{10}(X)$

**Description** The `log10` function operates element-by-element on arrays. Its domain includes complex numbers, which may lead to unexpected results if used unintentionally.

$Y = \log_{10}(X)$  returns the base 10 logarithm of the elements of  $X$ .

**Examples** `log10(realmax)` is 308.2547

and

`log10(eps)` is -15.6536

**See Also** `exp`, `log`, `log2`, `logm`

# log1p

---

**Purpose** Compute  $\log(1+x)$  accurately for small values of  $x$

**Syntax**  $y = \log1p(x)$

**Description**  $y = \log1p(x)$  computes  $\log(1+x)$ , compensating for the roundoff in  $1+x$ .  $\log1p(x)$  is more accurate than  $\log(1+x)$  for small values of  $x$ . For small  $x$ ,  $\log1p(x)$  is approximately  $x$ , whereas  $\log(1+x)$  can be zero.

**See Also** `log`, `expm1`



**Purpose** Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

**Syntax**  $Y = \log_2(X)$   
 $[F, E] = \log_2(X)$

**Description**  $Y = \log_2(X)$  computes the base 2 logarithm of the elements of  $X$ .  
 $[F, E] = \log_2(X)$  returns arrays  $F$  and  $E$ . Argument  $F$  is an array of real values, usually in the range  $0.5 \leq \text{abs}(F) < 1$ . For real  $X$ ,  $F$  satisfies the equation:  $X = F \cdot 2.^E$ . Argument  $E$  is an array of integers that, for real  $X$ , satisfy the equation:  $X = F \cdot 2.^E$ .

**Remarks** This function corresponds to the ANSI C function `frexp()` and the IEEE floating-point standard function `logb()`. Any zeros in  $X$  produce  $F = 0$  and  $E = 0$ .

**Examples** For IEEE arithmetic, the statement  $[F, E] = \log_2(X)$  yields the values:

<b>X</b>	<b>F</b>	<b>E</b>
1	1/2	1
pi	pi/4	2
-3	-3/4	2
eps	1/2	-51
realmax	1 - eps/2	1024
realmin	1/2	-1021

**See Also** `log`, `pow2`

# logical

---

**Purpose** Convert numeric values to logical

**Syntax** `K = logical(A)`

**Description** `K = logical(A)` returns an array that can be used for logical indexing or logical tests.

`A(B)`, where `B` is a logical array that is the same size as `A`, returns the values of `A` at the indices where the real part of `B` is nonzero.

`A(B)`, where `B` is a logical array that is smaller than `A`, returns the values of column vector `A(:)` at the indices where the real part of column vector `B(:)` is nonzero.

**Remarks** Most arithmetic operations remove the logicalness from an array. For example, adding zero to a logical array removes its logical characteristic. `A = +A` is the easiest way to convert a logical array, `A`, to a numeric double array.

Logical arrays are also created by the relational operators (`==`, `<`, `>`, `~`, etc.) and functions like `any`, `all`, `isnan`, `isinf`, and `isfinite`.

**Examples** Given `A = [1 2 3; 4 5 6; 7 8 9]`, the statement `B = logical(eye(3))` returns a logical array

```
B =  
    1    0    0  
    0    1    0  
    0    0    1
```

which can be used in logical indexing that returns `A`'s diagonal elements:

```
A(B)
```

```
ans =  
    1  
    5  
    9
```

However, attempting to index into A using the *numeric* array `eye(3)` results in:

```
A(eye(3))  
??? Subscript indices must either be real positive integers or  
logicals.
```

## See Also

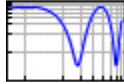
`islogical`, logical operators (elementwise and short-circuit),

# loglog


---

## Purpose

Log-log scale plot



## GUI Alternatives

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

## Syntax

```
loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,LineStyle,...)
loglog(...,'PropertyName',PropertyValue,...)
h = loglog(...)
hlines = loglog('v6',...)
```

## Description

`loglog(Y)` plots the columns of `Y` versus their index if `Y` contains real numbers. If `Y` contains complex numbers, `loglog(Y)` and `loglog(real(Y),imag(Y))` are equivalent. `loglog` ignores the imaginary component in all other uses of this function.

`loglog(X1,Y1,...)` plots all  $X_n$  versus  $Y_n$  pairs. If only  $X_n$  or  $Y_n$  is a matrix, `loglog` plots the vector argument versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`loglog(X1,Y1,LineStyle,...)` plots all lines defined by the  $X_n, Y_n, LineSpec$  triples, where `LineStyle` determines line type, marker symbol, and color of the plotted lines. You can mix  $X_n, Y_n, LineSpec$  triples with  $X_n, Y_n$  pairs, for example,

```
loglog(X1,Y1,X2,Y2,LineStyle,X3,Y3)
```

`loglog(..., 'PropertyName', PropertyValue, ...)` sets property values for all lineseries graphics objects created by `loglog`. See the [line reference page](#) for more information.

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

### Backward-Compatible Version

`hlines = loglog('v6', ...)` returns the handles to line objects instead of lineseries 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 MATLAB version.

---

See [Plot Objects and Backward Compatibility](#) for more information.

### Remarks

If you do not specify a color when plotting more than one line, `loglog` automatically cycles through the colors and line styles in the order specified by the current axes.

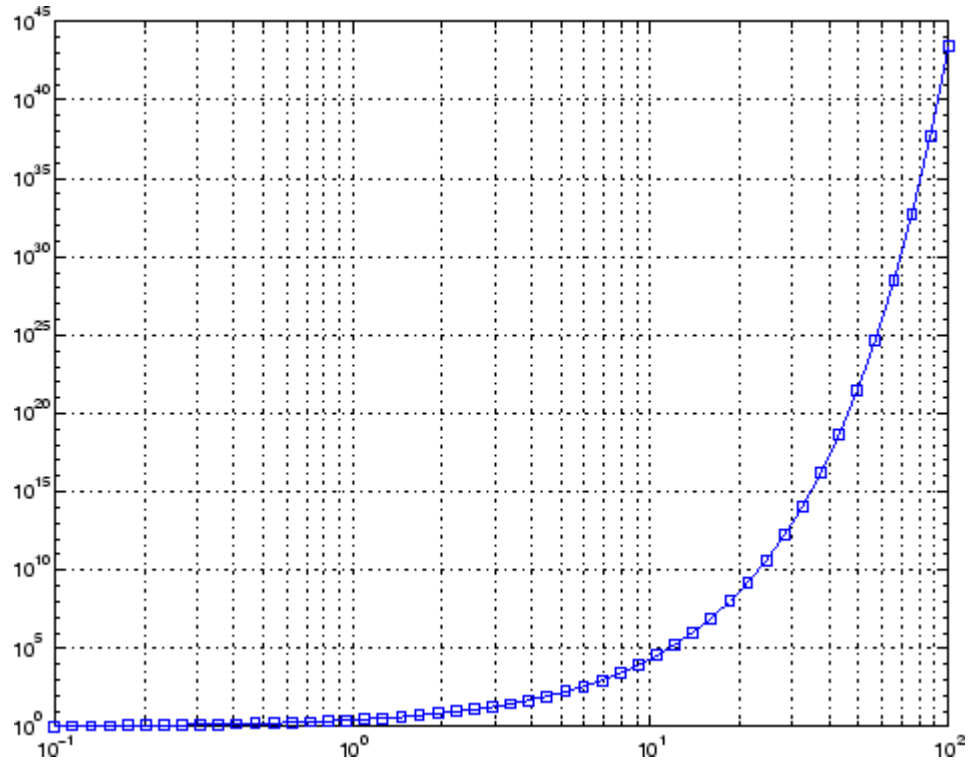
If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode will remain as it is and the new data will plot as linear.

### Examples

Create a simple `loglog` plot with square markers.

```
x = logspace(-1,2);
loglog(x,exp(x), '-s')
grid on
```

# loglog



## See Also

LineStyle, plot, semilogx, semilogy

“Basic Plots and Graphs” on page 1-90 for related functions

**Purpose** Matrix logarithm

**Syntax**  $L = \text{logm}(A)$   
 $[L, \text{exitflag}] = \text{logm}(A)$

**Description**  $L = \text{logm}(A)$  is the principal matrix logarithm of  $A$ , the inverse of  $\text{expm}(A)$ .  $L$  is the unique logarithm for which every eigenvalue has imaginary part lying strictly between  $-\pi$  and  $\pi$ . If  $A$  is singular or has any eigenvalues on the negative real axis, the principal logarithm is undefined. In this case, `logm` computes a non-principal logarithm and returns a warning message.

$[L, \text{exitflag}] = \text{logm}(A)$  returns a scalar `exitflag` that describes the exit condition of `logm`:

- If `exitflag = 0`, the algorithm was successfully completed.
- If `exitflag = 1`, too many matrix square roots had to be computed. However, the computed value of  $L$  might still be accurate. This is different from R13 and earlier versions that returned an expensive and often inaccurate error estimate as the second output argument.

The input  $A$  can have class `double` or `single`.

**Remarks** If  $A$  is real symmetric or complex Hermitian, then so is  $\text{logm}(A)$ .

Some matrices, like  $A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ , do not have any logarithms, real or complex, so `logm` cannot be expected to produce one.

**Limitations** For most matrices:

$$\text{logm}(\text{expm}(A)) = A = \text{expm}(\text{logm}(A))$$

These identities may fail for some  $A$ . For example, if the computed eigenvalues of  $A$  include an exact zero, then  $\text{logm}(A)$  generates infinity. Or, if the elements of  $A$  are too large,  $\text{expm}(A)$  may overflow.

## Examples

Suppose A is the 3-by-3 matrix

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & -1 \end{bmatrix}$$

and  $Y = \text{expm}(A)$  is

$$Y = \begin{bmatrix} 2.7183 & 1.7183 & 1.0862 \\ 0 & 1.0000 & 1.2642 \\ 0 & 0 & 0.3679 \end{bmatrix}$$

Then  $A = \text{logm}(Y)$  produces the original matrix A.

$$Y = \begin{bmatrix} 1.0000 & 1.0000 & 0.0000 \\ 0 & 0 & 2.0000 \\ 0 & 0 & -1.0000 \end{bmatrix}$$

But  $\text{log}(A)$  involves taking the logarithm of zero, and so produces

$$\text{ans} = \begin{bmatrix} 0.0000 & 0 & -35.5119 \\ -\text{Inf} & -\text{Inf} & 0.6931 \\ -\text{Inf} & -\text{Inf} & 0.0000 + 3.1416i \end{bmatrix}$$

## Algorithm

The algorithm `logm` uses is described in [1].

## See Also

`expm`, `funm`, `sqrtm`

## References

[1] Davies, P. I. and N. J. Higham, "A Schur-Parlett algorithm for computing matrix functions," *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Cheng, S. H., N. J. Higham, C. S. Kenney, and A. J. Laub, "Approximating the logarithm of a matrix to specified accuracy," *SIAM J. Matrix Anal. Appl.*, Vol. 22, Number 4, pp. 1112-1125, 2001.



[3] Higham, N. J., "Evaluating Pade approximants of the matrix logarithm," *SIAM J. Matrix Anal. Appl.*, Vol. 22, Number 4, pp. 1126-1135, 2001.

[4] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Johns Hopkins University Press, 1983, p. 384.

[5] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review* 20, 1978, pp. 801-836.

# logspace

---

## Purpose

Generate logarithmically spaced vectors

## Syntax

```
y = logspace(a,b)
y = logspace(a,b,n)
y = logspace(a,pi)
```

## Description

The `logspace` function generates logarithmically spaced vectors. Especially useful for creating frequency vectors, it is a logarithmic equivalent of `linspace` and the “:” or colon operator.

`y = logspace(a,b)` generates a row vector `y` of 50 logarithmically spaced points between decades  $10^a$  and  $10^b$ .

`y = logspace(a,b,n)` generates `n` points between decades  $10^a$  and  $10^b$ .

`y = logspace(a,pi)` generates the points between  $10^a$  and  $\pi$ , which is useful for digital signal processing where frequencies over this interval go around the unit circle.

## Remarks

All the arguments to `logspace` must be scalars.

## See Also

`linspace`

The colon operator :

<b>Purpose</b>	Search for keyword in all help entries
<b>GUI Alternatives</b>	As an alternative to the <code>lookfor</code> function, you can use the Function Browser.
<b>Syntax</b>	<pre>lookfor topic lookfor topic -all</pre>
<b>Description</b>	<p><code>lookfor topic</code> searches for the string <code>topic</code> in the first comment line (the H1 line) of the help text in all M-files found on the MATLAB search path. For all files in which a match occurs, <code>lookfor</code> displays the H1 line.</p> <p><code>lookfor topic -all</code> searches the entire first comment block of an M-file looking for <code>topic</code>.</p>
<b>Examples</b>	<p>For example:</p> <pre>lookfor inverse</pre> <p>finds at least a dozen matches, including H1 lines containing "inverse hyperbolic cosine," "two-dimensional inverse FFT," and "pseudoinverse." Contrast this with</p> <pre>which inverse</pre> <p>or</p> <pre>what inverse</pre> <p>These functions run more quickly, but probably fail to find anything because MATLAB does not have a function <code>inverse</code>.</p> <p>In summary, <code>what</code> lists the functions in a given directory, <code>which</code> finds the directory containing a given function or file, and <code>lookfor</code> finds all functions in all directories that might have something to do with a given keyword.</p> <p>Even more extensive than the <code>lookfor</code> function are the find features in the Current Directory browser. For example, you can look for all</p>

# lookfor

---

occurrences of a specified word in all the M-files in the current directory. For more information, see “Finding Files and Directories”.

## See Also

dir, doc, filebrowser, findstr, help, helpdesk, helpwin, regexp, what, which, who

Related topics in the MATLAB Desktop Tools and Development Environment documentation:

- “Searching Documentation and Demos with the Help Browser”
- “Finding Functions Using the Function Browser”
- “Help and Related Resources”

**Purpose** Convert string to lowercase

**Syntax**  
`t = lower('str')`  
`B = lower(A)`

**Description** `t = lower('str')` returns the string formed by converting any uppercase characters in `str` to the corresponding lowercase characters and leaving all other characters unchanged.

`B = lower(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `lower` to each string within `A`.

**Examples** `lower('MathWorks')` is `mathworks`.

**Remarks** Character sets supported:

- PC: Latin-1 for the Microsoft Windows operating system
- Other: ISO Latin-1 (ISO 8859-1)

**See Also** `upper`

# ls

---

**Purpose** Directory contents

**Syntax** `ls`

**Graphical Interface** As an alternative to the `ls` function, you can use the Current Directory browser to view directory contents.

**Description** `ls` lists the contents of the current directory. On UNIX<sup>17</sup> platforms, `ls` returns a character row vector of file names separated by tab and space characters. On Microsoft Windows platforms, `ls` returns an  $m$ -by- $n$  character array of file names, where  $m$  is the number of file names and  $n$  is the number of characters in the longest file name found. File names shorter than  $n$  characters are padded with space characters.

On UNIX platforms, you can pass any flags to `ls` that your operating system supports.

**See Also** `dir`, `pwd`

“Managing Files and Working with the Current Directory”

17. UNIX is a registered trademark of The Open Group in the United States and other countries.

**Purpose**

Least-squares solution in presence of known covariance

**Syntax**

```
x = lscov(A,b)
x = lscov(A,b,w)
x = lscov(A,b,V)
x = lscov(A,b,V,alg)
[x,stdx] = lscov(...)
[x,stdx,mse] = lscov(...)
[x,stdx,mse,S] = lscov(...)
```

**Description**

`x = lscov(A,b)` returns the ordinary least squares solution to the linear system of equations  $A*x = b$ , i.e.,  $x$  is the  $n$ -by- $1$  vector that minimizes the sum of squared errors  $(b - A*x)'*(b - A*x)$ , where  $A$  is  $m$ -by- $n$ , and  $b$  is  $m$ -by- $1$ .  $b$  can also be an  $m$ -by- $k$  matrix, and `lscov` returns one solution for each column of  $b$ . When  $\text{rank}(A) < n$ , `lscov` sets the maximum possible number of elements of  $x$  to zero to obtain a "basic solution".

`x = lscov(A,b,w)`, where  $w$  is a vector length  $m$  of real positive weights, returns the weighted least squares solution to the linear system  $A*x = b$ , that is,  $x$  minimizes  $(b - A*x)'*diag(w)*(b - A*x)$ .  $w$  typically contains either counts or inverse variances.

`x = lscov(A,b,V)`, where  $V$  is an  $m$ -by- $m$  real symmetric positive definite matrix, returns the generalized least squares solution to the linear system  $A*x = b$  with covariance matrix proportional to  $V$ , that is,  $x$  minimizes  $(b - A*x)'*inv(V)*(b - A*x)$ .

More generally,  $V$  can be positive semidefinite, and `lscov` returns  $x$  that minimizes  $e'*e$ , subject to  $A*x + T*e = b$ , where the minimization is over  $x$  and  $e$ , and  $T*T' = V$ . When  $V$  is semidefinite, this problem has a solution only if  $b$  is consistent with  $A$  and  $V$  (that is,  $b$  is in the column space of  $[A \ T]$ ), otherwise `lscov` returns an error.

By default, `lscov` computes the Cholesky decomposition of  $V$  and, in effect, inverts that factor to transform the problem into ordinary least squares. However, if `lscov` determines that  $V$  is semidefinite, it uses an orthogonal decomposition algorithm that avoids inverting  $V$ .

`x = lscov(A,b,V,alg)` specifies the algorithm used to compute `x` when `V` is a matrix. `alg` can have the following values:

- 'chol' uses the Cholesky decomposition of `V`.
- 'orth' uses orthogonal decompositions, and is more appropriate when `V` is ill-conditioned or singular, but is computationally more expensive.

`[x,stdx] = lscov(...)` returns the estimated standard errors of `x`. When `A` is rank deficient, `stdx` contains zeros in the elements corresponding to the necessarily zero elements of `x`.

`[x,stdx,mse] = lscov(...)` returns the mean squared error.

`[x,stdx,mse,S] = lscov(...)` returns the estimated covariance matrix of `x`. When `A` is rank deficient, `S` contains zeros in the rows and columns corresponding to the necessarily zero elements of `x`. `lscov` cannot return `S` if it is called with multiple right-hand sides, that is, if `size(B,2) > 1`.

The standard formulas for these quantities, when `A` and `V` are full rank, are

- $x = \text{inv}(A' \cdot \text{inv}(V) \cdot A) \cdot A' \cdot \text{inv}(V) \cdot B$
- $\text{mse} = B' \cdot (\text{inv}(V) - \text{inv}(V) \cdot A \cdot \text{inv}(A' \cdot \text{inv}(V) \cdot A) \cdot A' \cdot \text{inv}(V)) \cdot B ./ (m-n)$
- $S = \text{inv}(A' \cdot \text{inv}(V) \cdot A) \cdot \text{mse}$
- $\text{stdx} = \text{sqrt}(\text{diag}(S))$

However, `lscov` uses methods that are faster and more stable, and are applicable to rank deficient cases.

`lscov` assumes that the covariance matrix of `B` is known only up to a scale factor. `mse` is an estimate of that unknown scale factor, and `lscov` scales the outputs `S` and `stdx` appropriately. However, if `V` is known to be exactly the covariance matrix of `B`, then that scaling is unnecessary.



To get the appropriate estimates in this case, you should rescale  $S$  and  $stdx$  by  $1/mse$  and  $\sqrt{1/mse}$ , respectively.

## Algorithm

The vector  $x$  minimizes the quantity  $(A*x-b)'*inv(V)*(A*x-b)$ . The classical linear algebra solution to this problem is

$$x = inv(A'*inv(V)*A)*A'*inv(V)*b$$

but the `lscov` function instead computes the QR decomposition of  $A$  and then modifies  $Q$  by  $V$ .

## Examples

### Example 1 – Computing Ordinary Least Squares

The MATLAB backslash operator (`\`) enables you to perform linear regression by computing ordinary least-squares (OLS) estimates of the regression coefficients. You can also use `lscov` to compute the same OLS estimates. By using `lscov`, you can also compute estimates of the standard errors for those coefficients, and an estimate of the standard deviation of the regression error term:

```
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
X = [ones(size(x1)) x1 x2];
y = [.17 .26 .28 .23 .27 .34]';
```

```
a = X\y
a =
    0.1203
    0.3284
   -0.1312
```

```
[b,se_b,mse] = lscov(X,y)
b =
    0.1203
    0.3284
   -0.1312
se_b =
    0.0643
```

```
0.2267
0.1488
mse =
0.0015
```

## Example 2 – Computing Weighted Least Squares

Use `lscov` to compute a weighted least-squares (WLS) fit by providing a vector of relative observation weights. For example, you might want to downweight the influence of an unreliable observation on the fit:

```
w = [1 1 1 1 1 .1]';

[bw,sew_b,msew] = lscov(X,y,w)
bw =
0.1046
0.4614
-0.2621
sew_b =
0.0309
0.1152
0.0814
msew =
3.4741e-004
```

## Example 3 – Computing General Least Squares

Use `lscov` to compute a general least-squares (GLS) fit by providing an observation covariance matrix. For example, your data may not be independent:

```
V = .2*ones(length(x1)) + .8*diag(ones(size(x1)));

[bg,sew_b,mseg] = lscov(X,y,V)
bg =
0.1203
0.3284
-0.1312
sew_b =
```

```

0.0672
0.2267
0.1488
mse =
0.0019

```

#### Example 4 – Estimating the Coefficient Covariance Matrix

Compute an estimate of the coefficient covariance matrix for either OLS, WLS, or GLS fits. The coefficient standard errors are equal to the square roots of the values on the diagonal of this covariance matrix:

```
[b, se_b, mse, S] = lscov(X, y);
```

```

S
S =
    0.0041   -0.0130    0.0075
   -0.0130    0.0514   -0.0328
    0.0075   -0.0328    0.0221

```

```

[se_b sqrt(diag(S))]
ans =
    0.0643    0.0643
    0.2267    0.2267
    0.1488    0.1488

```

#### See Also

lsqnonneg, qr

The arithmetic operator \

#### Reference

[1] Strang, G., *Introduction to Applied Mathematics*, Wellesley-Cambridge, 1986, p. 398.

# lsqnonneg

---

**Purpose** Solve nonnegative least-squares constraints problem

**Syntax**

```
x = lsqnonneg(C,d)
x = lsqnonneg(C,d,x0)
x = lsqnonneg(C,d,x0,options)
[x,resnorm] = lsqnonneg(...)
[x,resnorm,residual] = lsqnonneg(...)
[x,resnorm,residual,exitflag] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)
```

**Description** `x = lsqnonneg(C,d)` returns the vector `x` that minimizes  $\text{norm}(C*x-d)$  subject to  $x \geq 0$ . `C` and `d` must be real.

`x = lsqnonneg(C,d,x0)` uses `x0` as the starting point if all `x0`  $\geq 0$ ; otherwise, the default is used. The default start point is the origin (the default is used when `x0` is `[]` or when only two input arguments are provided).

`x = lsqnonneg(C,d,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `lsqnonneg` uses these `options` structure fields:

<code>Display</code>	Level of display. 'off' displays no output; 'final' displays just the final output; 'notify' (default) displays output only if the function does not converge.
<code>TolX</code>	Termination tolerance on <code>x</code> .
<code>OutputFcn</code>	User-defined function that is called at each iteration. See "Output Function" in the Optimization Toolbox for more information.
<code>PlotFcns</code>	User-defined plot function that is called at each iteration. See "Plot Functions" in the Optimization Toolbox for more information.

`[x,resnorm] = lsqnonneg(...)` returns the value of the squared 2-norm of the residual:  $\text{norm}(C*x-d)^2$ .

`[x,resnorm,residual] = lsqnonneg(...)` returns the residual,  $d-C*x$ .

`[x,resnorm,residual,exitflag] = lsqnonneg(...)` returns a value `exitflag` that describes the exit condition of `lsqnonneg`:

- >0            Indicates that the function converged to a solution  $x$ .
- 0             Indicates that the iteration count was exceeded. Increasing the tolerance (`TolX` parameter in options) may lead to a solution.

`[x,resnorm,residual,exitflag,output] = lsqnonneg(...)` returns a structure `output` that contains information about the operation:

- `output.algorithm`            The algorithm used
- `output.iterations`         The number of iterations taken

`[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)` returns the dual vector (Lagrange multipliers) `lambda`, where `lambda(i) <= 0` when `x(i)` is (approximately) 0, and `lambda(i)` is (approximately) 0 when `x(i) > 0`.

## Examples

Compare the unconstrained least squares solution to the `lsqnonneg` solution for a 4-by-2 problem:

```
C = [  
    0.0372    0.2869  
    0.6861    0.7071  
    0.6233    0.6245  
    0.6344    0.6170];  
d = [  
    0.8587  
    0.1781  
    0.0747
```

# lsqnonneg

---

```
0.8405];  
[C\d lsqnonneg(C,d)] =  
-2.5627    0  
3.1108    0.6929  
[norm(C*(C\d)-d) norm(C*lsqnonneg(C,d)-d)] =  
0.6674 0.9118
```

The solution from `lsqnonneg` does not fit as well (has a larger residual), as the least squares solution. However, the nonnegative least squares solution has no negative components.

## Algorithm

`lsqnonneg` uses the algorithm described in [1]. The algorithm starts with a set of possible basis vectors and computes the associated dual vector `lambda`. It then selects the basis vector corresponding to the maximum value in `lambda` in order to swap out of the basis in exchange for another possible candidate. This continues until `lambda <= 0`.

## See Also

The arithmetic operator `\`, `optimset`

## References

[1] Lawson, C.L. and R.J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, 1974, Chapter 23, p. 161.

**Purpose**

LSQR method

**Syntax**

```

x = lsqr(A,b)
lsqr(A,b,tol)
lsqr(A,b,tol,maxit)
lsqr(A,b,tol,maxit,M)
lsqr(A,b,tol,maxit,M1,M2)
lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,x0)

```

**Description**

`x = lsqr(A,b)` attempts to solve the system of linear equations  $A^*x=b$  for  $x$  if  $A$  is consistent, otherwise it attempts to solve the least squares solution  $x$  that minimizes  $\text{norm}(b-A^*x)$ . The  $m$ -by- $n$  coefficient matrix  $A$  need not be square but it should be large and sparse. The column vector  $b$  must have length  $m$ .  $A$  can be a function handle `afun` such that `afun(x, 'notransp')` returns  $A^*x$  and `afun(x, 'transp')` returns  $A'^*x$ . See “Function Handles” in the MATLAB Programming documentation for more information.

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

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

`lsqr(A,b,tol,maxit)` specifies the maximum number of iterations.

# lsqr

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

`lsqr(A,b,tol,maxit,M1,M2,x0)` specifies the  $n$ -by-1 initial guess. If `x0` is `[]`, then `lsqr` uses the default, an all zero vector.

`[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a convergence flag.

Flag	Convergence
0	<code>lsqr</code> converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.
1	<code>lsqr</code> iterated <code>maxit</code> times but did not converge.
2	Preconditioner $M$ was ill-conditioned.
3	<code>lsqr</code> stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during <code>lsqr</code> became too small or too large to continue computing.

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 you specify the `flag` output.

`[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns an estimate of the relative residual norm  $(b-A*x)/norm(b)$ . If `flag` is 0, `relres`  $\leq$  `tol`.

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

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



`[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of estimates of the scaled normal equations residual at each iteration:  $\text{norm}((A \cdot \text{inv}(M))' * (B - A * X)) / \text{norm}(A \cdot \text{inv}(M), 'fro')$ . Note that the estimate of  $\text{norm}(A \cdot \text{inv}(M), 'fro')$  changes, and hopefully improves, at each iteration.

## Examples

### Example 1

```
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 = lsqr(A,b,tol,maxit,M1,M2);
```

displays the following message:

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-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_lsqr` that

- Calls `lsqr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_lsqr` are available to `afun`.

The following shows the code for `run_lsqr`:

```
function x1 = run_lsqr
```

# lsqr

---

```
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 = lsqr(@afun,b,tol,maxit,M1,M2);

function y = afun(x,transp_flag)
    if strcmp(transp_flag,'transp') % y = A'*x
        y = 4 * x;
        y(1:n-1) = y(1:n-1) - 2 * x(2:n);
        y(2:n) = y(2:n) - x(1:n-1);
    elseif strcmp(transp_flag,'notransp') % y = A*x
        y = 4 * x;
        y(2:n) = y(2:n) - 2 * x(1:n-1);
        y(1:n-1) = y(1:n-1) - x(2:n);
    end
end
end
end
```

When you enter

```
x1=run_lsqr;
```

MATLAB software displays the message

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

## See Also

bicg, bicgstab, cgs, gmres, minres, norm, pcg, qmr, symmlq,  
function\_handle (@)

## References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "LSQR: An Algorithm for Sparse Linear Equations And Sparse Least Squares," *ACM Trans. Math. Soft.*, Vol.8, 1982, pp. 43-71.

**Purpose** Test for less than

**Syntax** `A < B`  
`lt(A, B)`

**Description** `A < B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (true) where `A` is less than `B`, or set to logical 0 (false) where `A` is greater than or equal to `B`. 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.

`lt(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 less than the corresponding elements of `B`:

```
A = magic(6);  
B = repmat(3*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     1     0     1
```

---

0	1	0	0	1	1
1	0	0	0	1	0

**See Also**

gt, le, ge, ne, eq, “Relational Operators” in the MATLAB Programming documentation

**Purpose** LU matrix factorization

**Syntax**

```
Y = lu(A)
[L,U] = lu(A)
[L,U,P] = lu(A)
[L,U,P,Q] = lu(A)
[L,U,P,Q,R] = lu(A)
[...] = lu(A,'vector')
[...] = lu(A,thresh)
[...] = lu(A,thresh,'vector')
```

**Description** The `lu` function expresses a matrix  $A$  as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The factorization is often called the  $LU$ , or sometimes the  $LR$ , factorization.  $A$  can be rectangular. For a full matrix  $A$ , `lu` uses the Linear Algebra Package (LAPACK) routines described in “Algorithm” on page 2-2166.

`Y = lu(A)` returns matrix  $Y$  that, for sparse  $A$ , contains the strictly lower triangular  $L$ , i.e., without its unit diagonal, and the upper triangular  $U$  as submatrices. That is, if `[L,U,P] = lu(A)`, then  $Y = U + L - \text{eye}(\text{size}(A))$ . For nonsparse  $A$ ,  $Y$  is the output from the LAPACK `dgetrf` or `zgetrf` routine. The permutation matrix  $P$  is not returned.

`[L,U] = lu(A)` returns an upper triangular matrix in  $U$  and a permuted lower triangular matrix in  $L$  such that  $A = L * U$ . Return value  $L$  is a product of lower triangular and permutation matrices.

`[L,U,P] = lu(A)` returns an upper triangular matrix in  $U$ , a lower triangular matrix  $L$  with a unit diagonal, and a permutation matrix  $P$ , such that  $L * U = P * A$ . The statement `lu(A,'matrix')` returns identical output values.

`[L,U,P,Q] = lu(A)` for sparse nonempty  $A$ , returns a unit lower triangular matrix  $L$ , an upper triangular matrix  $U$ , a row permutation matrix  $P$ , and a column reordering matrix  $Q$ , so that  $P * A * Q = L * U$ . This syntax uses UMFPACK and is significantly more time and memory efficient than the other syntaxes, even when used with `colamd`. If  $A$

is empty or not sparse, `lu` displays an error message. The statement `lu(A, 'matrix')` returns identical output values.

`[L,U,P,Q,R] = lu(A)` returns unit lower triangular matrix `L`, upper triangular matrix `U`, permutation matrices `P` and `Q`, and a diagonal scaling matrix `R` so that  $P*(R\backslash A)*Q = L*U$  for sparse non-empty `A`. This uses `UMFPACK` as well. Typically, but not always, the row-scaling leads to a sparser and more stable factorization. Note that this factorization is the same as that used by `sparse mldivide` when `UMFPACK` is used. The statement `lu(A, 'matrix')` returns identical output values.

`[...]` = `lu(A, 'vector')` returns the permutation information in two row vectors `p` and `q`. You can specify from 1 to 5 outputs. Output `p` is defined as  $A(p,:) = L*U$ , output `q` is defined as  $A(p,q) = L*U$ , and output `R` is defined as  $R(:,p)\backslash A(:,q) = L*U$ .

`[...]` = `lu(A,thresh)` controls pivoting in `UMFPACK`. This syntax applies to sparse matrices only. The `thresh` input is a one- or two-element vector of type `single` or `double` that defaults to `[0.1, 0.001]`. If `A` is a square matrix with a mostly symmetric structure and mostly nonzero diagonal, `UMFPACK` uses a symmetric pivoting strategy. For this strategy, the diagonal where

$$A(i,j) \geq \text{thresh}(2) * \max(\text{abs}(A(j:m,j)))$$

is selected. If the diagonal entry fails this test, a pivot entry below the diagonal is selected, using `thresh(1)`. In this case, `L` has entries with absolute value  $1/\min(\text{thresh})$  or less.

If `A` is not as described above, `UMFPACK` uses an asymmetric strategy. In this case, the sparsest row `i` where

$$A(i,j) \geq \text{thresh}(1) * \max(\text{abs}(A(j:m,j)))$$

is selected. A value of 1.0 results in conventional partial pivoting. Entries in `L` have an absolute value of  $1/\text{thresh}(1)$  or less. The second element of the `thresh` input vector is not used when `UMFPACK` uses an asymmetric strategy.

Smaller values of `thresh(1)` and `thresh(2)` tend to lead to sparser LU factors, but the solution can become inaccurate. Larger values can lead to a more accurate solution (but not always), and usually an increase in the total work and memory usage. The statement `lu(A,thresh,'matrix')` returns identical output values.

`[...]` = `lu(A,thresh,'vector')` controls the pivoting strategy and also returns the permutation information in row vectors, as described above. The `thresh` input must precede `'vector'` in the input argument list.

---

**Note** In rare instances, incorrect factorization results in  $P*A*Q \neq L*U$ . Increase `thresh`, to a maximum of 1.0 (regular partial pivoting), and try again.

---

## Remarks

Most of the algorithms for computing LU factorization are variants of Gaussian elimination. The factorization is a key step in obtaining the inverse with `inv` and the determinant with `det`. It is also the basis for the linear equation solution or matrix division obtained with `\` and `/`.

## Arguments

A	Rectangular matrix to be factored.
thresh	Pivot threshold for sparse matrices. Valid values are in the interval [0, 1]. If you specify the fourth output Q, the default is 0.1. Otherwise, the default is 1.0.
L	Factor of A. Depending on the form of the function, L is either a unit lower triangular matrix, or else the product of a unit lower triangular matrix with P'.
U	Upper triangular matrix that is a factor of A.
P	Row permutation matrix satisfying the equation $L*U = P*A$ , or $L*U = P*A*Q$ . Used for numerical stability.



- Q Column permutation matrix satisfying the equation  $P*A*Q = L*U$ . Used to reduce fill-in in the sparse case.
- R Row-scaling matrix

## Examples

### Example 1

Start with

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

To see the LU factorization, call `lu` with two output arguments.

$$[L1,U] = \text{lu}(A)$$

$$L1 = \begin{bmatrix} 0.1429 & 1.0000 & 0 \\ 0.5714 & 0.5000 & 1.0000 \\ 1.0000 & 0 & 0 \end{bmatrix}$$

$$U = \begin{bmatrix} 7.0000 & 8.0000 & 0 \\ 0 & 0.8571 & 3.0000 \\ 0 & 0 & 4.5000 \end{bmatrix}$$

Notice that `L1` is a permutation of a lower triangular matrix: if you switch rows 2 and 3, and then switch rows 1 and 2, the resulting matrix is lower triangular and has 1s on the diagonal. Notice also that `U` is upper triangular. To check that the factorization does its job, compute the product

$$L1*U$$

which returns the original `A`. The inverse of the example matrix,  $X = \text{inv}(A)$ , is actually computed from the inverses of the triangular factors

$$X = \text{inv}(U) * \text{inv}(L1)$$

Using three arguments on the left side to get the permutation matrix as well,

$$[L2, U, P] = \text{lu}(A)$$

returns a truly lower triangular L2, the same value of U, and the permutation matrix P.

L2 =

$$\begin{bmatrix} 1.0000 & 0 & 0 \\ 0.1429 & 1.0000 & 0 \\ 0.5714 & 0.5000 & 1.0000 \end{bmatrix}$$

U =

$$\begin{bmatrix} 7.0000 & 8.0000 & 0 \\ 0 & 0.8571 & 3.0000 \\ 0 & 0 & 4.5000 \end{bmatrix}$$

P =

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Note that  $L2 = P * L1$ .

$P * L1$

ans =

$$\begin{bmatrix} 1.0000 & 0 & 0 \\ 0.1429 & 1.0000 & 0 \\ 0.5714 & 0.5000 & 1.0000 \end{bmatrix}$$

To verify that  $L2 * U$  is a permuted version of A, compute  $L2 * U$  and subtract it from  $P * A$ :

$$P*A - L2*U$$

$$\text{ans} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

In this case,  $\text{inv}(U) * \text{inv}(L)$  results in the permutation of  $\text{inv}(A)$  given by  $\text{inv}(P) * \text{inv}(A)$ .

The determinant of the example matrix is

$$d = \det(A)$$

$$d = 27$$

It is computed from the determinants of the triangular factors

$$d = \det(L) * \det(U)$$

The solution to  $Ax = b$  is obtained with matrix division

$$x = A \setminus b$$

The solution is actually computed by solving two triangular systems

$$y = L \setminus b$$

$$x = U \setminus y$$

## Example 2

The 1-norm of their difference is within roundoff error, indicating that  $L*U = P*B*Q$ .

Generate a 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster-Fuller geodesic dome.

$$B = \text{bucky};$$

Use the sparse matrix syntax with four outputs to get the row and column permutation matrices.

```
[L,U,P,Q] = lu(B);
```

Apply the permutation matrices to B, and subtract the product of the lower and upper triangular matrices.

```
Z = P*B*Q - L*U;
norm(Z,1)
```

```
ans =
    7.9936e-015
```

### Example 3

This example illustrates the benefits of using the 'vector' option. Note how much memory is saved by using the `lu(F, 'vector')` syntax.

```
rand('state',0);
F = rand(1000,1000);
g = sum(F,2);
[L,U,P] = lu(F);
[L,U,p] = lu(F,'vector');
whos P p
```

Name	Size	Bytes	Class	Attributes
P	1000x1000	8000000	double	
p	1x1000	8000	double	

The following two statements are equivalent. The first typically requires less time:

```
x = U \ (L \ (g(p, :)));
y = U \ (L \ (P*g));
```

## Algorithm

For full matrices X, `lu` uses the LAPACK routines listed in the following table.

---

	<b>Real</b>	<b>Complex</b>
X double	DGETRF	ZGETRF
X single	SGETRF	CGETRF

For sparse X, with four outputs, lu uses UMFPACK routines. With three or fewer outputs, lu uses its own sparse matrix routines.

### See Also

cond, det, inv, luinc, qr, rref

The arithmetic operators \ and /

### References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* ([http://www.netlib.org/lapack/lug/lapack\\_lug.html](http://www.netlib.org/lapack/lug/lapack_lug.html)), Third Edition, SIAM, Philadelphia, 1999.

[2] Davis, T. A., *UMFPACK Version 4.6 User Guide* (<http://www.cise.ufl.edu/research/sparse/umfpack>), Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, 2002.

# luinc

---

**Purpose** Sparse incomplete LU factorization

**Syntax**

```
luinc(A, '0')  
luinc(A, droptol)  
luinc(A, options)  
[L,U] = luinc(A,0)  
[L,U] = luinc(A,options)  
[L,U,P] = luinc(...)
```

**Description** `luinc` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`luinc(A, '0')` computes the incomplete LU factorization of level 0 of a square sparse matrix. The triangular factors have the same sparsity pattern as the permutation of the original sparse matrix `A`, and their product agrees with the permuted `A` over its sparsity pattern. `luinc(A, '0')` returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost, but `nnz(luinc(A, '0')) = nnz(A)`, with the possible exception of some zeros due to cancellation.

`luinc(A, droptol)` computes the incomplete LU factorization of any sparse matrix using the drop tolerance specified by the non-negative scalar `droptol`. The result is an approximation of the complete LU factors returned by `lu(A)`. For increasingly smaller values of the drop tolerance, this approximation improves until the drop tolerance is 0, at which time the complete LU factorization is produced, as in `lu(A)`.

As each column `j` of the triangular incomplete factors is being computed, the entries smaller in magnitude than the local drop tolerance (the product of the drop tolerance and the norm of the corresponding column of `A`)

$$\text{droptol} * \text{norm}(A(:, j))$$

are dropped from the appropriate factor.

The only exceptions to this dropping rule are the diagonal entries of the upper triangular factor, which are preserved to avoid a singular factor.

`luinc(A,options)` computes the factorization with up to four options. These options are specified by fields of the input structure `options`. The fields must be named exactly as shown in the table below. You can include any number of these fields in the structure and define them in any order. Any additional fields are ignored.

Field Name	Description
<code>droptol</code>	Drop tolerance of the incomplete factorization.
<code>milu</code>	If <code>milu</code> is 1, <code>luinc</code> produces the modified incomplete LU factorization that subtracts the dropped elements in any column from the diagonal element of the upper triangular factor. The default value is 0.
<code>udiag</code>	If <code>udiag</code> is 1, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is 0.
<code>thresh</code>	Pivot threshold between 0 (forces diagonal pivoting) and 1, the default, which always chooses the maximum magnitude entry in the column to be the pivot. <code>thresh</code> is described in greater detail in the <code>lu</code> reference page.

`luinc(A,options)` is the same as `luinc(A,droptol)` if `options` has `droptol` as its only field.

`[L,U] = luinc(A,0)` returns the product of permutation matrices and a unit lower triangular matrix in `L` and an upper triangular matrix in `U`. The exact sparsity patterns of `L`, `U`, and `A` are not comparable but the number of nonzeros is maintained with the possible exception of some zeros in `L` and `U` due to cancellation:

$$\text{nnz}(L) + \text{nnz}(U) = \text{nnz}(A) + n, \text{ where } A \text{ is } n\text{-by-}n.$$

The product `L*U` agrees with `A` over its sparsity pattern. `(L*U) .* spones(A) - A` has entries of the order of `eps`.

`[L,U] = luinc(A,options)` returns a permutation of a unit lower triangular matrix in `L` and an upper triangular matrix in `U`. The product `L*U` is an approximation to `A`. `luinc(A,options)` returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost.

`[L,U,P] = luinc(...)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`.

`[L,U,P] = luinc(A,'0')` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U` and a permutation matrix in `P`. `L` has the same sparsity pattern as the lower triangle of permuted `A`

$$\text{spones}(L) = \text{spones}(\text{tril}(P*A))$$

with the possible exceptions of 1s on the diagonal of `L` where `P*A` may be zero, and zeros in `L` due to cancellation where `P*A` may be nonzero. `U` has the same sparsity pattern as the upper triangle of `P*A`

$$\text{spones}(U) = \text{spones}(\text{triu}(P*A))$$

with the possible exceptions of zeros in `U` due to cancellation where `P*A` may be nonzero. The product `L*U` agrees within rounding error with the permuted matrix `P*A` over its sparsity pattern. `(L*U).*spones(P*A)-P*A` has entries of the order of `eps`.

`[L,U,P] = luinc(A,options)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`. The nonzero entries of `U` satisfy

$$\text{abs}(U(i,j)) \geq \text{droptol} * \text{norm}(A(:,j)),$$

with the possible exception of the diagonal entries, which were retained despite not satisfying the criterion. The entries of `L` were tested against the local drop tolerance before being scaled by the pivot, so for nonzeros in `L`

$$\text{abs}(L(i,j)) \geq \text{droptol} * \text{norm}(A(:,j))/U(j,j).$$

The product `L*U` is an approximation to the permuted `P*A`.



## Remarks

These incomplete factorizations may be useful as preconditioners for solving large sparse systems of linear equations. The lower triangular factors all have 1s along the main diagonal but 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 `udiag` option to replace a zero diagonal only gets rid of the symptoms of the problem but does not solve it. The preconditioner may not be singular, but it probably is not useful and a warning message is printed.

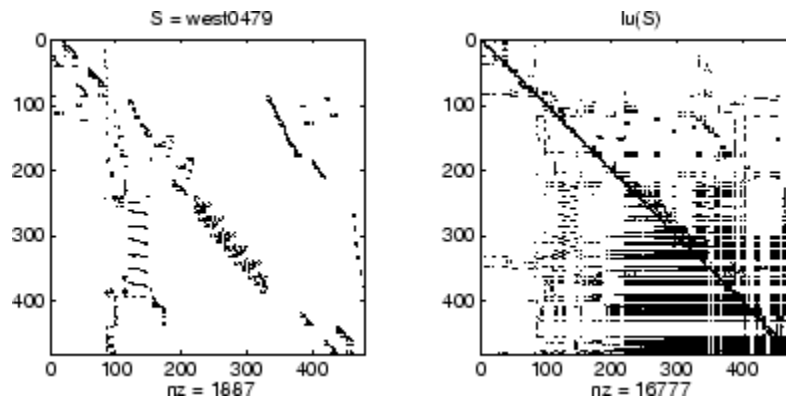
## Limitations

`luinc(X, '0')` works on square matrices only.

## Examples

Start with a sparse matrix and compute its LU factorization.

```
load west0479;
S = west0479;
[L,U] = lu(S);
```

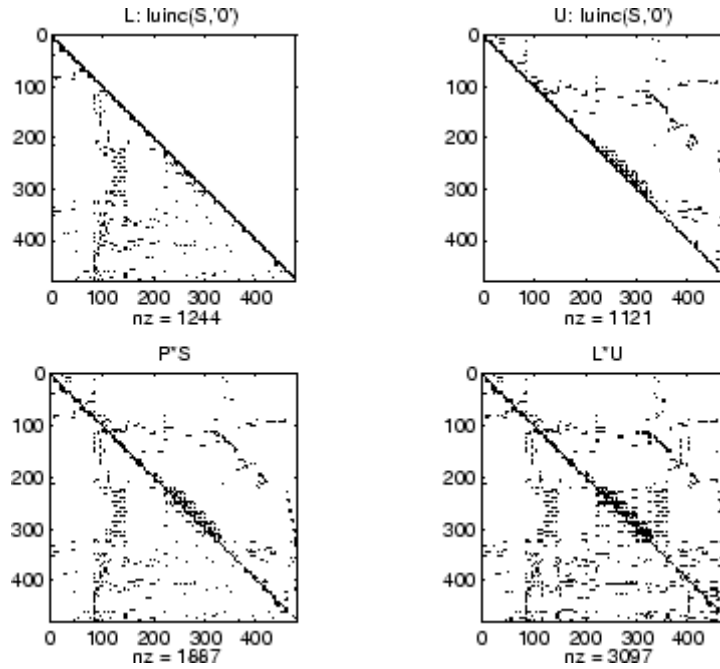


Compute the incomplete LU factorization of level 0.

```
[L,U,P] = luinc(S, '0');
D = (L*U).*spones(P*S)-P*S;
```

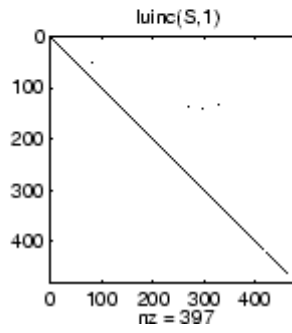
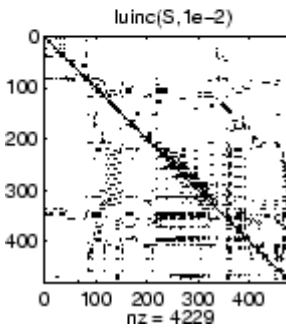
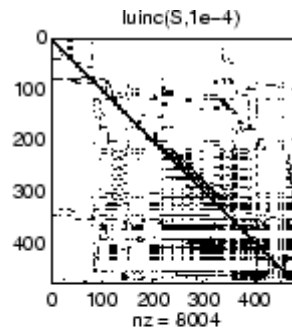
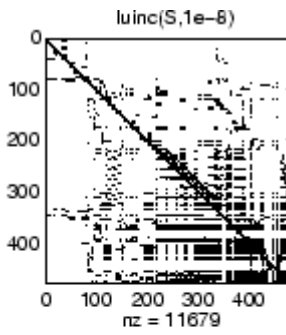
`spones(U)` and `spones(triu(P*S))` are identical.

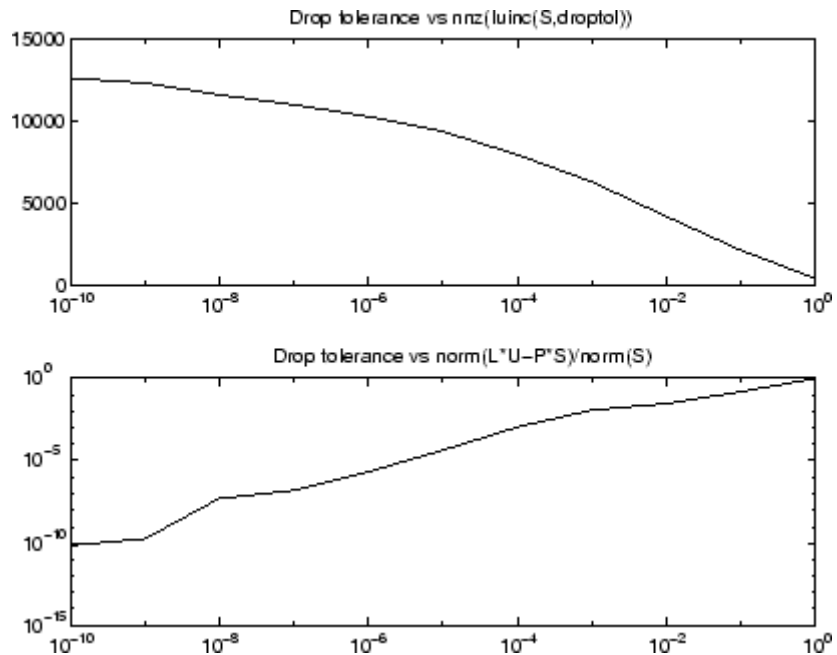
`spones(L)` and `spones(tril(P*S))` disagree at 73 places on the diagonal, where `L` is 1 and `P*S` is 0, and also at position (206,113), where `L` is 0 due to cancellation, and `P*S` is -1. `D` has entries of the order of `eps`.



```
[ILO,IU0,IP0] = luinc(S,0);
[IL1,IU1,IP1] = luinc(S,1e-10);
.
.
.
```

A drop tolerance of 0 produces the complete LU factorization. Increasing the drop tolerance increases the sparsity of the factors (decreases the number of nonzeros) but also increases the error in the factors, as seen in the plot of drop tolerance versus  $\text{norm}(L*U - P*S, 1) / \text{norm}(S, 1)$  in the second figure below.





## Algorithm

`luinc(A, '0')` is based on the “KJJ” variant of the LU factorization with partial pivoting. Updates are made only to positions which are nonzero in A.

`luinc(A,droptol)` and `luinc(A,options)` are based on the column-oriented lu for sparse matrices.

## See Also

`bicg`, `cholinc`, `ilu`, `lu`

## References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

**Purpose** Magic square

**Syntax** `M = magic(n)`

**Description** `M = magic(n)` returns an  $n$ -by- $n$  matrix constructed from the integers 1 through  $n^2$  with equal row and column sums. The order  $n$  must be a scalar greater than or equal to 3.

**Remarks** A magic square, scaled by its magic sum, is doubly stochastic.

**Examples** The magic square of order 3 is

```
M = magic(3)
```

```
M =
```

```
8   1   6
3   5   7
4   9   2
```

This is called a magic square because the sum of the elements in each column is the same.

```
sum(M) =
```

```
15   15   15
```

And the sum of the elements in each row, obtained by transposing twice, is the same.

```
sum(M')' =
```

```
15
15
15
```

This is also a special magic square because the diagonal elements have the same sum.

```
sum(diag(M)) =
```

```
15
```

The value of the characteristic sum for a magic square of order  $n$  is

```
sum(1:n^2)/n
```

which, when  $n = 3$ , is 15.

## Algorithm

There are three different algorithms:

- $n$  odd
- $n$  even but not divisible by four
- $n$  divisible by four

To make this apparent, type

```
for n = 3:20
    A = magic(n);
    r(n) = rank(A);
end
```

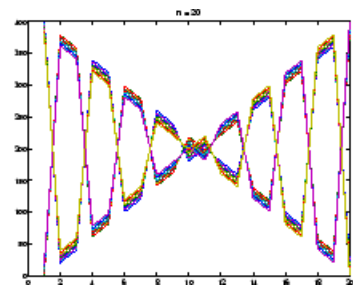
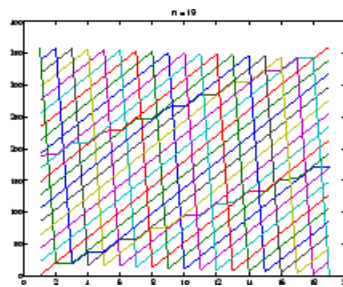
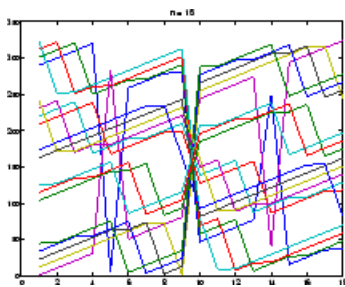
For  $n$  odd, the rank of the magic square is  $n$ . For  $n$  divisible by 4, the rank is 3. For  $n$  even but not divisible by 4, the rank is  $n/2 + 2$ .

```
[(3:20)', r(3:20)']
```

```
ans =
     3     3
     4     3
     5     5
     6     5
     7     7
     8     3
     9     9
    10     7
    11    11
```

12	3
13	13
14	9
15	15
16	3
17	17
18	11
19	19
20	3

Plotting A for  $n = 18, 19, 20$  shows the characteristic plot for each category.



**Limitations**

If you supply  $n$  less than 3, magic returns either a nonmagic square, or else the degenerate magic squares 1 and [].

**See Also**

ones, rand

# makehgtform

---

**Purpose** Create 4-by-4 transform matrix

**Syntax**

```
M = makehgtform
M = makehgtform('translate',[tx ty tz])
M = makehgtform('scale',s)
M = makehgtform('scale',[sx,sy,sz])
M = makehgtform('xrotate',t)
M = makehgtform('yrotate',t)
M = makehgtform('zrotate',t)
M = makehgtform('axisrotate',[ax,ay,az],t)
```

**Description** Use `makehgtform` to create transform matrices for translation, scaling, and rotation of graphics objects. Apply the transform to graphics objects by assigning the transform to the `Matrix` property of a parent `hgtransform` object. See [Examples](#) for more information.

`M = makehgtform` returns an identity transform.

`M = makehgtform('translate',[tx ty tz])` or `M = makehgtform('translate',tx,ty,tz)` returns a transform that translates along the *x*-axis by `tx`, along the *y*-axis by `ty`, and along the *z*-axis by `tz`.

`M = makehgtform('scale',s)` returns a transform that scales uniformly along the *x*-, *y*-, and *z*-axes.

`M = makehgtform('scale',[sx,sy,sz])` returns a transform that scales along the *x*-axis by `sx`, along the *y*-axis by `sy`, and along the *z*-axis by `sz`.

`M = makehgtform('xrotate',t)` returns a transform that rotates around the *x*-axis by `t` radians.

`M = makehgtform('yrotate',t)` returns a transform that rotates around the *y*-axis by `t` radians.

`M = makehgtform('zrotate',t)` returns a transform that rotates around the *z*-axis by `t` radians.

`M = makehgtform('axisrotate',[ax,ay,az],t)` Rotate around axis [`ax ay az`] by `t` radians.



Note that you can specify multiple operations in one call to `makehgtform` and the MATLAB software returns a transform matrix that is the result of concatenating all specified operations. For example,

```
m = makehgtform('xrotate',pi/2,'yrotate',pi/2);
```

is the same as

```
m = makehgtform('xrotate',pi/2)*makehgtform('yrotate',pi/2);
```

## See Also

`hggroup`, `hgtransform`

Tomas Moller and Eric Haines, *Real-Time Rendering*, A K Peters, Ltd., 1999 for more information about transforms.

“Group Objects” in MATLAB Graphics documentation for more information and examples.

`Hgtransform` Properties for property descriptions.

# containers.Map

---

**Purpose** Construct containers.Map object

**Syntax**

```
M = containers.Map  
M = containers.Map(keys, values)  
M = containers.Map(keys, values, 'uniformvalues', tf)
```

**Description** The Map object is a data structure that is a container for other data. A Map container is similar to an array except for the means by which you index into the data stored inside the map. Instead of being restricted to the use of integer array indices 1 through N, you select elements of a Map container using indices of various data types, (strings, for example). A map consists of **keys** (the indices) and **data values** and is a handle object.

`M = containers.Map` constructs a Map container object `M` to hold data values that you can easily reference using keys that you establish. `M` is a handle object with properties `Count`, `KeyType`, and `ValueType`. These properties represent the number of keys in the map, the type of these keys, and the type of the values assigned to those keys respectively.

When you call the `containers.Map` constructor with no input arguments, MATLAB constructs an empty Map object, setting the `Count`, `KeyType`, and `'ValueType'` properties to 0 to char, and `'any'`, respectively.

`M = containers.Map(keys, values)` constructs a Map object `M` that contains one or more keys and a value for each of these keys, as specified in the `keys` and `values` arguments. A `value` is some unit of data that you want stored in the Map object, and a `key` is a unique reference to that data. Valid data types for the `keys` argument are any real-valued scalars or character arrays. This includes `char`, `double`, `int32`, `uint32`, `int64`, or `uint64`. Valid values are the same, plus the string `'any'`.

To specify multiple keys, make the `keys` argument a 1-by-`n` cell array, where `n` is the number of keys to be stored in the map. Elements of this cell array should belong to the same type. You can specify values of mixed numeric, or numeric and logical, types, without generating an error. If you do this, the Map constructor converts all elements to the type of the leftmost element in the `keys` cell array. To specify multiple

values, use a 1-by-n cell array, where n is equal to numel (keys). There must be exactly one value for each key argument.

Object M has properties Count, KeyType, and ValueType. Count is a string that contains the number of key-value pairs in the Map object once the map has been constructed. KeyType is a character array containing the data type of the keys in the map. All keys belong to the same data type. ValueType is a character array containing the data type of the values in the map. If these values are of different data types, ValueType is set to the string 'any'.

`M = containers.Map(keys, values, 'uniformvalues', tf)` constructs Map M in which all values are required to be of the same type when uniformvalues is set to logical 1 (true). If they are not, MATLAB throws an error. If you want to be able to store values of mixed types, set uniformvalues to logical 0 (false). This flag is only needed if you want to override the default.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

## Properties

Property	Description
Count	Unsigned 64-bit integer that represents the total number of key-value pairs contained in the Map object when the initial object is constructed.
KeyType	Character array that indicates the data type of all keys contained in the Map object.
ValueType	Character array that indicates the data type of all values contained in the Map object. If not all values have the same data type, then ValueType is 'any'.

## Methods

Method	Description
isKey	Check if containers.Map object contains key.

# containers.Map

---

Method	Description
keys	Return all keys of containers.Map object.
size	Return size of containers.Map object.
length	Return length of containers.Map object.
values	Return all values of containers.Map object in cell array.
remove	Remove key-value pairs from containers.Map.

## Examples

### Example 1 – Constructing a New Map Object

Construct a one-member Map object, US\_Capitals:

```
US_Capitals = containers.Map('Arizona', 'Phoenix');
```

To call methods of the class, just use the method name followed by the name of the Map object in parentheses. Call the keys and values methods of the US\_Capitals object you just constructed:

```
keys(US_Capitals), values(US_Capitals)
ans =
    'Arizona'
ans =
    'Phoenix'
```

### Example 2 – Finding Map Properties

List the properties of the US\_Capitals object:

```
properties(US_Capitals)
```

```
Properties for class containers.Map:
    Count
    KeyType
    ValueType
```

Examine each property:

```
US_Capitals.Count,  
ans =  
    1
```

```
US_Capitals.KeyType  
ans =  
    char
```

```
US_Capitals.ValueType  
ans =  
    char
```

### Example 3 – Storing Multiple Keys and Values

Construct a new Map object and store 6 key and value parameters in it. Specify multiple keys and values by listing them in a cell array as shown here:

```
US_Capitals = containers.Map( ...  
{ 'Arizona', 'Nebraska', 'Nevada', ...      % 6 States  
'New York', 'Georgia', 'Alaska'}, ...  
{ 'Phoenix', 'Lincoln', 'Carson City', ...  % 6 Capitals  
'Albany', 'Atlanta', 'Juneau'})
```

```
US_Capitals =  
containers.Map handle  
package: containers  
properties:  
    Count: 6  
    KeyType: 'char'  
    ValueType: 'char'  
lists of methods, events, superclasses
```

## Example 4 – Displaying the Map Contents

Use the `keys` and `values` methods to see the mapping order defined within the map. The `keys` method lists all keys of the character array type in alphabetical order. The `values` method lists the value associated with each of these keys. These are listed in an order determined by their associated keys:

```
keys(US_Capitals), values(US_Capitals)
ans =
    'Alaska'  'Arizona'  'Georgia'  'Nebraska'  'Nevada'

    'New York'
ans =
    'Juneau'  'Phoenix'  'Atlanta'  'Lincoln'  'Carson City'
    'Albany'
```

When using the `values` method, you can either list all values in the map, as shown above, or list only those values that belong to those keys you specify in the command:

```
values(US_Capitals, {'Arizona', 'New York', 'Nebraska'})
ans =
    'Phoenix'  'Albany'  'Lincoln'
```

## Example 5 – Adding More Keys and Values Later

Once the object has been created, store two additional keys (Vermont and Oregon), and their related values (Montpelier and Salem) in it. You do not need to call the constructor this time as you are adding to an existing Map object:

```
US_Capitals('Vermont') = 'Montpelier';
US_Capitals('Oregon') = 'Salem';
keys(US_Capitals)
ans =
    'Alaska'  'Arizona'  'Georgia'  'Nebraska'
    'Nevada'  'New York'  'Oregon'  'Vermont'
```

Note that the Count property has gone from 6 to 8:

```
US_Capitals.Count
ans =
     8
```

If you want to add more than one key-value pair at a time, you can concatenate your existing map with a new map that contains the new keys and values that you want to add. Only horizontal concatenation is allowed. This example adds four more state/capital pairs to the US\_Capitals map in just one concatenation operation. The US\_Capitals map now contains twelve key-value pairs:

```
newSta = {'New Jersey', 'Ohio', 'Delaware', 'Montana'};
newCap = {'Trenton', 'Columbus', 'Dover', 'Helena'};
newMap = containers.Map(newSta, newCap);

% Construct a new map. Concatenate it to the existing one.
US_Capitals = [US_Capitals newMap];
```

## Example 6— Looking Up Values with the Map

Use the map to find the capital cities of two states in the US:

```
S1 = 'Alaska';    S2 = 'Arizona'
sprintf('\nThe capitals of %s and %s are %s and %s.', ...
        S1, S2, US_Capitals(S1), US_Capitals(S2))

ans =
The capitals of Alaska and Arizona are Juneau and Phoenix.
```

## Example 7— Removing Keys and Values

To remove a key-value pair, use the remove method of the Map class, as shown here:

```
keys(US_Capitals)
ans =
    'Alaska'    'Arizona'    'Georgia'    'Nebraska'
```

## containers.Map

---

```
        'Nevada'      'New York'      'Oregon'      'Vermont'
remove(US_Capitals, {'Nebraska', 'Nevada', 'New York'});

keys(US_Capitals)
ans =
    'Alaska'      'Arizona'      'Georgia'      'Oregon'
    'Vermont'
```

Removing keys and their values from the Map object also decrements the setting of the Count property.

### See Also

keys(Map), values(Map), size(Map), length(Map), isKey(Map),  
remove(Map), handle



**Purpose** Divide matrix into cell array of matrices

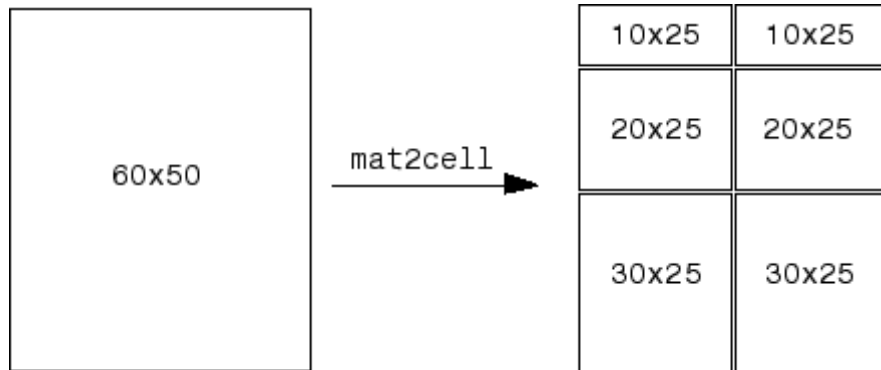
**Syntax**

```
c = mat2cell(x, m, n)
c = mat2cell(x, d1, d2, ..., dn)
c = mat2cell(x, r)
```

**Description** `c = mat2cell(x, m, n)` divides the two-dimensional matrix `x` into adjacent submatrices, each contained in a cell of the returned cell array `c`. Vectors `m` and `n` specify the number of rows and columns, respectively, to be assigned to the submatrices in `c`.

The example shown below divides a 60-by-50 matrix into six smaller matrices. The MATLAB software returns the new matrices in a 3-by-2 cell array:

```
mat2cell(x, [10 20 30], [25 25])
```



The sum of the element values in `m` must equal the total number of rows in `x`. And the sum of the element values in `n` must equal the number of columns in `x`.

The elements of `m` and `n` determine the size of each cell in `c` by satisfying the following formula for `i = 1:length(m)` and `j = 1:length(n)`:

```
size(c{i,j}) == [m(i) n(j)]
```

# mat2cell

---

`c = mat2cell(x, d1, d2, ..., dn)` divides the multidimensional array `x` and returns a multidimensional cell array of adjacent submatrices of `x`. Each of the vector arguments `d1` through `dn` should sum to the respective dimension sizes of `x` such that, for `p = 1:n`,

$$\text{size}(x,p) == \text{sum}(dp)$$

The elements of `d1` through `dn` determine the size of each cell in `c` by satisfying the following formula for `ip = 1:length(dp)`:

$$\text{size}(c\{i1,i2,\dots,in\}) == [d1(i1) \ d2(i2) \ \dots \ dn(in)]$$

If `x` is an empty array, `mat2cell` returns an empty cell array. This requires that all `dn` inputs that correspond to the zero dimensions of `x` be equal to `[]`.

For example,

```
a = rand(3,0,4);  
c = mat2cell(a, [1 2], [], [2 1 1]);
```

`c = mat2cell(x, r)` divides an array `x` by returning a single-column cell array containing full rows of `x`. The sum of the element values in vector `r` must equal the number of rows of `x`.

The elements of `r` determine the size of each cell in `c`, subject to the following formula for `i = 1:length(r)`:

$$\text{size}(c\{i\},1) == r(i)$$

## Remarks

`mat2cell` supports all array types.

## Examples

Divide matrix `X` into 2-by-3 and 2-by-2 matrices contained in a cell array:

```
X = [1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15; 16 17 18 19 20]  
X =  
     1     2     3     4     5  
     6     7     8     9    10  
    11    12    13    14    15
```

```
        16    17    18    19    20
C = mat2cell(X, [2 2], [3 2])
C =
    [2x3 double]    [2x2 double]
    [2x3 double]    [2x2 double]

C{1,1}
ans =
     1     2     3
     6     7     8

C{1,2}
ans =
     4     5
     9    10

C{2,1}
ans =
    11    12    13
    16    17    18

C{2,2}
ans =
    14    15
    19    20
```

**See Also**

cell2mat, num2cell

# mat2str

---

**Purpose** Convert matrix to string

**Syntax**

```
str = mat2str(A)
str = mat2str(A,n)
str = mat2str(A, 'class')
str = mat2str(A, n, 'class')
```

**Description**

`str = mat2str(A)` converts matrix `A` into a string. This string is suitable for input to the `eval` function such that `eval(str)` produces the original matrix to within 15 digits of precision.

`str = mat2str(A,n)` converts matrix `A` using `n` digits of precision.

`str = mat2str(A, 'class')` creates a string with the name of the class of `A` included. This option ensures that the result of evaluating `str` will also contain the class information.

`str = mat2str(A, n, 'class')` uses `n` digits of precision and includes the class information.

**Limitations** The `mat2str` function is intended to operate on scalar, vector, or rectangular array inputs only. An error will result if `A` is a multidimensional array.

## Examples

### Example 1

Consider the matrix

```
x = [3.85 2.91; 7.74 8.99]
x =
    3.8500    2.9100
    7.7400    8.9900
```

The statement

```
A = mat2str(x)
```

produces

```
A =  
    [3.85 2.91;7.74 8.99]
```

where A is a string of 21 characters, including the square brackets, spaces, and a semicolon.

`eval(mat2str(x))` reproduces x.

## Example 2

Create a 1-by-6 matrix of signed 16-bit integers, and then use `mat2str` to convert the matrix to a 1-by-33 character array, A. Note that output string A includes the class name, `int16`:

```
x1 = int16([-300 407 213 418 32 -125]);  
  
A = mat2str(x1, 'class')  
A =  
    int16([-300 407 213 418 32 -125])  
  
class(A)  
ans =  
    char
```

Evaluating the string A gives you an output x2 that is the same as the original `int16` matrix:

```
x2 = eval(A);  
  
if isnumeric(x2) && isa(x2, 'int16') && all(x2 == x1)  
    disp 'Conversion back to int16 worked'  
end  
  
Conversion back to int16 worked
```

## See Also

`num2str`, `int2str`, `str2num`, `sprintf`, `fprintf`

# material

---

**Purpose** Control reflectance properties of surfaces and patches

**Syntax**

```
material shiny
material dull
material metal
material([ka kd ks])
material([ka kd ks n])
material([ka kd ks n sc])
material default
```

**Description** `material` sets the lighting characteristics of surface and patch objects.

`material shiny` sets the reflectance properties so that the object has a high specular reflectance relative to the diffuse and ambient light, and the color of the specular light depends only on the color of the light source.

`material dull` sets the reflectance properties so that the object reflects more diffuse light and has no specular highlights, but the color of the reflected light depends only on the light source.

`material metal` sets the reflectance properties so that the object has a very high specular reflectance, very low ambient and diffuse reflectance, and the color of the reflected light depends on both the color of the light source and the color of the object.

`material([ka kd ks])` sets the ambient/diffuse/specular strength of the objects.

`material([ka kd ks n])` sets the ambient/diffuse/specular strength and specular exponent of the objects.

`material([ka kd ks n sc])` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects.

`material default` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects to their defaults.

## Remarks

The `material` command sets the `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, and `SpecularColorReflectance` properties of all surface and patch objects in the axes. There must be visible `light` objects in the axes for lighting to be enabled. Look at the `material.m` M-file to see the actual values set (enter the command `type material`).

## See Also

`light`, `lighting`, `patch`, `surface`

Lighting as a Visualization Tool for more information on lighting

“Lighting” on page 1-105 for related functions

# matlabcolon (matlab:)

---

**Purpose** Run specified function via hyperlink

**Syntax** `disp('<a href="matlab: stmtnt_1; stmtnt_n;">hyperlink_text</a>')`

**Description** `matlab:` executes `stmtnt_1` through `stmtnt_n` when you click (or press **Ctrl+Enter**) in `hyperlink_text`. This must be used with another function, such as `disp`, where `disp` creates and displays underlined and colored `hyperlink_text` in the Command Window. Use `disp`, `error`, `fprintf`, `help`, or `warning` functions to display the hyperlink. The `hyperlink_text` is interpreted as HTML—you might need to use HTML character entity references or ASCII values for some special characters. Include the full hypertext string, from '`<a href= to </a>`' within a single line, that is, do not continue a long string on a new line. No spaces are allowed after the opening `<` and before the closing `>`. A single space is required between `a` and `href`.

**Remarks** The `matlab:` function behaves differently with `diary`, `notebook`, `type`, and similar functions than might be expected. For example, if you enter the following statement

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

the `diary` file, when viewed in a text editor, shows

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
<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 does display it as a hyperlink.

## Examples **Single Function**

The statement

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

displays



[Generate magic square](#)

in the Command Window. When you click the link Generate magic square, the MATLAB software runs `magic(4)`.

## Multiple Functions

You can include multiple functions in the statement, such as

```
disp('<a href="matlab: x=0:1:8;y=sin(x);plot(x,y)">Plot  
x,y</a>')
```

which displays

[Plot x,y](#)

in the Command Window. When you click the link, MATLAB runs

```
x = 0:1:8;  
y = sin(x);  
plot(x,y)
```

## Clicking the Hyperlink Again

After running the statements in the hyperlink `Plot x,y` defined in the previous example, “Multiple Functions” on page 2-2195, you can subsequently redefine `x` in the base workspace, for example, as

```
x = -2*pi:pi/16:2*pi;
```

If you then click the hyperlink, `Plot x,y`, it changes the current value of `x` back to

```
0:1:8
```

because the `matlab:` statement defines `x` in the base workspace. In the `matlab:` statement that displayed the hyperlink, `Plot x,y`, `x` was defined as `0:1:8`.

# matlabcolon (matlab:)

---

## Presenting Options

Use multiple `matlab:` statements in an M-file to present options, such as

```
disp('<a href = "matlab:state = 0">Disable feature</a>')
disp('<a href = "matlab:state = 1">Enable feature</a>')
```

The Command Window displays

[Disable feature](#)  
[Enable feature](#)

and depending on which link is clicked, sets `state` to 0 or 1.

## Special Characters

MATLAB correctly interprets most strings that includes special characters, such as a greater than sign (>). For example, the following statement includes a >

```
disp('<a href="matlab:str = ''Value > 0''">Positive</a>')
```

and generates the following hyperlink.

[Positive](#)

Some symbols might not be interpreted correctly and you might need to use the HTML character entity reference for the symbol. For example, an alternative way to run the same statement is to use the `&gt;` character entity reference instead of the `>` symbol:

```
disp('<a href="matlab:str = ''Value &gt; 0''">Positive</a>')
```

Instead of the HTML character entity reference, you can use the ASCII value for the symbol. For example, the greater than sign, `>`, is ASCII 62. The above example becomes

```
disp('<a href="matlab:str=[''Value '' char(62) '' 0'']">Positive</a>')
```

Here are some values for common special characters.

Character	HTML Character Entity Reference	ASCII Value
>	&gt;	62
<	&lt;	60
&	&amp;	38
"	&quot;	34

For a list of all HTML character entity references, see <http://www.w3.org/>.

## Links from M-File Help

For functions you create, you can include `matlab:` links within the M-file help, but you do not need to include a `disp` or similar statement because the `help` function already includes it for displaying hyperlinks. Use the links to display additional help in a browser when the user clicks them. The M-file `soundspeed` contains the following statements:

```
function c=soundspeed(s,t,p)

% Speed of sound in water, using
% <a href="matlab: web('http://www.zu.edu')">Wilson's formula</a>
% Where c is the speed of sound in water in m/s
```

etc.

Run `help soundspeed` and MATLAB displays the following in the Command Window.

```
>> help soundspeed
Speed of sound in water, using
Wilson's formula
Where c is the speed of sound in water in m/s
```

When you click the link `Wilson's formula`, MATLAB displays the HTML page `http://www.zu.edu` in the Web browser. Note that this URL is only an example and is intentionally invalid.

## matlabcolon (matlab:)

---

### **See Also**

disp, error, fprintf, input, run, warning

<b>Purpose</b>	Startup M-file for MATLAB program
<b>Description</b>	<p>At startup time, MATLAB automatically executes the master M-file <code>matlabrc.m</code> and, if it exists, <code>startup.m</code>. On multiuser or networked systems, <code>matlabrc.m</code> is reserved for use by the system manager. The file <code>matlabrc.m</code> invokes the file <code>startup.m</code> if it exists on the search path MATLAB uses.</p> <p>As an individual user, you can create a startup file in your own MATLAB directory. Use the startup file to define physical constants, engineering conversion factors, graphics defaults, or anything else you want predefined in your workspace.</p>
<b>Algorithm</b>	<p>Only <code>matlabrc</code> is actually invoked by MATLAB at startup. However, <code>matlabrc.m</code> contains the statements</p> <pre>if exist('startup') == 2     startup end</pre> <p>that invoke <code>startup.m</code>. Extend this process to create additional startup M-files, if required.</p>
<b>Remarks</b>	You can also start MATLAB using options you define at the Command Window prompt or in your Microsoft Windows shortcut for MATLAB.
<b>Examples</b>	<p><b>Turning Off the Figure Window Toolbar</b></p> <p>If you do not want the toolbar to appear in the figure window, remove the comment marks from the following line in the <code>matlabrc.m</code> file, or create a similar line in your own <code>startup.m</code> file.</p> <pre>% set(0,'defaultfiguretoolbar','none')</pre>
<b>See Also</b>	<p><code>matlabroot</code>, <code>quit</code>, <code>restoredefaultpath</code>, <code>startup</code></p> <p>Startup Options in the MATLAB Desktop Tools and Development Environment documentation</p>

# matlabroot

---

**Purpose** Root directory

**Syntax** matlabroot  
rd = matlabroot

**Description** matlabroot returns the name of the directory where the MATLAB software is installed. Use matlabroot to create a path to MATLAB and toolbox directories that does not depend on a specific platform, MATLAB version, or installation directory.

rd = matlabroot returns the name of the directory in which the MATLAB software is installed and assigns it to rd.

**Remarks** **matlabroot as Directory Name**

The term *matlabroot* also represents the directory where MATLAB files are installed and should not be confused with the matlabroot function. For example, “save to *matlabroot/toolbox/local*” means save to the *toolbox/local* directory in the MATLAB root directory.

## Using \$matlabroot as a Literal

In some files, such as `info.xml` and `classpath.txt`, `$matlabroot` is literal. In those files, MATLAB interprets `$matlabroot` as the full path to the MATLAB root directory. For example, including the line

```
$matlabroot/toolbox/local/myfile.jar
```

in `classpath.txt`, adds `myfile.jar`, which is located in the `toolbox/local` directory, to `classpath.txt`.

Sometimes, particularly in older code examples, the term `$matlabroot` or `$MATLABROOT` is not meant to be interpreted literally but is used to represent the value returned by the `matlabroot` function.

## matlabroot on Macintosh Platforms

In R2008b (V7,7), running `matlabroot` returns

```
/Applications/MATLAB_R2008b.app
```

In prior versions, such as R2008a (V7.6), running `matlabroot` returns, for example

```
/Applications/MATLAB_R2008a
```

When you use GUIs on Macintosh platforms, you cannot directly view the contents of the MATLAB root directory. For more information, see “Using File Browser GUIs on Macintosh Platforms to Navigate Within the MATLAB Root Directory”.

### Use in Compiled Code

To return the path to the executable in compiled mode, use the MATLAB® Compiler™ `ctfroot` function, or the MATLAB `toolboxdir` function. For details, see the MATLAB or MATLAB Compiler documentation.

## Examples

Run

```
matlabroot
```

MATLAB returns, for example,

```
C:\Program Files\MATLAB\R2008b
```

To produce a full path to the `toolbox/matlab/general` directory that is correct for the platform on which it is executed, run

```
fullfile(matlabroot, 'toolbox', 'matlab', 'general')
```

To change the current directory to the MATLAB root directory, run

```
cd(matlabroot)
```

To add the directory `myfiles` to the MATLAB search path, run

```
addpath([matlabroot ' /toolbox/local/myfiles'])
```

# matlabroot

---

## **See Also**

`ctfroot` (in MATLAB Compiler product), `fullfile`, `partialpath`,  
`path`, `toolboxdir`

“Managing Files and Working with the Current Directory”



**Purpose** Start MATLAB program (UNIX platforms)

**Syntax**

```
matlab helpOption
matlab archOption
matlab dispOption
matlab modeOption
matlab -c licensefile
matlab -r matlab_command
matlab -logfile filename
matlab -mwvisual visualid
matlab -nosplash
matlab -timing
matlab -debug
matlab -Ddebugger options
matlab mgrOption
```

---

**Note** You can enter more than one of these options in the same `matlab` command. If you use `-Ddebugger` to start MATLAB in debug mode, the first option in the command must be `-Ddebugger`.

---

# matlab (UNIX)

---

## Description

`matlab` is a Bourne shell script that starts the MATLAB executable on UNIX<sup>18</sup> platforms. (In this document, `matlab` refers to this script; MATLAB refers to the application program). Before actually initiating the execution of MATLAB, this script configures the run-time environment by

- Determining the MATLAB root directory
- Determining the host machine architecture
- Processing any command line options
- Reading the MATLAB startup file, `.matlab7rc.sh`
- Setting MATLAB environment variables

There are two ways in which you can control the way the `matlab` script works:

- By specifying command line options
- By assigning values in the MATLAB startup file, `.matlab7rc.sh`

## Specifying Options at the Command Line

Options that you can enter at the command line are as follows:

`matlab helpOption` displays information that matches the specified `helpOption` argument without starting MATLAB. `helpOption` can be any one of the keywords shown in the table below. Enter only one `helpOption` keyword in a `matlab` command.

### Values for helpOption

Option	Description
<code>-help</code>	Display <code>matlab</code> command usage.
<code>-h</code>	The same as <code>-help</code> .

18. UNIX is a registered trademark of The Open Group in the United States and other countries.

## Values for helpOption (Continued)

Option	Description
-n	Display all the final values of the environment variables and arguments passed to the MATLAB executable as well as other diagnostic information.
-e	Display <i>all</i> environment variables and their values just prior to exiting. This argument must have been parsed before exiting for anything to be displayed. The last possible exiting point is just before the MATLAB image would have been executed and a status of 0 is returned. If the exit status is not 0 on return, then the variables and values may not be correct.

`matlab archOption` starts MATLAB and assumes that you are running on the system architecture specified by `arch`, or using the MATLAB version specified by `variant`, or both. The values for the `archOption` argument are shown in the table below. Enter only one of these options in a `matlab` command.

## Values for archOption

Option	Description
-arch	Run MATLAB assuming this architecture rather than the actual architecture of the machine you are using. Replace the term <code>arch</code> with a string representing a recognized system architecture.

# matlab (UNIX)

---

## Values for archOption (Continued)

Option	Description
v=variant	Execute the version of MATLAB found in the directory bin/\$ARCH/variant instead of bin/\$ARCH. Replace the term variant with a string representing a MATLAB version.
v=arch/variant	Execute the version of MATLAB found in the directory bin/arch/variant instead of bin/\$ARCH. Replace the terms arch and variant with strings representing a specific architecture and MATLAB version.

matlab dispOption starts MATLAB using one of the display options shown in the table below. Enter only one of these options in a matlab command.

## Values for dispOption

Option	Description
-display xDisp	Send X commands to X Window Server display xDisp. This supersedes the value of the DISPLAY environment variable.
-nodisplay	Start the Sun Microsystems JVM software, but do not start the MATLAB desktop. Do not display any X commands, and ignore the DISPLAY environment variable,

matlab modeOption starts MATLAB without its usual desktop component. Enter only one of the options shown below.

**Values for modeOption**

<b>Option</b>	<b>Description</b>
<b>-desktop</b>	Allow the MATLAB desktop to be started by a process without a controlling terminal. This is usually a required command line argument when attempting to start MATLAB from a window manager menu or desktop icon.
<b>-nodesktop</b>	Start MATLAB without bringing up the MATLAB desktop. The JVM software <i>is</i> started. Use the current window in the operating system to enter commands. Use this option to run without an X-window, for example, in VT100 mode, or in batch processing mode. Note that if you pipe to MATLAB using the > constructor, the nodesktop option is used automatically. With nodesktop, MATLAB does not save statements to the Command History. With nodesktop, you can still use most development environment tools by starting them via a function. For example, use preferences to open the Preferences dialog box and helpbrowser to open the Help browser. Do not use nodesktop to provide a Command Window-only interface; instead, select <b>Desktop &gt; Desktop Layout &gt; Command Window Only</b> .
<b>-nojvm</b>	Start MATLAB without the JVM software. Use the current window to enter commands. The MATLAB desktop does not open. Any tools that require Java software, such as the desktop tools, cannot be used. Handle Graphics and related functionality are not supported; MATLAB produces a warning when you use them.

# matlab (UNIX)

---

`matlab -c licensefile` starts MATLAB using the specified license file. The `licensefile` argument can have the form `port@host` or it can be a colon-separated list of license filenames. This option causes the `LM_LICENSE_FILE` and `MLM_LICENSE_FILE` environment variables to be ignored.

`matlab -r matlab_command` starts MATLAB and executes the specified MATLAB command.

`matlab -logfile filename` starts MATLAB and makes a copy of any output to the command window in file `log`. This includes all crash reports.

`matlab -mwvisual visualid` starts MATLAB and uses `visualid` as the default X visual for figure windows. `visualid` is a hexadecimal number that can be found using `xdpyinfo`.

`matlab -nosplash` starts MATLAB but does not display the splash screen during startup.

`matlab -timing` starts MATLAB and prints a summary of startup time to the command window. This information is also recorded in a timing log, the name of which is printed to the shell window in which MATLAB is started. This option should be used only when working with a Technical Support Representative from The MathWorks, Inc.

`matlab -debug` starts MATLAB and displays debugging information that can be useful, especially for X based problems. This option should be used only when working with a Technical Support Representative from The MathWorks, Inc.

`matlab -Ddebugger options` starts MATLAB in debug mode, using the named debugger (e.g., `dbx`, `gdb`, `xdb`, `cvd`). A full path can be specified for debugger.

The `options` argument can include *only* those options that follow the debugger name in the syntax of the actual debug command. For most debuggers, there is a very limited number of such options. Options that would normally be passed to the MATLAB executable should be used as parameters of a command inside the debugger (like `run`). They should not be used when running the `matlab` script.

If any other `matlab` command options are placed before the `-Ddebugger` argument, they will be handled as if they were part of the options after the `-Ddebugger` argument and will be treated as illegal options by most debuggers. The `MATLAB_DEBUG` environment variable is set to the filename part of the debugger argument.

To customize your debugging session, use a startup file. See your debugger documentation for details.

---

**Note** For certain debuggers like `gdb`, the `SHELL` environment variable is *always* set to `/bin/sh`.

---

**Note** `matlab mgrOption` This syntax is deprecated and support will be removed in a future release. The `mgrOption` option starts MATLAB in the memory management mode specified by one of the following options.

## Values for mgrOption

Option	Description
<code>-memmgr manager</code>	Set environment variable <code>MATLAB_MEM_MGR</code> to <code>manager</code> . The <code>manager</code> argument can have one of the following values: <ul style="list-style-type: none"><li>• <b>cache</b></li><li>• <b>compact</b> — This is useful for large models or MATLAB code that uses many structure or object variables. It is not helpful for large arrays. (This option applies only to 32-bit architectures.)</li><li>• <b>debug</b> — Does memory integrity checking and is useful for debugging memory problems caused by user-created MEX files.</li></ul>
<code>-check_malloc</code>	The same as using <code>'-memmgr debug'</code> .

## Specifying Options in the MATLAB Startup File

The `.matlab7rc.sh` shell script contains definitions for a number of variables that the `matlab` script uses. These variables are defined within the `matlab` script, but can be redefined in `.matlab7rc.sh`. When invoked, `matlab` looks for the first occurrence of `.matlab7rc.sh` in the current directory, in the home directory (`$HOME`), and in the `matlabroot/bin` directory, where the template version of `.matlab7rc.sh` is located.



You can edit the template file to redefine information used by the `matlab` script. If you do not want your changes applied systemwide, copy the edited version of the script to your current or home directory. Ensure that you edit the section that applies to your machine architecture.

The following table lists the variables defined in the `.matlab7rc.sh` file. See the comments in the `.matlab7rc.sh` file for more information about these variables.

<b>Variable</b>	<b>Definition and Standard Assignment Behavior</b>
ARCH	<p>The machine architecture.</p> <p>The value ARCH passed with the <code>-arch</code> or <code>-arch/ext</code> argument to the script is tried first, then the value of the environment variable <code>MATLAB_ARCH</code> is tried next, and finally it is computed. The first one that gives a valid architecture is used.</p>
AUTOMOUNT_MAP	<p>Path prefix map for automounting.</p> <p>The value set in <code>.matlab7rc.sh</code> (initially by the installer) is used unless the value differs from that determined by the script, in which case the value in the environment is used.</p>
DISPLAY	<p>The hostname of the X Window display MATLAB uses for output.</p> <p>The value of <code>Xdisplay</code> passed with the <code>-display</code> argument to the script is used; otherwise, the value in the environment is used. <code>DISPLAY</code> is ignored by MATLAB if the <code>-nodisplay</code> argument is passed.</p>

# matlab (UNIX)

---

Variable	Definition and Standard Assignment Behavior
LD_LIBRARY_PATH	<p>Final Load library path. The name LD_LIBRARY_PATH is platform dependent.</p> <p>The final value is normally a colon-separated list of four sublists, each of which could be empty. The first sublist is defined in <code>.matlab7rc.sh</code> as <code>LDPATH_PREFIX</code>. The second sublist is computed in the script and includes directories inside the MATLAB root directory and relevant Sun Microsystems Java directories. The third sublist contains any nonempty value of LD_LIBRARY_PATH from the environment possibly augmented in <code>.matlab7rc.sh</code>. The final sublist is defined in <code>.matlab7rc.sh</code> as <code>LDPATH_SUFFIX</code>.</p>
LM_LICENSE_FILE	<p>The FLEX lm license variable.</p> <p>The license file value passed with the <code>-c</code> argument to the script is used; otherwise it is the value set in <code>.matlab7rc.sh</code>. In general, the final value is a colon-separated list of license files and/or <code>port@host</code> entries. The shipping <code>.matlab7rc.sh</code> file starts out the value by prepending <code>LM_LICENSE_FILE</code> in the environment to a default <code>license.file</code>.</p> <p>Later in the <code>matlab</code> script, if the <code>-c</code> option is not used, the <code>matlabroot/etc</code> directory is searched for the files that start with <code>license.dat.DEMO</code>. These files are assumed to contain demo licenses and are added automatically to the end of the current list.</p>

<b>Variable</b>	<b>Definition and Standard Assignment Behavior</b>
MATLAB	<p>The MATLAB root directory.</p> <p>The default computed by the script is used unless <code>MATLABdefault</code> is reset in <code>.matlab7rc.sh</code>.</p> <p>Currently <code>MATLABdefault</code> is not reset in the shipping <code>.matlab7rc.sh</code>.</p>
MATLAB_DEBUG	<p>Normally set to the name of the debugger.</p> <p>The <code>-Ddebugger</code> argument passed to the script sets this variable. Otherwise, a nonempty value in the environment is used.</p>
MATLAB_JAVA	<p>The path to the root of the Java Runtime Environment.</p> <p>The default set in the script is used unless <code>MATLAB_JAVA</code> is already set. Any nonempty value from <code>.matlab7rc.sh</code> is used first, then any nonempty value from the environment. Currently there is no value set in the shipping <code>.matlab67rc.sh</code>, so that environment alone is used.</p>

# matlab (UNIX)

<b>Variable</b>	<b>Definition and Standard Assignment Behavior</b>
MATLAB_MEM_MGR	<hr/> <p><b>Note</b> Turns on MATLAB memory integrity checking. In the future, MATLAB will not support the MATLAB_MEM_MGR environment variable.</p> <p>The <code>-check_malloc</code> argument passed to the script sets this variable to 'debug'. Otherwise, a nonempty value set in <code>.matlab7rc.sh</code> is used, or a nonempty value in the environment is used. If a nonempty value is not found, the variable is not exported to the environment.</p> <hr/>
MATLABPATH	<p>The MATLAB search path.</p> <p>The final value is a colon-separated list with the MATLABPATH from the environment prepended to a list of computed defaults. You can add subdirectories of <code>userpath</code> to the MATLAB search path upon startup. See <code>userpath</code> for details.</p>

Variable	Definition and Standard Assignment Behavior
SHELL	<p>The shell to use when the “!” or unix command is issued in MATLAB. This is taken from the environment unless SHELL is reset in <code>.matlab7rc.sh</code>.</p> <p>Note that an additional environment variable called <code>MATLAB_SHELL</code> takes precedence over <code>SHELL</code>. MATLAB checks internally for <code>MATLAB_SHELL</code> first and, if empty or not defined, then checks <code>SHELL</code>. If <code>SHELL</code> is also empty or not defined, MATLAB uses <code>/bin/sh</code>. The value of <code>MATLAB_SHELL</code> should be an absolute path, i.e. <code>/bin/sh</code>, not simply <code>sh</code>.</p> <p>Currently, the shipping <code>.matlab7rc.sh</code> file does not reset <code>SHELL</code> and also does not reference or set <code>MATLAB_SHELL</code>.</p>
TOOLBOX	<p>Path of the toolbox directory.</p> <p>A nonempty value in the environment is used first. Otherwise, <code>matlabroot/toolbox</code>, computed by the script, is used unless <code>TOOLBOX</code> is reset in <code>.matlab7rc.sh</code>. Currently <code>TOOLBOX</code> is not reset in the shipping <code>.matlab7rc.sh</code>.</p>

## matlab (UNIX)

---

Variable	Definition and Standard Assignment Behavior
XAPPLRESDIR	The X application resource directory.  A nonempty value in the environment is used first unless XAPPLRESDIR is reset in <code>.matlab7rc.sh</code> . Otherwise, <i>matlabroot</i> /X11/app-defaults, computed by the script, is used.
XKEYSYMDB	The X keysym database file.  A nonempty value in the environment is used first unless XKEYSYMDB is reset in <code>.matlab7rc.sh</code> . Otherwise, <i>matlabroot</i> /X11/app-defaults/XKeysymDB, computed by the script, is used. The <code>matlab</code> script determines the path of the MATLAB root directory as one level up the directory tree from the location of the script. Information in the AUTOMOUNT_MAP variable is used to fix the path so that it is correct to force a mount. This can involve deleting part of the pathname from the front of the MATLAB root path. The MATLAB variable is then used to locate all files within the MATLAB directory tree.

The `matlab` script determines the path of the MATLAB root directory by looking up the directory tree from the *matlabroot*/bin directory (where the `matlab` script is located). The MATLAB variable is then used to locate all files within the MATLAB directory tree.

You can change the definition of MATLAB if, for example, you want to run a different version of MATLAB or if, for some reason, the path determined by the `matlab` script is not correct. (This can happen when certain types of automounting schemes are used by your system.)

AUTOMOUNT\_MAP is used to modify the MATLAB root directory path. The pathname that is assigned to AUTOMOUNT\_MAP is deleted from the

front of the MATLAB root path. (It is unlikely that you will need to use this option.)

### **See Also**

matlab (Windows), mex

“Starting the MATLAB Program on UNIX Platforms”, “Starting the MATLAB Program on Macintosh Platforms”, and “Startup Options” in the MATLAB Desktop Tools and Development Environment documentation

# matlab (Windows)

---

**Purpose** Start MATLAB program (Windows platforms)

**Syntax**

```
matlab helpOption
matlab -automation
matlab -c licensefile
matlab -logfile filename
matlab -nosplash
matlab -noFigureWindows
matlab -r "statement"
matlab -regserver
matlab -sd "startdir"
matlab shieldOption
matlab -timing
matlab -unregserver
matlab mgrOption
```

---

**Note** You can enter more than one of these options in the same `matlab` command.

---

**Description** `matlab` is a script that runs the main MATLAB executable on Microsoft Windows platforms. (In this document, the term `matlab` refers to the script, and MATLAB refers to the main executable). Before actually initiating the execution of MATLAB, it configures the run-time environment by

- Determining the MATLAB root directory
- Determining the host machine architecture
- Selectively processing command line options with the rest passed to MATLAB.
- Setting certain MATLAB environment variables

There are two ways in which you can control the way `matlab` works:



- By specifying command line options
- By setting environment variables before calling the program

## Specifying Options at the Command Line

Options that you can enter at the command line are as follows:

`matlab helpOption` displays information that matches the specified *helpOption* argument without starting MATLAB. *helpOption* can be any one of the keywords shown in the table below. Enter only one *helpOption* keyword in a `matlab` statement.

### Values for helpOption

Option	Description
<b>-help</b>	Display <code>matlab</code> command usage.
<b>-h</b>	The same as <b>-help</b> .
<b>-?</b>	The same as <b>-help</b> .

`matlab -automation` starts MATLAB as an automation server. The server window is minimized, and the MATLAB splash screen does not display on startup.

`matlab -c licensefile` starts MATLAB using the specified license file. The `licensefile` argument can have the form `port@host`. This option causes MATLAB to ignore the `LM_LICENSE_FILE` and `MLM_LICENSE_FILE` environment variables.

`matlab -logfile filename` starts MATLAB and makes a copy of any output to the Command Window in `filename`. This includes all crash reports.

`matlab -nosplash` starts MATLAB, but does not display the splash screen during startup.

`matlab -noFigureWindows` starts MATLAB, but disables the display of any figure windows in MATLAB.

## matlab (Windows)

---

`matlab -r "statement"` starts MATLAB and executes the specified MATLAB statement. Any required file must be on the MATLAB search path or in the startup directory.

`matlab -regserver` registers MATLAB as a Component Object Model (COM) server.

`matlab -sd "startdir"` specifies the startup directory for MATLAB (the current directory in MATLAB after startup). The `-sd` option has been deprecated. For information about alternatives, see “Startup Directory (Folder) on Windows Platforms”.

`matlab shieldOption` provides the specified level of protection of the address space used by MATLAB during startup on Windows 32-bit platforms. It attempts to help ensure the largest contiguous block of memory available after startup, which is useful for processing large data sets. The *shieldOption* does this by ensuring resources such as DLLs, are loaded into locations that will not fragment the address space. With *shieldOption* set to a value other than none, address space is protected up to or after the processing of `matlabrc`. Use higher levels of protection to secure larger initial blocks of contiguous memory, however a higher level might not always provide a larger size block and might cause startup problems. Therefore, start with a lower level of protection, and if successful, try the next higher level. You can use the `memory` function after startup to see the size of the largest contiguous block of memory; this helps you determine the actual effect of the *shieldOption* setting you used. If your `matlabrc` (or `startup.m`) requires significant memory, a higher level of protection for *shieldOption* might cause startup to fail; in that event try a lower level. Values for *shieldOption* can be any one of the keywords shown in the table below.

Option	Description
<p><b>-shield minimum</b></p>	<p>This is the default setting. It protects the range 0x50000000 to 0x70000000 during MATLAB startup until just before startup processes <code>matlabrc</code>. It ensures there is at least approximately 500 MB of contiguous memory up to this point.</p> <p>Start with this, the default value. To use the default, do not specify a shield option upon startup.</p> <p>If MATLAB fails to start successfully using the default option, <b>-shield minimum</b>, instead use <b>-shield none</b>.</p> <p>If MATLAB starts successfully with the default value for <i>shieldOption</i> and you want to try to ensure an even larger contiguous block after startup, try using the <b>-shield medium</b> option.</p>
<p><b>-shield medium</b></p>	<p>This protects the same range as for minimum, 0x50000000 to 0x70000000, but protects the range until just after startup processes <code>matlabrc</code>. It ensures there is at least approximately 500 MB of contiguous memory up to this point.</p> <p>If MATLAB fails to start successfully with the <b>-shield medium</b> option, instead use the default option (<b>-shield minimum</b>).</p> <p>If MATLAB starts successfully with the <b>-shield medium</b> option and you want to try to ensure an even larger contiguous block after startup, try using the <b>-shield maximum</b> option.</p>

## matlab (Windows)

---

Option	Description
<b>-shield maximum</b>	This protects the maximum possible range, which can be up to approximately 1.5 GB, until just after startup processes <code>matlabrc</code> .  If MATLAB fails to start successfully with the <b>-shield maximum</b> option, instead use the <b>-shield medium</b> option.
<b>-shield none</b>	This completely disables address shielding. Use this if MATLAB fails to start successfully with the default ( <b>-shield minimum</b> ) option.

`matlab -timing` starts MATLAB and prints a summary of startup time to the command window. MATLAB also records this information in a timing log, whose name displays in the MATLAB Command Window. Use this option only when working with a Technical Support Representative from The MathWorks.

`matlab -unregserver` removes all MATLAB COM server entries from the registry.

**Note** `matlab mgrOption` This syntax is deprecated and support will be removed in a future release. The `mgrOption` option starts MATLAB in the memory management mode specified by one of the following options.

## Values for mgrOption

Option	Description
<code>-memmgr manager</code>	<p>Set environment variable <code>MATLAB_MEM_MGR</code> to <i>manager</i>. The <i>manager</i> argument can have one of the following values:</p> <ul style="list-style-type: none"> <li>• <b>cache</b></li> <li>• <b>fast</b> — For large models or MATLAB code that uses many structure or object variables. It is not helpful for large arrays.</li> <li>• <b>debug</b> — Does memory integrity checking and is useful for debugging memory problems caused by user-created MEX files.</li> </ul>
<code>-check_malloc</code>	The same as using <code>'-memmgr debug'</code> .

## Setting Environment Variables

You can set the following environment variables before starting MATLAB.

# matlab (Windows)

---

Variable Name	Description
LM_LICENSE_FILE	This is the license variable for Macrovision® FLEXlm® license file. The license file value passed with the <b>-c</b> argument to the script is used; otherwise it is the value set in the environment. The final value is a colon-separated list of license files and/or port@host entries.
MATLAB_MEM_MGR	<hr/> <b>Note</b> This variable determines the type of memory manager used by MATLAB. In the future, MATLAB will not support the MATLAB_MEM_MGR environment variable. <hr/>

## See Also

matlab (UNIX), mex, userpath

“Starting the MATLAB Program on Windows Platforms”, and “Startup Options” in the MATLAB Desktop Tools and Development Environment documentation

---

<b>Purpose</b>	Largest elements in array
<b>Syntax</b>	$C = \max(A)$ $C = \max(A,B)$ $C = \max(A, [], \text{dim})$ $[C, I] = \max(\dots)$
<b>Description</b>	<p><math>C = \max(A)</math> returns the largest elements along different dimensions of an array.</p> <p>If <math>A</math> is a vector, <math>\max(A)</math> returns the largest element in <math>A</math>.</p> <p>If <math>A</math> is a matrix, <math>\max(A)</math> treats the columns of <math>A</math> as vectors, returning a row vector containing the maximum element from each column.</p> <p>If <math>A</math> is a multidimensional array, <math>\max(A)</math> treats the values along the first non-singleton dimension as vectors, returning the maximum value of each vector.</p> <p><math>C = \max(A,B)</math> returns an array the same size as <math>A</math> and <math>B</math> with the largest elements taken from <math>A</math> or <math>B</math>. The dimensions of <math>A</math> and <math>B</math> must match, or they may be scalar.</p> <p><math>C = \max(A, [], \text{dim})</math> returns the largest elements along the dimension of <math>A</math> specified by scalar <math>\text{dim}</math>. For example, <math>\max(A, [], 1)</math> produces the maximum values along the first dimension (the rows) of <math>A</math>.</p> <p><math>[C, I] = \max(\dots)</math> finds the indices of the maximum values of <math>A</math>, and returns them in output vector <math>I</math>. If there are several identical maximum values, the index of the first one found is returned.</p>
<b>Remarks</b>	<p>For complex input <math>A</math>, <math>\max</math> returns the complex number with the largest complex modulus (magnitude), computed with <math>\max(\text{abs}(A))</math>. Then computes the largest phase angle with <math>\max(\text{angle}(x))</math>, if necessary.</p> <p>The <math>\max</math> function ignores NaNs.</p>
<b>See Also</b>	<code>isnan</code> , <code>mean</code> , <code>median</code> , <code>min</code> , <code>sort</code>

# max (timeseries)

---

**Purpose** Maximum value of timeseries data

**Syntax** `ts_max = max(ts)`  
`ts_max = max(ts, 'PropertyName1', PropertyValue1, ...)`

**Description** `ts_max = max(ts)` returns the maximum value in the time-series data. When `ts.Data` is a vector, `ts_max` is the maximum value of `ts.Data` values. When `ts.Data` is a matrix, `ts_max` is a row vector containing the maximum value of each column of `ts.Data` (when `IsTimeFirst` is true and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `max` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_max = max(ts, 'PropertyName1', PropertyValue1, ...)` specifies the following optional input arguments:

- 'MissingData' property has two possible values, 'remove' (default) or 'interpolate', indicating how to treat missing data during the calculation.
- 'Quality' values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- 'Weighting' property has two possible values, 'none' (default) or 'time'.  
When you specify 'time', larger time values correspond to larger weights.

**Examples** The following example illustrates how to find the maximum values in multivariate time-series data.

**1** Load a 24-by-3 data array.

```
load count.dat
```

**2** Create a timeseries object with 24 time values.



```
count_ts = timeseries(count,[1:24], 'Name', 'CountPerSecond')
```

**3** Find the maximum in each data column for this timeseries object.

```
max(count_ts)
```

```
ans =
```

```
114    145    257
```

The maximum is found independently for each data column in the timeseries object.

### See Also

```
iqr (timeseries), min (timeseries), median (timeseries), mean  
(timeseries), std (timeseries), timeseries, var (timeseries)
```

# MaximizeCommandWindow

---

**Purpose** Open server window on Microsoft Windows desktop

**Syntax**

**MATLAB Client**

```
h.MaximizeCommandWindow  
MaximizeCommandWindow(h)  
invoke(h, 'MaximizeCommandWindow')
```

**Method Signature**

```
HRESULT MaximizeCommandWindow(void)
```

**Microsoft Visual Basic Client**

```
MaximizeCommandWindow
```

**Description** MaximizeCommandWindow displays the window for the server attached to handle h, and makes it the currently active window on the desktop. If the server window was not in a minimized state to begin with, then MaximizeCommandWindow does nothing.

---

**Note** MaximizeCommandWindow does not maximize the server window to its maximum possible size on the desktop. It restores the window to the size it had at the time it was minimized.

---

**Remarks** Server function names, like MaximizeCommandWindow, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

**Examples** Create a COM server and minimize its window. Then maximize the window and make it the currently active window.

**MATLAB Client**

```
h = actxserver('matlab.application');
```

```
h.MinimizeCommandWindow;  
% Now return the server window to its former state on  
% the desktop and make it the currently active window.  
h.MaximizeCommandWindow;
```

## Visual Basic .NET Client

```
Dim Matlab As Object  
  
Matlab = CreateObject("matlab.application")  
Matlab.MinimizeCommandWindow  
  
'Now return the server window to its former state on  
'the desktop and make it the currently active window.  
  
Matlab.MaximizeCommandWindow
```

## See Also

[MinimizeCommandWindow](#)

# maxNumCompThreads

---

**Purpose** Controls maximum number of computational threads

**Syntax**  
N = maxNumCompThreads  
LASTN = maxNumCompThreads(N)  
LASTN = maxNumCompThreads('automatic')

**Description** N = maxNumCompThreads returns the current maximum number of computational threads N.

LASTN = maxNumCompThreads(N) sets the maximum number of computational threads to N, and returns the previous maximum number of computational threads, LASTN.

LASTN = maxNumCompThreads('automatic') sets the maximum number of computational threads using what the MATLAB software determines to be the most desirable. It additionally returns the previous maximum number of computational threads, LASTN.

Currently, the maximum number of computational threads is equal to the number of computational cores on your machine.

---

**Note** Unlike enabling multithreading using the Preferences panel, setting the maximum number of computational threads using maxNumCompThreads will not propagate to your next MATLAB session.

---

**Purpose** Average or mean value of array

**Syntax**  
`M = mean(A)`  
`M = mean(A,dim)`

**Description** `M = mean(A)` returns the mean values of the elements along different dimensions of an array.

If `A` is a vector, `mean(A)` returns the mean value of `A`.

If `A` is a matrix, `mean(A)` treats the columns of `A` as vectors, returning a row vector of mean values.

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

`M = mean(A,dim)` returns the mean values for elements along the dimension of `A` specified by scalar `dim`. For matrices, `mean(A,2)` is a column vector containing the mean value of each row.

**Examples**

```
A = [1 2 3; 3 3 6; 4 6 8; 4 7 7];
mean(A)
ans =
    3.0000    4.5000    6.0000

mean(A,2)
ans =
    2.0000
    4.0000
    6.0000
    6.0000
```

**See Also** `corrcoef`, `cov`, `max`, `median`, `min`, `mode`, `std`, `var`

# mean (timeseries)

---

**Purpose** Mean value of timeseries data

**Syntax**

```
ts_mn = mean(ts)
ts_mn = mean(ts, 'PropertyName1', PropertyValue1, ...)
```

**Description** `ts_mn = mean(ts)` returns the mean value of `ts.Data`. When `ts.Data` is a vector, `ts_mn` is the mean value of `ts.Data` values. When `ts.Data` is a matrix, `ts_mn` is a row vector containing the mean value of each column of `ts.Data` (when `IsTimeFirst` is true and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `mean` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_mn = mean(ts, 'PropertyName1', PropertyValue1, ...)` specifies the following optional input arguments:

- 'MissingData' property has two possible values, 'remove' (default) or 'interpolate', indicating how to treat missing data during the calculation.
- 'Quality' values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- 'Weighting' property has two possible values, 'none' (default) or 'time'.  
When you specify 'time', larger time values correspond to larger weights.

**Examples** The following example illustrates how to find the mean values in multivariate time-series data.

**1** Load a 24-by-3 data array.

```
load count.dat
```

**2** Create a timeseries object with 24 time values.

```
count_ts = timeseries(count,[1:24], 'Name', 'CountPerSecond')
```

**3** Find the mean of each data column for this `timeseries` object.

```
mean(count_ts)

ans =

    32.0000    46.5417    65.5833
```

The mean is found independently for each data column in the `timeseries` object.

### See Also

```
iqr (timeseries), max (timeseries), min (timeseries), median  
(timeseries), std (timeseries), timeseries, var (timeseries)
```

# median

---

**Purpose** Median value of array

**Syntax** `M = median(A)`  
`M = median(A,dim)`

**Description** `M = median(A)` returns the median values of the elements along different dimensions of an array.

If `A` is a vector, `median(A)` returns the median value of `A`.

If `A` is a matrix, `median(A)` treats the columns of `A` as vectors, returning a row vector of median values.

If `A` is a multidimensional array, `median(A)` treats the values along the first nonsingleton dimension as vectors, returning an array of median values.

`M = median(A,dim)` returns the median values for elements along the dimension of `A` specified by scalar `dim`.

**Examples**

```
A = [1 2 4 4; 3 4 6 6; 5 6 8 8; 5 6 8 8];  
median(A)
```

```
ans =  
  
     4     5     7     7
```

```
median(A,2)
```

```
ans =  
  
     3  
     5  
     7  
     7
```

**See Also** `corrcoef`, `cov`, `max`, `mean`, `min`, `mode`, `std`, `var`



**Purpose** Median value of timeseries data

**Syntax**

```
ts_med = median(ts)
ts_med = median(ts, 'PropertyName1', PropertyValue1, ...)
```

**Description** `ts_med = median(ts)` returns the median value of `ts.Data`. When `ts.Data` is a vector, `ts_med` is the median value of `ts.Data` values. When `ts.Data` is a matrix, `ts_med` is a row vector containing the median value of each column of `ts.Data` (when `IsTimeFirst` is true and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `median` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_med = median(ts, 'PropertyName1', PropertyValue1, ...)` specifies the following optional input arguments:

- 'MissingData' property has two possible values, 'remove' (default) or 'interpolate', indicating how to treat missing data during the calculation.
- 'Quality' values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- 'Weighting' property has two possible values, 'none' (default) or 'time'.  
When you specify 'time', larger time values correspond to larger weights.

**Examples** The following example illustrates how to find the median values in multivariate time-series data.

**1** Load a 24-by-3 data array.

```
load count.dat
```

**2** Create a timeseries object with 24 time values.

## median (timeseries)

---

```
count_ts = timeseries(count,[1:24], 'Name', 'CountPerSecond')
```

**3** Find the median of each data column for this `timeseries` object.

```
median(count_ts)
```

```
ans =
```

```
23.5000 36.0000 39.0000
```

The median is found independently for each data column in the `timeseries` object.

### See Also

```
iqr (timeseries), max (timeseries), min (timeseries), mean  
(timeseries), std (timeseries), timeseries, var (timeseries)
```

**Purpose** Construct memmapfile object

**Syntax**

```
m = memmapfile(filename)
m = memmapfile(filename, prop1, value1, prop2, value2, ...)
```

**Description**

`m = memmapfile(filename)` constructs an object of the `memmapfile` class that maps file `filename` to memory using the default property values. The `filename` input is a quoted string that specifies the path and name of the file to be mapped into memory. `filename` must include a filename extension if the name of the file being mapped has an extension. The `filename` argument cannot include any wildcard characters (e.g., `*` or `?`), is case sensitive on The Open Group UNIX platforms, but is not case sensitive on Microsoft Windows platforms.

`m = memmapfile(filename, prop1, value1, prop2, value2, ...)` constructs an object of the `memmapfile` class that maps file `filename` into memory and sets the properties of that object that are named in the argument list (`prop1`, `prop2`, etc.) to the given values (`value1`, `value2`, etc.). All property name arguments must be quoted strings (e.g., `'Writable'`). Any properties that are not specified are given their default values.

Optional properties are shown in the table below and are described in the sections that follow.

Property	Description	Data Type	Default
Format	Format of the contents of the mapped region, including data type, array shape, and variable or field name by which to access the data	char array or N-by-3 cell array	uint8

# memmapfile

---

Property	Description	Data Type	Default
Offset	Number of bytes from the start of the file to the start of the mapped region. This number is zero-based. That is, offset 0 represents the start of the file.	double	0
Repeat	Number of times to apply the specified format to the mapped region of the file	double	Inf
Writable	Type of access allowed to the mapped region	logical	false

There are three different ways you can specify a value for the Format property. See the following sections in the MATLAB Programming Fundamentals documentation for more information on this:

- “Mapping a Single Data Type”
- “Formatting the Mapped Data to an Array”
- “Mapping Multiple Data Types and Arrays”

Any of the following data types can be used when you specify a Format value. The default type is `uint8`.

Format String	Data Type Description
'int8'	Signed 8-bit integers
'int16'	Signed 16-bit integers
'int32'	Signed 32-bit integers
'int64'	Signed 64-bit integers
'uint8'	Unsigned 8-bit integers
'uint16'	Unsigned 16-bit integers
'uint32'	Unsigned 32-bit integers
'uint64'	Unsigned 64-bit integers
'single'	32-bit floating-point
'double'	64-bit floating-point

## Remarks

You can only map an existing file. You cannot create a new file and map that file to memory in one operation. Use the MATLAB file I/O functions to create the file before attempting to map it to memory.

Once `memmapfile` locates the file, MATLAB stores the absolute pathname for the file internally, and then uses this stored path to locate the file from that point on. This enables you to work in other directories outside your current work directory and retain access to the mapped file.

Once a `memmapfile` object has been constructed, you can change the value of any of its properties. Use the `objname.property` syntax in assigning the new value. To set a new offset value for memory map object `m`, type

```
m.Offset = 2048;
```

Property names are not case sensitive. For example, MATLAB considers `m.Offset` to be the same as `m.offset`.

# memmapfile

---

## Examples

### Example 1

To construct a map for the file `records.dat` that resides in your current working directory, type the following:

```
m = memmapfile('records.dat');
```

MATLAB constructs an instance of the `memmapfile` class, assigns it to the variable `m`, and maps the entire `records.dat` file to memory, setting all properties of the object to their default values. In this example, the command maps the entire file as a sequence of unsigned 8-bit integers and gives the caller read-only access to its contents.

### Example 2

To construct a map using nondefault values for the `Offset`, `Format`, and `Writable` properties, type the following, enclosing all property names in single quotation marks:

```
m = memmapfile('records.dat',      ...
               'Offset', 1024,     ...
               'Format', 'uint32', ...
               'Writable', true);
```

Type the object name to see the current settings for all properties:

```
m
m =
  Filename: 'd:\matlab\mfiles\records.dat'
  Writable: true
  Offset: 1024
  Format: 'uint32'
  Repeat: Inf
  Data: 4778x1 uint32 array
```

### Example 3

Construct a `memmapfile` object for the entire file `records.dat` and set the `Format` property for that object to `uint64`. Any read or write

operations made via the memory map will read and write the file contents as a sequence of unsigned 64-bit integers:

```
m = memmapfile('records.dat', 'Format', 'uint64');
```

## Example 4

Construct a `memmapfile` object for a region of `records.dat` such that the contents of the region are handled by MATLAB as a 4-by-10-by-18 array of unsigned 32-bit integers, and can be referenced in the structure of the returned object using the field name `x`:

```
m = memmapfile('records.dat', ...
               'Offset', 1024, ...
               'Format', {'uint32' [4 10 18] 'x'});
```

```
A = m.Data.x;
```

```
whos A
  Name      Size      Bytes  Class
  ----      -
  A         4x10x18    2880   uint32 array
```

Grand total is 720 elements using 2880 bytes

## Example 5

Map a 24 kilobyte file containing data of three different data types: `int16`, `uint32`, and `single`. The `int16` data is mapped as a 2-by-2 matrix that can be accessed using the field name `model`. The `uint32` data is a scalar value accessed as field `serialno`. The `single` data is a 1-by-3 matrix named `expenses`.

Each of these fields belongs to the 800-by-1 structure array `m.Data`:

```
m = memmapfile('records.dat', ...
               'Offset', 2048, ...
               'Format', {
                   'int16' [2 2] 'model'; ...
                   'uint32' [1 1] 'serialno'; ...
```

# memmapfile

---

```
'single' [1 3] 'expenses'});
```

## Example 6

Map a file region identical to that of the previous example, except repeat the pattern of `int16`, `uint32`, and `single` data types only three times within the mapped region of the file. Allow write access to the file by setting the `Writable` property to `true`:

```
m = memmapfile('records.dat',           ...
               'Offset', 2048,           ...
               'Format', {               ...
                   'int16' [2 2] 'model'; ...
                   'uint32' [1 1] 'serialno'; ...
                   'single' [1 3] 'expenses'}, ...
               'Repeat', 3,             ...
               'Writable', true);
```

## See Also

`disp(memmapfile)`, `get(memmapfile)`



**Purpose** Display memory information

**Syntax**

```
memory
userview = memory
[userview systemview] = memory
```

**Description** `memory` displays information showing how much memory is available and how much the MATLAB software is currently using. The information displayed at your computer screen includes the following items, each of which is described in a section below:

- “Maximum Possible Array” on page 2-2244
- “Memory Available for All Arrays” on page 2-2245
- “Memory Used By MATLAB” on page 2-2246
- “Total Physical Memory (RAM)” on page 2-2246

`userview = memory` returns user-focused information on memory use in structure `userview`. The information returned in `userview` includes the following items, each of which is described in a section below:

- “Maximum Possible Array” on page 2-2244
- “Memory Available for All Arrays” on page 2-2245
- “Memory Used By MATLAB” on page 2-2246

`[userview systemview] = memory` returns both user- and system-focused information on memory use in structures `userview` and `systemview`, respectively. The `userview` structure is described in the command syntax above. The information returned in `systemview` includes the following items, each of which is described in a section below:

- “Virtual Address Space” on page 2-2247
- “System Memory” on page 2-2247

- “Physical Memory” on page 2-2248

## Output

Each of the sections below describes a value that is displayed or returned by the `memory` function.

### Maximum Possible Array

Maximum Possible Array is the size of the largest contiguous free memory block. As such, it is an upper bound on the largest single array MATLAB can create at this time.

MATLAB derives this number from the smaller of the following two values:

- The largest contiguous memory block found in the MATLAB virtual address space
- The total available system memory

To see how many array elements this number represents, divide by the number of bytes in the array class. For example, for a `double` array, divide by 8. The actual number of elements MATLAB can create is always fewer than this number.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
<code>memory</code>	String labelled Maximum possible array:
<code>user = memory</code>	Structure field <code>user.MaxPossibleArrayBytes</code>

All values are double-precision and in units of bytes.

### Footnotes

When you enter the `memory` command without specifying any outputs, MATLAB may also display one of the following footnotes. 32-bit systems

show either the first or second footnote; 64-bit systems show only the second footnote:

Limited by contiguous virtual address space available.

There is sufficient system memory to allow mapping of all virtual addresses in the largest available block of the MATLAB process. The maximum amount of total MATLAB virtual address space is either 2 GB or 3 GB, depending on whether the /3GB switch is in effect or not.

Limited by System Memory (physical + swap file) available.

There is insufficient system memory to allow mapping of all virtual addresses in the largest available block of the MATLAB process.

### Memory Available for All Arrays

Memory Available for All Arrays is the total amount of memory available to hold data. The amount of memory available is guaranteed to be at least as large as this field.

MATLAB derives this number from the smaller of the following two values:

- The total available MATLAB virtual address space
- The total available system memory

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
<code>memory</code>	String labelled Memory available for all arrays:
<code>user = memory</code>	Structure field <code>user.MemAvailableAllArrays</code>

## Footnotes

When you enter the `memory` command without specifying any outputs, MATLAB may also display one of the following footnotes. 32-bit systems show either the first or second footnote; 64-bit systems show only the latter footnote:

Limited by virtual address space available.

There is sufficient system memory to allow mapping of all available virtual addresses in the MATLAB process virtual address space to system memory. The maximum amount of total MATLAB virtual address space is either 2 GB or 3 GB, depending on whether the `/3GB` switch is in effect or not.

Limited by System Memory (physical + swap file) available.

There is insufficient system memory to allow mapping of all available virtual addresses in the MATLAB process.

## Memory Used By MATLAB

Memory Used By MATLAB is the total amount of system memory reserved for the MATLAB process. It is the sum of the physical memory and potential swap file usage.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
<code>memory</code>	String labelled Memory used by MATLAB:
<code>user = memory</code>	Structure field <code>user.MemUsedMATLAB</code>

## Total Physical Memory (RAM)

Physical Memory (RAM) is the total physical memory (or RAM) in the computer.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. See the table below.

Command	Returned in
memory	String labelled Physical Memory (RAM):

### Virtual Address Space

Virtual Address Space is the amount of available and total virtual memory for the MATLAB process. MATLAB returns the information in two fields of the return structure: Available and Total.

Command	Return Value	Returned in Structure Field
[user, sys] = memory	Available memory	sys.VirtualAddressSpace.Available
	Total memory	sys.VirtualAddressSpace.Total

You can monitor the difference:

```
VirtualAddressSpace.Total - VirtualAddressSpace.Available
```

as the Virtual Bytes counter in the WindowsPerformance program. (e.g., Windows XP Control Panel/Administrative Tool/Performance program).

### System Memory

System Memory is the amount of available system memory on your computer system. This number includes the amount of available physical memory and the amount of available swap file space on the computer running MATLAB. MATLAB returns the information in the SystemMemory field of the return structure.

Command	Return Value	Returned in Structure Field
[user, sys] = memory	Available memory	sys.SystemMemory

This is the same as the difference:

```
limit - total (in bytes)
```

# memory

---

found in the Windows Task Manager: Performance/Commit Charge.

## Physical Memory

Physical Memory is the available and total amounts of physical memory (RAM) on the computer running MATLAB. MATLAB returns the information in two fields of the return structure: Available and Total.

Command	Value	Returned in Structure Field
[user,sys] = memory	Available memory	sys.PhysicalMemory.Available
	Total memory	sys.PhysicalMemory.Total

Available physical memory is the same as:

Available (in bytes)

found in the Windows Task Manager: Performance/Physical Memory

The total physical memory is the same as

Total (in bytes)

found in the Windows Task Manager: Performance/Physical Memory

You can use the amount of available physical memory as a measure of how much data you can access quickly.

## Remarks

The memory function is currently available on Microsoft Windows systems only. Results vary, depending on the computer running MATLAB, the load on that computer, and what MATLAB is doing at the time.

## Details on Memory Used By MATLAB

MATLAB computes the value for Memory Used By MATLAB by walking the MATLAB process memory structures and summing all the sections that have physical storage allocated in memory or in the paging file on disk.

Using the Windows Task Manager, you have for the MATLAB.exe image:

$$\text{Mem Usage} < \text{MemUsedMATLAB} < \text{Mem Usage} + \text{VM Size (in bytes)}$$

where both of the following are true:

- Mem Usage is the working set size in kilobytes.
- VM Size is the page file usage, or private bytes, in kilobytes.

The working set size is the portion of the MATLAB virtual address space that is *currently* resident in RAM and can be referenced without a memory page fault. The page file usage gives the portion of the MATLAB virtual address space that requires a backup that doesn't already exist. Another name for page file usage is *private bytes*. It includes all MATLAB variables and workspaces. Since some of the pages in the page file may also be part of the working set, this sum is an overestimate of MemUsedMATLAB. Note that there are virtual pages in the MATLAB process space that already have a backup. For example, code loaded from EXEs and DLLs and memory-mapped files. If any part of those files is in memory when the memory builtin is called, that memory will be counted as part of MemUsedMATLAB.

### Reserved Addresses

Reserved addresses are addresses set aside in the process virtual address space for some specific future use. These reserved addresses reduce the size of MemAvailableAllArrays and can reduce the size of the current or future value of MaxPossibleArrayBytes.

#### Example 1 — Java Virtual Machine (JVM)

At MATLAB startup, part of the MATLAB virtual address space is reserved by the Java Virtual Machine (JVM) and cannot be used for storing MATLAB arrays.

#### Example 2 — Standard Windows Heap Manager

MATLAB, by default, uses the standard Windows heap manager except for a set of small preselected allocation sizes. One characteristic of this heap manager is that its behavior depends upon whether the requested

# memory

---

allocation is less than or greater than the fixed number of 524,280 bytes. For, example, if you create a sequence of MATLAB arrays, each less than 524,280 bytes, and then clear them all, the MemUsedMATLAB value before and after shows little change, and the MemAvailableAllArrays value is now smaller by the total space allocated.

The result is that, instead of globally freeing the extra memory, the memory becomes reserved. It can *only* be reused for arrays less than 524,280 bytes. You cannot reclaim this memory for a larger array except by restarting MATLAB.

## Examples

Display memory statistics on a 32-bit Windows system:

```
memory
```

```
Maximum possible array:          677 MB (7.101e+008 bytes) *
Memory available for all arrays: 1601 MB (1.679e+009 bytes) **
Memory used by MATLAB:          446 MB (4.681e+008 bytes)
Physical Memory (RAM):          3327 MB (3.489e+009 bytes)
```

\* Limited by contiguous virtual address space available.

\*\* Limited by virtual address space available.

Return in the structure `userview`, information on the largest array MATLAB can create at this time, how much memory is available to hold data, and the amount of memory currently being used by your MATLAB process:

```
userview = memory
```

```
userview =
    MaxPossibleArrayBytes: 710127616
    MemAvailableAllArrays: 1.6792e+009
    MemUsedMATLAB: 468127744
```

Assign the output to two structures, `user` and `sys`, to obtain the information shown here:



```
[user sys] = memory;

% --- Largest array MATLAB can create ---
user.MaxPossibleArrayBytes
ans =
    710127616

% --- Memory available for data ---
user.MemAvailableAllArrays
ans =
    1.6797e+009

% --- Memory used by MATLAB process ---
user.MemUsedMATLAB
ans =
    467603456

% --- Virtual memory for MATLAB process ---
sys.VirtualAddressSpace
ans =
    Available: 1.6797e+009
    Total: 2.1474e+009

% --- Physical memory and paging file ---
sys.SystemMemory
ans =
    Available: 4.4775e+009

% --- Computer's physical memory ---
sys.PhysicalMemory
ans =
    Available: 2.3941e+009
    Total: 3.4889e+009
```

**See Also**

clear, pack, whos, inmem, save, load, mlock, munlock

# menu

---

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

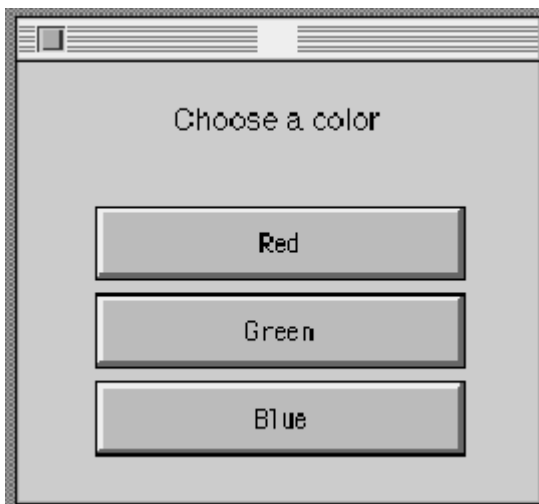
**Syntax** `k = menu('mtitle','opt1','opt2',..., 'optn')`

**Description** `k = menu('mtitle','opt1','opt2',..., 'optn')` displays the menu whose title is in the string variable `'mtitle'` and whose choices are string variables `'opt1'`, `'opt2'`, and so on. `menu` returns the number of the selected menu item.

If the user's terminal provides a graphics capability, `menu` displays the menu items as push buttons in a figure window (Example 1), otherwise they will be given as a numbered list in the command window (Example 2).

**Remarks** To call `menu` from another ui object, set that object's `Interruptible` property to `'yes'`. For more information, see the MATLAB Graphics documentation.

**Examples** **Example 1**  
`k = menu('Choose a color', 'Red', 'Green', 'Blue')` displays



After input is accepted, use `k` to control the color of a graph.

```
color = ['r','g','b']  
plot(t,s,color(k))
```

### Example 2

```
K = menu('Choose a color','Red','Blue','Green')
```

displays on the Command Window

```
----- Choose a color -----  
1) Red  
2) Blue  
3) Green  
Select a menu number:
```

The number entered by the user in response to the prompt is returned as `K` (i.e. `K = 2` implies that the user selected `Blue`).

### See Also

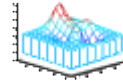
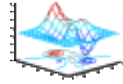
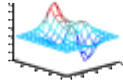
`guide`, `input`, `uicontrol`, `uimenu`

# mesh, meshc, meshz


---

## Purpose

Mesh plots



## GUI Alternatives

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

## Syntax

```
mesh(X,Y,Z)
mesh(Z)
mesh(...,C)
mesh(...,'PropertyName',PropertyValue,...)
mesh(axes_handles,...)
meshc(...)
meshz(...)
h = mesh(...)
hsurface = mesh('v6',...) hsurface = meshc('v6',...),
```

## Description

mesh, meshc, and meshz create wireframe parametric surfaces specified by X, Y, and Z, with color specified by C.

mesh(X,Y,Z) draws a wireframe mesh with color determined by Z so color is proportional to surface height. If X and Y are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case,  $(X(j), Y(i), Z(i,j))$  are the intersections of the wireframe grid lines; X and Y correspond to the columns and rows of Z, respectively. If X and Y are matrices,  $(X(i,j), Y(i,j), Z(i,j))$  are the intersections of the wireframe grid lines.

mesh(Z) draws a wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, Z, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`mesh(...,C)` draws a wireframe mesh with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`mesh(...,'PropertyName',PropertyValue,...)` sets the value of the specified surface property. Multiple property values can be set with a single statement.

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

`meshc(...)` draws a contour plot beneath the mesh.

`meshz(...)` draws a curtain plot (i.e., a reference plane) around the mesh.

`h = mesh(...)`, `h = meshc(...)`, and `h = meshz(...)` return a handle to a surfaceplot graphics object.

## Backward-Compatible Version

`hsurface = mesh('v6',...)`, `hsurface = meshc('v6',...)`, and `hsurface = meshz('v6',...)` returns the handles of surface objects instead of surfaceplot objects for compatibility with MATLAB 6.5 and earlier.

---

**Note** The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

---

See Plot Objects and Backward Compatibility for more information.

## Remarks

`mesh`, `meshc`, and `meshz` do not accept complex inputs.

A mesh is drawn as a surface graphics object with the viewpoint specified by `view(3)`. The face color is the same as the background color (to simulate a wireframe with hidden-surface elimination), or none when drawing a standard see-through wireframe. The current colormap determines the edge color. The `hidden` command controls the

# mesh, meshc, meshz

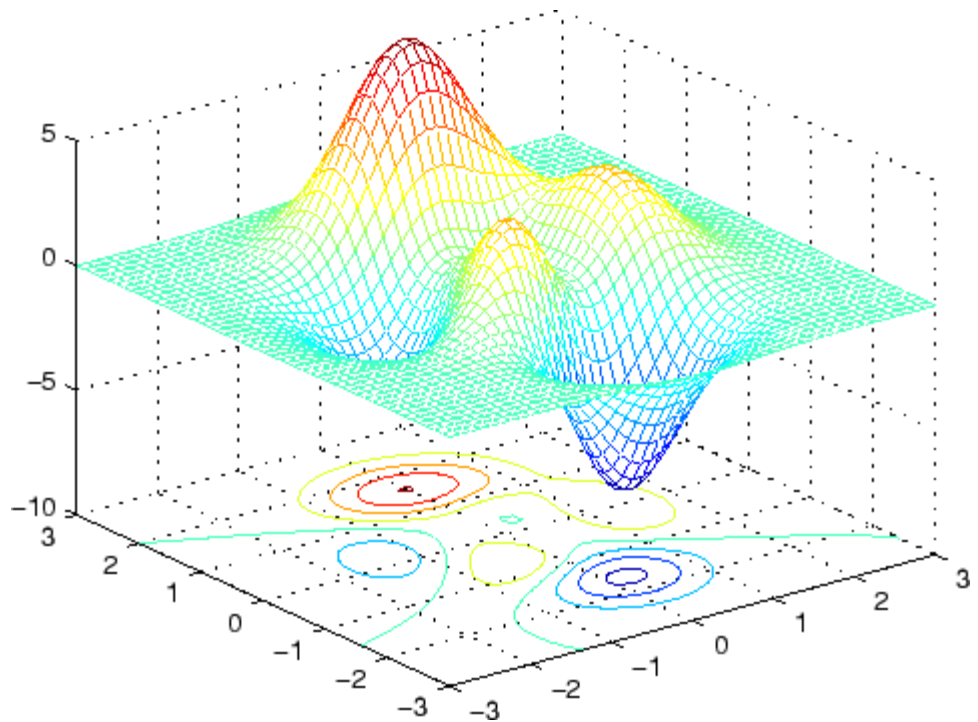
---

simulation of hidden-surface elimination in the mesh, and the shading command controls the shading model.

## Examples

Produce a combination mesh and contour plot of the peaks surface:

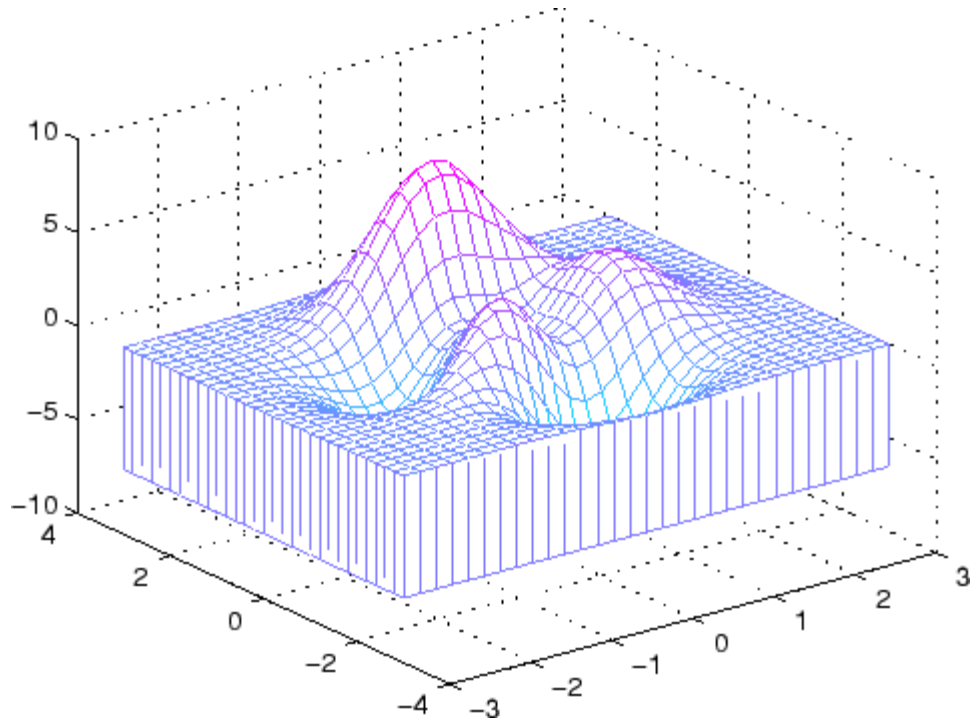
```
[X,Y] = meshgrid(-3:.125:3);  
Z = peaks(X,Y);  
meshc(X,Y,Z);  
axis([-3 3 -3 3 -10 5])
```



Generate the curtain plot for the peaks function:

```
[X,Y] = meshgrid(-3:.125:3);  
Z = peaks(X,Y);
```

`meshz(X,Y,Z)`



## Algorithm

The range of  $X$ ,  $Y$ , and  $Z$ , or the current settings of the axes `XLimMode`, `YLimMode`, and `ZLimMode` properties, determine the axis limits. `axis` sets these properties.

The range of  $C$ , or the current settings of the axes `CLim` and `CLimMode` properties (also set by the `caxis` function), determine the color scaling. The scaled color values are used as indices into the current colormap.

The mesh rendering functions produce color values by mapping the  $z$  data values (or an explicit color array) onto the current colormap. The MATLAB default behavior is to compute the color limits automatically using the minimum and maximum data values (also set using `caxis`

## mesh, meshc, meshz

---

auto). The minimum data value maps to the first color value in the colormap and the maximum data value maps to the last color value in the colormap. MATLAB performs a linear transformation on the intermediate values to map them to the current colormap.

meshc calls mesh, turns hold on, and then calls contour and positions the contour on the  $x$ - $y$  plane. For additional control over the appearance of the contours, you can issue these commands directly. You can combine other types of graphs in this manner, for example surf and pcolor plots.

meshc assumes that  $X$  and  $Y$  are monotonically increasing. If  $X$  or  $Y$  is irregularly spaced, contour3 calculates contours using a regularly spaced contour grid, then transforms the data to  $X$  or  $Y$ .

### See Also

contour, hidden, meshgrid, surface, surf, surfc, surf1, waterfall  
“Surface and Mesh Creation” on page 1-101 for related functions

Surfaceplot Properties for a list of surfaceplot properties

The functions axis, caxis, colormap, hold, shading, and view all set graphics object properties that affect mesh, meshc, and meshz.

For a discussion of parametric surfaces plots, refer to surf.



**Purpose** Generate X and Y arrays for 3-D plots

**Syntax**

```
[X,Y] = meshgrid(x,y)
[X,Y] = meshgrid(x)
[X,Y,Z] = meshgrid(x,y,z)
```

**Description** `[X,Y] = meshgrid(x,y)` transforms the domain specified by vectors `x` and `y` into arrays `X` and `Y`, which can be used to evaluate functions of two variables and three-dimensional mesh/surface plots. The rows of the output array `X` are copies of the vector `x`; columns of the output array `Y` are copies of the vector `y`.

`[X,Y] = meshgrid(x)` is the same as `[X,Y] = meshgrid(x,x)`.

`[X,Y,Z] = meshgrid(x,y,z)` produces three-dimensional arrays used to evaluate functions of three variables and three-dimensional volumetric plots.

**Remarks** The `meshgrid` function is similar to `ndgrid` except that the order of the first two input and output arguments is switched. That is, the statement

```
[X,Y,Z] = meshgrid(x,y,z)
```

produces the same result as

```
[Y,X,Z] = ndgrid(y,x,z)
```

Because of this, `meshgrid` is better suited to problems in two- or three-dimensional Cartesian space, while `ndgrid` is better suited to multidimensional problems that aren't spatially based.

`meshgrid` is limited to two- or three-dimensional Cartesian space.

**Examples**

```
[X,Y] = meshgrid(1:3,10:14)
```

`X =`

```

1     2     3
1     2     3
```

# meshgrid

---

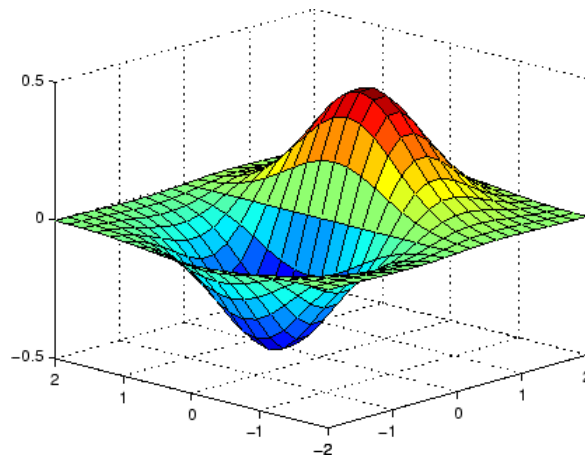
```
1 2 3
1 2 3
1 2 3
```

Y =

```
10 10 10
11 11 11
12 12 12
13 13 13
14 14 14
```

The following example shows how to use meshgrid to create a surface plot of a function.

```
[X,Y] = meshgrid(-2:.2:2, -2:.2:2);
Z = X .* exp(-X.^2 - Y.^2);
surf(X,Y,Z)
```



## See Also

griddata, mesh, ndgrid, slice, surf

## Purpose

`meta.class` class describes MATLAB classes

## Description

The `meta.class` class contains information about MATLAB classes. The read/write properties of the `meta.class` class correspond to class attributes and are set only from within class definitions on the `classdef` line. You can query the read-only properties of the `meta.class` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class).

You can construct a `meta.class` object from an instance of a class or using the class name:

- `metaclass` — returns a `meta.class` object representing the object passed as an argument.
- `?classname` — returns a `meta.class` object representing the named class.
- `fromName` — static method returns a `meta.class` object representing the named class.

For example, the `metaclass` function returns the `meta.class` object representing `myclass`.

```
ob = myclass;  
obmeta = metaclass(ob);  
obmeta.Name  
ans =  
myclass
```

You can use the class name to obtain the `meta.class` object:

```
obmeta = ?myclass;
```

You can also use the `fromName` static method:

```
obmeta = meta.class.fromName('myclass');
```

# meta.class

---

## Properties

Property	Purpose
ConstructOnLoad attribute, default = false	If true, the class constructor is called automatically when loading an object from a MAT-file. Therefore, the construction must be implemented so that calling it with no arguments does not produce an error.  See “Saving and Loading Objects”
ContainingPackage read only	A <code>meta.package</code> object describing the package within which this class is contained, or an empty object if this class is not in a package.
Description read only	Currently not used
DetailedDescription read only	Currently not used
Events read only	A cell array of <code>meta.event</code> objects describing each event defined by this class, including all inherited events.
Hidden attribute, default = false	If set to true, the class does not appear in the output of MATLAB commands or tools that display class names.
InferiorClasses attribute, default = {}	A cell array of <code>meta.class</code> objects defining the precedence of classes represented by the list as inferior to this class.  See “Specifying Class Precedence”
Methods read only	A cell array of <code>meta.method</code> objects describing each method defined by this class, including all inherited public and protected methods.
Name read only	Name of the class associated with this <code>meta.class</code> object (char array)

Property	Purpose
Properties read only	A cell array of <code>meta.property</code> objects describing each property defined by this class, including all inherited public and protected properties.
Sealed attribute, default = false	If true, the class can be not be specialized with subclasses.
SuperClasses read only	A cell array of <code>meta.class</code> objects describing each base class from which this class is derived.

## Methods

Method	Purpose
<code>fromName</code>	Returns the <code>meta.class</code> object associated with the specified class name.
<code>tf = eq(cls)</code>	Equality function ( <code>a == b</code> ). Use to test if two variables refer to equal classed (classes that contain exactly the same list of elements).
<code>tf = ne(cls)</code>	Not equal function ( <code>a ~= b</code> ). Use to test if two variables refer to different meta-classes.
<code>tf = lt(clsa, clsb)</code>	Less than function ( <code>clsA &lt; clsB</code> ). Use to determine if <code>clsA</code> is a strict subclass of <code>clsB</code> (i.e., <code>clsB &lt; clsB</code> is false).
<code>tf = le(clsa, clsb)</code>	Less than or equal to function ( <code>clsA &lt;= clsB</code> ). Use to determine if <code>clsA</code> is a subclass of <code>clsB</code> .

# meta.class

---

Method	Purpose
tf = gt(clsa, clsb)	Greater than function (clsb > clsa). Use to determine if clsb is a strict superclass of clsa (i.e., clsb > clsb is false).
tf = ge(clsa, clsb)	Greater than or equal to function (clsb >= clsa). Use to determine if clsb is a superclass of clsa.

## See Also

See “Obtaining Information About Classes with Meta-Classes”,  
fromName, meta.property, meta.method, meta.event, meta.package

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

**Syntax** `mcls = meta.class.fromName('className')`

**Description** `mcls = meta.class.fromName('className')` is a static method that returns the `meta.class` object associated with the class `className`. Note that you can also use the `?` operator to obtain the `meta.class` object for a class name:

```
mcls = ?className;
```

The equivalent call to `fromName` is:

```
mcls = meta.class.fromName('className');
```

**See Also** `meta.class`

# meta.DynamicProperty

---

## Purpose

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

## Description

The `meta.DynamicProperty` class contains descriptive information about dynamic properties that you have added to an instance of a MATLAB classes. The MATLAB class must be a subclass of `dynamicprops`. The properties of the `meta.DynamicProperty` class correspond to property attributes that you specify from within class definitions. Dynamic properties are not defined in `classdef` blocks, but you can set their attributes by setting the `meta.DynamicProperty` object properties.

You add a dynamic property to an object using the `addprop` method of the `dynamicprops` class. The `addprop` method returns a `meta.DynamicProperty` instance representing the new dynamic property. You can modify the properties of the `meta.DynamicProperty` object to set the attributes of the dynamic property or to add set and get access methods, which would be defined in the `classdef` for regular properties.

To remove the dynamic property, call the `delete` handle class method on the `meta.DynamicProperty` object.

Obtain a `meta.DynamicProperty` object from the `addprops` method, which returns an array of `meta.DynamicProperty` objects, one for each dynamic property.

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

## Properties

Property	Purpose
Name	Name of the property.
Description	Can contain text
DetailedDescription	Can contain text



Property	Purpose
Abstract attribute, default = false	<p>If true, the property has no implementation, but a concrete subclass must redefine this property without <code>Abstract</code> being set to true.</p> <ul style="list-style-type: none"><li>• Abstract properties cannot define set or get access methods. See “Controlling Property Access”</li><li>• Abstract properties cannot define initial values. “Assigning an Initial Value”</li><li>• All subclasses must specify the same values as the superclass for the property <code>SetAccess</code> and <code>GetAccess</code> attributes.</li><li>• <code>Abstract=true</code> should be used with the class attribute <code>Sealed=false</code> (the default).</li></ul>
constant attribute, default = false	<p>Set to true if you want only one value for this property in all instances of the class.</p> <ul style="list-style-type: none"><li>• Subclasses inherit constant properties, but cannot change them.</li><li>• Constant properties cannot be <code>Dependent</code></li><li>• <code>SetAccess</code> is ignored.</li></ul> <p>See</p>
<code>GetAccess</code> attribute, default = public	<p><code>public</code> – unrestricted access</p> <p><code>protected</code> – access from class or derived classes</p> <p><code>private</code> – access by class members only</p>

# meta.DynamicProperty

---

Property	Purpose
SetAccess attribute, default = public	public – unrestricted access protected – access from class or derived classes private – access by class members only
Dependent attribute, default = false	If false, property value is stored in object. If true, property value is not stored in object and the set and get functions cannot access the property by indexing into the object using the property name. See “Property Get Methods”
Transient attribute, default = false	If true, property value is not saved when object is saved to a file. See “Saving and Loading Objects” for more about saving objects.
Hidden attribute, default = false	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to <code>properties</code> , etc.).
GetObservable attribute, default = false	If true, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See “Property-Set and Query Events”
SetObservable attribute, default = false	If true, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See “Property-Set and Query Events”
GetMethod	Function handle of the get method associated with this property. Empty if there is no get method specified. See “Property Get Method”

Property	Purpose
SetMethod	Function handle of the set method associated with this property. Empty if there is no set method specified. See “Property Set Methods”
DefiningClass	The meta.class object representing the class that defines this property.

## Events

See for information on using property events.

Event Name	Purpose
PreGet	Event occurs just before property is queried.
PostGet	Event occurs just after property has been queried
PreSet	Event occurs just before this property is modified
PostSet	Event occurs just after this property has been modified
ObjectBeingDestroyed	Inherited from handle

## See Also

addprop, handle

# meta.event

---

## Purpose

meta.event class describes MATLAB class events

## Description

The meta.event class provides information about MATLAB class events. The read/write properties of the meta.event class correspond to event attributes and are specified only from within class definitions.

You can query the read-only properties of the meta.event object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class defining the event).

Obtain a meta.event object from the meta.class Events property, which contains a cell array of meta.event objects, one for each event defined by the class. For example:

```
mco = ?classname;  
eventcell = mco.Events;  
eventcell{1}.Name; % name of first event
```

Use the metaclass function to obtain a meta.class object from a class instance:

```
mco = metaclass(obj);
```

## Properties

Property	Purpose
Name read only	Name of the event.
Description read only	Currently not used
DetailedDescription read only	Currently not used
Hidden	If true, the event does not appear in the list of events returned by the events function (or other event listing functions or viewers)

Property	Purpose
ListenAccess	Determines where you can create listeners for the event. <ul style="list-style-type: none"><li>• <code>public</code> — unrestricted access</li><li>• <code>protected</code> — access from methods in class or derived classes</li><li>• <code>private</code> — access by class methods only (not from derived classes)</li></ul>
NotifyAccess	Determines where code can trigger the event. <ul style="list-style-type: none"><li>• <code>public</code> — any code can trigger event</li><li>• <code>protected</code> — can trigger event from methods in class or derived classes</li><li>• <code>private</code> — can trigger event by class methods only (not from derived classes)</li></ul>
DefiningClass	The <code>meta.class</code> object representing the class that defines this event.

**See Also**

`meta.class`, `meta.property`, `meta.method`, `metaclass`

“Events — Sending and Responding to Messages”

“Obtaining Information About Classes with Meta-Classes”

# meta.method

---

## Purpose

`meta.method` class describes MATLAB class methods

## Description

The `meta.method` class provides information about the methods of MATLAB classes. The read/write properties of the `meta.method` class correspond to method attributes and are specified only from within class definitions.

You can query the read-only properties of the `meta.method` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class defining a method).

Obtain a `meta.method` object from the `meta.class` `Methods` property, which contains a cell array of `meta.method` objects, one for each class method. For example:

```
mco = ?classname;  
methodcell = mco.Methods;  
methodcell{1}.Name; % name of first method
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

**Properties**

Property	Purpose
Abstract	<p>If true, the method has no implementation. The method has a syntax line that can include arguments, which subclasses use when implementing the method.</p> <ul style="list-style-type: none"> <li>• Subclasses are not required to define the same number of input and output arguments.</li> <li>• The method can have comments after the function line</li> <li>• Does not contain <code>function</code> or <code>end</code> keywords, only the function syntax (e.g., <code>[a,b] = myMethod(x,y)</code>)</li> </ul>
Access attribute, default = false	<p>Determines what code can call this method.</p> <ul style="list-style-type: none"> <li>• <code>public</code> — unrestricted access</li> <li>• <code>protected</code> — access from methods in class</li> </ul>
DefiningClass	<p>The <code>meta.class</code> object representing the class that defines this method.</p>
Description read only	<p>Currently not used</p>
DetailedDescription read only	<p>Currently not used</p>
Hidden attribute, default = false	<p>When <code>false</code>, the method name shows in the list of methods displayed using the <code>methods</code> or <code>methodsview</code> commands. If set to <code>true</code>, the method name is not included in these listings.</p>
Name read only	<p>Name of the method.</p>

# meta.method

---

Property	Purpose
Sealed attribute, default = false	If true, the method cannot be redefined in a subclass. Attempting to define a method with the same name in a subclass causes an error.
Static attribute, default = false	Set to true to define a method that is not depend on an object of the class and does not require an object argument. You must use class name to call the method: <i>classname.methodname</i> See “Static Methods”

## See Also

`meta.class`, `meta.property`, `meta.event`, `metaclass`

“Methods — Defining Class Operations”

“Obtaining Information About Classes with Meta-Classes”



**Purpose** meta.package class describes MATLAB packages

**Description** The meta.package class contains information about MATLAB packages.

## Properties

Property	Purpose
Name read only	Name of the package associated with this meta.package object
Packages read only	List of packages that are scoped to this package. A cell array of meta.package objects.
Classes read only	List of classes that are scoped to this package. A cell array of meta.class objects.
Functions read only	List of functions that are scoped to this package. A cell array of function handles.
ContainingPackage read only	A meta.package object describing the package within which this package is contained, or an empty object if this package is not nested.

## Methods

Method	Purpose
fromName	Static method returns a meta.package object for a specified package name.
getAllPackages	Static method returns a cell array of meta.package objects representing all top-level packages.

## See Also

See “Obtaining Information About Classes with Meta-Classes”  
meta.class, meta.property, meta.method, meta.event

# meta.package.fromName

---

**Purpose** Return meta.package object for specified package

**Syntax** mpkg = meta.package.fromName('pkgname')

**Description** mpkg = meta.package.fromName('pkgname') is a static method that returns the meta.package object associated with the named package. If pkgname is a nested package, then you must provide the fully qualified name (e.g., 'pkgname1.pkgname2').

**Examples** List the classes in the event package:

```
mev = meta.package.fromName('event');
for k=1:length(mev.Classes)
    disp(mev.Classes{k}.Name)
end
event.EventData
event.PropertyEvent
event.listener
event.proplistener
```

**See Also** meta.package, meta.package.getAllPackages

**Purpose** Get all top-level packages

**Syntax** `P = meta.package.getAllPackages`

**Description** `P = meta.package.getAllPackages` is a static method that returns a cell array of `meta.package` objects representing all the top-level packages that are visible on the MATLAB path or defined as top-level built-in packages. You can access subpackages using the `Packages` property of each `meta.package` object.

Note that the time required to find all the packages on the path might be excessively long in some cases. You should therefore avoid using this method in any code where execution time is a consideration. `getAllPackages` is generally intended for interactive use only.

**See Also** `meta.package`, `meta.package.fromName`

# meta.property

---

**Purpose** meta.property class describes MATLAB class properties

**Description** The meta.property class provides information about the properties of MATLAB classes. The read/write properties of the meta.property class correspond to property attributes and are specified only from within your class definitions.

You can query the read-only properties of the meta.property object to obtain information that is specified syntactically by the class (for example, to obtain the function handle of a property's set access method).

Obtain a meta.property object from the meta.class Properties property, which contains a cell array of meta.property objects, one for each class property. For example:

```
mco = ?classname;  
propcell = mco.Properties;  
propcell{1}.Name; % name of first property
```

Use the metaclass function to obtain a meta.class object from a class instance:

```
mco = metaclass(obj);
```

## Properties

Property	Purpose
Name read only	Name of the property.
Description read only	Currently not used
DetailedDescription read only	Currently not used

Property	Purpose
<p>Abstract attribute, default = false</p>	<p>If true, the property has no implementation, but a concrete subclass must redefine this property without <code>Abstract</code> being set to true.</p> <ul style="list-style-type: none"> <li>• Abstract properties cannot define set or get access methods. See “Controlling Property Access”</li> <li>• Abstract properties cannot define initial values. “Assigning an Initial Value”</li> <li>• All subclasses must specify the same values as the superclass for the property <code>SetAccess</code> and <code>GetAccess</code> attributes.</li> <li>• <code>Abstract=true</code> should be used with the class attribute <code>Sealed=false</code> (the default).</li> </ul>
<p>constant attribute, default = false</p>	<p>Set to true if you want only one value for this property in all instances of the class.</p> <ul style="list-style-type: none"> <li>• Subclasses inherit constant properties, but cannot change them.</li> <li>• Constant properties cannot be <code>Dependent</code></li> <li>• <code>SetAccess</code> is ignored.</li> </ul> <p>See</p>
<p><code>GetAccess</code> attribute, default = public</p>	<p><code>public</code> – unrestricted access</p> <p><code>protected</code> – access from class or derived classes</p> <p><code>private</code> – access by class members only</p>

# meta.property

---

Property	Purpose
SetAccess attribute, default = public	public – unrestricted access protected – access from class or derived classes private – access by class members only
Dependent attribute, default = false	If false, property value is stored in object. If true, property value is not stored in object and the set and get functions cannot access the property by indexing into the object using the property name. See “Property Get Methods”
Transient attribute, default = false	If true, property value is not saved when object is saved to a file. See “Saving and Loading Objects” for more about saving objects.
Hidden attribute, default = false	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to <code>properties</code> , etc.).
GetObservable attribute, default = false	If true, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See “Property-Set and Query Events”
SetObservable attribute, default = false	If true, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See “Property-Set and Query Events”
GetMethod read only	Function handle of the get method associated with this property. Empty if there is no get method specified. See “Property Get Method”

Property	Purpose
SetMethod read only	Function handle of the set method associated with this property. Empty if there is no set method specified. See “Property Set Methods”
DefiningClass	The meta.class object representing the class that defines this property.

## Events

See for information on using property events.

Event Name	Purpose
PreGet	Event occurs just before property is queried.
PostGet	Event occurs just after property has been queried
PreSet	Event occurs just before this property is modified
PostSet	Event occurs just after this property has been modified

## See Also

meta.class, meta.method, meta.event, metaclass

“Properties — Storing Class Data”

“Obtaining Information About Classes with Meta-Classes”

# metaclass

---

**Purpose** Return `meta.class` object

**Syntax** `mc = metaclass(object)`  
`mc = ?classname`

**Description** `mc = metaclass(object)` returns the `meta.class` object for the class of `object`. `object` can be a scalar or an array of objects. However, `metaclass` always returns a scalar `meta.class` object.

`mc = ?classname` returns the `meta.class` object for the class with name `classname`. The `?` operator works only with a class name, not a class instance.

---

**Note** If you pass a class name as a string to the `metaclass` function, it returns the `meta.class` object for the `char` class. Use the `?` operator to obtain the `meta.class` object from a class name.

---

**Examples** Return the `meta.class` object for an instance of the `MException` class:

```
obj = MException('Msg:ID', 'MsgTxt');  
mc = metaclass(obj);
```

Use the `?` operator to get the `meta.class` object for the `hgsetget` class:

```
mc = ?hgsetget;
```

**See Also** See “Obtaining Information About Classes with Meta-Classes” for more information.

`meta.class`



**Purpose** Information on class methods

**Syntax**

```
methods('classname')
methods(obj)
methods(..., '-full')
m = methods(...)
```

**Description**

`methods('classname')` displays the names of the methods for the class *classname*. If *classname* is a MATLAB or Java class, then `methods` displays only public methods, including those inherited from superclasses.

`methods(obj)` displays the names of the methods for the class of the object `obj`.

`methods(..., '-full')` displays a full description of the methods, including inheritance information and, for MATLAB and Java methods, method attributes and signatures. Duplicate method names with different signatures are not removed. Note that this option does not work with classes using pre MATLAB Version 7.6 class definitions (i.e., classes not using the `classdef` syntax).

`m = methods(...)` returns the method names in a cell array of strings.

---

**Note** `methods` is also a keyword used in MATLAB class definition. See `classdef` for more information on class definition keywords.

---

**Examples** Retrieve the names of the static methods in class `memmapfile`:

```
m = methods('memmapfile', '-full');
for k=1:length(m)
    if ~isempty(findstr(m{k}, 'Static'))
        disp(m(k))
    end
end
```

# methods

---

```
'Static DeleteDataHandle(dh)'
```

```
'Static E empty(varargin)'
```

```
'Static obj loadobj(obj)'
```

Construct a `java.lang.String` instance and display the names of the public methods of that instance:

```
s = java.lang.String;  
methods(s)  
Methods for class java.lang.String:
```

String	endsWith	lastIndexOf	startsWith
charAt	equals	length	subSequence
codePointAt	equalsIgnoreCase	matches	substring
codePointBefore	format	notify	toCharArray
codePointCount	getBytes	notifyAll	toLowerCase
compareTo	getChars	offsetByCodePoints	toString
compareToIgnoreCase	getClass	regionMatches	toUpperCase
concat	hashCode	replace	trim
contains	indexOf	replaceAll	valueOf
contentEquals	intern	replaceFirst	wait
copyValueOf	isEmpty	split	

## See Also

`methodsview`, `properties`, `events`, `what`, `which`

**Purpose**

Information on class methods in separate window

**Syntax**

```
methodsview packagename.classname
methodsview classname
methodsview(object)
```

**Description**

`methodsview packagename.classname` displays information about the methods in the class, `classname`. You must include `packagename` if the class is in a package. If `classname` is a MATLAB or Sun Java class, `methodsview` lists only public methods, including those inherited from superclasses.

`methodsview classname` displays information describing the class `classname`.

`methodsview(object)` displays information about the methods of the class of `object`.

`methodsview` creates a new window that displays the methods defined in the specified class and lists the following information;

- **Qualifiers**—information about the method (for example, `abstract` or `synchronized`)
- **Returned Type**—class of returned values
- **Name**—name of the method
- **Arguments**—Arguments passed to the method
- **Other**—Possible exceptions thrown
- **Inherited From**—Superclass of the specified class

**Examples**

The following command lists information on all methods in the `java.awt.MenuItem` class.

```
methodsview java.awt.MenuItem
```

# methodsview

MATLAB displays this information in a new window, as shown below

Qualifiers	Return Type	Name	Arguments	Other	Inherited From
		MenuItem	{java.lang.String}	throws java.awt.HeadlessExcept...	
		MenuItem	{java.lang.String, java.awt.MenuSh...	throws java.awt.HeadlessExcept...	
		MenuItem	()	throws java.awt.HeadlessExcept...	
synchroni...	void	addActionListener	{java.awt.event.ActionListener}		
	void	addNotify	()		
	void	deleteShortcut	()		
synchroni...	void	disable	()		
	void	dispatchEvent	{java.awt.AWTEvent}		java.awt.MenuCo...
synchroni...	void	enable	()		
	void	enable	{boolean}		
	boolean	equals	{java.lang.Object}		java.lang.Object
	javax.accessibility.AccessibleCon...	getAccessibleCo...	()		
	java.lang.String	getActionCommand	()		
synchroni...	java.awt.event.ActionListener[]	getActionListeners	()		
	java.lang.Class	getClass	()		java.lang.Object
	java.awt.Font	getFont	()		java.awt.MenuCo...
	java.lang.String	getLabel	()		
	java.util.EventListener[]	getListeners	{java.lang.Class}		
	java.lang.String	getName	()		java.awt.MenuCo...
	java.awt.MenuContainer	getParent	()		java.awt.MenuCo...
	java.awt.peer.MenuComponent...	getPeer	()		java.awt.MenuCo...
	java.awt.MenuShortcut	getShortcut	()		
	int	hashCode	()		java.lang.Object
	boolean	isEnabled	()		
	void	notify	()		java.lang.Object
	void	notifyAll	()		java.lang.Object

## See Also

methods, import, class, javaArray

**Purpose**

Compile MEX-function from C/ C++ or Fortran source code

**Syntax**

```
mex -help
mex -setup
mex filenames
mex options filenames
```

**Description**

`mex -help` displays the M-file help for `mex`.

`mex -setup` lets you select or change the compiler configuration. For more information, see “Building MEX-Files”.

`mex filenames` compiles and links one or more C/C++ or Fortran source files specified in `filenames` into a shared library called a binary MEX-file from MATLAB software.

`mex options filenames` compiles and links one or more source files specified in `filenames` using one or more of the specified command-line options.

The MEX-file has a platform-dependent extension. Use the `mexext` function to return the extension for the current machine or for all supported platforms.

`filenames` can be any combination of source files, object files, and library files. Include both the file name and the file extension in `filenames`. A non-source-code `filenames` parameter is passed to the linker without being compiled.

All valid command-line options are shown in the MEX Script Switches on page 2-2288 table. These options are available on all platforms except where noted.

`mex` also can build executable files for stand alone MATLAB engine and MAT-file applications. For more information, see “Engine/MAT Stand Alone Application Details” on page 2-2293.

You can run `mex` from the MATLAB Command Prompt, the Microsoft Windows Command Prompt, or the UNIX<sup>19</sup> shell. `mex` is a script named `mex.bat` on Windows systems and `mex` on UNIX systems. It is located in the `matlabroot/bin` directory.

The first file listed in `filenames` becomes the name of the binary MEX-file. You can list other source, object, or library files as additional `filenames` parameters to satisfy external references.

`mex` uses an options file to specify variables and values that are passed as arguments to the compiler, linker, and other tools (e.g., the resource linker on Windows systems). For more information, see “Options File Details” on page 2-2292. The default name for the options file is `mexopts.bat` (Windows systems) or `mexopts.sh` (UNIX systems).

The `setup` option causes `mex` to search for installed compilers and allows you to choose an options file as the default for future invocations of `mex`.

Command-line options to `mex` may supplement or override contents of the options file. For more information, see “Override Option Details” on page 2-2292.

For an up-to-date list of supported compilers, see Technical Note 1601.

## MEX Script Switches

Switch	Function
<code>@rsp_file</code>	(Windows systems only) Include the contents of the text file <code>rsp_file</code> as command-line arguments to <code>mex</code> .

19. UNIX is a registered trademark of The Open Group in the United States and other countries.

**MEX Script Switches (Continued)**

Switch	Function
<code>-arch</code>	Build an output file for architecture <i>arch</i> . To determine the value for <i>arch</i> , type <code>computer('arch')</code> at the MATLAB Command Prompt on the target machine. Valid values for <i>arch</i> depend on the architecture of the build platform.
<code>-ada sfcn.ads</code>	Use this option to compile a Simulink S-function written in Ada, where <i>sfcn.ads</i> is the Package Specification for the S-function. When this option is specified, only the <code>-v</code> (verbose) and <code>-g</code> (debug) options are relevant. All other options are ignored. For examples and information on supported compilers and other requirements, see <code>README</code> in the <code>simulink/ada/examples</code> directory.
<code>-argcheck</code>	(C functions only) Add argument checking. This adds code so arguments passed incorrectly to MATLAB API functions cause assertion failures.
<code>-c</code>	Compile only. Creates an object file, but not a binary MEX-file.
<code>-compatibleArrayDims</code>	Build a binary MEX-file using the MATLAB Version 7.2 array-handling API, which limits arrays to $2^{31}-1$ elements. This option is the default, but in the future the <code>-largeArrayDims</code> option will be the default.

**MEX Script Switches (Continued)**

<b>Switch</b>	<b>Function</b>
-cxx	(UNIX systems only) Use the C++ linker to link the MEX-file if the first source file is in C and there are one or more C++ source or object files. This option overrides the assumption that the first source file in the list determines which linker to use.
-Dname	Define a symbol name to the C preprocessor. Equivalent to a <code>#define name</code> directive in the source.
-Dname=value	Define a symbol name and value to the C preprocessor. Equivalent to a <code>#define name value</code> directive in the source.
-f optionsfile	Specify location and name of options file to use. Overrides the mex default-options-file search mechanism.
-fortran	(UNIX systems only) Specify that the gateway routine is in Fortran. This option overrides the assumption that the first source file in the list determines which linker to use.
-g	Create a binary MEX-file containing additional symbolic information for use in debugging. This option disables the mex default behavior of optimizing built object code (see the -O option).
-h[elp]	Print help for mex.
-Ipathname	Add <i>pathname</i> to the list of directories to search for <code>#include</code> files.



**MEX Script Switches (Continued)**

Switch	Function
-inline	Inline matrix accessor functions (mx*). The generated MEX-function may not be compatible with future versions of MATLAB.
-lname	Link with object library. On Windows systems, <i>name</i> expands to <i>name.lib</i> or <i>libname.lib</i> and on UNIX systems, to <i>libname.so</i> or <i>libname.dylib</i> .
-Ldirectory	Add <i>directory</i> to the list of directories to search for libraries specified with the -l option. On UNIX systems, you must also set the run-time library path, as explained in “Setting Run-Time Library Path”.
-largeArrayDims	Build a binary MEX-file using the MATLAB large-array-handling API. This API can handle arrays with more than $2^{31}-1$ elements when compiled on 64-bit platforms. (See also the -compatibleArrayDims option.)
-n	No execute mode. Print any commands that <i>mex</i> would otherwise have executed, but do not actually execute any of them.
-O	Optimize the object code. Optimization is enabled by default and by including this option on the command line. If the -g option appears without the -O option, optimization is disabled.
-outdir <i>dirname</i>	Place all output files in directory <i>dirname</i> .
-output <i>resultname</i>	Create binary MEX-file named <i>resultname</i> . Automatically appends the appropriate MEX-file extension. Overrides the default MEX-file naming mechanism.

**MEX Script Switches (Continued)**

<b>Switch</b>	<b>Function</b>
-setup	Specify the compiler options file to use when calling the <code>mex</code> function. When you use this option, all other command-line options are ignored.
-U <i>name</i>	Remove any initial definition of the C preprocessor symbol <i>name</i> . (Inverse of the -D option.)
-v	Verbose mode. Print the values for important internal variables after the options file is processed and all command-line arguments are considered. Prints each compile step and final link step fully evaluated.
<i>name=value</i>	Override an options file variable for variable <i>name</i> . For examples, see Override Option Details in the Remarks section of the <code>mex</code> reference page.

**Remarks****Options File Details**

MATLAB provides template options files for the compilers that are supported by `mex`. These templates are located in the `matlabroot\bin\win32\mexopts` or the `matlabroot\bin\win64\mexopts` directories on Windows systems, or the `matlabroot/bin` directory on UNIX systems. These template options files are used by the `-setup` option to define the selected default options file.

**Override Option Details**

Use the `name=value` command-line argument to override a variable specified in the options file at the command line. When using this option, you may need to use the shell's quoting syntax to protect characters such as spaces, which have a meaning in the shell syntax.

This option is processed after the options file is processed and all command line arguments are considered.

On Windows platforms, at either the MATLAB prompt or the DOS prompt, use double quotes ("). For example:

```
mex -v COMPFLAGS="$COMPFLAGS -Wall" LINKFLAGS="$LINKFLAGS /VERBOSE"
```

At the MATLAB command line on UNIX platforms, use double quotes ("). Use the backslash (\) escape character before the dollar sign (\$). For example:

```
mex -v CFLAGS="\$CFLAGS -Wall" LDFLAGS="\$LDFLAGS-w" yprime.c
```

At the shell command line on UNIX platforms, use single quotes ('). For example:

```
mex -v CFLAGS='$CFLAGS -Wall' LDFLAGS='$LDFLAGS -w' yprime.c
```

## Engine/MAT Stand Alone Application Details

mex can build executable files for stand alone MATLAB engine and MAT-file applications. For these applications, mex does not use the default options file; you must use the -f option to specify an options file.

The options files used to generate stand alone MATLAB engine and MAT-file executables are named \*engmatopts.bat on Windows systems, or engopts.sh and matopts.sh on UNIX systems, and are located in the same directory as the template options files referred to above in Options File Details.

## Examples

### Compiling a C File

The following command compiles yprime.c:

```
mex yprime.c
```

### Using Verbose Mode

When debugging, it is often useful to use verbose mode, as well as include symbolic debugging information:

```
mex -v -g yprime.c
```

## **Overriding Command Line Options**

For examples, see “Override Option Details” on page 2-2292.

## **See Also**

computer, dbmex, inmem, loadlibrary, mexext, pcode, prefdir, system

**Purpose** Get compiler configuration information for building MEX-files

**Syntax**

```
cc = mex.getCompilerConfigurations()  
cc = mex.getCompilerConfigurations('lang')  
cc = mex.getCompilerConfigurations('lang','list')
```

**Description** `cc = mex.getCompilerConfigurations()` returns a `mex.CompilerConfiguration` object `cc` containing information about the selected compiler configuration used by `mex`. The selected compiler is the one you choose when you run the `mex -setup` command. For details about the `mex.CompilerConfiguration` class, see “`mex.CompilerConfiguration`” on page 2-2296.

`cc = mex.getCompilerConfigurations('lang')` returns an array of `mex.CompilerConfiguration` objects `cc` containing information about the selected configuration for the given `lang`. If the language of the selected compiler is different from `lang`, then `cc` is empty.

Language `lang` is a string with one of the following values:

- **Any** — All supported languages. This is the default value.
- **C** — All C compiler configurations, including C++ configurations.
- **C++** or **CPP** — All C++ compiler configurations.
- **Fortran** — All Fortran compiler configurations.

`cc = mex.getCompilerConfigurations('lang','list')` returns an array of `mex.CompilerConfiguration` objects `cc` containing information about configurations for the given language and the given `list`. Values for `list` are:

- **Selected** — The compiler you choose when you run `mex -setup`. This is the default value.
- **Installed** — All supported compilers `mex` finds installed on your system.

# mex.getCompilerConfigurations

---

- **Supported** — All compilers supported in the current release. For an up-to-date list of supported compilers, see [Technical Note 1601](#).

## Classes

### mex.CompilerConfiguration

The `mex.CompilerConfiguration` class contains the following read-only properties about compiler configurations.

Property	Purpose
Name	Name of the compiler
Manufacturer	Name of the manufacturer of the compiler
Language	Compiler language
Version	(Windows platforms only) Version of the compiler
Location	(Windows platforms only) Directory where compiler is installed
Details	A <code>mex.CompilerConfigurationDetails</code> object containing specific information about build options. For details about this class, see “ <code>mex.CompilerConfigurationDetails</code> ” on page 2-2296.

### mex.CompilerConfigurationDetails

The `mex.CompilerConfigurationDetails` class provides information about the command options used by the compiler, linker and other build programs used to create MEX-files. These properties are read-only.

Property
CompilerExecutable
CompilerFlags
OptimizationFlags
DebugFlags

Property
LinkerExecutable
LinkerFlags
LinkerOptimizationFlags
LinkerDebugFlags

## Examples

### Selected Compiler Example

```
myCompiler = mex.getCompilerConfigurations()
```

MATLAB software displays information similar to the following (depending on your architecture, your version of MATLAB, and what you selected when you ran `mex -setup`):

```
myCompiler =  
  
mex.CompilerConfiguration  
Package: mex  
  
Properties:  
    Name: 'Microsoft Visual C++ 2005'  
    Manufacturer: 'Microsoft'  
    Language: 'C++'  
    Version: '8.0'  
    Location: '%VS80COMNTOOLS%\\..\\..'  
    Details: [1x1 mex.CompilerConfigurationDetails]  
  
Methods
```

### Supported Compiler Configurations Example

```
allCC = mex.getCompilerConfigurations('Any','Supported')
```

MATLAB displays information similar to the following:

```
allCC =
```

# mex.getCompilerConfigurations

---

```
1x11 mex.CompilerConfiguration
```

```
Package: mex
```

```
Properties:
```

```
  Name
```

```
  Manufacturer
```

```
  Language
```

```
  Version
```

```
  Location
```

```
  Details
```

```
Methods
```

This version of MATLAB supports eleven configurations, hence, `a11CC` is a 1-by-11 matrix.

## Supported C Compilers Example

To see what C compilers MATLAB supports, type:

```
cLanguageCC = mex.getCompilerConfigurations('C','Supported')
```

MATLAB displays the following information (the number of compilers for your version of MATLAB may be different):

```
cLanguageCC =
```

```
1x9 mex.CompilerConfiguration
```

```
Package: mex
```

```
Properties:
```

```
  Name
```

```
  Manufacturer
```

```
  Language
```

```
  Version
```

```
  Location
```



## Details

## Methods

To display the compiler names, type:

```
format compact
cLanguageCC.Name
```

MATLAB displays information similar to the following:

```
ans =
Intel C++
ans =
Lcc-win32
ans =
Microsoft Visual C++
ans =
Microsoft Visual C++ 2003
ans =
Microsoft Visual C++ 2005 Express Edition
ans =
Microsoft Visual C++ 2005
ans =
Microsoft Visual C++ 2008
ans =
Open Watcom C/C++
ans =
Open Watcom C/C++
```

## Example – Viewing Build Options for a C Compiler

To see what build options MATLAB uses with a particular C compiler, create an array `CC` of all supported C compiler configurations:

```
CC = mex.getCompilerConfigurations('C','Supported');
disp('Compiler Name')
for i = 1:3; disp(CC(i).Name); end;
```

# mex.getCompilerConfigurations

---

MATLAB displays a list similar to:

```
Intel C++  
Lcc-win32  
Microsoft Visual C++
```

To see the build options for the Microsoft Visual C++ compiler, type:

```
CC(3).Details
```

MATLAB displays information similar to the following (output is formatted):

```
ans =  
  mex.CompilerConfigurationDetails  
  Package: mex  
  
  Properties:  
    CompilerExecutable: 'cl'  
    CompilerFlags: '-c -Zp8 -G5 -W3 -EHS  
                  -DMATLAB_MEX_FILE -nologo /MD'  
    OptimizationFlags: '-O2 -Oy- -DNDEBUG'  
    DebugFlags: '-Zi  
                -Fd"%OUTDIR%%MEX_NAME%.pdb" '  
    LinkerExecutable: 'link'  
    LinkerFlags: [1x258 char]  
    LinkerOptimizationFlags: ''  
    LinkerDebugFlags: '/debug'  
  
  Methods
```

## See Also

mex

## Purpose

Construct MException object

## Syntax

```
ME = MException(msgID,a errmsg, v1, v2, ...)  
ME = MException(errmsg, v1, v2, ...)  
ME = MException(errmsg)
```

## Description

ME = MException(msgID,a errmsg, v1, v2, ...) constructs an object ME of class MException and assigns to that object a message identifier msgID and error message string errmsg. This object then provides properties and methods that you can use in generating or responding to errors in your program code.

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.

# MException

---

All string input arguments must be enclosed in single quotation marks. If `errmsg` is an empty string, the error command has no effect.

`ME = MException(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.

`ME = MException(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 `\t` in `'C:\testdir'` for example, as a backslash character followed by the letter `t`, and not as a horizontal tab.

There are two ways to generate an error in your MATLAB code. Although the latter method is more work, it can provide you with a more extensible system for reporting and handling errors:

- Call the MATLAB error function.
- Construct an `MException` object, store identifying information in the object, and use the `throw` or `throwAsCaller` methods of that object to generate the error.

## Properties

The `MException` object has four properties: `identifier`, `message`, `stack`, and `cause`.

Property	Description
<code>identifier</code>	Identifies the <code>MException</code> string.
<code>message</code>	Formatted error message that is displayed.
<code>stack</code>	Structure containing stack trace information such as M-file function name and line number where the <code>MException</code> was thrown.
<code>cause</code>	Cell array of <code>MException</code> that caused this exception to be created.

## Methods

Method	Description
addCause	Appends an MException to the cause field of another MException.
eq	Compares two MException objects for equality.
getReport	Returns a formatted message string based on the current exception that uses the same format as errors thrown by internal MATLAB code.
isequal	Compares two MException objects for equality.
last	Returns an MException object for the most recently thrown exception.
ne	Compares two MException objects for inequality.
rethrow	Reissues an exception that has been caught, causing the program to stop.
throw	Issues an exception from the currently running M-file.
throwAsCaller	Issues an exception from the currently running M-file, also omitting the current stack frame from the stack field of the MException.

## Remarks

When MATLAB encounters an error in its internal code or in your own program code, it *throws an exception*. In this exception process, MATLAB:

- Interrupts the program at the point of the error.
- Constructs an object of the MException class.

# MException

---

- Records information about the error in that object.
- Displays this information at the user's terminal.
- Aborts the program.

If your program code implements a try-catch mechanism to intercept the error before MATLAB aborts the program, you can obtain access to the MException object that MATLAB associates with this error instance via the catch statement, and then handle the condition based on the records you can retrieve from the object.

## Examples

### Example 1 – Formatted Messages

If your message string requires formatting specifications like those available with the `sprintf` function, you can use this syntax for the MException constructor:

```
ME = MException(identifier, formatstring, arg1, arg2, ...)
```

For example,

```
S = 'Accounts'; f1 = 'ClientName';  
ME = MException('AcctError:Incomplete', ...  
    'Field '%s.%s' is not defined.', S, f1);
```

```
ME.message  
ans =  
    Field 'Accounts.ClientName' is not defined.
```

### Example 2 – Error Recovery

This example reads the contents of an image file. The attempt to open and then read the file is done in a try block. If either the open or read fails, the program catches the resulting exception and saves the MException object in the variable ME1.

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 file name 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
        case '.jpg' % Change jpg to jpeg
            filename = strrep(filename, '.jpg', '.jpeg')
        case '.jpeg' % Change jpeg to jpg
            filename = strrep(filename, '.jpeg', '.jpg')
        case '.tif' % Change tif to tiff
            filename = strrep(filename, '.tif', '.tiff')
        case '.tiff' % Change tiff to tif
            filename = strrep(filename, '.tiff', '.tif')
        otherwise
            fprintf('File %s not found\n', filename);
            rethrow(ME1);
        end

        % Try again, with modified filenames.
        try
            fid = fopen(filename, 'r');
```

# MException

---

```
        d_in = fread(fid);
    catch ME2
        fprintf('Unable to access file %s\n', filename);
        ME2 = addCause(ME2, ME1);
        rethrow(ME2)
    end
end
end
end
```

## Example 3 – Nested try-catch

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

```
function data = read_it(filename);
try
    fid = fopen(filename, 'r');
    data = fread(fid);
catch ME1
    if strcmp(ME1.identifier, 'MATLAB:FileIO:InvalidFid')
        msg = sprintf('\n%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);
        try
            fid = fopen(filename, 'r');
            data = fread(fid);
        catch ME2
            ME3 = addCause(ME2, ME1)
            throw(ME3);
        end
    end
end
```



```
        rmpath(newdir);
    end
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.

## See Also

```
throw(MException), rethrow(MException),
throwAsCaller(MException), addCause(MException),
getReport(MException), disp(MException), isequal(MException),
eq(MException), ne(MException), last(MException), error, try,
catch
```

# mexext

---

**Purpose** Binary MEX-file name extension

**Syntax**  
`ext = mexext`  
`extlist = mexext('all')`

**Description** `ext = mexext` returns the file name extension for the current platform.  
`extlist = mexext('all')` returns a struct with fields `arch` and `ext` describing MEX-file name extensions for the all platforms.

**Remarks** For a table of file extensions, see “Using Binary MEX-Files”.

**Examples** Find the MEX-file extension for the system you are currently working on:

```
ext = mexext
```

```
ext =  
    mexw32
```

Find the MEX-file extension for an Apple Macintosh system:

```
extlist = mexext('all');  
  
for k=1:length(extlist)  
    if strcmp(extlist(k).arch, 'maci')  
        disp(sprintf('Arch: %s      Ext: %s', ...  
                    extlist(k).arch, extlist(k).ext))  
    end, end
```

```
Arch: maci      Ext: mexmaci
```

**See Also** `mex`

<b>Purpose</b>	Name of currently running M-file
<b>Syntax</b>	<pre>mfilename p = mfilename('fullpath') c = mfilename('class')</pre>
<b>Description</b>	<p>mfilename returns a string containing the name of the most recently invoked M-file. When called from within an M-file, it returns the name of that M-file, allowing an M-file to determine its name, even if the filename has been changed.</p> <p>p = mfilename('fullpath') returns the full path and name of the M-file in which the call occurs, not including the filename extension.</p> <p>c = mfilename('class') in a method, returns the class of the method, not including the leading @ sign. If called from a nonmethod, it yields the empty string.</p>
<b>Remarks</b>	<p>If mfilename is called with any argument other than the above two, it behaves as if it were called with no argument.</p> <p>When called from the command line, mfilename returns an empty string.</p> <p>To get the names of the callers of an M-file, use dbstack with an output argument.</p>
<b>See Also</b>	dbstack, function, nargin, nargout, inputname

# mget

---

**Purpose** Download file from FTP server

**Syntax**

```
mget(f, 'filename')
mget(f, 'dirname')
mget(..., 'target')
```

**Description**

`mget(f, 'filename')` retrieves `filename` from the FTP server `f` into the MATLAB current directory, where `f` was created using `ftp`.

`mget(f, 'dirname')` retrieves the directory `dirname` and its contents from the FTP server `f` into the MATLAB current directory, where `f` was created using `ftp`. You can use a wildcard (\*) in `dirname`.

`mget(..., 'target')` retrieves the specified items from the FTP server `f`, where `f` was created using `ftp`, into the local directory specified by `target`, where `target` is an absolute path name.

**Examples**

Connect to an FTP server, change to the `documents/rfc` directory, and retrieve the file `rfc0959.txt` into the current MATLAB directory.

```
ftpobj = ftp('nic.merit.edu');
cd(ftpobj, 'documents/rfc');

mget(ftpobj, 'rfc0959.txt')
ans =
    'C:\work\rfc0959.txt'
```

**See Also** `cd (ftp)`, `ftp`, `mput`

---

<b>Purpose</b>	Smallest elements in array
<b>Syntax</b>	<pre>C = min(A) C = min(A,B) C = min(A,[],dim) [C,I] = min(...)</pre>
<b>Description</b>	<p><code>C = min(A)</code> returns the smallest elements along different dimensions of an array.</p> <p>If <code>A</code> is a vector, <code>min(A)</code> returns the smallest element in <code>A</code>.</p> <p>If <code>A</code> is a matrix, <code>min(A)</code> treats the columns of <code>A</code> as vectors, returning a row vector containing the minimum element from each column.</p> <p>If <code>A</code> is a multidimensional array, <code>min</code> operates along the first nonsingleton dimension.</p> <p><code>C = min(A,B)</code> returns an array the same size as <code>A</code> and <code>B</code> with the smallest elements taken from <code>A</code> or <code>B</code>. The dimensions of <code>A</code> and <code>B</code> must match, or they may be scalar.</p> <p><code>C = min(A,[],dim)</code> returns the smallest elements along the dimension of <code>A</code> specified by scalar <code>dim</code>. For example, <code>min(A,[],1)</code> produces the minimum values along the first dimension (the rows) of <code>A</code>.</p> <p><code>[C,I] = min(...)</code> finds the indices of the minimum values of <code>A</code>, and returns them in output vector <code>I</code>. If there are several identical minimum values, the index of the first one found is returned.</p>
<b>Remarks</b>	<p>For complex input <code>A</code>, <code>min</code> returns the complex number with the smallest complex modulus (magnitude), computed with <code>min(abs(A))</code>. Then computes the smallest phase angle with <code>min(angle(x))</code>, if necessary.</p> <p>The <code>min</code> function ignores NaNs.</p>
<b>See Also</b>	<code>max</code> , <code>mean</code> , <code>median</code> , <code>sort</code>

# min (timeseries)

---

**Purpose** Minimum value of timeseries data

**Syntax**  
`ts_min = min(ts)`  
`ts_min = min(ts, 'PropertyName1', PropertyValue1, ...)`

**Description** `ts_min = min(ts)` returns the minimum value in the time-series data. When `ts.Data` is a vector, `ts_min` is the minimum value of `ts.Data` values. When `ts.Data` is a matrix, `ts_min` is a row vector containing the minimum value of each column of `ts.Data` (when `IsTimeFirst` is true and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `min` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_min = min(ts, 'PropertyName1', PropertyValue1, ...)` specifies the following optional input arguments:

- 'MissingData' property has two possible values, 'remove' (default) or 'interpolate', indicating how to treat missing data during the calculation.
- 'Quality' values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- 'Weighting' property has two possible values, 'none' (default) or 'time'.  
When you specify 'time', larger time values correspond to larger weights.

**Examples** The following example illustrates how to find the minimum values in multivariate time-series data.

**1** Load a 24-by-3 data array.

```
load count.dat
```

**2** Create a timeseries object with 24 time values.

```
count_ts = timeseries(count,[1:24], 'Name', 'CountPerSecond')
```

**3** Find the minimum in each data column for this `timeseries` object.

```
min(count_ts)
```

```
ans =
```

```
7     9     7
```

The minimum is found independently for each data column in the `timeseries` object.

### See Also

```
iqr (timeseries), max (timeseries), median (timeseries), mean  
(timeseries), std (timeseries), timeseries, var (timeseries)
```

# MinimizeCommandWindow

---

**Purpose** Minimize size of server window

**Syntax** **MATLAB Client**  
h.MinimizeCommandWindow  
MinimizeCommandWindow(h)  
invoke(h, 'MinimizeCommandWindow')

**Method Signature**  
HRESULT MinimizeCommandWindow(void)

**Microsoft Visual Basic Client**  
MinimizeCommandWindow

**Description** MinimizeCommandWindow minimizes the window for the server attached to handle h, and makes it inactive. If the server window was already in a minimized state to begin with, then MinimizeCommandWindow does nothing.

**Remarks** Server function names, like MinimizeCommandWindow, are case sensitive when using the first syntax shown.  
  
There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

**Examples** Create a COM server and minimize its window. Then maximize the window and make it the currently active window.

**MATLAB Client**

```
h = actxserver('matlab.application');  
h.MinimizeCommandWindow;  
% Now return the server window to its former state on  
% the desktop and make it the currently active window.  
h.MaximizeCommandWindow;
```



## Visual Basic .NET Client

Create a COM server and minimize its window.

```
Dim Matlab As Object

Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow

'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

## See Also

MaximizeCommandWindow

# minres

---

## Purpose

Minimum residual method

## Syntax

```
x = minres(A,b)
minres(A,b,tol)
minres(A,b,tol,maxit)
minres(A,b,tol,maxit,M)
minres(A,b,tol,maxit,M1,M2)
minres(A,b,tol,maxit,M1,M2,x0)
[x,flag] = minres(A,b,...)
[x,flag,relres] = minres(A,b,...)
[x,flag,relres,iter] = minres(A,b,...)
[x,flag,relres,iter,resvec] = minres(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...)
```

## Description

`x = minres(A,b)` attempts to find a minimum norm residual solution  $x$  to the system of linear equations  $A*x=b$ . The  $n$ -by- $n$  coefficient matrix  $A$  must be symmetric but need not be positive definite. It 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 `minres` converges, a message to that effect is displayed. If `minres` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual  $\text{norm}(b-A*x)/\text{norm}(b)$  and the iteration number at which the method stopped or failed.

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

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

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

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

`[x,flag] = minres(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	minres converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.
1	minres iterated <code>maxit</code> times but did not converge.
2	Preconditioner $M$ was ill-conditioned.
3	minres stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during <code>minres</code> became too small or too large to continue computing.

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

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

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

`[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...)` also returns a vector of estimates of the Conjugate Gradients residual norms at each iteration.

## Examples

### Example 1

```
n = 100; on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],[-1:1,n,n]);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M1 = spdiags(4*on,0,n,n);

x = minres(A,b,tol,maxit,M1);
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

### 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_minres` that

- Calls `minres` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_minres` are available to `afun`.

The following shows the code for `run_minres`:

```
function x1 = run_minres
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],[-1:1,n,n]);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M = spdiags(4*on,0,n,n);
x1 = minres(@afun,b,tol,maxit,M);
```

```

function y = afun(x)
    y = 4 * x;
    y(2:n) = y(2:n) - 2 * x(1:n-1);
    y(1:n-1) = y(1:n-1) - 2 * x(2:n);
end
end

```

When you enter

```
x1=run_minres;
```

MATLAB software displays the message

```

minres converged at iteration 49 to a solution with relative
residual 4.7e-014

```

### Example 3

Use a symmetric indefinite matrix that fails with `pcg`.

```

A = diag([20:-1:1, -1:-1:-20]);
b = sum(A,2);           % The true solution is the vector of all ones.
x = pcg(A,b);          % Errors out at the first iteration.

```

displays the following message:

```

pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1

```

However, `minres` can handle the indefinite matrix `A`.

```

x = minres(A,b,1e-6,40);
minres converged at iteration 39 to a solution with relative
residual 1.3e-007

```

### See Also

`bicg`, `bicgstab`, `cgs`, `cholinc`, `gmres`, `lsqr`, `pcg`, `qmr`, `symmlq`

function\_handle (@), mldivide (\)

## References

- [1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.
- [2] Paige, C. C. and M. A. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

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

**Syntax** `mislocked`  
`mislocked(fun)`

**Description** `mislocked` by itself returns logical 1 (true) if the currently running M-file or MEX-file is locked, and logical 0 (false) otherwise.

`mislocked(fun)` returns logical 1 (true) if the function named *fun* is locked in memory, and logical 0 (false) otherwise. Locked M-files and MEX-files cannot be removed with the `clear` function.

**See Also** `mlock`, `munlock`

# mkdir

---

## Purpose

Make new directory

## Graphical Interface

As an alternative to `mkdir`, you can use the Current Directory browser to add a directory.

## Syntax

```
mkdir('dirname')  
mkdir('parentdir','dirname')  
status = mkdir(...,'dirname')  
[status,message,messageid] = mkdir(...,'dirname')
```

## Description

`mkdir('dirname')` creates the directory `dirname` in the current directory, if `dirname` represents a relative path. Otherwise, `dirname` represents an absolute path and `mkdir` attempts to create the absolute directory `dirname` in the root of the current volume. An absolute path starts with any one of the following: a drive letter on the Microsoft Windows platform, a UNC path '\\ ' string, or a '/' character on UNIX<sup>20</sup> platforms.

`mkdir('parentdir','dirname')` creates the directory `dirname` in the existing directory `parentdir`, where `parentdir` is an absolute or relative path. If `parentdir` does not exist, the MATLAB software attempts to create it. See the Remarks section below.

`status = mkdir(...,'dirname')` creates the specified directory and returns a `status` of logical 1 if the operation is successful. It returns logical 0 if the operation is unsuccessful.

`[status,message,messageid] = mkdir(...,'dirname')` creates the specified directory, and returns `status`, message string, and MATLAB error message ID. The value given to `status` is logical 1 for success and logical 0 for error.

See the help for `error` and `lasterror` for more information.)

20. UNIX is a registered trademark of The Open Group in the United States and other countries.



**Remarks**

If an argument specifies a path that includes one or more nonexistent directories, MATLAB attempts to create the nonexistent directory. For example, for

```
mkdir('mydir\mdir1\mdir2\targetdir')
```

if `mdir1` does not exist, MATLAB creates `mdir1`, creates `mdir2` within `mdir1`, and creates `targetdir` within `mdir2`.

**Examples**

**Creating a Subdirectory in the Current Directory**

To create a subdirectory in the current directory called `newdir`, type

```
mkdir('newdir')
```

**Creating a Subdirectory in the Specified Parent Directory**

To create a subdirectory called `newdir` in the directory `testdata`, which is at the same level as the current directory, type

```
mkdir('../testdata','newdir')
```

**Returning Status When Creating a Directory**

In this example, the first attempt to create `newdir` succeeds, returning a status of 1, and no error or warning message or message identifier:

```
[s, mess, messid] = mkdir('../testdata', 'newdir')
s =
    1
mess =
    ''
messid =
    ''
```

If you attempt to create the same directory again, `mkdir` again returns a success status, and also a warning and message identifier informing you that the directory already existed:

```
[s,mess,messid] = mkdir('../testdata','newdir')
s =
```

# mkdir

---

```
1
mess =
    Directory "newdir" already exists.
messid =
    MATLAB:MKDIR:DirectoryExists
```

## See Also

copyfile, cd, dir, fileattrib, filebrowser, fileparts, ls, mfilename, movefile, rmdir

“Managing Files and Working with the Current Directory”

**Purpose** Create new directory on FTP server

**Syntax** `mkdir(f, 'dirname')`

**Description** `mkdir(f, 'dirname')` creates the directory `dirname` in the current directory of the FTP server `f`, where `f` was created using `ftp`, and where `dirname` is a path name relative to the current directory on `f`.

**Examples** Connect to server `testsite`, view the contents, and create the directory `newdir` in the directory `testdir`.

```
test=ftp('ftp.testsite.com')
dir(test)
.          ..          otherfile.m          testdir
mkdir(test, 'testdir/newdir');
dir(test, 'testdir')
.          ..          newdir
```

**See Also** `dir (ftp)`, `ftp`, `rmdir (ftp)`

**Purpose** Make piecewise polynomial

**Syntax**  
`pp = mkpp(breaks,coefs)`  
`pp = mkpp(breaks,coefs,d)`

**Description** `pp = mkpp(breaks,coefs)` builds a piecewise polynomial `pp` from its `breaks` and coefficients. `breaks` is a vector of length `L+1` with strictly increasing elements which represent the start and end of each of `L` intervals. `coefs` is an `L-by-k` matrix with each row `coefs(i,:)` containing the coefficients of the terms, from highest to lowest exponent, of the order `k` polynomial on the interval `[breaks(i),breaks(i+1)]`.

`pp = mkpp(breaks,coefs,d)` indicates that the piecewise polynomial `pp` is `d`-vector valued, i.e., the value of each of its coefficients is a vector of length `d`. `breaks` is an increasing vector of length `L+1`. `coefs` is a `d-by-L-by-k` array with `coefs(r,i,:)` containing the `k` coefficients of the `i`th polynomial piece of the `r`th component of the piecewise polynomial.

Use `ppval` to evaluate the piecewise polynomial at specific points. Use `unmkpp` to extract details of the piecewise polynomial.

**Note.** The *order* of a polynomial tells you the number of coefficients used in its description. A *k*th order polynomial has the form

$$c_1x^{k-1} + c_2x^{k-2} + \dots + c_{k-1}x + c_k$$

It has *k* coefficients, some of which can be 0, and maximum exponent *k*-1. So the order of a polynomial is usually one greater than its degree. For example, a cubic polynomial is of order 4.

**Examples** The first plot shows the quadratic polynomial

$$1 - \left(\frac{x}{2} - 1\right)^2 = \frac{-x^2}{4} + x$$

shifted to the interval [-8,-4]. The second plot shows its negative

$$\left(\frac{x}{2} - 1\right)^2 - 1 = \frac{x^2}{4} - x$$

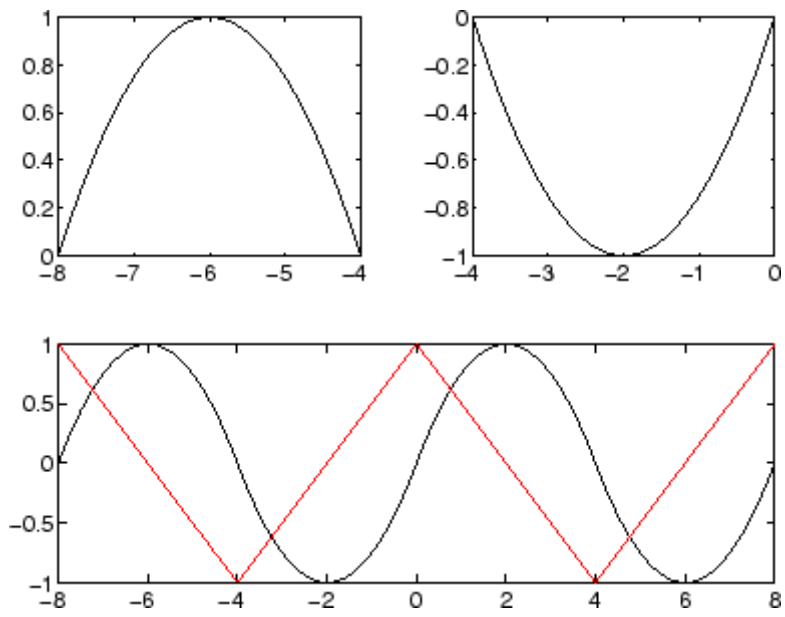
but shifted to the interval [-4,0].

The last plot shows a piecewise polynomial constructed by alternating these two quadratic pieces over four intervals. It also shows its first derivative, which was constructed after breaking the piecewise polynomial apart using `unmkpp`.

```
subplot(2,2,1)
cc = [-1/4 1 0];
pp1 = mkpp([-8 -4],cc);
xx1 = -8:0.1:-4;
plot(xx1,ppval(pp1,xx1),'k-')

subplot(2,2,2)
pp2 = mkpp([-4 0],-cc);
xx2 = -4:0.1:0;
plot(xx2,ppval(pp2,xx2),'k-')

subplot(2,1,2)
pp = mkpp([-8 -4 0 4 8],[cc;-cc;cc;-cc]);
xx = -8:0.1:8;
plot(xx,ppval(pp,xx),'k-')
[breaks,coefs,l,k,d] = unmkpp(pp);
dpp = mkpp(breaks,repmat(k-1:-1:1,d*1,1).*coefs(:,1:k-1),d);
hold on, plot(xx,ppval(dpp,xx),'r-'), hold off
```



## See Also

`ppval`, `spline`, `unmkpp`

**Purpose** Left or right matrix division

**Syntax**

mldivide(A,B)	A\B
mrdivide(B,A)	B/A

**Description** mldivide(A,B) and the equivalent A\B perform matrix left division (back slash). A and B must be matrices that have the same number of rows, unless A is a scalar, in which case A\B performs element-wise division — that is, A\B = A.\B.

If A is a square matrix, A\B is roughly the same as inv(A)\*B, except it is computed in a different way. If A is an n-by-n matrix and B is a column vector with n elements, or a matrix with several such columns, then X = A\B is the solution to the equation AX = B computed by Gaussian elimination with partial pivoting (see “Algorithm” on page 2-2333 for details). A warning message is displayed if A is badly scaled or nearly singular.

If A is an m-by-n matrix with m ≈ n and B is a column vector with m components, or a matrix with several such columns, then X = A\B is the solution in the least squares sense to the under- or overdetermined system of equations AX = B. In other words, X minimizes norm(A\*X - B), the length of the vector AX - B. The rank k of A is determined from the QR decomposition with column pivoting (see “Algorithm” on page 2-2333 for details). The computed solution X has at most k nonzero elements per column. If k < n, this is usually not the same solution as x = pinv(A)\*B, which returns a least squares solution.

mrdivide(B,A) and the equivalent B/A perform matrix right division (forward slash). B and A must have the same number of columns.

If A is a square matrix, B/A is roughly the same as B\*inv(A). If A is an n-by-n matrix and B is a row vector with n elements, or a matrix with several such rows, then X = B/A is the solution to the equation XA = B computed by Gaussian elimination with partial pivoting. A warning message is displayed if A is badly scaled or nearly singular.

If B is an m-by-n matrix with m ≈ n and A is a column vector with m components, or a matrix with several such columns, then X = B/A is

# mldivide \, mrdivide /

---

the solution in the least squares sense to the under- or overdetermined system of equations  $XA = B$ .

---

**Note** Matrix right division and matrix left division are related by the equation  $B/A = (A' \backslash B')'$ .

---

## Least Squares Solutions

If the equation  $Ax = b$  does not have a solution (and  $A$  is not a square matrix),  $x = A \backslash b$  returns a *least squares solution* — in other words, a solution that minimizes the length of the vector  $Ax - b$ , which is equal to  $\text{norm}(A*x - b)$ . See “Example 3” on page 2-2332 for an example of this.

## Examples

### Example 1

Suppose that  $A$  and  $b$  are the following.

```
A = magic(3)
```

```
A =
```

```
     8     1     6
     3     5     7
     4     9     2
```

```
b = [1;2;3]
```

```
b =
```

```
     1
     2
     3
```

To solve the matrix equation  $Ax = b$ , enter

```
x=A\b
```



```
x =  
  
    0.0500  
    0.3000  
    0.0500
```

You can verify that  $x$  is the solution to the equation as follows.

```
A*x  
  
ans =  
  
    1.0000  
    2.0000  
    3.0000
```

## Example 2 – A Singular

If  $A$  is singular,  $A \setminus b$  returns the following warning.

```
Warning: Matrix is singular to working precision.
```

In this case,  $Ax = b$  might not have a solution. For example,

```
A = magic(5);  
A(:,1) = zeros(1,5); % Set column 1 of A to zeros  
b = [1;2;5;7;7];  
x = A \ b  
Warning: Matrix is singular to working precision.  
  
ans =  
  
    NaN  
    NaN  
    NaN  
    NaN  
    NaN
```

## mldivide \, mrdivide /

---

If you get this warning, you can still attempt to solve  $Ax = b$  using the pseudoinverse function `pinv`.

```
x = pinv(A)*b
```

```
x =
```

```
    0  
    0.0209  
    0.2717  
    0.0808  
   -0.0321
```

The result  $x$  is least squares solution to  $Ax = b$ . To determine whether  $x$  is an exact solution — that is, a solution for which  $Ax - b = 0$  — simply compute

```
A*x - b
```

```
ans =
```

```
   -0.0603  
    0.6246  
   -0.4320  
    0.0141  
    0.0415
```

The answer is not the zero vector, so  $x$  is not an exact solution.

“Pseudoinverses”, in the online MATLAB Mathematics documentation, provides more examples of solving linear systems using `pinv`.

### Example 3

Suppose that

```
A = [1 0 0; 1 0 0];  
b = [1; 2];
```

Note that  $Ax = b$  cannot have a solution, because  $A*x$  has equal entries for any  $x$ . Entering

```
x = A\b
```

returns the least squares solution

```
x =  
  
    1.5000  
         0  
         0
```

along with a warning that  $A$  is rank deficient. Note that  $x$  is not an exact solution:

```
A*x - b  
  
ans =  
  
    0.5000  
   -0.5000
```

## Data Type Support

When computing  $X = A \setminus B$  or  $X = A/B$ , the matrices  $A$  and  $B$  can have data type `double` or `single`. The following rules determine the data type of the result:

- If both  $A$  and  $B$  have type `double`,  $X$  has type `double`.
- If either  $A$  or  $B$  has type `single`,  $X$  has type `single`.

## Algorithm

The specific algorithm used for solving the simultaneous linear equations denoted by  $X = A \setminus B$  and  $X = B/A$  depends upon the structure of the coefficient matrix  $A$ . To determine the structure of  $A$  and select the appropriate algorithm, MATLAB software follows this precedence:

- 1 If  $A$  is **sparse and diagonal**,  $X$  is computed by dividing by the diagonal elements of  $A$ .

## mldivide \, mrdivide /

---

**2 If A is sparse, square, and banded**, then banded solvers are used. Band density is (# nonzeros in the band)/(# nonzeros in a full band). Band density = 1.0 if there are no zeros on any of the three diagonals.

- If A is real and tridiagonal, i.e., band density = 1.0, and B is real with only one column, X is computed quickly using Gaussian elimination without pivoting.
- If the tridiagonal solver detects a need for pivoting, or if A or B is not real, or if B has more than one column, but A is banded with band density greater than the spparms parameter 'bandden' (default = 0.5), then X is computed using the Linear Algebra Package (LAPACK) routines in the following table.

	<b>Real</b>	<b>Complex</b>
A and B double	DGBTRF, DGBTRS	ZGBTRF, ZGBTRS
A or B single	SGBTRF, SGBTRS	CGBTRF, CGBTRS

**3 If A is an upper or lower triangular matrix**, then X is computed quickly with a backsubstitution algorithm for upper triangular matrices, or a forward substitution algorithm for lower triangular matrices. The check for triangularity is done for full matrices by testing for zero elements and for sparse matrices by accessing the sparse data structure.

If A is a full matrix, computations are performed using the Basic Linear Algebra Subprograms (BLAS) routines in the following table.

	<b>Real</b>	<b>Complex</b>
A and B double	DTRSV, DTRSM	ZTRSV, ZTRSM
A or B single	STRSV, STRSM	CTRSV, CTRSM

**4 If A is a permutation of a triangular matrix**, then X is computed with a permuted backsubstitution algorithm.

**5** If **A is symmetric, or Hermitian, and has real positive diagonal elements**, then a Cholesky factorization is attempted (see chol). If A is found to be positive definite, the Cholesky factorization attempt is successful and requires less than half the time of a general factorization. Nonpositive definite matrices are usually detected almost immediately, so this check also requires little time.

If successful, the Cholesky factorization for full A is

$$A = R' * R$$

where R is upper triangular. The solution X is computed by solving two triangular systems,

$$X = R \setminus (R' \setminus B)$$

Computations are performed using the LAPACK routines in the following table.

	<b>Real</b>	<b>Complex</b>
A and B double	DLANSY, DPOTRF, DPOTRS, DPOCON	ZLANHE, ZPOTRF, ZPOTRS, ZPOCON
A or B single	SLANSY, SPOTRF, SPOTRS, SDPOCON	CLANHE, CPOTRF, CPOTRS, CPOCON

**6** If A is sparse, then MATLAB software uses CHOLMOD to compute X. The computations result in

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

where P is a permutation matrix generated by amd, and R is an upper triangular matrix. In this case,

$$X = P * (R \setminus (R' \setminus (P' * B)))$$

## mldivide \, mrdivide /

- 7** if A is not sparse but is symmetric, and the Cholesky factorization failed, then MATLAB solves the system using a symmetric, indefinite factorization. That is, MATLAB computes the factorization  $P' * A * P = L * D * L'$ , and computes the solution X by  $X = P * (L' \setminus (D \setminus (L \setminus (P * B))))$ . Computations are performed using the LAPACK routines in the following table:

	<b>Real</b>	<b>Complex</b>
A and B double	DLANSY, DSYTRF, DSYTRS, DSYCON	ZLANHE, ZHETRF, ZHETRS, ZHECON
A or B single	SLANSY, SSYTRF, SSYTRS, SSYCON	CLANHE, CHETRF, CHETRS, CHECON

- 8** If A is **Hessenberg**, but not sparse, it is reduced to an upper triangular matrix and that system is solved via substitution.
- 9** If A is **square** and does not satisfy criteria 1 through 6, then a general triangular factorization is computed by Gaussian elimination with partial pivoting (see lu). This results in

$$A = L * U$$

where L is a permutation of a lower triangular matrix and U is an upper triangular matrix. Then X is computed by solving two permuted triangular systems.

$$X = U \setminus (L \setminus B)$$

If A is not sparse, computations are performed using the LAPACK routines in the following table.

	<b>Real</b>	<b>Complex</b>
A and B double	DLANGE, DGESV, DGECON	ZLANGE, ZGESV, ZGECON
A or B single	SLANGE, SGESV, SGECON	CLANGE, CGESV, CGECON

If  $A$  is sparse, then UMFPACK is used to compute  $X$ . The computations result in

$$P*(R\A)*Q = L*U$$

where

- $P$  is a row permutation matrix
- $R$  is a diagonal matrix that scales the rows of  $A$
- $Q$  is a column reordering matrix.

Then  $X = Q*(U\L\ (P*(R\B)))$ .

---

**Note** The factorization  $P*(R\A)*Q = L*U$  differs from the factorization used by the function `lu`, which does not scale the rows of  $A$ .

---

**10 If  $A$  is not square**, then Householder reflections are used to compute an orthogonal-triangular factorization.

$$A*P = Q*R$$

where  $P$  is a permutation,  $Q$  is orthogonal and  $R$  is upper triangular (see `qr`). The least squares solution  $X$  is computed with

$$X = P*(R\ (Q'*B))$$

If  $A$  is sparse, MATLAB computes a least squares solution using the sparse `qr` factorization of  $A$ .

If  $A$  is full, MATLAB uses the LAPACK routines listed in the following table to compute these matrix factorizations.

# mldivide \, mrdivide /

---

	<b>Real</b>	<b>Complex</b>
A and B double	DGEQP3, DORMQR, DTRTRS	ZGEQP3, ZORMQR, ZTRTRS
A or B single	SGEQP3, SORMQR, STRTRS	CGEQP3, CORMQR, CTRTRS

---

**Note** To see information about choice of algorithm and storage allocation for sparse matrices, set the spparms parameter 'spumoni' = 1.

---

---

**Note** mldivide and mrdivide are not implemented for sparse matrices A that are complex but not square.

---

## See Also


Arithmetic Operators, linsolve, ldivide, rdivide



**Purpose**

Check M-files for possible problems

**GUI Alternatives**

From the Current Directory browser, click the **Actions** button , and then select **Reports > M-Lint Code Check Report**. See also the automatic “M-Lint Code Analyzer” in the Editor.

**Syntax**

```
mlint('filename')
mlint('filename', '-config=settings.txt')
mlint('filename', '-config=factory')
inform=mlint('filename', -struct )
msg=mlint('filename', -string )
[inform,filepath]=mlint('filename')
inform=mlint('filename', -id )
inform=mlint('filename', -fullpath )
inform=mlint('filename', -notok')
mlint('filename', -cyc )
mlint('filename', -eml )
%#eml
%#ok
```

**Description**

`mlint('filename')` displays M-Lint information about `filename`, where the information reports potential problems and opportunities for code improvement, referred to as suspicious constructs. The line number in the message is a hyperlink that opens the file in the Editor, scrolled to that line. If `filename` is a cell array, information is displayed for each file. For `mlint(F1,F2,F3,...)`, where each input is a character array, MATLAB software displays information about each input file name. You cannot combine cell arrays and character arrays of file names. Note that the exact text of the `mlint` messages is subject to some change between versions.

`mlint('filename', '-config=settings.txt')` overrides the default M-lint active settings file with the M-Lint settings that enable or suppress messages as indicated in the specified `settings.txt` file.

---

**Note** If used, you must specify the full path to the `settings.txt` file specified with the `-config` option.

---

For information about creating a `settings.txt` file, see the information on “Preferences for M-Lint”. If you specify an invalid file, `mlint` returns a message indicating that it cannot open or read the file you specified. In that case, `mlint` uses the factory default settings.

`mlint('filename', '-config=factory')` ignores all settings files and uses the factory default M-lint preference settings.

`inform=mlint('filename', '-struct')` returns the M-Lint information in a structure array whose length is the number of suspicious constructs found. The structure has the fields that follow.

Field	Description
<code>message</code>	Message describing the suspicious construct that M-Lint caught.
<code>line</code>	Vector of M-file line numbers to which the message refers.
<code>column</code>	Two-column array of M-file columns (column extents) to which the message applies. The first column of the array specifies the column in the Editor where the M-Lint message begins. The second column of the array specifies the column in the Editor where the M-Lint message ends. There is one row in the two-column array for each occurrence of an M-Lint message.

If you specify multiple file names as input, or if you specify a cell array as input, `inform` contains a cell array of structures.

`msg=mlint('filename', '-string')` returns the M-Lint information as a string to the variable `msg`. If you specify multiple file names as input,

or if you specify a cell array as input, `msg` contains a string where each file's information is separated by 10 equal sign characters (=), a space, the file name, a space, and 10 equal sign characters.

If you omit the **-struct** or **-string** argument and you specify an output argument, the default behavior is **-struct**. If you omit the argument and there are no output arguments, the default behavior is to display the information to the command line.

`[inform,filepath]=mlint('filename')` additionally returns `filepath`, the absolute paths to the file names, in the same order as you specified them.

`inform=mlint('filename', -id )` requests the message ID from M-Lint, where ID is a string of the form ABC... When returned to a structure, the output also has the `id` field, which is the ID associated with the message.

`inform=mlint('filename', -fullpath )` assumes that the input file names are absolute paths, so that M-Lint does not try to locate them.

`inform=mlint('filename', -notok)` runs `mlint` for all lines in `filename`, even those lines that end with the `mlint` suppression syntax, `##ok`.

`mlint('filename', -cyc )` displays the McCabe complexity (also referred to as cyclomatic complexity) of each function in the file. Higher McCabe complexity values indicate higher complexity, and there is some evidence to suggest that programs with higher complexity values are more likely to contain errors. Frequently, you can lower the complexity of a function by dividing it into smaller, simpler functions. In general, smaller complexity values indicate programs that are easier to understand and modify. Some people advocate splitting up programs that have a complexity rating over 10.

`mlint('filename', -eml )` enables Embedded MATLAB™ messages for display in the Command Window.

If you include `##eml` anywhere within an M-file, except within a comment, it causes `mlint` to behave as though you specified `eml` for

that file. For more information, see “Adding the Compilation Directive `%#eml`”. MATLAB comments can follow the `%#eml` directive.

If you include `%#ok` at the end of a line in an M-file, `mlint` ignores that line. `mlint` ignores specified messages `id1` through `idn` on a given line when `%#ok< id1,id2,...idn>` appears at the end of that line. `mlint` ignores specified messages 1 through `n` throughout the file when `%#ok<*id1,*id2,...*idn>` appears at the end of a line. To determine the id for a given message, use the following command, where `filename` is the name of the file that elicits the message:

```
mlint filename -id
```

For information on adding the `%#ok` directive using the Editor context menu, see “Suppressing M-Lint Indicators and Messages”.

## Examples

The following examples use `lengthofline.m`, which is a sample M-file with code that can be improved. You can find it in `matlabroot/help/techdoc/matlab_env/examples`. If you want to run the examples, save a copy of `lengthofline.m` to a location on your MATLAB path.

### Running mlint on a File with No Options

To run `mlint` on the example file, `lengthofline.m`, run

```
mlint('lengthofline')
```

MATLAB displays M-Lint messages for `lengthofline.m` in the Command Window:

```
L 22 (C 1-9): The value assigned here to variable 'nohandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMP1(str1,str2) instead of using LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
Type 'doc struct' for more information.
```

```
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop.Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary. Use parentheses
to group, if needed.
```

For details about these messages and how to improve the code, see “Making Changes Based on M-Lint Messages” in the MATLAB Desktop Tools and Development Environment documentation.

### Running mlint with Options to Show IDs and Return Results to a Structure

To store the results to a structure and include message IDs, run

```
inform=mlint('lengthofline', '-id')
```

MATLAB returns

```
inform =

19x1 struct array with fields:
    message
    line
    column
    id
```

To see values for the first message, run

```
inform(1)
```

MATLAB displays

```
ans =  
  
message: 'The value assigned here to variable 'nohandle' might never be used.'  
line: 22  
column: [1 9]  
id: 'NASGU'
```

Here, the message is for the value that appears on line 22 that extends from column 1–9 in the M-file. NASGU is the ID for the message 'The value assigned here to variable 'nohandle' might never be used.'.

## Suppressing Specific Messages with mlint

When you add `%#ok` to a line, it suppresses all `mlint` messages for that line. However, suppose there are multiple messages in a line and you want to suppress some, but not all of them. Or, suppose you want to suppress a specific message, but not all messages that might arise in the future due to changes you make to that line. Use the `%#ok` syntax in conjunction with message IDs.

This example uses the following code, `displayAnonymousFunction.m`:

```
function displayAnonymousFunction  
% mini tutorial on anonymous function handles.  
  
disp(' ');  
disp(' Here is an example of an anonymous function that');  
disp(' retrieves the last modified date of a given file:');  
disp(' ');  
fileDate = @(f)getfield(dir(f),'date')  
  
disp(' ');  
disp(' You can call it by passing a filename into the ');  
disp(' function_handle variable. We will use the currently');  
disp(' running M-file for example purposes:');  
disp(' ');
```

```

thisFile = which(mfilename('fullpath'))

disp(' ');
disp(' Now call the anonymous function handle as you would');
disp(' call any function or function_handle: fileDate(thisFile)');
disp(' ');
fileDate(thisFile)

```

Run `mlint` with the `-id` option on `displayAnonymousFunction.m`:

```
mlint('displayAnonymousFunction','-id')
```

Results displayed to the Command Window show two messages for line 8:

```

L 8 (C 10): NOPRT: Terminate statement with semicolon to suppress output (in functions).
L 8 (C 16-23): GFLD: Use dynamic fieldnames with structures instead of GETFIELD.
                Type 'doc struct' for more information.

```

To suppress the first message on the line (about using a semicolon), use its message ID, `NOPRT`, with the `%#ok` syntax as shown here:

```
fileDate = @(f)getfield(dir(f),'date') %#ok<NOPRT>
```

When you run `mlint` for `displayAnonymousFunction.m`, only one message now displays for line 8.

To suppress multiple specific messages for a line, separate message IDs with commas in the `%#ok` syntax:

```
fileDate = @(f)getfield(dir(f),'date') %#ok<NOPRT,GFLD>
```

Now when you run `mlint` for `displayAnonymousFunction.m`, no messages display for line 8.

## Suppressing Specific Messages Throughout a File with `mlint`

To suppress a specific message throughout a file, use the `%#ok` syntax in conjunction with a message ID preceded by an asterisk (\*).

Run `mlint` with the `-id` option on the original `displayAnonymousFunction.m` code presented in the previous example:

```
mlint('displayAnonymousFunction','-id')
```

Results displayed to the Command Window show two messages for line 8:

```
L 8 (C 10): NOPRT: Terminate statement with semicolon to suppress output (in functions).
L 8 (C 16-23): GFLD: Use dynamic fieldnames with structures instead of GETFIELD.
                Type 'doc struct' for more information.
```

To suppress the semicolon message throughout the file, use its message ID, `NOPRT`, with an asterisk in the `%#ok` syntax as shown here:

```
fileDate = @(f)getfield(dir(f),'date') %#ok<*NOPRT>
```

When you run `mlint` for `displayAnonymousFunction.m`, the semicolon message is suppressed throughout the file and only one message displays for line 8.

To suppress multiple specific messages throughout a file, separate message IDs with commas in the `%#ok` syntax, and precede each message ID with an asterisk:

```
fileDate = @(f)getfield(dir(f),'date') %#ok<*NOPRT,*GFLD>
```

Now when you run `mlint` for `displayAnonymousFunction.m`, both the `NOPRT` and `GFLD` messages are suppressed throughout the file.

## **Error Message: An M-Lint message Was Once Suppressed Here, But the Message No Longer Appears**

This examples shows how to interpret the message, “An M-Lint message was once suppressed here, but the message no longer appears.”

Suppose you direct `mlint` to ignore line 15, in the M-file, `displayAnonymousFunction.m` (the code for which is presented in the third example in this section) by adding `%#ok` to the end of line 15:



```
thisFile = which(mfilename('fullpath') %#ok
```

When you run `mlint` for `displayAnonymousFunction.m`, typically no message is shown for line 15, because it contains the `%#ok` message suppression syntax. However, there are some exceptions, as follows:

- If you change the code so that it would not elicit the message, “Terminate statement with semicolon to suppress output (in functions)” if you removed the `%#ok` directive
- If you disable the message in M-Lint preferences after you add the `%#ok` directive
- If the rules M-Lint uses for generating the message change

If any one of these cases is true for line 15, then the following message now appears at line 15:

```
An M-Lint message was once suppressed here, but the message no longer
```

To remove this message, use the context menu and select **Remove the Message Suppression**. The `%#ok` directive is removed and now no M-Lint messages appear for line 15 of `displayAnonymousFunction.m`.

### Displaying McCabe Complexity with `mlint`

To display the McCabe complexity of an M-File, run `mlint` with the `-cyc` option, as shown in the following example (assuming you have saved `lengthofline.m` to a local directory).

```
mlint lengthofline.m -cyc
```

Results displayed in the Command Window show the McCabe complexity of the file, followed by the M-File messages, as shown here:

```
L 1 (C 23-34): The McCabe complexity of 'lengthofline' is 12.  
L 22 (C 1-9): The value assigned here to variable 'nohandle' might never be used.  
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).  
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.  
L 24 (C 44-49): Use STRCMP1(str1,str2) instead of using UPPER/LOWER in a call to STRCMP.
```

L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).

L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.

L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD. Type 'doc struct' for more.

L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.

L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.

L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.

L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.

L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.

L 45 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.

L 48 (C 52): There may be a parenthesis imbalance around here.

L 48 (C 53): There may be a parenthesis imbalance around here.

L 48 (C 54): There may be a parenthesis imbalance around here.

L 48 (C 55): There may be a parenthesis imbalance around here.


L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).

L 49 (C 23): Use of brackets [] is unnecessary. Use parentheses to group, if needed.

## See Also

mlintrpt, profile

---

<b>Purpose</b>	Run <code>mlint</code> for file or directory, reporting results in browser
<b>GUI Alternatives</b>	From the Current Directory browser, click the <b>Actions</b> button  , and then select <b>Reports &gt; M-Lint Code Check Report</b> . See also the automatic “M-Lint Code Analyzer” in the Editor.
<b>Syntax</b>	<pre>mlintrpt mlintrpt('filename', file ) mlintrpt('dirname', dir ) mlintrpt('filename', file , 'settings.txt') mlintrpt('dirname', dir , 'settings.txt')</pre>
<b>Description</b>	<p><code>mlintrpt</code> scans all M-files in the current directory for M-Lint messages and reports the results in a MATLAB Web browser.</p> <p><code>mlintrpt('filename', file )</code> scans the M-file <code>filename</code> for messages and reports results. You can omit <code>file</code> in this form of the syntax because it is the default.</p> <p><code>mlintrpt('dirname', dir )</code> scans the specified directory. Here, <code>dirname</code> can be in the current directory or can be a full path.</p> <p><code>mlintrpt('filename', file , 'settings.txt')</code> applies the M-Lint settings to enable or suppress messages as indicated in the specified <code>settings.txt</code> file. For information about creating a <code>settings.txt</code> file, select <b>File &gt; Preferences &gt; M-Lint</b>, and click <b>Help</b>.</p> <p><code>mlintrpt('dirname', dir , 'settings.txt')</code> applies the M-Lint settings indicated in the specified <code>settings.txt</code> file.</p> <hr/> <p><b>Note</b> If you specify a <code>settings.txt</code> file, you must specify the full path to the file.</p> <hr/>
<b>Examples</b>	<code>lengthofline.m</code> is an example M-file with code that can be improved. It is found in <code>matlabroot/matlab/help/techdoc/matlab_env/examples</code> .

## Run Report for All Files in a Directory

Run

```
mlintrpt(fullfile(matlabroot,'help','techdoc','matlab_env','examples'),'dir')
```

and MATLAB displays a report of potential problems and improvements for all M-files in the `examples` directory.

**Web Browser - M-Lint Code Check Report**

File Edit View Go Debug Desktop Window Help

**M-Lint Code Check Report**

Rerun This Report Run Report on Current Directory

Report for directory matlab\help\techdoc\matlab\_env\examples

<a href="#">collatz</a> 1 message	<a href="#">17:</a> 'sequence' might be growing inside a loop. Consider preallocating for speed.
<a href="#">collatzplot</a> 2 messages	<a href="#">11:</a> The value assigned here to variable 'seq_length' might never be used. <a href="#">11:</a> 'seq_length' might be growing inside a loop. Consider preallocating for speed.
<a href="#">lengthoffline</a> 19 messages	<a href="#">22:</a> The value assigned here to variable 'nohandle' might never be used. <a href="#">23:</a> NUMEL(x) is usually faster than PROD(SIZE(x)). <a href="#">24:</a> 'notline' might be growing inside a loop. Consider preallocating for speed. <a href="#">24:</a> Use STRCMP1(str1,str2) instead of using LOWER in a call to STRCMP. <a href="#">28:</a> NUMEL(x) is usually faster than PROD(SIZE(x)). <a href="#">34:</a> 'data' might be growing inside a loop. Consider preallocating for speed. <a href="#">34:</a> Use dynamic fieldnames with structures instead of GETFIELD. Type 'doc struct' for more information. <a href="#">38:</a> Use    instead of   as the OR operator in (scalar) conditional

For details about these messages and how to improve the code, see “Making Changes Based on M-Lint Messages” in the MATLAB Desktop Tools and Development Environment documentation.

## Run Report Using M-Lint Preference Settings

In **File > Preferences > M-Lint**, save preference settings to a file, for example, `MLintNoSemis.txt`. To apply those settings when you run `mlintrpt`, use the `file` option and supply the full path to the settings file name as shown in this example:

```
mlintrpt('lengthofline.m', 'file', ...  
        'C:\WINNT\Profiles\me\Application Data\MathWorks\MATLAB\R2007a\MLintNoSemis.txt')
```

Alternatively, use `fullfile` if the settings file is stored in the preferences directory:

```
mlintrpt('lengthofline.m', 'file', fullfile(prefdir, 'MLintNoSemis.txt'))
```

Assuming that in that example `MLintNoSemis.txt` file, the setting for `Terminate statement with semicolon to suppress output` has been disabled, the results of `mlintrpt` for `lengthofline` do not show that message for line 49.

When `mlintrpt` cannot locate the settings file, the first message in the report is

```
0: Unable to open or read the configuration file 'MLintNoSemis.txt'--u
```

## See Also

`mlint`

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

**Syntax** `mlock`

**Description** `mlock` locks the currently running M-file or MEX-file in memory so that subsequent `clear` functions do not remove it.

Use the `munlock` function to return the file to its normal, clearable state.

Locking an M-file or MEX-file in memory also prevents any persistent variables defined in the file from getting reinitialized.

**Examples** The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
.
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked('testfun')
ans =
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock('testfun')

mislocked('testfun')
ans =
     0
```

**See Also** `mislocked`, `munlock`, `persistent`

# mmfileinfo

---

**Purpose** Information about multimedia file

**Syntax** `info = mmfileinfo(filename)`

**Description** `info = mmfileinfo(filename)` returns a structure, `info`, with fields containing information about the contents of the multimedia file identified by `filename`. The `filename` input is a string enclosed in single quotes.

---

**Note** `mmfileinfo` can be used only on Microsoft Windows and Macintosh operating systems.

---

If `filename` is a URL, `mmfileinfo` might take a long time to return because it must first download the file. For large files, downloading can take several minutes. To avoid blocking the MATLAB command line while this processing takes place, download the file before calling `mmfileinfo`.

The `info` structure contains the following fields, listed in the order they appear in the structure.

Field	Description
Filename	String indicating the name of the file
Duration	Length of the file in seconds
Audio	Structure containing information about the audio data in the file. See “Audio Data” on page 2-2355 for more information about this data structure.
Video	Structure containing information about the video data in the file. See “Video Data” on page 2-2355 for more information about this data structure.



## Audio Data

The Audio structure contains the following fields, listed in the order they appear in the structure. If the file does not contain audio data, the fields in the structure are empty.

Field	Description
Format	Text string, indicating the audio format
NumberOfChannels	Number of audio channels

## Video Data

The Video structure contains the following fields, listed in the order they appear in the structure.

Field	Description
Format	Text string, indicating the video format
Height	Height of the video frame
Width	Width of the video frame

## Examples

This example gets information about the contents of a file containing audio data.

```
info = mmfileinfo('my_audio_data.mp3')  
  
info =  
  
    Filename: 'my_audio_data.mp3'  
    Duration: 1.6030e+002  
    Audio: [1x1 struct]  
    Video: [1x1 struct]
```

To look at the information returned about the audio data in the file, examine the fields in the Audio structure.

# mmfileinfo

---

```
audio_data = info.Audio  
  
audio_data =  
  
           Format: 'MPEGLAYER3'  
           NumberOfChannels: 2
```

Because the file contains only audio data, the fields in the Video structure are empty.

```
info.Video  
  
ans =  
  
           Format: ''  
           Height: []  
           Width: []
```

**Purpose** Create multimedia reader object for reading video files

**Syntax**  
`obj = mmreader(filename)`  
`obj = mmreader(filename, 'P1', V1, 'P2', V2, ...)`

**Description** `obj = mmreader(filename)` constructs a multimedia reader object, `obj`, that can read video data from a multimedia file. `filename` is a string specifying the name of a multimedia file. There are no restrictions on file extensions. By default, the MATLAB software looks for the file `filename` on the MATLAB path. `mmreader` supports the file formats listed in the following table:

Platform	Supported File Formats
Windows	AVI (.avi), MPEG-1 (.mpg), Windows Media® Video (.wmv, .asf, .asx), and any format supported by Microsoft DirectShow.
Macintosh	AVI (.avi), MPEG-1 (.mpg), MPEG-4 (.mp4, .m4v), Apple QuickTime® Movie (.mov), and any format supported by QuickTime as listed on .

If the object cannot be constructed for any reason (for example, if the file cannot be opened or does not exist, or if the file format is not recognized or supported), MATLAB throws an error.

`obj = mmreader(filename, 'P1', V1, 'P2', V2, ...)` constructs a multimedia reader object, assigning values `V1`, `V2`, etc. to the specified properties `P1`, `P2`, etc., respectively. If an invalid property name or property value is specified, MATLAB throws an error and the object is not created. Note that the property value pairs can be in any format supported by the `set` function, i.e., parameter-value string pairs, structures, or parameter-value cell array pairs. The `mmreader` object supports the following properties.

# mmreader

---

<b>Property</b>	<b>Description</b>	<b>Read-Only</b>	<b>Default Value</b>
Duration	Total length of file in seconds	Yes	
Name	Name of the file from which the reader object was created	Yes	
Path	String containing the full path to the file associated with the reader	Yes	
Tag	Generic string for the user to set	No	''
Type	Classname of the object	Yes	mmreader
UserData	Generic field for any user-defined data	No	[ ]
BitsPerPixel	Bits per pixel of the video data	Yes	
FrameRate	Frame rate of the video in frames per second	Yes	
Height	Height of the video frame in pixels	Yes	
NumberOfFrames	Total number of frames in the video stream	Yes	

Property	Description	Read-Only	Default Value
VideoFormat	String indicating the video format as it is represented in MATLAB, e.g., RGB24	Yes	
Width	Width of the video frame in pixels	Yes	

## Remarks

### Working with Variable Frame Rate Video

If the video file provided to mmreader is a variable frame rate file (as with many Windows Media Video files), the MATLAB software shows a warning, as in this hypothetical case:

```
>> obj = mmreader('VarFrameRate.wmv')
Warning: Unable to determine the number of frames in this file.

Summary of Multimedia Reader Object for 'VarFrameRate.wmv'.

Video Parameters: 23.98 frames per second, RGB24 1280x720.
                  Unable to determine video frames available.
```

Because the file `VarFrameRate.wmv` was encoded as a variable frame rate video, the number of frames is not known when you construct the mmreader object.

### Attempting to Read Beyond the End of the File

You can still read from a variable frame rate file by specifying the number of frames, but mmreader and read will behave slightly differently depending on the context of the read request.

If you ask for a frame range beyond the end of the file, the system generates an error. For example, suppose you attempt to read frame 3000 in a file that has only 2825 frames:

```
>> images = read(obj, 3000);
```

??? The frame range requested is beyond the end of the file.

If the requested frame range straddles the end of the file, the system returns a warning as shown in the next example, where frames 2800–3000 are requested in a file that has only 2825 frames:

```
>> images = read(obj, [2800 3000]);  
Warning: The end of file was reached before the  
requested frames were read completely.  
Frames 2800 through 2825 were returned.
```

## Examples

Construct a multimedia reader object associated with file `xylophone.mpg` with the user tag property set to `'myreader1'`.

```
readerobj = mmreader('xylophone.mpg', 'tag', 'myreader1');
```

Read in all the video frames.

```
vidFrames = read(readerobj);
```

Find out how many frames there are.

```
numFrames = get(readerobj, 'numberOfFrames');
```

Create a MATLAB movie struct from the video frames.

```
for k = 1 : numFrames  
    mov(k).cdata = vidFrames(:,:,k);  
    mov(k).colormap = [];  
end
```

Play back the movie once at the video's frame rate.

```
movie(mov, 1, readerobj.FrameRate);
```

## See Also

`get`, `mmfileinfo`, `read`, `set`

**Purpose** Modulus after division

**Syntax** `M = mod(X,Y)`

**Description** `M = mod(X,Y)` if `Y ~= 0`, returns `X - n.*Y` where `n = floor(X./Y)`. If `Y` is not an integer and the quotient `X./Y` is within roundoff error of an integer, then `n` is that integer. The inputs `X` and `Y` must be real arrays of the same size, or real scalars.

The following are true by convention:

- `mod(X,0)` is `X`
- `mod(X,X)` is `0`
- `mod(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `Y`.

**Remarks** `rem(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `X`.

`mod(X,Y)` and `rem(X,Y)` are equal if `X` and `Y` have the same sign, but differ by `Y` if `X` and `Y` have different signs.

The `mod` function is useful for congruence relationships: *x and y are congruent (mod m)* if and only if `mod(x,m) == mod(y,m)`.

**Examples**

```

mod(13,5)
ans =
    3

mod([1:5],3)
ans =
    1    2    0    1    2

mod(magic(3),3)
ans =
    2    1    0
    0    2    1
    1    0    2
    
```

# mod

---

## See Also

rem



---

<b>Purpose</b>	Most frequent values in array
<b>Syntax</b>	$M = \text{mode}(X)$ $M = \text{mode}(X, \text{dim})$ $[M, F] = \text{mode}(X, \dots)$ $[M, F, C] = \text{mode}(X, \dots)$
<b>Description</b>	<p><math>M = \text{mode}(X)</math> for vector <math>X</math> computes the sample mode <math>M</math>, (i.e., the most frequently occurring value in <math>X</math>). If <math>X</math> is a matrix, then <math>M</math> is a row vector containing the mode of each column of that matrix. If <math>X</math> is an <math>N</math>-dimensional array, then <math>M</math> is the mode of the elements along the first nonsingleton dimension of that array.</p> <p>When there are multiple values occurring equally frequently, <code>mode</code> returns the smallest of those values. For complex inputs, this is taken to be the first value in a sorted list of values.</p> <p><math>M = \text{mode}(X, \text{dim})</math> computes the mode along the dimension <code>dim</code> of <math>X</math>.</p> <p><math>[M, F] = \text{mode}(X, \dots)</math> also returns array <math>F</math>, each element of which represents the number of occurrences of the corresponding element of <math>M</math>. The <math>M</math> and <math>F</math> output arrays are of equal size.</p> <p><math>[M, F, C] = \text{mode}(X, \dots)</math> also returns cell array <math>C</math>, each element of which is a sorted vector of all values that have the same frequency as the corresponding element of <math>M</math>. All three output arrays <math>M</math>, <math>F</math>, and <math>C</math> are of equal size.</p>
<b>Remarks</b>	<p>The <code>mode</code> function is most useful with discrete or coarsely rounded data. The mode for a continuous probability distribution is defined as the peak of its density function. Applying the <code>mode</code> function to a sample from that distribution is unlikely to provide a good estimate of the peak; it would be better to compute a histogram or density estimate and calculate the peak of that estimate. Also, the <code>mode</code> function is not suitable for finding peaks in distributions having multiple modes.</p>
<b>Examples</b>	<b>Example 1</b> Find the mode of the 3-by-4 matrix shown here:

# mode

---

```
X = [3 3 1 4; 0 0 1 1; 0 1 2 4]
X =
     3     3     1     4
     0     0     1     1
     0     1     2     4

mode(X)
ans =
     0     0     1     4
```

Find the mode along the second (row) dimension:

```
mode(X, 2)
ans =
     3
     0
     0
```

## Example 2

Find the mode of a continuous variable grouped into bins:

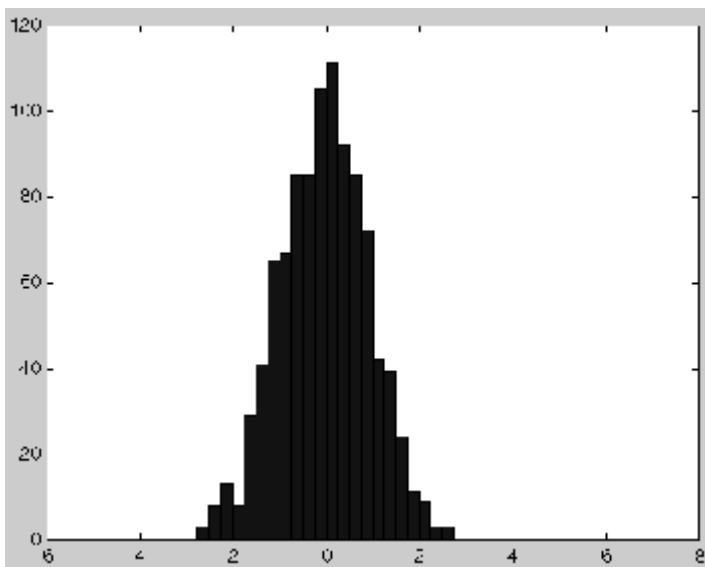
```
randn('state', 0);           % Reset the random number generator

y = randn(1000,1);
edges = -6:.25:6;
[n,bin] = histc(y,edges);

m = mode(bin)
m =
    22

edges([m, m+1])
ans =
   -0.7500   -0.5000

hist(y,edges+.125)
```



**See Also** `mean`, `median`, `hist`, `histc`

# more

---

**Purpose** Control paged output for Command Window

**Syntax**

```
more on
more off
more(n)
A = more(state)
```

**Description** `more on` enables paging of the output in the MATLAB Command Window. MATLAB displays output one page at a time. Use the keys defined in the table below to control paging.

`more off` disables paging of the output in the MATLAB Command Window.

`more(n)` defines the length of a page to be `n` lines.

`A = more(state)` returns in `A` the number of lines that are currently defined to be a page. The `state` input can be one of the quoted strings 'on' or 'off', or the number of lines to set as the new page length.

By default, the length of a page is equal to the number of lines available for display in the MATLAB Command Window. Manually changing the size of the command window adjusts the page length accordingly.

If you set the page length to a specific value, MATLAB uses that value for the page size, regardless of the size of the command window. To have MATLAB return to matching page size to window size, type `more off` followed by `more on`.

To see the status of `more`, type `get(0, 'More')`. MATLAB returns either `on` or `off` indicating the `more` status. You can also set status for `more` by using `set(0, More, 'status')`, where 'status' is either 'on' or 'off'.

When you have enabled `more` and are examining output, you can do the following.

Press the...	To...
Return key	Advance to the next line of output.

<b>Press the...</b>	<b>To...</b>
Space bar	Advance to the next page of output.
Q (for quit) key	Terminate display of the text. Do not use <b>Ctrl+C</b> to terminate <b>more</b> or you might generate error messages in the Command Window.

**more** is in the **off** state, by default.

**See Also**

diary

# move

---

**Purpose** Move or resize control in parent window

**Syntax** `V = h.move(position)`  
`V = move(h, position)`

**Description** `V = h.move(position)` moves the control to the position specified by the `position` argument. When you use `move` with only the handle argument, `h`, it returns a four-element vector indicating the current position of the control.

`V = move(h, position)` is an alternate syntax for the same operation.

The `position` argument is a four-element vector specifying the position and size of the control in the parent figure window. The elements of the vector are

`[x, y, width, height]`

where `x` and `y` are offsets, in pixels, from the bottom left corner of the figure window to the same corner of the control, and `width` and `height` are the size of the control itself.

**Examples** This example moves the control:

```
f = figure('Position', [100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200], f);
pos = h.move([50 50 200 200])
pos =
    50    50   200   200
```

The next example resizes the control to always be centered in the figure as you resize the figure window. Start by creating the script `resizectl.m` that contains

```
% Get the new position and size of the figure window
fpos = get(gcbo, 'position');

% Resize the control accordingly
```

```
h.move([0 0 fpos(3) fpos(4)]);
```

Now execute the following in MATLAB or in an M-file:

```
f = figure('Position', [100 100 200 200]);  
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200]);  
set(f, 'ResizeFcn', 'resizectrl');
```

As you resize the figure window, notice that the circle moves so that it is always positioned in the center of the window.

## See Also

set, get

# movefile

---

## Purpose

Move file or directory

## Graphical Interface

As an alternative to the `movefile` function, you can use the Current Directory browser to move files and directories.

## Syntax

```
movefile('source')
movefile('source','destination')
movefile('source','destination','f')
[status,message,messageid]=movefile('source','destination',
    'f')
```

## Description

`movefile('source')` moves the file or directory named `source` to the current directory, where `source` is the absolute or relative path name for the directory or file. To move multiple files or directories, you can use one or more wildcard characters `*` after the last file separator in `source`. Note that the archive attribute of `source` is not preserved.

`movefile('source','destination')` moves the file or directory named `source` to the location `destination`, where `source` and `destination` are the absolute or relative paths for the directory or files. To move 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`. To rename a file or directory when moving it, make `destination` a different name than `source`, and specify only one file for `source`.

`movefile('source','destination','f')` moves the file or directory named `source` to the location `destination`, regardless of the read-only attribute of `destination`.

`[status,message,messageid]=movefile('source','destination','f')` moves the file or directory named `source` to the location `destination`, returning the status, a message, and the MATLAB error message ID (see `error` and `lasterror`). Here, `status` is logical 1 for success or logical 0 for error. Only one output argument is required and the `f` input argument is optional.

You can use the `*` (wildcard character) in a path string.



## Examples

### Moving a File to the Current Directory

To move the file `myfiles/myfunction.m` to the current directory, type

```
movefile('myfiles/myfunction.m')
```

If the current directory is `projects/testcases` and you want to move `projects/myfiles` and its contents to the current directory, use `../` in the source path to navigate up one level to get to the directory.

```
movefile('../myfiles')
```

### Using a Wildcard to Move All Matching Files

To move all files in the directory `myfiles` whose names begin with `my` to the current directory, type

```
movefile('myfiles/my*')
```

### Moving a File to a Destination Directory

To move the file `myfunction.m` from the current directory to the directory `projects`, where `projects` and the current directory are at the same level, type

```
movefile('myfunction.m', '../projects')
```

### Moving a Directory Down One Level

This example moves the a directory down a level. For example to move the directory `projects/testcases` and all its contents down a level in `projects` to `projects/myfiles`, type

```
movefile('projects/testcases', 'projects/myfiles/')
```

The directory `testcases` and its contents now appear in the directory `myfiles`.

### Moving a File to Read-Only Directory and Renaming the File

Move the file `myfile.m` from the current directory to `d:/work/restricted`, assigning it the name `test1.m`, where `restricted` is a read-only directory.

# movefile

---

```
movefile('myfile.m','d:/work/restricted/test1.m','f')
```

The read-only file `myfile.m` is no longer in the current directory. The file `test1.m` is in `d:/work/restricted` and is read only.

## Returning Status When Moving Files

This example aims to move to the current directory all files in the directory `myfiles` whose names start with `new`. However, `new*` is mistyped as `nex*`. As a result, `movefile` is unsuccessful because no files are found that start with `nex`, and the status, message, and messageid returned indicate this:

```
[s,mess,messid]=movefile('myfiles/nex*')
```

```
s =  
    0
```

```
mess =
```

```
A duplicate filename exists, or the file cannot be found.
```

```
messid =
```

```
MATLAB:MOVEFILE:OSError
```

## See Also

`cd`, `copyfile`, `delete`, `dir`, `fileattrib`, `filebrowser`, `ls`, `mkdir`, `rmdir`  
“Managing Files and Working with the Current Directory”

---

<b>Purpose</b>	Move GUI figure to specified location on screen
<b>Syntax</b>	<pre>movegui(h, 'position') movegui('position') movegui(h) movegui</pre>
<b>Description</b>	<p><code>movegui(h, 'position')</code> moves the figure identified by handle <code>h</code> to the specified screen location, preserving the figure's size. The <i>position</i> argument can be any of the following strings:</p> <ul style="list-style-type: none"><li>• north – top center edge of screen</li><li>• south – bottom center edge of screen</li><li>• east – right center edge of screen</li><li>• west – left center edge of screen</li><li>• northeast – top right corner of screen</li><li>• northwest – top left corner of screen</li><li>• southeast – bottom right corner of screen</li><li>• southwest – bottom left corner</li><li>• center – center of screen</li><li>• onscreen – nearest location with respect to current location that is on screen</li></ul> <p>The <i>position</i> argument can also be a two-element vector <code>[h, v]</code>, where depending on sign, <code>h</code> specifies the figure's offset from the left or right edge of the screen, and <code>v</code> specifies the figure's offset from the top or bottom of the screen, in pixels. The following table summarizes the possible values.</p>

# movegui

---

<code>h</code> (for <code>h &gt;= 0</code> )	offset of left side from left edge of screen
<code>h</code> (for <code>h &lt; 0</code> )	offset of right side from right edge of screen
<code>v</code> (for <code>v &gt;= 0</code> )	offset of bottom edge from bottom of screen
<code>v</code> (for <code>v &lt; 0</code> )	offset of top edge from top of screen

`movegui('position')` move the callback figure (`gcbf`) or the current figure (`gcf`) to the specified position.

`movegui(h)` moves the figure identified by the handle `h` to the onscreen position.

`movegui` moves the callback figure (`gcbf`) or the current figure (`gcf`) to the onscreen position. This is useful as a string-based `CreateFcn` callback for a saved figure. It ensures the figure appears on screen when reloaded, regardless of its saved position.

## Examples

This example demonstrates the usefulness of `movegui` to ensure that saved GUIs appear on screen when reloaded, regardless of the target computer's screen sizes and resolution. It creates a figure off the screen, assigns `movegui` as its `CreateFcn` callback, then saves and reloads the figure.

```
f = figure('Position',[10000,10000,400,300]);
set(f,'CreateFcn','movegui')
hgsave(f,'onscreenfig')
close(f)
f2 = hgload('onscreenfig');
```

## See Also

`guide`

"Creating GUIs" in the MATLAB documentation

**Purpose** Play recorded movie frames

**Syntax**

```
movie
movie(M)
movie(M,n)
movie(M,n,fps)
movie(h,...)
movie(h,M,n,fps,loc)
```

**Description** `movie` plays the movie defined by a matrix whose columns are movie frames (usually produced by `getframe`).

`movie(M)` plays the movie in matrix `M` once, using the current axes as the default target. If you want to play the movie in the figure instead of the axes, specify the figure handle (or `gcf`) as the first argument: `movie(figure_handle,...)`. `M` must be an array of movie frames (usually from `getframe`).

`movie(M,n)` plays the movie `n` times. If `n` is negative, each cycle is shown forward then backward. If `n` is a vector, the first element is the number of times to play the movie, and the remaining elements make up a list of frames to play in the movie.

For example, if `M` has four frames then `n = [10 4 4 2 1]` plays the movie ten times, and the movie consists of frame 4 followed by frame 4 again, followed by frame 2 and finally frame 1.

`movie(M,n,fps)` plays the movie at `fps` frames per second. The default is 12 frames per second. Computers that cannot achieve the specified speed play as fast as possible.

`movie(h,...)` plays the movie centered in the figure or axes identified by the handle `h`.

`movie(h,M,n,fps,loc)` specifies `loc`, a four-element location vector, `[x y 0 0]`, where the lower left corner of the movie frame is anchored (only the first two elements in the vector are used). The location is relative to the lower left corner of the figure or axes specified by handle `h` and in units of pixels, regardless of the object's `Units` property.

## Remarks

The `movie` function uses a default figure size of 560-by-420 and does not resize figures to fit movies with larger or smaller frames. To accommodate other frame sizes, you can resize the figure to fit the movie, as shown in the second example below.

`movie` only accepts 8-bit image frames; it does not accept 16-bit grayscale or 24-bit truecolor image frames.

Buffering the movie places all frames in memory. As a result, on Microsoft Windows and perhaps other platforms, a long movie (on the order of several hundred frames) can exhaust memory, depending on system resources. In such cases an error message is issued that says

```
??? Error using ==> movie
    Could not create movie frame
```

You can abort a movie by typing **Ctrl-C**.

## Examples

Example 1: Animate the peaks function as you scale the values of `Z`:

```
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
% Record the movie
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    F(j) = getframe;
end
% Play the movie ten times
movie(F,10)
```

Example 2: Specify figure when calling `movie` to fit the movie to the figure:

```
r = subplot(2,1,1)
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
```

```
s = subplot(2,1,2)
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
% Record the movie
for j = 1:20
    axes(r)
        surf(sin(2*pi*j/20)*Z,Z)
    axes(s)
        surf(sin(2*pi*(j+5)/20)*Z,Z)
        F(j) = getframe(gcf);
    pause(.0333)
end
% Play the movie; note that it does not fit the figure properly:
h2 = figure;
movie(F,10)
% Use the figure handle to make the frames fit:
movie(h2,F,10)
```

Example 3: With larger frames, first adjust the figure's size to fit the movie:

```
figure('position',[100 100 850 600])
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
% Record the movie
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    F(j) = getframe;
end
[h, w, p] = size(F(1).cdata); % use 1st frame to get dimensions
hf = figure;
% resize figure based on frame's w x h, and place at (150, 150)
set(hf, 'position', [150 150 w h]);
axis off
% tell movie command to place frames at bottom left
```

# movie

---

```
movie(hf,F,4,30,[0 0 0 0]);
```

## See Also

`aviread`, `getframe`, `frame2im`, `im2frame`

“Animation” on page 1-95 for related functions

See Example – Visualizing an FFT as a Movie for another example



**Purpose** Create Audio/Video Interleaved (AVI) movie from MATLAB movie

**Syntax** `movie2avi(mov, filename)`  
`movie2avi(mov, filename, param, value, param, value...)`

**Description** `movie2avi(mov, filename)` creates the AVI movie filename from the MATLAB movie mov. The filename input is a string enclosed in single quotes.

`movie2avi(mov, filename, param, value, param, value...)` creates the AVI movie filename from the MATLAB movie mov using the specified parameter settings.

Parameter	Value	Default
'colormap'	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).  This parameter can be specified only when the 'compression' parameter is set to 'MSVC', 'RLE', or 'None'	There is no default colormap.

Parameter	Value	Default
'compression'	<p>A text string specifying the compression codec to use.</p> <p>On Microsoft Windows operating systems:</p> <ul style="list-style-type: none"><li>• 'Indeo3'</li><li>• 'Indeo5'</li><li>• 'Cinepak'</li><li>• 'MSVC'</li><li>• 'RLE'</li><li>• 'None'</li><li>• 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 <code>addframe</code> function reports an error if it can not find the specified custom compressor.</li></ul> <p>On UNIX operating systems:</p> <ul style="list-style-type: none"><li>• 'None'</li></ul>	<p>'Indeo5' on Windows systems.</p> <p>'None' on UNIX systems.</p>
'fps'	<p>A scalar value specifying the speed of the AVI movie in frames per second (fps).</p>	15 fps
'keyframe'	<p>For compressors that support temporal compression, this is the number of key frames per second.</p>	2.1429 key frames per second.

Parameter	Value	Default
'quality'	A number between 0 and 100 the specifies the desired quality of the output. Higher numbers result in higher video quality and larger file sizes. Lower numbers result in lower video quality and smaller file sizes. This parameter has no effect on uncompressed movies.	75
'videoname'	A descriptive name for the video stream. This parameter must be no greater than 64 characters long.	The default is the filename.

**See Also**      avifile, aviread, aviinfo, movie

# mput

---

**Purpose** Upload file or directory to FTP server

**Syntax**

```
mput(f, 'filename')  
mput(ftp, 'directoryname')  
mput(f, 'wildcard')
```

**Description** `mput(f, 'filename')` uploads `filename` from the MATLAB current directory to the current directory of the FTP server `f`, where `filename` is a file, and where `f` was created using `ftp`. You can use a wildcard (\*) in `filename`. MATLAB returns a cell array listing the full path to the uploaded files on the server.

`mput(ftp, 'directoryname')` uploads the directory `directoryname` and its contents. MATLAB returns a cell array listing the full path to the uploaded files on the server.

`mput(f, 'wildcard')` uploads a set of files or directories specified by a wildcard. MATLAB returns a cell array listing the full path to the uploaded files on the server.

**See Also** `ftp`, `mget`, `mkdir (ftp)`, `rename`

**Purpose** Create and open message box

**Syntax**

```
h = msgbox(Message)
h = msgbox(Message,Title)
h = msgbox(Message,Title,Icon)
h = msgbox(Message,Title,'custom',IconData,IconCMap)
h = msgbox(...,CreateMode)
```

**Description** `h = msgbox(Message)` creates a message dialog box that automatically wraps `Message` to fit an appropriately sized figure. `Message` is a string vector, string matrix, or cell array. `msgbox` returns the handle of the message box in `h`.

`h = msgbox(Message,Title)` specifies the title of the message box.

`h = msgbox(Message,Title,Icon)` specifies which icon to display in the message box. `Icon` is 'none', 'error', 'help', 'warn', or 'custom'. The default is 'none'.



Error Icon



Help Icon



Warning Icon

`h = msgbox(Message,Title,'custom',IconData,IconCMap)` defines a customized icon. `IconData` contains image data defining the icon. `IconCMap` is the colormap used for the image.

`h = msgbox(...,CreateMode)` specifies whether the message box is modal or nonmodal. Optionally, it can also specify an interpreter for `Message` and `Title`.

If `CreateMode` is a string, it must be one of the values shown in the following table.

CreateMode Value	Description
'modal'	Replaces the message box having the specified Title, that was last created or clicked on, with a modal message box as specified. All other message boxes with the same title are deleted. The message box which is replaced can be either modal or nonmodal.
'non-modal' (default)	Creates a new nonmodal message box with the specified parameters. Existing message boxes with the same title are not deleted.
'replace'	Replaces the message box having the specified Title, that was last created or clicked on, with a nonmodal message box as specified. All other message boxes with the same title are deleted. The message box which is replaced can be either modal or nonmodal.

---

**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 `WindowState` in the Figure Properties.

---

If `CreateMode` is a structure, it can have fields `WindowState` and `Interpreter`. The `WindowState` field must be one of the values in the

table above. Interpreter is one of the strings 'tex' or 'none'. The default value for Interpreter is 'none'.

**See Also**

dialog, errordlg, helpdlg, inputdlg, listdlg, questdlg, warndlg  
figure, textwrap, uiwait, uiresume

“Predefined Dialog Boxes” on page 1-108 for related functions

# mtimes

---

**Purpose** Matrix multiplication

**Syntax**  $C = A*B$

**Description**  $C = A*B$  is the linear algebraic product of the matrices  $A$  and  $B$ . If  $A$  is an  $m$ -by- $p$  and  $B$  is a  $p$ -by- $n$  matrix, the  $i, j$  entry of  $C$  is defined by

$$C(i, j) = \sum_{k=1}^p A(i, k)B(k, j)$$

The product  $C$  is an  $m$ -by- $n$  matrix. For nonscalar  $A$  and  $B$ , the number of columns of  $A$  must equal the number of rows of  $B$ . You can multiply a scalar by a matrix of any size.

The preceding definition says that  $C(i, j)$  is the inner product of the  $i$ th row of  $A$  with the  $j$ th column of  $B$ . You can write this definition using the MATLAB colon operator as

$$C(i, j) = A(i, :)*B(:, j)$$

where  $A(i, :)$  is the  $i$ th row of  $A$  and  $B(:, j)$  is the  $j$ th row of  $B$ .

---

**Note** If  $A$  is an  $m$ -by-0 empty matrix and  $B$  is a 0-by- $n$  empty matrix, where  $m$  and  $n$  are positive integers,  $A*B$  is an  $m$ -by- $n$  matrix of all zeros.

---

## Examples

### Example 1

If  $A$  is a row vector and  $B$  is a column vector with the same number of elements as  $A$ ,  $A*B$  is simply the inner product of  $A$  and  $B$ . For example,

$$A = [5 \ 3 \ 2 \ 6]$$

$$A =$$

$$5 \quad 3 \quad 2 \quad 6$$



$$B = [-4 \ 9 \ 0 \ 1]'$$

$$B =$$

-4  
9  
0  
1

$$A*B$$

$$\text{ans} =$$

13

### Example 2

$$A = [1 \ 3 \ 5; \ 2 \ 4 \ 7]$$

$$A =$$

1    3    5  
2    4    7

$$B = [-5 \ 8 \ 11; \ 3 \ 9 \ 21; \ 4 \ 0 \ 8]$$

$$B =$$

-5    8    11  
3    9    21  
4    0    8

The product of A and B is

$$C = A*B$$

$$C =$$

24    35    114  
30    52    162

Note that the second row of A is

```
A(2,:)
ans =
     2     4     7
```

while the third column of B is

```
B(:,3)
ans =
    11
    21
     8
```

The inner product of A(2,:) and B(:,3) is

```
A(2,:)*B(:,3)
ans =
    162
```

which is the same as C(2,3).

## Algorithm

mtimes uses the following Basic Linear Algebra Subroutines (BLAS):

- DDOT
- DGEMV
- DGEMM
- DSYRK
- DSYRZK

For inputs of type `single`, `mtimes` using corresponding routines that begin with “S” instead of “D”.

**See Also**

Arithmetic Operators

# mu2lin

---

**Purpose** Convert mu-law audio signal to linear

**Syntax** `y = mu2lin(mu)`

**Description** `y = mu2lin(mu)` converts mu-law encoded 8-bit audio signals, stored as “flints” in the range  $0 \leq \mu \leq 255$ , to linear signal amplitude in the range  $-s < Y < s$  where  $s = 32124/32768 \approx .9803$ . The input `mu` is often obtained using `fread(..., 'uchar')` to read byte-encoded audio files. “Flints” are MATLAB integers — floating-point numbers whose values are integers.

**See Also** `auread`, `lin2mu`

## Purpose

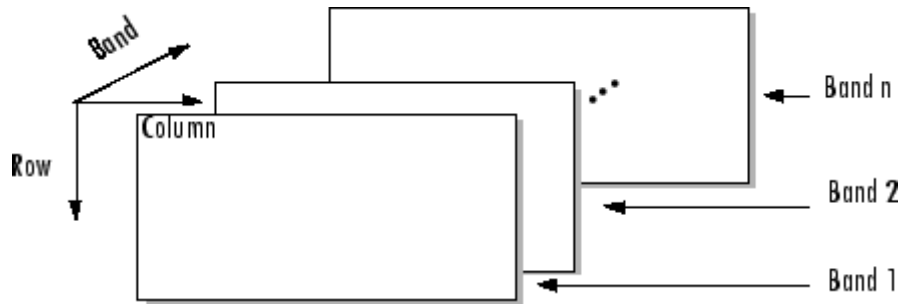
Read band-interleaved data from binary file

## Syntax

```
X = multibandread(filename, size, precision, offset,
    interleave, byteorder)
X = multibandread(...,subset1,subset2,subset3)
```

## Description

`X = multibandread(filename, size, precision, offset, interleave, byteorder)` reads band-sequential (BSQ), band-interleaved-by-line (BIL), or band-interleaved-by-pixel (BIP) data from the binary file `filename`. The `filename` input is a string enclosed in single quotes. This function defines *band* as the third dimension in a 3-D array, as shown in this figure.



You can use the parameters to `multibandread` to specify many aspects of the read operation, such as which bands to read. See “Parameters” on page 2-2392 for more information.

`X` is a 2-D array if only one band is read; otherwise it is 3-D. `X` is returned as an array of data type `double` by default. Use the `precision` parameter to map the data to a different data type.

`X = multibandread(...,subset1,subset2,subset3)` reads a subset of the data in the file. You can use up to three subsetting parameters to specify the data subset along row, column, and band dimensions. See “Subsetting Parameters” on page 2-2393 for more information.

# multibandread

---

---

**Note** In addition to BSQ, BIL, and BIP files, multiband imagery may be stored using the TIFF file format. In that case, use the `imread` function to import the data.

---

## Parameters

This table describes the arguments accepted by `multibandread`.

Argument	Description
<code>filename</code>	String containing the name of the file to be read.
<code>size</code>	Three-element vector of integers consisting of [height, width, N], where <ul style="list-style-type: none"><li>• height is the total number of rows</li><li>• width is the total number of elements in each row</li><li>• N is the total number of bands.</li></ul> This will be the dimensions of the data if it is read in its entirety.
<code>precision</code>	String specifying the format of the data to be read, such as 'uint8', 'double', 'integer*4', or any of the other precisions supported by the <code>fread</code> function.  Note: You can also use the <code>precision</code> parameter to specify the format of the output data. For example, to read <code>uint8</code> data and output a <code>uint8</code> array, specify a precision of 'uint8=>uint8' (or '*uint8'). To read <code>uint8</code> data and output it in the MATLAB software in single precision, specify 'uint8=>single'. See <code>fread</code> for more information.

Argument	Description
<code>offset</code>	Scalar specifying the zero-based location of the first data element in the file. This value represents the number of bytes from the beginning of the file to where the data begins.
<code>interleave</code>	String specifying the format in which the data is stored <ul style="list-style-type: none"> <li>• 'bsq' — Band-Sequential</li> <li>• 'bil' — Band-Interleaved-by-Line</li> <li>• 'bip' — Band-Interleaved-by-Pixel</li> </ul> For more information about these interleave methods, see the <code>multibandwrite</code> reference page.
<code>byteorder</code>	String specifying the byte ordering (machine format) in which the data is stored, such as <ul style="list-style-type: none"> <li>• 'ieee-le' — Little-endian</li> <li>• 'ieee-be' — Big-endian</li> </ul> See <code>fopen</code> for a complete list of supported formats.

## Subsetting Parameters

You can specify up to three subsetting parameters. Each subsetting parameter is a three-element cell array, `{dim, method, index}`, where

Parameter	Description
<code>dim</code>	Text string specifying the dimension to subset along. It can have any of these values: <ul style="list-style-type: none"> <li>• 'Column'</li> <li>• 'Row'</li> <li>• 'Band'</li> </ul>

# multibandread

---

Parameter	Description
<i>method</i>	<p>Text string specifying the subsetting method. It can have either of these values:</p> <ul style="list-style-type: none"><li>• 'Direct'</li><li>• 'Range'</li></ul> <p>If you leave out this element of the subset cell array, <code>multibandread</code> uses 'Direct' as the default.</p>
<i>index</i>	<p>If <code>method</code> is 'Direct', <code>index</code> is a vector specifying the indices to read along the Band dimension.</p> <p>If <code>method</code> is 'Range', <code>index</code> is a three-element vector of [<code>start</code>, <code>increment</code>, <code>stop</code>] specifying the range and step size to read along the dimension specified in <code>dim</code>. If <code>index</code> is a two-element vector, <code>multibandread</code> assumes that the value of <code>increment</code> is 1.</p>

## Examples

### Example 1

Setup initial parameters for a data set.

```
rows=3; cols=3; bands=5;
filename = tempname;
```

Define the data set.

```
fid = fopen(filename, 'w', 'ieee-le');
fwrite(fid, 1:rows*cols*bands, 'double');
fclose(fid);
```

Read every other band of the data using the Band-Sequential format.

```
im1 = multibandread(filename, [rows cols bands], ...
    'double', 0, 'bsq', 'ieee-le', ...
```



```
{'Band', 'Range', [1 2 bands]} )
```

Read the first two rows and columns of data using Band-Interleaved-by-Pixel format.

```
im2 = multibandread(filename, [rows cols bands], ...
    'double', 0, 'bip', 'ieee-le', ...
    {'Row', 'Range', [1 2]}, ...
    {'Column', 'Range', [1 2]} )
```

Read the data using Band-Interleaved-by-Line format.

```
im3 = multibandread(filename, [rows cols bands], ...
    'double', 0, 'bil', 'ieee-le')
```

Delete the file created in this example.

```
delete(filename);
```

## Example 2

Read int16 BIL data from the FITS file `tst0012.fits`, starting at byte 74880.

```
im4 = multibandread('tst0012.fits', [31 73 5], ...
    'int16', 74880, 'bil', 'ieee-be', ...
    {'Band', 'Range', [1 3]} );
im5 = double(im4)/max(max(max(im4)));
imagesc(im5);
```

## See Also

`fread`, `fwrite`, `imread`, `memmapfile`, `multibandwrite`

# multibandwrite

---

**Purpose** Write band-interleaved data to file

**Syntax** `multibandwrite(data,filename,interleave)`  
`multibandwrite(data,filename,interleave,start,totalsize)`  
`multibandwrite(...,param,value...)`

**Description** `multibandwrite(data,filename,interleave)` writes `data`, a two- or three-dimensional numeric or logical array, to the binary file specified by `filename`. The `filename` input is a string enclosed in single quotes. The length of the third dimension of `data` determines the number of bands written to the file. The bands are written to the file in the form specified by `interleave`. See “Interleave Methods” on page 2-2398 for more information about this argument.

If `filename` already exists, `multibandwrite` overwrites it unless you specify the optional `offset` parameter. See the last alternate syntax for `multibandwrite` for information about other optional parameters.

`multibandwrite(data,filename,interleave,start,totalsize)` writes `data` to the binary file `filename` in chunks. In this syntax, `data` is a subset of the complete data set.

`start` is a 1-by-3 array [`firstrow firstcolumn firstband`] that specifies the location to start writing data. `firstrow` and `firstcolumn` specify the location of the upper left image pixel. `firstband` gives the index of the first band to write. For example, `data(I,J,K)` contains the data for the pixel at [`firstrow+I-1, firstcolumn+J-1`] in the (`firstband+K-1`)-th band.

`totalsize` is a 1-by-3 array, [`totalrows,totalcolumns,totalbands`], which specifies the full, three-dimensional size of the data to be written to the file.

---

**Note** In this syntax, you must call `multibandwrite` multiple times to write all the data to the file. The first time it is called, `multibandwrite` writes the complete file, using the fill value for all values outside the data subset. In each subsequent call, `multibandwrite` overwrites these fill values with the data subset in `data`. The parameters `filename`, `interleave`, `offset`, and `totalsize` must remain constant throughout the writing of the file.

---

`multibandwrite(..., param, value...)` writes the multiband data to a file, specifying any of these optional parameter/value pairs.

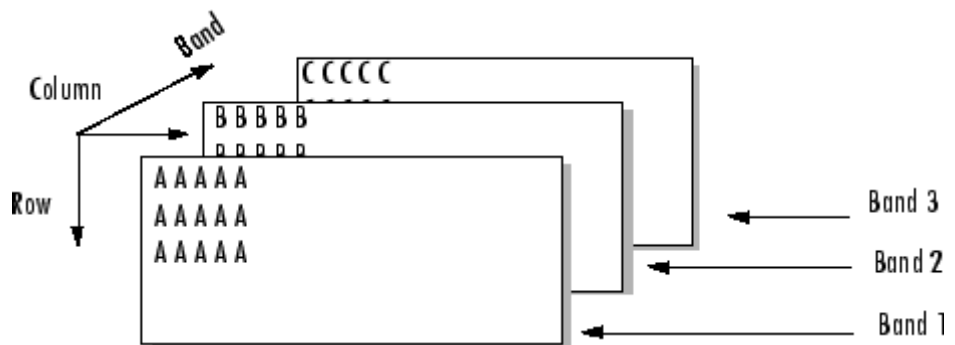
Parameter	Description
'precision'	String specifying the form and size of each element written to the file. See the help for <code>fwrite</code> for a list of valid values. The default precision is the class of the data.
'offset'	The number of bytes to skip before the first data element. If the file does not already exist, <code>multibandwrite</code> writes ASCII null values to fill the space. To specify a different fill value, use the parameter <code>'fillvalue'</code> .  This option is useful when you are writing a header to the file before or after writing the data. When writing the header to the file after the data is written, open the file with <code>fopen</code> using <code>'r+'</code> permission.

# multibandwrite

Parameter	Description
'machfmt'	String to control the format in which the data is written to the file. Typical values are 'ieee-le' for little endian and 'ieee-be' for big endian. See the help for <code>fopen</code> for a complete list of available formats. The default machine format is the local machine format.
'fillvalue'	A number specifying the value to use in place of missing data. 'fillvalue' can be a single number, specifying the fill value for all missing data, or a 1-by-Number-of-bands vector of numbers specifying the fill value for each band. This value is used to fill space when data is written in chunks.

## Interleave Methods

`interleave` is a string that specifies how `multibandwrite` interleaves the bands as it writes data to the file. If data is two-dimensional, `multibandwrite` ignores the `interleave` argument. The following table lists the supported methods and uses this example multiband file to illustrate each method.



Supported methods of interleaving bands include those listed below.

Method	String	Description	Example
Band-Interleaved-by-Line	'bil'	Write an entire row from each band	AAAAABBBBBCCCC AAAAABBBBBCCCC AAAAABBBBBCCCC
Band-Interleaved-by-Pixel	'bip'	Write a pixel from each band	ABCABCABCABC...
Band-Sequential	'bsq'	Write each band in its entirety	AAAAA AAAAA AAAAA BBBBB BBBBB BBBBB CCCCC CCCCC CCCCC

## Examples

---

**Note** To run these examples successfully, you must be in a writable directory.

---

### Example 1

Write all data (interleaved by line) to the file in one call.

```
data = reshape(uint16(1:600), [10 20 3]);
multibandwrite(data, 'data.bil', 'bil');
```

# multibandwrite

---

## Example 2

Write the bands (interleaved by pixel) to the file in separate calls.

```
totalRows = size(data, 1);
totalColumns = size(data, 2);
totalBands = size(data, 3);
for i = 1:totalBands
    bandData = data(:, :, i);
    multibandwrite(bandData, 'data.bip', 'bip', [1 1 i],...
        [totalColumns, totalRows, totalBands]);
end
```

## Example 3

Write a single-band tiled image with one call for each tile. This is only useful if a subset of each band is available at each call to `multibandwrite`.

```
numBands = 1;
dataDims = [1024 1024 numBands];
data = reshape(uint32(1:(1024 * 1024 * numBands)), dataDims);

for band = 1:numBands
    for row = 1:2
        for col = 1:2

            subsetRows = ((row - 1) * 512 + 1):(row * 512);
            subsetCols = ((col - 1) * 512 + 1):(col * 512);

            upperLeft = [subsetRows(1), subsetCols(1), band];
            multibandwrite(data(subsetRows, subsetCols, band), ...
                'banddata.bsq', 'bsq', upperLeft, dataDims);

        end
    end
end
```

**See Also**      multibandread, fwrite, fread

# munlock

---

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

**Syntax**

```
munlock
munlock fun
munlock('fun')
```

**Description** `munlock` unlocks the currently running M-file or MEX-file in memory so that subsequent `clear` functions can remove it.

`munlock fun` unlocks the M-file or MEX-file named `fun` from memory. By default, these files are unlocked so that changes to the file are picked up. Calls to `munlock` are needed only to unlock M-files or MEX-files that have been locked with `mlock`.

`munlock('fun')` is the function form of `munlock`.

**Examples** The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
.
.
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked testfun
ans =
    1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock testfun

mislocked testfun
ans =
```



0

**See Also** mlock, mislocked, persistent

# namelengthmax

---

**Purpose** Maximum identifier length

**Syntax** `len = namelengthmax`

**Description** `len = namelengthmax` returns the maximum length allowed for MATLAB identifiers. MATLAB identifiers are

- Variable names
- Function and subfunction names
- Structure fieldnames
- Object names
- M-file names
- MEX-file names
- MDL-file names

Rather than hard-coding a specific maximum name length into your programs, use the `namelengthmax` function. This saves you the trouble of having to update these limits should the identifier length change in some future MATLAB release.

**Examples** Call `namelengthmax` to get the maximum identifier length:

```
maxid = namelengthmax
maxid =
    63
```

**See Also** `isvarname`, `genvarname`

**Purpose** Not-a-Number

**Syntax** NaN

**Description** NaN returns the IEEE arithmetic representation for Not-a-Number (NaN). These result from operations which have undefined numerical results.

NaN( 'double' ) is the same as NaN with no inputs.

NaN( 'single' ) is the single precision representation of NaN.

NaN( n ) is an n-by-n matrix of NaNs.

NaN( m, n ) or NaN( [ m, n ] ) is an m-by-n matrix of NaNs.

NaN( m, n, p, ... ) or NaN( [ m, n, p, ... ] ) is an m-by-n-by-p-by-... array of NaNs.

---

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

---

NaN( ... , classname ) is an array of NaNs of class specified by classname. classname must be either 'single' or 'double'.

**Examples** These operations produce NaN:

- Any arithmetic operation on a NaN, such as `sqrt(NaN)`
- Addition or subtraction, such as magnitude subtraction of infinities as `(+Inf)+(-Inf)`
- Multiplication, such as `0*Inf`
- Division, such as `0/0` and `Inf/Inf`
- Remainder, such as `rem(x,y)` where y is zero or x is infinity

# NaN

---

## Remarks

Because two NaNs are not equal to each other, logical operations involving NaNs always return false, except `~=` (not equal). Consequently,

```
NaN ~= NaN
ans =
     1
NaN == NaN
ans =
     0
```

and the NaNs in a vector are treated as different unique elements.

```
unique([1 1 NaN NaN])
ans =
     1 NaN NaN
```

Use the `isnan` function to detect NaNs in an array.

```
isnan([1 1 NaN NaN])
ans =
     0     0     1     1
```

## See Also

`Inf`, `isnan`

## Purpose

Validate number of input arguments

## Syntax

```
msgstring = nargchk(minargs, maxargs, numargs)
msgstring = nargchk(minargs, maxargs, numargs, 'string')
msgstruct = nargchk(minargs, maxargs, numargs, 'struct')
```

## Description

Use `nargchk` inside an M-file function to check that the desired number of input arguments is specified in the call to that function.

`msgstring = nargchk(minargs, maxargs, numargs)` returns an error message string `msgstring` if the number of inputs specified in the call `numargs` is less than `minargs` or greater than `maxargs`. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty matrix.

It is common to use the `nargin` function to determine the number of input arguments specified in the call.

`msgstring = nargchk(minargs, maxargs, numargs, 'string')` is essentially the same as the command shown above, as `nargchk` returns a string by default.

`msgstruct = nargchk(minargs, maxargs, numargs, 'struct')` returns an error message structure `msgstruct` instead of a string. The fields of the return structure contain the error message string and a message identifier. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty structure.

When too few inputs are supplied, the message string and identifier are

```
message: 'Not enough input arguments.'
identifier: 'MATLAB:nargchk:notEnoughInputs'
```

When too many inputs are supplied, the message string and identifier are

```
message: 'Too many input arguments.'
identifier: 'MATLAB:nargchk:tooManyInputs'
```

# nargchk

---

## Remarks

nargchk is often used together with the error function. The error function accepts either type of return value from nargchk: a message string or message structure. For example, this command provides the error function with a message string and identifier regarding which error was caught:

```
error(nargchk(2, 4, nargin, 'struct'))
```

If nargchk detects no error, it returns an empty string or structure. When nargchk is used with the error function, as shown here, this empty string or structure is passed as an input to error. When error receives an empty string or structure, it simply returns and no error is generated.

## Examples

Given the function foo,

```
function f = foo(x, y, z)
    error(nargchk(2, 3, nargin))
```

Then typing foo(1) produces

```
Not enough input arguments.
```

## See Also

nargoutchk, nargin, nargout, varargin, varargout, error

**Purpose** Number of function arguments

**Syntax** nargin  
nargin(fun)  
nargsout  
nargsout(fun)

**Description** In the body of a function M-file, nargin and nargsout indicate how many input or output arguments, respectively, a user has supplied. Outside the body of a function M-file, nargin and nargsout indicate the number of input or output arguments, respectively, for a given function. The number of arguments is negative if the function has a variable number of arguments.

nargin returns the number of input arguments specified for a function.

nargin(fun) returns the number of declared inputs for the function fun. If the function has a variable number of input arguments, nargin returns a negative value. fun may be the name of a function, or the name of “Function Handles” that map to specific functions.

nargsout returns the number of output arguments specified for a function.

nargsout(fun) returns the number of declared outputs for the function fun. fun may be the name of a function, or the name of “Function Handles” that map to specific functions.

**Examples** This example shows portions of the code for a function called myplot, which accepts an optional number of input and output arguments:

```
function [x0, y0] = myplot(x, y, npts, angle, subdiv)
% MYPLOT Plot a function.
% MYPLOT(x, y, npts, angle, subdiv)
%     The first two input arguments are
%     required; the other three have default values.
...
if nargin < 5, subdiv = 20; end
if nargin < 4, angle = 10; end
```

## nargin, nargsout

---

```
if nargin < 3, npts = 25; end
...
if nargsout == 0
    plot(x, y)
else
    x0 = x;
    y0 = y;
end
```

### See Also

inputname, varargin, varargout, nargchk, nargsoutchk



## Purpose

Validate number of output arguments

## Syntax

```
msgstring = nargoutchk(minargs, maxargs, numargs)
msgstring = nargoutchk(minargs, maxargs, numargs, 'string')
msgstruct = nargoutchk(minargs, maxargs, numargs, 'struct')
```

## Description

Use `nargoutchk` inside an M-file function to check that the desired number of output arguments is specified in the call to that function.

`msgstring = nargoutchk(minargs, maxargs, numargs)` returns an error message string `msgstring` if the number of outputs specified in the call, `numargs`, is less than `minargs` or greater than `maxargs`. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargoutchk` returns an empty matrix.

It is common to use the `nargout` function to determine the number of output arguments specified in the call.

`msgstring = nargoutchk(minargs, maxargs, numargs, 'string')` is essentially the same as the command shown above, as `nargoutchk` returns a string by default.

`msgstruct = nargoutchk(minargs, maxargs, numargs, 'struct')` returns an error message structure `msgstruct` instead of a string. The fields of the return structure contain the error message string and a message identifier. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargoutchk` returns an empty structure.

When too few outputs are supplied, the message string and identifier are

```
message: 'Not enough output arguments.'
identifier: 'MATLAB:nargoutchk:notEnoughOutputs'
```

When too many outputs are supplied, the message string and identifier are

```
message: 'Too many output arguments.'
identifier: 'MATLAB:nargoutchk:tooManyOutputs'
```

# nargoutchk

---

## Remarks

nargoutchk is often used together with the error function. The error function accepts either type of return value from nargoutchk: a message string or message structure. For example, this command provides the error function with a message string and identifier regarding which error was caught:

```
error(nargoutchk(2, 4, nargout, 'struct'))
```

If nargoutchk detects no error, it returns an empty string or structure. When nargoutchk is used with the error function, as shown here, this empty string or structure is passed as an input to error. When error receives an empty string or structure, it simply returns and no error is generated.

## Examples

You can use nargoutchk to determine if an M-file has been called with the correct number of output arguments. This example uses nargout to return the number of output arguments specified when the function was called. The function is designed to be called with one, two, or three output arguments. If called with no arguments or more than three arguments, nargoutchk returns an error message:

```
function [s, varargout] = mysize(x)
msg = nargoutchk(1, 3, nargout);
if isempty(msg)
    nout = max(nargout, 1) - 1;
    s = size(x);
    for k = 1:nout, varargout(k) = {s(k)}; end
else
    disp(msg)
end
```

## See Also

nargchk, nargout, nargin, varargout, varargin, error

**Purpose** Convert numeric bytes to Unicode characters

**Syntax**  
`unicodestr = native2unicode(bytes)`  
`unicodestr = native2unicode(bytes, encoding)`

**Description** `unicodestr = native2unicode(bytes)` takes a vector containing numeric values in the range [0,255] and converts these values as a stream of 8-bit bytes to Unicode characters. The stream of bytes is assumed to be in the MATLAB default character encoding scheme. Return value `unicodestr` is a char vector that has the same general array shape as `bytes`.

`unicodestr = native2unicode(bytes, encoding)` does the conversion with the assumption that the byte stream is in the character encoding scheme specified by the string `encoding`. `encoding` must be the empty string ('') or a name or alias for an encoding scheme. Some examples are 'UTF-8', 'latin1', 'US-ASCII', and 'Shift\_JIS'. For common names and aliases, see the Web site <http://www.iana.org/assignments/character-sets>. If `encoding` is unspecified or is the empty string (''), the MATLAB default encoding scheme is used.

---

**Note** If `bytes` is a char vector, it is returned unchanged.

---

**Examples** This example begins with a vector of bytes in an unknown character encoding scheme. The user-written function `detect_encoding` determines the encoding scheme. If successful, it returns the encoding scheme name or alias as a string. If unsuccessful, it throws an error. The example calls `native2unicode` to convert the bytes to Unicode characters.

```
try
    enc = detect_encoding(bytes);
    str = native2unicode(bytes, enc);
    disp(str);
```

# native2unicode

---

```
catch
    rethrow(lasterror);
end
```

Note that the computer must be configured to display text in a language represented by the detected encoding scheme for the output of `disp(str)` to be correct.

## See Also

`unicode2native`

<b>Purpose</b>	Binomial coefficient or all combinations																				
<b>Syntax</b>	$C = \text{nchoosek}(n, k)$ $C = \text{nchoosek}(v, k)$																				
<b>Description</b>	<p><math>C = \text{nchoosek}(n, k)</math> where <math>n</math> and <math>k</math> are nonnegative integers, returns <math>n!/((n-k)! k!)</math>. This is the number of combinations of <math>n</math> things taken <math>k</math> at a time.</p> <p><math>C = \text{nchoosek}(v, k)</math>, where <math>v</math> is a row vector of length <math>n</math>, creates a matrix whose rows consist of all possible combinations of the <math>n</math> elements of <math>v</math> taken <math>k</math> at a time. Matrix <math>C</math> contains <math>n!/((n-k)! k!)</math> rows and <math>k</math> columns.</p> <p>Inputs <math>n</math>, <math>k</math>, and <math>v</math> support classes of <code>float double</code> and <code>float single</code>.</p>																				
<b>Examples</b>	<p>The command <code>nchoosek(2:2:10,4)</code> returns the even numbers from two to ten, taken four at a time:</p> <table><tbody><tr><td>2</td><td>4</td><td>6</td><td>8</td></tr><tr><td>2</td><td>4</td><td>6</td><td>10</td></tr><tr><td>2</td><td>4</td><td>8</td><td>10</td></tr><tr><td>2</td><td>6</td><td>8</td><td>10</td></tr><tr><td>4</td><td>6</td><td>8</td><td>10</td></tr></tbody></table>	2	4	6	8	2	4	6	10	2	4	8	10	2	6	8	10	4	6	8	10
2	4	6	8																		
2	4	6	10																		
2	4	8	10																		
2	6	8	10																		
4	6	8	10																		
<b>Limitations</b>	<p>When <math>C = \text{nchoosek}(n, k)</math> has a large coefficient, a warning will be produced indicating possible inexact results. In such cases, the result is only accurate to 15 digits for double-precision inputs, or 8 digits for single-precision inputs.</p> <p><math>C = \text{nchoosek}(v, k)</math> is only practical for situations where <math>n</math> is less than about 15.</p>																				
<b>See Also</b>	<code>perms</code>																				

# ndgrid

## Purpose

Generate arrays for N-D functions and interpolation

## Syntax

```
[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)  
[X1,X2,...] = ndgrid(x)
```

## Description

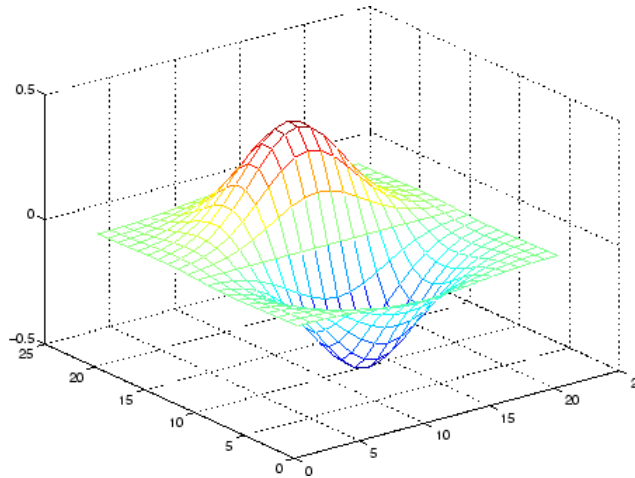
`[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)` transforms the domain specified by vectors `x1,x2,x3...` into arrays `X1,X2,X3...` that can be used for the evaluation of functions of multiple variables and multidimensional interpolation. The *i*th dimension of the output array `Xi` are copies of elements of the vector `xi`.

`[X1,X2,...] = ndgrid(x)` is the same as `[X1,X2,...] = ndgrid(x,x,...)`.

## Examples

Evaluate the function  $x_1 e^{-x_1^2 - x_2^2}$  over the range  $-2 < x_1 < 2, -2 < x_2 < 2$ .

```
[X1,X2] = ndgrid(-2:.2:2, -2:.2:2);  
Z = X1 .* exp(-X1.^2 - X2.^2);  
mesh(Z)
```



**Remarks**

The `ndgrid` function is like `meshgrid` except that the order of the first two input arguments are switched. That is, the statement

```
[X1,X2,X3] = ndgrid(x1,x2,x3)
```

produces the same result as

```
[X2,X1,X3] = meshgrid(x2,x1,x3)
```

Because of this, `ndgrid` is better suited to multidimensional problems that aren't spatially based, while `meshgrid` is better suited to problems in two- or three-dimensional Cartesian space.

**See Also**

`meshgrid`, `interp`

# ndims

---

**Purpose**            Number of array dimensions

**Syntax**            `n = ndims(A)`

**Description**        `n = ndims(A)` returns the number of dimensions in the array `A`. The number of dimensions in an array is always greater than or equal to 2. Trailing singleton dimensions are ignored. A singleton dimension is any dimension for which `size(A,dim) = 1`.

**Algorithm**            `ndims(x)` is `length(size(x))`.

**See Also**            `size`



**Purpose** Test for inequality

**Syntax** `A ~= B`  
`ne(A, B)`

**Description** `A ~= B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (true) where `A` and `B` are unequal, or logical 0 (false) where they are 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.

`ne(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 not equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(magic(3), 2, 2);
```

```
A ~= B
ans =
     1     0     0     1     1     1
     0     1     0     1     1     1
     1     0     0     1     1     1
     0     1     1     1     1     1
     1     0     1     1     1     1
```

**ne**

---

0 1 1 1 1 1

**See Also**

eq, le, ge, lt, gt, relational operators

<b>Purpose</b>	Compare MException objects for inequality
<b>Syntax</b>	<code>eObj1 ~= eObj2</code>
<b>Description</b>	<code>eObj1 ~= eObj2</code> tests MException objects <code>eObj1</code> and <code>eObj2</code> for inequality, returning logical 1 (true) if the two objects are not identical, otherwise returning logical 0 (false).
<b>See Also</b>	<code>try</code> , <code>catch</code> , <code>error</code> , <code>assert</code> , <code>MException</code> , <code>isequal(MException)</code> , <code>eq(MException)</code> , <code>getReport(MException)</code> , <code>disp(MException)</code> , <code>throw(MException)</code> , <code>rethrow(MException)</code> , <code>throwAsCaller(MException)</code> , <code>addCause(MException)</code> , <code>last(MException)</code>

## Purpose

Summary of MATLAB Network Common Data Form (netCDF) capabilities

## Description

MATLAB provides access to more than 30 functions in the Network Common Data Form (netCDF) interface. This interface provides an API that you can use to enable reading data from and writing data to netCDF files (known as *datasets* in netCDF terminology).

To use these MATLAB functions, you must be familiar with the netCDF C interface. To get this information, consult the [NetCDF C Interface Guide](#).

---

**Note** For information about MATLAB support for the Common Data Format (CDF), which is a completely separate, incompatible format, see “Common Data Format (CDF) Files”.

---

In most cases, the syntax of the MATLAB function is similar to the syntax of the netCDF library function. The functions are implemented as a package called `netcdf`. To use these functions, prefix the function name with package name `netcdf`. For example, to call the netCDF library routine used to open existing netCDF files, use the following MATLAB syntax:

```
ncid = netcdf.open( ncfile, mode );
```

## netCDF Library Functions

The following tables list all of the functions in the MATLAB netCDF package, grouped by category.

### File Operations

<code>netcdf</code>	Summary of MATLAB Network Common Data Form (netCDF) capabilities
<code>netcdf.abort</code>	Revert recent netCDF file definitions

---

<code>netcdf.close</code>	Close netCDF file
<code>netcdf.create</code>	Create new netCDF dataset
<code>netcdf.endDef</code>	End netCDF file define mode
<code>netcdf.getConstant</code>	Return numeric value of named constant
<code>netcdf.getConstantNames</code>	Return list of constants known to netCDF library
<code>netcdf.inq</code>	Return information about netCDF file
<code>netcdf.inqLibVers</code>	Return netCDF library version information
<code>netcdf.open</code>	Open netCDF file
<code>netcdf.reDef</code>	Put open netCDF file into define mode
<code>netcdf.setDefaultFormat</code>	Change default netCDF file format
<code>netcdf.setFill</code>	Set netCDF fill mode
<code>netcdf.sync</code>	Synchronize netCDF file to disk

**Dimensions**

<code>netcdf.defDim</code>	Create netCDF dimension
<code>netcdf.inqDim</code>	Return netCDF dimension name and length
<code>netcdf.inqDimID</code>	Return dimension ID
<code>netcdf.renameDim</code>	Change name of netCDF dimension

## Variables

<code>netcdf.defVar</code>	Create netCDF variable
<code>netcdf.getVar</code>	Return data from netCDF variable
<code>netcdf.inqVar</code>	Return information about variable
<code>netcdf.inqVarID</code>	Return ID associated with variable name
<code>netcdf.putVar</code>	Write data to netCDF variable
<code>netcdf.renameVar</code>	Change name of netCDF variable

## Attributes

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

<b>Purpose</b>	Revert recent netCDF file definitions
<b>Syntax</b>	<code>netcdf.abort(ncid)</code>
<b>Description</b>	<p><code>netcdf.abort(ncid)</code> reverts a netCDF file to its previous state, backing out any definitions made since the file last entered define mode. A file enters define mode when you create it (using <code>netcdf.create</code>) or when you explicitly enter define mode (using <code>netcdf.redef</code>). Once you leave define mode (using <code>netcdf.endDef</code>), you cannot revert the definitions you made while in define mode. <code>ncid</code> is a netCDF file identifier returned by <code>netcdf.create</code> or <code>netcdf.open</code>. A call to <code>netcdf.abort</code> closes the file.</p> <p>This function corresponds to the <code>nc_abort</code> function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the <a href="#">NetCDF C Interface Guide</a> for more information.</p>
<b>Examples</b>	<p>This example creates a new file, performs an operation on the file, and then reverts the file back to its original state. To run this example, you must have write permission in your current directory.</p> <pre>% Create a netCDF file ncid = netcdf.create('foo.nc','NC_NOCLlobber');  % Perform an operation, such as defining a dimension. dimid = netcdf.defDim(ncid, 'lat', 50);  % Revert the file back to its previous state. netcdf.abort(ncid)  % Verify that the file is now closed. dimid = netcdf.defDim(ncid, 'lat', 50); % should fail ??? Error using ==&gt; netcdf.lib NetCDF: Not a valid ID  Error in ==&gt; defDim at 22</pre>

## netcdf.abort

---

```
dimid = netcdflib('def_dim', ncid,dimname,dimlen);
```

### **See Also**

`netcdf.create`, `netcdf.endDef`, `netcdf.reDef`



**Purpose** Close netCDF file

**Syntax** `netcdf.close(ncid)`

**Description** `netcdf.close(ncid)` terminates access to the netCDF file identified by `ncid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_close` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example creates a new netCDF file, and then closes the file. You must have write permission in your current directory to run this example.

```
ncid = netcdf.open('foo.nc','NC_WRITE')  
  
netcdf.close(ncid)
```

**See Also** `netcdf.create`, `netCDF.open`

# netcdf.copyAtt

---

**Purpose** Copy attribute to new location

**Syntax** netcdf.copyAtt(ncid\_in,varid\_in,attname,ncid\_out,varid\_out)

**Description** netcdf.copyAtt(ncid\_in,varid\_in,attname,ncid\_out,varid\_out) copies an attribute from one variable to another, possibly across files. ncid\_in and ncid\_out are netCDF file identifiers returned by netcdf.create or netcdf.open. varid\_in identifies the variable with an attribute that you want to copy. varid\_out identifies the variable to which you want to associate a copy of the attribute.

This function corresponds to the nc\_copy\_att function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example makes a copy of the attribute associated with the first variable in the netCDF example file, example.nc, in a new file. To run this example, you must have write permission in your current directory.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get identifier for a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Create new netCDF file.
ncid2 = netcdf.create('foo.nc','NC_NOCLlobber');

% Define a dimension in the new file.
dimid2 = netcdf.defDim(ncid2,'x',50);

% Define a variable in the new file.
varid2 = netcdf.defVar(ncid2,'myvar','double',dimid2);

% Copy the attribute named 'description' into the new file,
% associating the attribute with the new variable.
```

```
netcdf.copyAtt(ncid,varid,'description',ncid2,varid2);  
%  
% Check the name of the attribute in new file.  
attname = netcdf.inqAttName(ncid2,varid2,0)  
  
attname =  
  
description
```

## See Also

netcdf.inqAtt, netcdf.inqAttID, netcdf.inqAttName,  
netcdf.putAtt, netcdf.renameAtt

# netcdf.create

---

**Purpose** Create new netCDF dataset

**Syntax**

```
ncid = netcdf.create(filename, mode)
[chunksize_out, ncid]=netcdf.create(filename,mode,initSZ,
    chunksize)
```

**Description** `ncid = netcdf.create(filename, mode)` creates a new netCDF file according to the file creation mode. The return value, `ncid`, is a file ID. The type of access is described by the mode parameter, which can have any of the following values.

Value	Description
'NC_NOCLOBBER'	Prevent overwriting of existing file with the
'NC_SHARE'	Allow synchronous file updates.
'NC_64BIT_OFFSET'	Allow easier creation of files and variables which are larger than two gigabytes.

---

**Note** You can specify the mode as a numeric value, retrieved using the `netcdf.getConstant` function. To specify more than one mode, use a bitwise-OR of the numeric values of the modes.

---

`[chunksize_out, ncid]=netcdf.create(filename,mode,initSZ,chunksize)` creates a new netCDF file, but with additional performance tuning parameters. `initSZ` sets the initial size of the file. `chunksize` can affect I/O performance. The actual value chosen by the netCDF library might not correspond to the input value.

This function corresponds to the `nc_create` and `nc__create` functions in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

## Examples

This example creates a netCDF dataset named `foo.nc`, only if no other file with the same name exists in the current directory. To run this example, you must have write permission in your current directory.

```
ncid = netcdf.create('foo.nc', 'NC_NOCLlobber');
```

## See Also

`netcdf.getConstant`, `netcdf.open`

# netcdf.defDim

---

**Purpose** Create netCDF dimension

**Syntax** `dimid = netcdf.defDim(ncid,dimname,dimlen)`

**Description** `dimid = netcdf.defDim(ncid,dimname,dimlen)` creates a new dimension in the netCDF file specified by `ncid`, where `dimname` is a character string that specifies the name of the dimension and `dimlen` is a numeric value that specifies its length. To define an unlimited dimension, specify the predefined constant `'NC_UNLIMITED'` for `dimlen`, using `netcdf.getConstant` to retrieve the value.

`netcdf.defDim` returns `dimid`, a numeric ID corresponding to the new dimension.

This function corresponds to the `nc_def_dim` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example creates a new file and defines two dimensions in the file. One dimension is an unlimited dimension. To run this example, you must have write permission in your current directory.

```
% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_NOGLOBBER')

% Define a dimension.
latdimID = netcdf.defDim(ncid,'lat',50);

% Define an unlimited dimension.
lonDimID = netcdf.defDim(ncid,'lon',netcdf.getConstant('NC_UNLIMITED'))
```

**See Also** `netcdf.getConstant`

<b>Purpose</b>	Create netCDF variable
<b>Syntax</b>	<code>varid = netcdf.defVar(ncid,varname,xtype,dimids)</code>
<b>Description</b>	<p><code>varid = netcdf.defVar(ncid,varname,xtype,dimids)</code> creates a new variable in the dataset identified by <code>ncid</code>.</p> <p><code>varname</code> is a character string that specifies the name of the variable. <code>xtype</code> can be either a character string specifying the data type of the variable, such as <code>'double'</code>, or it can be the numeric equivalent returned by the <code>netcdf.getConstant</code> function. <code>dimids</code> specifies a list of dimension IDs.</p> <p><code>netcdf.defVar</code> returns <code>varid</code>, a numeric identifier for the new variable.</p> <p>This function corresponds to the <code>nc_def_var</code> function in the netCDF library C API. Because MATLAB uses FORTRAN-style ordering, the fastest-varying dimension comes first and the slowest comes last. Any unlimited dimension is therefore last in the list of dimension IDs. This ordering is the reverse of that found in the C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the <a href="#">NetCDF C Interface Guide</a> for more information.</p>
<b>Examples</b>	<p>This example creates a new netCDF file, defines a dimension in the file, and then defines a variable on that dimension. (In netCDF files, you must create a dimension before you can create a variable.) To run this example, you must have write permission in your current directory.</p> <pre>% Create netCDF file. ncid = netcdf.create('foo.nc','NC_NOGLOBBBER'); % % Define a dimension in the new file. dimid = netcdf.defDim(ncid,'x',50);  % Define a variable in the new file. varid = netcdf.defVar(ncid,'myvar','double',dimid)</pre>

# netcdf.defVar

---

## **See Also**

`netCDF.getConstant`, `netCDF.inqVar`, `netCDF.putVar`



**Purpose** Delete netCDF attribute

**Syntax** netcdf.delAtt(ncid,varid,attName)

**Description** netcdf.delAtt(ncid,varid,attName) deletes the attribute identified by the text string attName.

ncid is a netCDF file identifier returned by netcdf.create or netcdf.open.

varid is a numeric value that identifies the variable. To delete a global attribute, use netcdf.getConstant('GLOBAL') for the varid. You must be in define mode to delete an attribute.

This function corresponds to the nc\_del\_att function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example opens a local copy of the example netCDF file included with MATLAB, example.nc.

```
% Open a netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Determine number of global attributes in file.
[numdims numvars numatts unlimdimID] = netcdf.inq(ncid);

numatts =

    1

% Get name of attribute; it is needed for deletion.
attname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)

% Put file in define mode to delete an attribute.
netcdf.reDef(ncid);
```

# netcdf.delAtt

---

```
% Delete the global attribute in the netCDF file.
netcdf.delAtt(ncid,netcdf.getConstant('GLOBAL'),attname);

% Verify that the global attribute was deleted.
[numdims numvars numatts unlimdimID] = netcdf.inq(ncid);

numatts =

    0
```

## See Also

`netcdf.getConstant`, `netcdf.inqAttName`

<b>Purpose</b>	End netCDF file define mode
<b>Syntax</b>	<pre>netcdf.endDef(ncid) netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)</pre>
<b>Description</b>	<p><code>netcdf.endDef(ncid)</code> takes a netCDF file out of define mode and into data mode. <code>ncid</code> is a netCDF file identifier returned by <code>netcdf.create</code> or <code>netcdf.open</code>.</p> <p><code>netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)</code> takes a netCDF file out of define mode, specifying four additional performance tuning parameters. For example, one reason for using the performance parameters is to reserve extra space in the netCDF file header using the <code>h_minfree</code> parameter:</p> <pre>ncid = netcdf.endDef(ncid,20000,4,0,4);</pre> <p>This reserves 20,000 bytes in the header, which can be used later when adding attributes. This can be extremely efficient when working with very large files. To understand how to use these performance tuning parameters, see the netCDF library documentation.</p> <p>This function corresponds to the <code>nc_enddef</code> and <code>nc__enddef</code> functions in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the <a href="#">NetCDF C Interface Guide</a> for more information.</p>
<b>Examples</b>	<p>When you create a file using <code>netcdf.create</code>, the functions opens the file in define mode. This example uses <code>netcdf.endDef</code> to take the file out of define mode.</p> <pre>% Create a netCDF file. ncid = netcdf.create('foo.c','NC_NOGLOBBER');  % Define a dimension. dimid = netcdf.defDim(ncid, 'lat', 50);  % Leave define mode. netcdf.endDef(ncid)</pre>

## netcdf.endDef

---

```
% Test if still in define mode.
dimid = netcdf.defDim(ncid, 'lon', 50); % should fail
??? Error using ==> netcdf.lib
NetCDF: Operation not allowed in data mode

Error in ==> defDim at 22
dimid = netcdf.lib('def_dim', ncid,dimname,dimlen);
```

### See Also

netcdf.create, netcdf.reDef

**Purpose** Return netCDF attribute

**Syntax**

```
attrvalue = netcdf.getAtt(ncid,varid,attname)
attrvalue = netcdf.getAtt(ncid,varid,attname,output_datatype)
```

**Description** `attrvalue = netcdf.getAtt(ncid,varid,attname)` returns `attrvalue`, the value of the attribute specified by the text string `attname`. When it chooses the data type of `attrvalue`, MATLAB attempts to match the netCDF class of the attribute. For example, if the attribute has the netCDF data type `NC_INT`, MATLAB uses the `int32` class for the output data. If an attribute has the netCDF data type `NC_BYTE`, the class of the output data is `int8` value.

`attrvalue = netcdf.getAtt(ncid,varid,attname,output_datatype)` returns `attrvalue`, the value of the attribute specified by the text string `attname`, using the output class specified by `output_datatype`. You can specify any of the following strings for the output data type.

'int'	'double'	'int16'
'short'	'single'	'int8'
'float'	'int32'	'uint8'

This function corresponds to several attribute I/O functions in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example opens the example netCDF file included with MATLAB, `example.nc`, and gets the value of the attribute associated with the first variable. The example also gets the value of the global variable in the file.

```
% Open a netCDF file.
ncid = netcdf.open('example.nc','NC_NOWRITE');
```

## netcdf.getAtt

---

```
% Get name of first variable.
[varname vartype vardimIDs varatts] = netcdf.inqVar(ncid,0);

% Get ID of variable, given its name.
varid = netcdf.inqVarID(ncid,varname);

% Get attribute name, given variable id.
attname = netcdf.inqAttName(ncid,varid,0);

% Get value of attribute.
attval = netcdf.getAtt(ncid,varid,attname);

% Get name of global attribute
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0);

% Get value of global attribute.
gattval = netcdf.getAtt(ncid,netcdf.getConstant('NC_GLOBAL'),gattname)

gattval =

09-Jun-2008
```

### See Also

netcdf.inqAtt,netcdf.putAtt

**Purpose** Return numeric value of named constant

**Syntax** `val = netcdf.getConstant(param_name)`

**Description** `val = netcdf.getConstant(param_name)` returns the numeric value corresponding to the name of a constant defined by the netCDF library. For example, `netcdf.getConstant('NC_NOGLOBBER')` returns the numeric value corresponding to the netCDF constant `NC_NOGLOBBER`.

The value for `param_name` can be either upper- or lowercase, and does not need to include the leading three characters `'NC_'`. To retrieve a list of all the names defined by the netCDF library, use the `netcdf.getConstantNames` function.

This function has no direct equivalent in the netCDF C interface. However, to learn about the constants defined by the API, see the [NetCDF C Interface Guide](#).

**Examples** This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Determine contents of the file.
[ndims nvars natts dimm] = netcdf.inq(ncid);

% Get name of global attribute.
% Note: You must use netcdf.getConstant to specify NC_GLOBAL.
attname = netcdf.inqattname(ncid,netcdf.getConstant('NC_GLOBAL'),0)

attname =

creation_date
```

**See Also** `netcdf.getConstantNames`

# netcdf.getConstantNames

---

**Purpose** Return list of constants known to netCDF library

**Syntax** `val = netcdf.getConstantNames(param_name)`

**Description** `val = netcdf.getConstantNames(param_name)` returns a list of names of netCDF library constants, definitions, and enumerations. When these strings are supplied as actual parameters to MATLAB netCDF package functions, the functions automatically convert the constant to the appropriate numeric value.

This MATLAB function has no direct equivalent in the netCDF C interface. However, to learn about the constants defined by the API, see the NetCDF C Interface Guide.

**Examples** `nc_constants = netcdf.getConstantNames`

```
nc_constants =  
  
    'NC2_ERR'  
    'NC_64BIT_OFFSET'  
    'NC_BYTE'  
    'NC_CHAR'  
    'NC_CLOBBER'  
    'NC_DOUBLE'  
    'NC_EBADDIM'  
    'NC_EBADID'  
    'NC_EBADNAME'  
    'NC_EBADTYPE'  
    ...
```

**See Also** `netCDF.getConstantNames`



**Purpose** Return data from netCDF variable

**Syntax**

```
data = netcdf.getVar(ncid,varid)
data = netcdf.getVar(ncid,varid,start)
data = netcdf.getVar(ncid,varid,start,count)
data = netcdf.getVar(ncid,varid,start,count,stride)
data = netcdf.getVar(...,output_type)
```

**Description** `data = netcdf.getVar(ncid,varid)` returns `data`, the value of the variable specified by `varid`. MATLAB attempts to match the class of the output data to netCDF class of the variable.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`data = netcdf.getVar(ncid,varid,start)` returns a single value starting at the specified index, `start`.

`data = netcdf.getVar(ncid,varid,start,count)` returns a contiguous section of a variable. `start` specifies the starting point and `count` specifies the amount of data to return.

`data = netcdf.getVar(ncid,varid,start,count,stride)` returns a subset of a section of a variable. `start` specifies the starting point, `count` specifies the extent of the section, and `stride` specifies which values to return.

`data = netcdf.getVar(...,output_type)` specifies the data type of the return value `data`. For example, to read in an entire integer variable as double precision, use:

```
data=netcdf.getVar(ncid,varid,'double');
```

You can specify any of the following strings for the output data type.

'int'	'double'	'int16'
'short'	'single'	'int8'
'float'	'int32'	'uint8'

# netcdf.getVar

---

This function corresponds to several functions in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

## Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and uses several functions to get the value of a variable.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get the name of the first variable.
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

% Get variable ID of the first variable, given its name.
varid = netcdf.inqVarID(ncid,varname);

% Get the value of the first variable, given its ID.
data = netcdf.getVar(ncid,varid)

data =

    6.0221e+023
```

## See Also

`netcdf.create`, `netcdf.inqVarID`, `netcdf.open`

**Purpose** Return information about netCDF file

**Syntax** [ndims,nvars,ngatts,unlimdimid] = netcdf.inq(ncid)

**Description** [ndims,nvars,ngatts,unlimdimid] = netcdf.inq(ncid) returns the number of dimensions, variables, and global attributes in a netCDF file. The function also returns the ID of the dimension defined with unlimited length, if one exists.

ncid is a netCDF file identifier returned by netcdf.create or netcdf.open. You can call netcdf.inq in either define mode or data mode.

This function corresponds to the nc\_inq function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the NetCDF C Interface Guide for more information.

**Examples** This example opens the example netCDF file included with MATLAB, example.nc, and uses the netcdf.inq function to get information about the contents of the file.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE')

% Get information about the contents of the file.
[numdims, numvars, numglobalatts, unlimdimID] = netcdf.inq(ncid)

numdims =

     4

numvars =

     4
```

```
numglobalatts =
```

```
1
```

```
unlimdimID =
```

```
3
```

**See Also**

`netcdf.create`, `netcdf.open`

<b>Purpose</b>	Return information about netCDF attribute
<b>Syntax</b>	<code>[xtype,attlen] = netcdf.inqAtt(ncid,varid,attname)</code>
<b>Description</b>	<p><code>[xtype,attlen] = netcdf.inqAtt(ncid,varid,attname)</code> returns the data type, <code>xtype</code>, and length, <code>attlen</code>, of the attribute identified by the text string <code>attname</code>.</p> <p><code>ncid</code> is a netCDF file identifier returned by <code>netcdf.create</code> or <code>netcdf.open</code>.</p> <p><code>varid</code> identifies the variable that the attribute is associated with. To get information about a global attribute, specify <code>netcdf.getConstant('NC_GLOBAL')</code> in place of <code>varid</code>.</p> <p>This function corresponds to the <code>nc_inq_att</code> function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the <a href="#">NetCDF C Interface Guide</a> for more information.</p>

**Examples** This example opens the example netCDF file included with MATLAB, `example.nc`, and gets information about an attribute in the file.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NOWRITE');

% Get identifier of a variable in the file, given its name.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Get attribute name, given variable id and attribute number.
attname = netcdf.inqAttName(ncid,varid,0);

% Get information about the attribute.
[xtype,attlen] = netcdf.inqAtt(ncid,varid,'description')

xtype =
```

## netcdf.inqAtt

---

```
attlen =  
  
    31  
  
% Get name of global attribute  
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0);  
  
% Get information about global attribute.  
[gxtype gattlen] = netcdf.inqAtt(ncid,netcdf.getConstant('NC_GLOBAL'),  
  
gxtype =  
  
    2  
  
gattlen =  
  
    11
```

### See Also

netcdf.inqAttID netcdf.inqAttName

## Purpose

Return ID of netCDF attribute

## Syntax

```
attnum = netcdf.inqAttID(ncid,varid,attname)
```

## Description

`attnum = netcdf.inqAttID(ncid,varid,attname)` retrieves `attnum`, the identifier of the attribute specified by the text string `attname`.

`varid` specifies the variable the attribute is associated with.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_attid` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

## Examples

This example opens the netCDF example file included with MATLAB, `example.nc`.

```
% Open the netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get the identifier of a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Retrieve the identifier of the attribute associated with the variable.
attid = netcdf.inqAttID(ncid,varid,'description');
```

## See Also

`netcdf.inqAttnetcdf.inqAttName`

# netcdf.inqAttName

---

**Purpose** Return name of netCDF attribute

**Syntax** `attname = netcdf.inqAttName(ncid,varid,attnum)`

**Description** `attname = netcdf.inqAttName(ncid,varid,attnum)` returns `attname`, a text string specifying the name of an attribute.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` is a numeric identifier of a variable in the file. If you want to get the name of a global attribute in the file, use `netcdf.getConstant('NC_GLOBAL')` in place of `attnum` is a zero-based numeric value specifying the attribute, with 0 indicating the first attribute, 1 the second attribute, and so on.

This function corresponds to the `nc_inq_attname` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get identifier of a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number')

% Get the name of the attribute associated with the variable.
attname = netcdf.inqAttName(ncid,varid,0)

attname =

description

% Get the name of the global attribute associated with the variable.
```



```
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0
gattname =
creation_date
```

### See Also

`netcdf.inqAtt` `netcdf.inqAttID`

# netcdf.inqDim

---

**Purpose** Return netCDF dimension name and length

**Syntax** `[dimname, dimlen] = netcdf.inqDim(ncid,dimid)`

**Description** `[dimname, dimlen] = netcdf.inqDim(ncid,dimid)` returns the name, `dimname`, and length, `dimlen`, of the dimension specified by `dimid`. If `ndims` is the number of dimensions defined for a netCDF file, each dimension has an ID between 0 and `ndims-1`. For example, the dimension identifier of the first dimension is 0, the second dimension is 1, and so on.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_dim` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** The example opens the example netCDF file include with MATLAB, `example.nc`.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name and length of first dimension
[dimname, dimlen] = netcdf.inqDim(ncid,0)

dimname =

x

dimlen =

50
```

**See Also** `netcdf.inqDimID`

<b>Purpose</b>	Return dimension ID
<b>Syntax</b>	<code>dimid = netcdf.inqDimID(ncid,dimname)</code>
<b>Description</b>	<p><code>dimid = netcdf.inqDimID(ncid,dimname)</code> returns <code>dimid</code>, the identifier of the dimension specified by the character string <code>dimname</code>. You can use the <code>netcdf.inqDim</code> function to retrieve the dimension name. <code>ncid</code> is a netCDF file identifier returned by <code>netcdf.create</code> or <code>netcdf.open</code>.</p> <p>This function corresponds to the <code>nc_inq_dimid</code> function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the <a href="#">NetCDF C Interface Guide</a> for more information.</p>
<b>Examples</b>	<p>This example opens the example netCDF file included with MATLAB, <code>example.nc</code>.</p> <pre>% Open netCDF example file. ncid = netcdf.open('example.nc','NC_NOWRITE');  % Get name and length of first dimension [dimname, dimlen] = netcdf.inqDim(ncid,0);  % Retrieve identifier of dimension. dimid = netcdf.inqDimID(ncid,dimname)  dimid =      0</pre>
<b>See Also</b>	<code>netcdf.inqDim</code>

# netcdf.inqLibVers

---

**Purpose** Return netCDF library version information

**Syntax** `libvers = netcdf.inqLibVers`

**Description** `libvers = netcdf.inqLibVers` returns a string identifying the version of the netCDF library.

This function corresponds to the `nc_inq_libvers` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** `libvers = netcdf.inqLibVers`

```
libvers =
```

```
3.6.2
```

**Purpose**

Return information about variable

**Syntax**

```
[varname,xtype,dimids,natts] = netcdf.inqVar(ncid,varid)
```

**Description**

```
[varname,xtype,dimids,natts] = netcdf.inqVar(ncid,varid)
```

returns the name, data type, dimensions IDs, and the number of attributes associated with the variable identified by `varid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_var` function in the netCDF library C API. Because MATLAB uses FORTRAN-style ordering, however, the order of the dimension IDs is reversed relative to what would be obtained from the C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples**

This example opens the example netCDF file included with MATLAB, `example.nc`, and gets information about a variable in the file.

```
% Open the example netCDF file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get information about third variable in the file.
[varname, xtype, dimids, numatts] = netcdf.inqVar(ncid,2)

varname =

peaks

xtype =

5

dimids =
```

# netcdf.inqVar

---

```
0 1
```

```
numatts =
```

```
1 1
```

## See Also

`netcdf.create`, `netcdf.inqVarID`, `netcdf.open`

**Purpose** Return ID associated with variable name

**Syntax** `varid = netcdf.inqVarID(ncid,varname)`

**Description** `varid = netcdf.inqVarID(ncid,varname)` returns `varid`, the ID of a netCDF variable specified by the text string, `varname`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_varid` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example opens the example netCDF file included with MATLAB, `example.nc`, and uses several inquiry functions to get the ID of the first variable.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get information about first variable in the file.
[varname, xtype, dimids,atts] = netcdf.inqVar(ncid,0);

% Get variable ID of the first variable, given its name
varid = netcdf.inqVarID(ncid,varname)

varid =

    0
```

**See Also** `netcdf.create`, `netcdf.inqVar`, `netcdf.open`

# netcdf.open

---

**Purpose** Open netCDF file

**Syntax**

```
ncid = netcdf.open(filename, mode)
[chosen_chunksize, ncid] = netcdf.open(filename, mode,
    chunksize)
```

**Description** `ncid = netcdf.open(filename, mode)` opens an existing netCDF file and returns a netCDF ID in `ncid`. `mode` describes the type of access to the file and can have any of the following values.

Value	Description
'NC_WRITE'	Read-write access
'NC_SHARE'	Synchronous file updates
'NC_NOWRITE'	Read-only access

You can also specify `mode` as a numeric value that can be retrieved using `netcdf.getConstant`. Use these numeric values when you want to specify a bitwise-OR of several mode.

`[chosen_chunksize, ncid] = netcdf.open(filename, mode, chunksize)` opens an existing netCDF file, specifying an additional I/O performance tuning parameter, `chunksize`. The actual value used by the netCDF library might not correspond to the input value you specify.

This function corresponds to the `nc_open` and `nc__open` functions in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example opens the example netCDF file included with MATLAB, `example.nc`.

```
ncid = netcdf.open('example.nc', 'NC_NOWRITE');
```

**See Also** `netcdf.create`, `netcdf.getConstant`



**Purpose** Write netCDF attribute

**Syntax** netcdf.putAtt(ncid,varid,attrname,attrvalue)

**Description** netcdf.putAtt(ncid,varid,attrname,attrvalue) writes the attribute named attrname with value attrvalue to the netCDF variable specified by varid. To specify a global attribute, use netcdf.getConstant('NC\_GLOBAL') for varid

ncid is a netCDF file identifier returned by netCDF.create or netCDF.open.

This function corresponds to several attribute I/O functions in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example creates a new netCDF file, defines a dimension and a variable, adds data to the variable, and then creates an attribute associated with the variable. To run this example, you must have writer permission in your current directory.

```
% Create a variable in the workspace.
my_vardata = linspace(0,50,50);

% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE');

% Define a dimension in the file.
dimid = netcdf.defDim(ncid,'my_dim',50);

% Define a new variable in the file.
varid = netcdf.defVar(ncid,'my_var','double',dimid);

% Leave define mode and enter data mode to write data.
netcdf.endDef(ncid);

% Write data to variable.
```

# netcdf.putAtt

---

```
netcdf.putVar(ncid,varid,my_vardata);

% Re-enter define mode.
netcdf.reDef(ncid);

% Create an attribute associated with the variable.
netcdf.putAtt(ncid,0,'my_att',10);

% Verify that the attribute was created.
[xtype xlen] = netcdf.inqAtt(ncid,0,'my_att')

xtype =

     6

xlen =

     1
```

**See Also** [netcdf.getatt](#)

**Purpose** Write data to netCDF variable

**Syntax**

```
netcdf.putVar(ncid,varid,data)
netcdf.putVar(ncid,varid,start,data)
netcdf.putVar(ncid,varid,start,count,data)
netcdf.putVar(ncid,varid,start,count,stride,data)
```

**Description**

`netcdf.putVar(ncid,varid,data)` writes data to a netCDF variable identified by `varid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`netcdf.putVar(ncid,varid,start,data)` writes a single data value into the variable at the index specified by `start`.

`netcdf.putVar(ncid,varid,start,count,data)` writes a section of values into the netCDF variable at the index specified by the vector `start` to the extent specified by the vector `count`, along each dimension of the specified variable.

`netcdf.putVar(ncid,varid,start,count,stride,data)` writes the subsection specified by sampling interval, `stride`, of the values in the section of the variable beginning at the index `start` and to the extent specified by `count`.

This function corresponds to several variable I/O functions in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example creates a new netCDF file and writes a variable to the file.

```
% Create a 50 element vector for a variable.
my_vardata = linspace(0,50,50);

% Open netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE')
```

## netcdf.putVar

---

```
% Define the dimensions of the variable.
dimid = netcdf.defDim(ncid, 'my_dim', 50);

% Define a new variable in the file.
my_varID = netcdf.defVar(ncid, 'my_var', 'double', dimid)

% Leave define mode and enter data mode to write data.
netcdf.endDef(ncid)

% Write data to variable.
netcdf.putVar(ncid, my_varID, my_vardata);

% Verify that the variable was created.
[varname xtype dimid natts ] = netcdf.inqVar(ncid, 0)

varname =

my_var

xtype =

    6

dimid =

    0

natts =

    0
```

### See Also

`netcdf.getVar`

---

<b>Purpose</b>	Put open netCDF file into define mode
<b>Syntax</b>	<code>netcdf.reDef(ncid)</code>
<b>Description</b>	<p><code>netcdf.reDef(ncid)</code> puts an open netCDF file into define mode so that dimensions, variables, and attributes can be added or renamed. Attributes can also be deleted in define mode. <code>ncid</code> is a valid NetCDF file ID, returned from a previous call to <code>netcdf.open</code> or <code>netcdf.create</code>.</p> <p>This function corresponds to the <code>nc_redef</code> function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the <a href="#">NetCDF C Interface Guide</a> for more information.</p>
<b>Examples</b>	<p>This example opens a local copy of the example netCDF file included with MATLAB, <code>example.nc</code>.</p> <pre>% Open a netCDF file. ncid = netcdf.open('my_example.nc','NC_WRITE')  % Try to define a dimension. dimid = netcdf.defdim(ncid, 'lat', 50); % should fail. ??? Error using ==&gt; netcdflib NetCDF: Operation not allowed in data mode  Error in ==&gt; defDim at 22 dimid = netcdflib('def_dim', ncid,dimname,dimlen);  % Put file in define mode. netcdf.reDef(ncid);  % Try to define a dimension again. Should succeed. dimid = netcdf.defDim(ncid, 'lat', 50);</pre>
<b>See Also</b>	<code>netcdfcreate</code> , <code>netcdf.endDef</code> , <code>netcdf.open</code>

# netcdf.renameAtt

---

**Purpose** Change name of attribute

**Syntax** `netcdf.renameAtt(ncid,varid,oldName,newName)`

**Description** `netcdf.renameAtt(ncid,varid,oldName,newName)` changes the name of the attribute specified by the character string `oldName`.

`newName` is a character string that specifies the new name.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` identifies the variable to which the attribute is associated. To specify a global attribute, use `netcdf.getConstant('NC_GLOBAL')` for `varid`.

This function corresponds to the `nc_rename_att` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

## Examples

This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Get the ID of a variable the attribute is associated with.
varID = netcdf.inqVarID(ncid,'avagadros_number')

% Rename the attribute.
netcdf.renameAtt(ncid,varID,'description','Description');

% Verify that the name changed.
attname = netcdf.inqAttName(ncid,varID,0)

attname =

Description
```

**See Also**      `netcdf.inqAttName`

# netcdf.renameDim

---

**Purpose** Change name of netCDF dimension

**Syntax** `netcdf.renameDim(ncid,dimid,newName)`

**Description** `netcdf.renameDim(ncid,dimid,newName)` renames the dimension identified by the dimension identifier, `dimid`.

`newName` is a character string specifying the new name. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`

This function corresponds to the `nc_rename_dim` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This examples modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Put file is define mode.
netcdf.reDef(ncid)

% Get the identifier of a dimension to rename.
dimid = netcdf.inqDimID(ncid,'x');

% Rename the dimension.
netcdf.renameDim(ncid,dimid,'Xdim')

% Verify that the name changed.
data = netcdf.inqDim(ncid,dimid)

data =

Xdim
```



**See Also**      `netcdf.defDim`

# netcdf.renameVar

---

**Purpose** Change name of netCDF variable

**Syntax** `netcdf.renameVar(ncid,varid,newName)`

**Description** `netcdf.renameVar(ncid,varid,newName)` renames the variable identified by `varid` in the netCDF file identified by `ncid`. `newName` is a character string specifying the new name.

This function corresponds to the `nc_rename_var` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc', 'NC_WRITE')

% Put file in define mode.
netcdf.redef(ncid)

% Get name of first variable
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

varname

varname =

avagadros_number

% Rename the variable, using a capital letter to start the name.
netcdf.renameVar(ncid,0, 'Avagadros_number')

% Verify that the name of the variable changed.
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);
```

varname

varname =

Avagadros\_number

## See Also

`netCDF.defVar`, `netCDF.inqVar`, `netCDF.putVar`

# netcdf.setDefaultFormat

---

**Purpose** Change default netCDF file format

**Syntax** `oldFormat = netcdf.setDefaultFormat(newFormat)`

**Description** `oldFormat = netcdf.setDefaultFormat(newFormat)` changes the default format used by `netcdf.create` when creating new netCDF files, and returns the value of the old format. You can use this function to change the format used by a netCDF file without having to change the creation mode flag used in each call to `netcdf.create`.

`newFormat` can be either of the following values.

Value	Description
'NC_FORMAT_CLASSIC'	Original netCDF file format
'NC_FORMAT_64BIT'	64-bit offset format; relaxes limitations on creating very large files

You can also specify the numeric equivalent of these values, as retrieved by `netcdf.getConstant`.

This function corresponds to the `nc_set_default_format` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** `oldFormat = netcdf.setDefaultFormat('NC_FORMAT_64BIT');`

**See Also** `netcdf.create`

<b>Purpose</b>	Set netCDF fill mode
<b>Syntax</b>	<code>old_mode = netcdf.setFill(ncid,new_mode)</code>
<b>Description</b>	<p><code>old_mode = netcdf.setFill(ncid,new_mode)</code> sets the fill mode for a netCDF file identified by <code>ncid</code>.</p> <p><code>new_mode</code> can be either 'FILL' or 'NOFILL' or their numeric equivalents, as retrieved by <code>netcdf.getConstant</code>. The default mode is 'FILL'. netCDF pre-fills data with fill values. Specifying 'NOFILL' can be used to enhance performance, because it avoids the duplicate writes that occur when the netCDF writes fill values that are later overwritten with data.</p> <p>This function corresponds to the <code>nc_set_fill</code> function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the <a href="#">NetCDF C Interface Guide</a> for more information.</p>
<b>Examples</b>	<p>This example creates a new file and specifies the fill mode used by netCDF with the file.</p> <pre>ncid = netcdf.open('foo.nc','NC_WRITE');  % Set filling behavior old_mode = netcdf.setFill(ncid,'NC_NOFILL');</pre>
<b>See Also</b>	<code>netcdf.getConstant</code>

# netcdf.sync

---

**Purpose** Synchronize netCDF file to disk

**Syntax** `netcdf.sync(ncid)`

**Description** `netcdf.sync(ncid)` synchronizes the state of a netCDF file to disk. The netCDF library normally buffers accesses to the underlying netCDF file, unless you specify the `NC_SHARE` mode when you opened the file with `netcdf.open` or `netcdf.create`. To call `netcdf.sync`, the netCDF file must be in data mode.

This function corresponds to the `nc_sync` function in the netCDF library C API. To use this and the other functions in the MATLAB netCDF package, you must be familiar with the netCDF C interface. See the [NetCDF C Interface Guide](#) for more information.

**Examples** This example creates a new netCDF file for write access, performs an operation on the file, takes the file out of define mode, and then synchronizes the file to disk.

```
% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE');

% Perform an operation.
dimid = netcdf.defDim(ncid,'Xdim',50);

% Take file out of define mode.
netcdf.endDef(ncid);

% Synchronize the file to disk.
netcdf.sync(ncid)
```

**See Also** `netcdf.close`, `netcdf.create`, `netcdf.open`, `netcdf.endDef`

<b>Purpose</b>	Determine where to draw graphics objects
<b>Syntax</b>	<pre>newplot h = newplot h = newplot(hsave)</pre>
<b>Description</b>	<p><code>newplot</code> prepares a figure and axes for subsequent graphics commands.</p> <p><code>h = newplot</code> prepares a figure and axes for subsequent graphics commands and returns a handle to the current axes.</p> <p><code>h = newplot(hsave)</code> prepares and returns an axes, but does not delete any objects whose handles you have assigned to the <code>hsave</code> argument, which can be a vector of handles. If <code>hsave</code> is not empty, the figure and axes containing <code>hsave</code> are prepared for plotting instead of the current axes of the current figure. If <code>hsave</code> is empty, <code>newplot</code> behaves as if it were called without any inputs.</p>
<b>Remarks</b>	<p>Use <code>newplot</code> at the beginning of high-level graphics M-files to determine which figure and axes to target for graphics output. Calling <code>newplot</code> can change the current figure and current axes. Basically, there are three options when you are drawing graphics in existing figures and axes:</p> <ul style="list-style-type: none"><li>• Add the new graphics without changing any properties or deleting any objects.</li><li>• Delete all existing objects whose handles are not hidden before drawing the new objects.</li><li>• Delete all existing objects regardless of whether or not their handles are hidden, and reset most properties to their defaults before drawing the new objects (refer to the following table for specific information).</li></ul> <p>The figure and axes <code>NextPlot</code> properties determine how <code>newplot</code> behaves. The following two tables describe this behavior with various property values.</p> <p>First, <code>newplot</code> reads the current figure's <code>NextPlot</code> property and acts accordingly.</p>

# newplot

---

<b>NextPlot</b>	<b>What Happens</b>
<code>new</code>	Create a new figure and use it as the current figure.
<code>add</code>	Draw to the current figure without clearing any graphics objects already present.
<code>replacechildren</code>	Remove all child objects whose <code>HandleVisibility</code> property is set to <code>on</code> and reset figure <code>NextPlot</code> property to <code>add</code> .  This clears the current figure and is equivalent to issuing the <code>clf</code> command.
<code>replace</code>	Remove all child objects (regardless of the setting of the <code>HandleVisibility</code> property) and reset figure properties to their defaults, except <code>NextPlot</code> is reset to <code>add</code> regardless of user-defined defaults. <ul style="list-style-type: none"><li>• <code>Position</code>, <code>Units</code>, <code>PaperPosition</code>, and <code>PaperUnits</code> are not reset.</li></ul> This clears and resets the current figure and is equivalent to issuing the <code>clf reset</code> command.

After `newplot` establishes which figure to draw in, it reads the current axes' `NextPlot` property and acts accordingly.

<b>NextPlot</b>	<b>Description</b>
<code>add</code>	Draw into the current axes, retaining all graphics objects already present.



NextPlot	Description
replacechildren	Remove all child objects whose HandleVisibility property is set to on, but do not reset axes properties. This clears the current axes like the cla command.
replace	Remove all child objects (regardless of the setting of the HandleVisibility property) and reset axes properties to their defaults, except Position and Units.  This clears and resets the current axes like the cla reset command.

## See Also

axes, cla, clf, figure, hold, ishold, reset

The NextPlot property for figure and axes graphics objects

“Figure Windows” on page 1-99 for related functions

Controlling Graphics Output for more examples.

# nextpow2

---

**Purpose** Next higher power of 2

**Syntax** `p = nextpow2(A)`

**Description** `p = nextpow2(A)` returns the smallest power of two that is greater than or equal to the absolute value of A. (That is, p that satisfies  $2^p \geq \text{abs}(A)$ ).

This function is useful for optimizing FFT operations, which are most efficient when sequence length is an exact power of two.

If A is non-scalar, `nextpow2` returns the smallest power of two greater than or equal to `length(A)`.

**Examples** For any integer n in the range from 513 to 1024, `nextpow2(n)` is 10.

For a 1-by-30 vector A, `length(A)` is 30 and `nextpow2(A)` is 5.

**See Also** `fft`, `log2`, `pow2`

---

<b>Purpose</b>	Number of nonzero matrix elements
<b>Syntax</b>	<code>n = nnz(X)</code>
<b>Description</b>	<code>n = nnz(X)</code> returns the number of nonzero elements in matrix <code>X</code> . The density of a sparse matrix is <code>nnz(X)/prod(size(X))</code> .
<b>Examples</b>	The matrix  <pre>w = sparse(wilkinson(21));</pre> is a tridiagonal matrix with 20 nonzeros on each of three diagonals, so <code>nnz(w) = 60</code> .
<b>See Also</b>	<code>find</code> , <code>isa</code> , <code>nonzeros</code> , <code>nzmax</code> , <code>size</code> , <code>whos</code>

# noanimate

---

**Purpose** Change EraseMode of all objects to normal

**Syntax** `noanimate(state, fig_handle)`  
`noanimate(state)`

**Description** `noanimate(state, fig_handle)` sets the EraseMode of all image, line, patch, surface, and text graphics objects in the specified figure to normal. `state` can be the following strings:

- 'save' — Set the values of the EraseMode properties to normal for all the appropriate objects in the designated figure.
- 'restore' — Restore the EraseMode properties to the previous values (i.e., the values before calling `noanimate` with the 'save' argument).

`noanimate(state)` operates on the current figure.

`noanimate` is useful if you want to print the figure to a TIFF or JPEG format.

**See Also** `print`

“Animation” on page 1-95 for related functions

**Purpose** Nonzero matrix elements

**Syntax** `s = nonzeros(A)`

**Description** `s = nonzeros(A)` returns a full column vector of the nonzero elements in A, ordered by columns.

This gives the s, but not the i and j, from `[i,j,s] = find(A)`.  
Generally,

`length(s) = nnz(A) <= nzmax(A) <= prod(size(A))`

**See Also** `find`, `isa`, `nnz`, `nzmax`, `size`, `whos`

# norm

---

**Purpose** Vector and matrix norms

**Syntax**  
`n = norm(A)`  
`n = norm(A,p)`

**Description** The *norm* of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. The `norm` function calculates several different types of matrix norms:

`n = norm(A)` returns the largest singular value of `A`, `max(svd(A))`.

`n = norm(A,p)` returns a different kind of norm, depending on the value of `p`.

If <code>p</code> is...	Then <code>norm</code> returns...
1	The 1-norm, or largest column sum of <code>A</code> , <code>max(sum(abs(A)))</code> .
2	The largest singular value (same as <code>norm(A)</code> ).
<code>inf</code>	The infinity norm, or largest row sum of <code>A</code> , <code>max(sum(abs(A')))</code> .
<code>'fro'</code>	The Frobenius-norm of matrix <code>A</code> , <code>sqrt(sum(diag(A'*A)))</code> .

When `A` is a vector:

<code>norm(A,p)</code>	Returns <code>sum(abs(A).^p)^(1/p)</code> , for any $1 \leq p \leq \infty$ .
<code>norm(A)</code>	Returns <code>norm(A,2)</code> .
<code>norm(A,inf)</code>	Returns <code>max(abs(A))</code> .
<code>norm(A,-inf)</code>	Returns <code>min(abs(A))</code> .

**Remarks** Note that `norm(x)` is the Euclidean length of a vector `x`. On the other hand, MATLAB software uses “length” to denote the number of elements `n` in a vector. This example uses `norm(x)/sqrt(n)` to obtain the root-mean-square (RMS) value of an `n`-element vector `x`.

```
x = [0 1 2 3]
x =
     0     1     2     3

sqrt(0+1+4+9) % Euclidean length
ans =
     3.7417

norm(x)
ans =
     3.7417

n = length(x) % Number of elements
n =
     4

rms = 3.7417/2 % rms = norm(x)/sqrt(n)
rms =
     1.8708
```

**See Also**

cond, condest, hypot, normest, rcond

# normest

---

**Purpose** 2-norm estimate

**Syntax**  
`nrm = normest(S)`  
`nrm = normest(S,tol)`  
`[nrm,count] = normest(...)`

**Description** This function is intended primarily for sparse matrices, although it works correctly and may be useful for large, full matrices as well.

`nrm = normest(S)` returns an estimate of the 2-norm of the matrix  $S$ .

`nrm = normest(S,tol)` uses relative error `tol` instead of the default tolerance `1.e-6`. The value of `tol` determines when the estimate is considered acceptable.

`[nrm,count] = normest(...)` returns an estimate of the 2-norm and also gives the number of power iterations used.

**Examples** The matrix  $W = \text{gallery}('wilkinson',101)$  is a tridiagonal matrix. Its order, 101, is small enough that `norm(full(W))`, which involves `svd(full(W))`, is feasible. The computation takes 4.13 seconds (on one computer) and produces the exact norm, 50.7462. On the other hand, `normest(sparse(W))` requires only 1.56 seconds and produces the estimated norm, 50.7458.

**Algorithm** The power iteration involves repeated multiplication by the matrix  $S$  and its transpose,  $S'$ . The iteration is carried out until two successive estimates agree to within the specified relative tolerance.

**See Also** `cond`, `condest`, `norm`, `rcond`, `svd`



**Purpose** Find logical NOT of array or scalar input

**Syntax** `~A`  
`not(A)`

**Description** `~A` performs a logical NOT of input array `A`, 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 the input array contains a zero value element at that same array location. Otherwise, that element is set to 0.

The input of the expression can be an array or can be a scalar value. If the input is an array, then the output is an array of the same dimensions. If the input is scalar, then the output is scalar.

`not(A)` is called for the syntax `~A` when `A` is an object.

**Example** If matrix `A` is

0	29	0	36	0
23	34	35	0	39
0	24	31	27	0
0	29	0	0	34

then

```

~A
ans =
    1     0     1     0     1
    0     0     0     1     0
    1     0     0     0     1
    1     0     1     1     0

```

**See Also** `bitcmp`, `and`, `or`, `xor`, `any`, `all`, “Logical Operators”, “Logical Classes”, “Bit-Wise Functions”

# notebook

---

**Purpose** Open M-book in Microsoft Word software (on Microsoft Windows platforms)

**Syntax** `notebook`  
`notebook('filename')`  
`notebook('-setup')`

**Description** `notebook` starts Microsoft Word software and creates a new M-book titled Document 1.

`notebook('filename')` starts Microsoft Word and opens the M-book `filename`, where `filename` is either in the MATLAB current directory or is a full path. If `filename` does not exist, MATLAB creates a new M-book titled `filename`. If the file name extension is not specified, MATLAB assumes `.doc`.

`notebook('-setup')` runs an interactive setup function for Notebook. It copies the Notebook template, `m-book.dot`, to the Microsoft Word template directory, whose location MATLAB automatically determines from the Windows system registry. Upon completion, MATLAB displays a message indicating whether or not the setup was successful.

**See Also** MATLAB Desktop Tools and Development Environment documentation

- Notebook for Publishing to Word
- “Overview of Publishing M-Files”

**Purpose** notify listeners that event is occurring

**Syntax** `notify(Hobj, 'EventName')`  
`notify(Hobj, 'EventName', data)`

**Description** `notify(Hobj, 'EventName')` notifies listeners that the specified event is taking place on the specified handle objects.

`notify(Hobj, 'EventName', data)` includes user-defined event data.

## Arguments

Hobj

Array of handle objects triggering the specified event.

EventName

Name of the event.

data

An `event.EventData` object encapsulating information about the event. You can define custom event data by subclassing `event.EventData` and passing an instance of your subclass as the `data` argument. See “Defining Event-Specific Data” for information on defining event data.

## See Also

See “Defining Events and Listeners — Syntax and Techniques”

`handle`, `addlistener`

# now

---

**Purpose** Current date and time

**Syntax** `t = now`

**Description** `t = now` returns the current date and time as a serial date number. To return the time only, use `rem(now, 1)`. To return the date only, use `floor(now)`.

**Examples** `t1 = now, t2 = rem(now, 1)`

```
t1 =  
  
    7.2908e+05
```

```
t2 =  
  
    0.4013
```

**See Also** `clock`, `date`, `datenum`

**Purpose** Real nth root of real numbers

**Syntax** `y = nthroot(X, n)`

**Description** `y = nthroot(X, n)` returns the real nth root of the elements of X. Both X and n must be real and n must be a scalar. If X has negative entries, n must be an odd integer.

**Example** `nthroot(-2, 3)`

returns the real cube root of -2.

`ans =`

`-1.2599`

By comparison,

`(-2)^(1/3)`

returns a complex cube root of -2.

`ans =`

`0.6300 + 1.0911i`

**See Also** `power`

# null

---

**Purpose** Null space

**Syntax** `Z = null(A)`  
`Z = null(A, 'r')`

**Description** `Z = null(A)` is an orthonormal basis for the null space of `A` obtained from the singular value decomposition. That is, `A*Z` has negligible elements, `size(Z,2)` is the nullity of `A`, and `Z'*Z = I`.

`Z = null(A, 'r')` is a “rational” basis for the null space obtained from the reduced row echelon form. `A*Z` is zero, `size(Z,2)` is an estimate for the nullity of `A`, and, if `A` is a small matrix with integer elements, the elements of the reduced row echelon form (as computed using `rref`) are ratios of small integers.

The orthonormal basis is preferable numerically, while the rational basis may be preferable pedagogically.

## Example

### Example 1

Compute the orthonormal basis for the null space of a matrix `A`.

```
A = [1  2  3
      1  2  3
      1  2  3];
```

```
Z = null(A);
A*Z
```

```
ans =
  1.0e-015 *
    0.2220    0.2220
    0.2220    0.2220
    0.2220    0.2220
```

```
Z'*Z
```

```
ans =
```

```
1.0000 -0.0000
-0.0000 1.0000
```

### Example 2

Compute the 1-norm of the matrix  $A*Z$  and determine that it is within a small tolerance.

```
norm(A*Z,1) < 1e-12
ans =
1
```

### Example 3

Compute the rational basis for the null space of the same matrix A.

```
ZR = null(A, 'r')
```

```
ZR =
-2 -3
1 0
0 1
```

```
A*ZR
```

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

### See Also

orth, rank, rref, svd

# num2cell

---

**Purpose** Convert numeric array to cell array

**Syntax**

```
C = num2cell(A)
C = num2cell(A, dim)
C = num2cell(A, [dim1, dim2, ...])
```

**Description** `C = num2cell(A)` converts numeric array `A` into cell array `C` by placing each element of `A` into a separate cell in `C`. The output array has the same size and dimensions as the input array. Each cell in `C` contains the same numeric value as its respective element in `A`.

`C = num2cell(A, dim)` converts numeric array `A` into a cell array of numeric vectors, the dimensions of which depend on the value of the `dim` argument. Return value `C` contains `numel(A)/size(A,dim)` vectors, each of length `size(A, dim)`. (See Example 2, below). The `dim` input must be an integer with a value from `ndims(A)` to 1.

`C = num2cell(A, [dim1, dim2, ...])` converts numeric array `A` into a cell array of numeric arrays, the dimensions of which depend on the values of arguments `[dim1, dim2, ...]`. Given the variables `X` and `Y`, where `X=size(A,dim1)` and `Y=size(A,dim2)`, return value `C` contains `numel(A)/prod(X,Y,...)` arrays, each of size `X-by-Y-by-....`. All `dimN` inputs must be an integer with a value from `ndims(A)` to 1.

## Examples

### Example 1 – Converting to a Cell Array of the Same Dimensions

Create a 4x7x3 array of random numbers:

```
A = rand(4,7,3);
```

Convert the numeric array to a cell array of the same dimensions:

```
C = num2cell(A);
whos C
```

Name	Size	Bytes	Class	Attributes
C	4x7x3	5712	cell	



### Example 2 – Converting to a Cell Array of Different Dimensions

The following diagrams show the output of `C=num2cell(A,dim)`, where `A` is a 4x7x3 numeric array, and `dim` is one dimension of the input array: either 1, 2, or 3 in this case. The shaded segment of each diagram represents one cell of the output cell array. Each of these cells contains those elements of `A` that are positioned along the dimension specified by `dim`:

### Example 3 – Specifying More Than One Dimension for the Output

Given a 4x7x3 numeric array `A`

```
A = rand(4,7,3)
```

and the following two values `X` and `Y`

```
X = size(A,dim1)  
Y = size(A,dim2)
```

then the statement

```
C = num2cell(A, [1 3])
```

returns in `C` a cell array of `numel(A)/prod(X,Y)` numeric arrays, each having the dimensions `X-by-Y`.

### See Also

`cat`, `mat2cell`, `cell2mat`

# num2hex

---

**Purpose** Convert singles and doubles to IEEE hexadecimal strings

**Syntax** num2hex(X)

**Description** If X is a single or double precision array with n elements, num2hex(X) is an n-by-8 or n-by-16 char array of the hexadecimal floating-point representation. The same representation is printed with format hex.

**Examples** num2hex([1 0 0.1 -pi Inf NaN])

returns

ans =

```
3ff0000000000000
0000000000000000
3fb999999999999a
c00921fb54442d18
7ff0000000000000
fff8000000000000
num2hex(single([1 0 0.1 -pi Inf NaN]))
```

returns

ans =

```
3f800000
00000000
3dcccccd
c0490fdb
7f800000
ffc00000
```

**See Also** hex2num, dec2hex, format

**Purpose**

Convert number to string

**Syntax**

```
str = num2str(A)
str = num2str(A, precision)
str = num2str(A, format)
```

**Description**

The `num2str` function converts numbers to their string representations. This function is useful for labeling and titling plots with numeric values.

`str = num2str(A)` converts array `A` into a string representation `str` with roughly four digits of precision and an exponent if required.

`str = num2str(A, precision)` converts the array `A` into a string representation `str` with maximum precision specified by `precision`. Argument `precision` specifies the number of digits the output string is to contain. The default is four.

`str = num2str(A, format)` converts array `A` using the supplied `format`. (See `fprintf` for format string details.) By default, `num2str` displays floating point values using `'%11.4g'` format (four significant digits in exponential or fixed-point notation, whichever is shorter).

If the input array is integer-valued, `num2str` returns the exact string representation of that integer. The term integer-valued includes large floating-point numbers that lose precision due to limitations of the hardware.

`num2str` removes any leading spaces from the output string. Thus, `num2str(42.67, '%10.2f')` returns a 1-by-5 character array `'42.67'`.

**Examples**

```
num2str(pi) is 3.142.
```

```
num2str(eps) is 2.22e-16.
```

`num2str` with a format of `%10.5e\n` returns a matrix of strings in exponential format, having 5 decimal places, with each element separated by a newline character:

```
x = rand(2,3) * 9999;           % Create a 2-by-3 matrix.
```

## num2str

---

```
A = num2str(x, '%10.5e\n')      % Convert to string array.  
A =  
6.87255e+003  
1.55597e+003  
8.55890e+003  
3.46077e+003  
1.91097e+003  
4.90201e+003
```

### See Also

mat2str, int2str, str2num, sprintf, fprintf

**Purpose** Number of elements in array or subscripted array expression

**Syntax**  
`n = numel(A)`  
`n = numel(A, index1, index2, ... indexn)`

**Description** `n = numel(A)` returns the number of elements, `n`, in array `A`.  
`n = numel(A, index1, index2, ... indexn)` returns the number of subscripted elements, `n`, in `A(index1, index2, ..., indexn)`. To handle the variable number of arguments, `numel` is typically written with the header function `n = numel(A, varargin)`, where `varargin` is a cell array with elements `index1, index2, ... indexn`.

The MATLAB software implicitly calls the `numel` built-in function whenever an expression generates a comma-separated list. This includes brace indexing (i.e., `A{index1,index2,...,indexN}`), and dot indexing (i.e., `A.fieldname`).

**Remarks** It is important to note the significance of `numel` with regards to the overloaded `subsref` and `subsasgn` functions. In the case of the overloaded `subsref` function for brace and dot indexing (as described in the last paragraph), `numel` is used to compute the number of expected outputs (`nargout`) returned from `subsref`. For the overloaded `subsasgn` function, `numel` is used to compute the number of expected inputs (`nargin`) to be assigned using `subsasgn`. The `nargin` value for the overloaded `subsasgn` function is the value returned by `numel` plus 2 (one for the variable being assigned to, and one for the structure array of subscripts).

As a class designer, you must ensure that the value of `n` returned by the built-in `numel` function is consistent with the class design for that object. If `n` is different from either the `nargout` for the overloaded `subsref` function or the `nargin` for the overloaded `subsasgn` function, then you need to overload `numel` to return a value of `n` that is consistent with the class' `subsref` and `subsasgn` functions. Otherwise, MATLAB produces errors when calling these functions.

# numel

---

## Examples

Create a 4-by-4-by-2 matrix. `numel` counts 32 elements in the matrix.

```
a = magic(4);
a(:,:,2) = a'

a(:,:,1) =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

a(:,:,2) =
    16     5     9     4
     2    11     7    14
     3    10     6    15
    13     8    12     1

numel(a)
ans =
    32
```

## See Also

`nargin`, `nargout`, `prod`, `size`, `subsasgn`, `subsref`

**Purpose**

Amount of storage allocated for nonzero matrix elements

**Syntax**

`n = nzmax(S)`

**Description**

`n = nzmax(S)` returns the amount of storage allocated for nonzero elements.

If `S` is a sparse matrix...      `nzmax(S)` is the number of storage locations allocated for the nonzero elements in `S`.

If `S` is a full matrix...      `nzmax(S) = prod(size(S))`.

Often, `nnz(S)` and `nzmax(S)` are the same. But if `S` is created by an operation which produces fill-in matrix elements, such as sparse matrix multiplication or sparse LU factorization, more storage may be allocated than is actually required, and `nzmax(S)` reflects this. Alternatively, `sparse(i, j, s, m, n, nzmax)` or its simpler form, `spalloc(m, n, nzmax)`, can set `nzmax` in anticipation of later fill-in.

**See Also**

`find`, `isa`, `nnz`, `nonzeros`, `size`, `whos`

# ode15i

---

**Purpose** Solve fully implicit differential equations, variable order method

**Syntax**

```
[T,Y] = ode15i(odefun,tspan,y0,yp0)
[T,Y] = ode15i(odefun,tspan,y0,yp0,options)
[T,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)
sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)
```

**Arguments** The following table lists the input arguments for ode15i.

odefun	A function handle that evaluates the left side of the differential equations, which are of the form $f(t, y, y') = \mathbf{0}$ . See “Function Handles” in the MATLAB Programming documentation for more information.
tspan	A vector specifying the interval of integration, [t0,tf]. To obtain solutions at specific times (all increasing or all decreasing), use tspan = [t0,t1,...,tf].
y0, yp0	Vectors of initial conditions for $y$ and $y'$ respectively.
options	Optional integration argument created using the odeset function. See odeset for details.

The following table lists the output arguments for ode15i.

T	Column vector of time points
Y	Solution array. Each row in $y$ corresponds to the solution at a time returned in the corresponding row of $t$ .

**Description** [T,Y] = ode15i(odefun,tspan,y0,yp0) with tspan = [t0 tf] integrates the system of differential equations  $f(t, y, y') = \mathbf{0}$  from time t0 to tf with initial conditions y0 and yp0. odefun is a function handle. Function ode15i solves ODEs and DAEs of index 1. The initial conditions must be consistent, meaning that  $f(t0, y0, yp0) = \mathbf{0}$ . You can use the function decic to compute consistent initial conditions



close to guessed values. Function `odefun(t,y,yp)`, for a scalar  $t$  and column vectors  $y$  and  $yp$ , must return a column vector corresponding to  $f(t, y, y')$ . Each row in the solution array  $Y$  corresponds to a time returned in the column vector  $T$ . To obtain solutions at specific times  $t_0, t_1, \dots, t_f$  (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `odefun`, if necessary.

`[T,Y] = ode15i(odefun,tspan,y0,yp0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used options include a scalar relative error tolerance `RelTol` ( $1e-3$  by default) and a vector of absolute error tolerances `AbsTol` (all components  $1e-6$  by default). See `odeset` for details.

`[T,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)` with the 'Events' property in `options` set to a function `events`, solves as above while also finding where functions of  $(t, y, y')$ , called event functions, are zero. The function `events` is of the form `[value,isterminal,direction] = events(t,y,yp)` and includes the necessary event functions. Code the function `events` so that the  $i$ th element of each output vector corresponds to the  $i$ th event. For the  $i$ th event function in `events`:

- `value(i)` is the value of the function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function and 0 otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), +1 if only the zeros where the event function increases, and -1 if only the zeros where the event function decreases.

Output `TE` is a column vector of times at which events occur. Rows of `YE` are the corresponding solutions, and indices in vector `IE` specify which

event occurred. See “Integrator Options” in the MATLAB Mathematics documentation for more information.

`sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)` returns a structure that can be used with `deval` to evaluate the solution at any point between `t0` and `tfinal`. The structure `sol` always includes these fields:

<code>sol.x</code>	Steps chosen by the solver. If you specify the <code>Events</code> option and a terminal event is detected, <code>sol.x(end)</code> contains the end of the step at which the event occurred.
<code>sol.y</code>	Each column <code>sol.y(:,i)</code> contains the solution at <code>sol.x(i)</code> .

If you specify the `Events` option and events are detected, `sol` also includes these fields:

<code>sol.xe</code>	Points at which events, if any, occurred. <code>sol.xe(end)</code> contains the exact point of a terminal event, if any.
<code>sol.ye</code>	Solutions that correspond to events in <code>sol.xe</code> .
<code>sol.ie</code>	Indices into the vector returned by the function specified in the <code>Events</code> option. The values indicate which event the solver detected.

## Options

`ode15i` accepts the following parameters in options. For more information, see `odeset` and Changing ODE Integration Properties in the MATLAB Mathematics documentation.

Error control	<code>RelTol</code> , <code>AbsTol</code> , <code>NormControl</code>
Solver output	<code>OutputFcn</code> , <code>OutputSel</code> , <code>Refine</code> , <code>Stats</code>
Event location	<code>Events</code>

Step size	MaxStep, InitialStep
Jacobian matrix	Jacobian, JPattern, Vectorized

## Solver Output

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, and `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

## Jacobian Matrices

The Jacobian matrices  $\partial f / \partial y$  and  $\partial f / \partial y'$  are critical to reliability and efficiency. You can provide these matrices as one of the following:

- Function of the form `[dfdy, dfdyp] = FJAC(t,y,yp)` that computes the Jacobian matrices. If `FJAC` returns an empty matrix `[]` for either `dfdy` or `dfdyp`, then `ode15i` approximates that matrix by finite differences.
- Cell array of two constant matrices `{dfdy, dfdyp}`, either of which could be empty.

Use `odeset` to set the `Jacobian` option to the function or cell array. If you do not set the `Jacobian` option, `ode15i` approximates both Jacobian matrices by finite differences.

For `ode15i`, `Vectorized` is a two-element cell array. Set the first element to `'on'` if `odefun(t,[y1,y2,...],yp)` returns

[odefun(t,y1,yp),odefun(t,y2,yp),...]. Set the second element to 'on' if odefun(t,y,[yp1,yp2,...]) returns [odefun(t,y,yp1),odefun(t,y,yp2),...]. The default value of Vectorized is {'off','off'}.

For ode15i, JPattern is also a two-element sparse matrix cell array. If  $\partial f/\partial y$  or  $\partial f/\partial y'$  is a sparse matrix, set JPattern to the sparsity patterns, {SPDY,SPDYP}. A sparsity pattern of  $\partial f/\partial y$  is a sparse matrix SPDY with SPDY(i,j) = 1 if component i of f(t,y,yp) depends on component j of y, and 0 otherwise. Use SPDY = [] to indicate that  $\partial f/\partial y$  is a full matrix. Similarly for  $\partial f/\partial y'$  and SPDYP. The default value of JPattern is {[],[ ]}.

## Examples

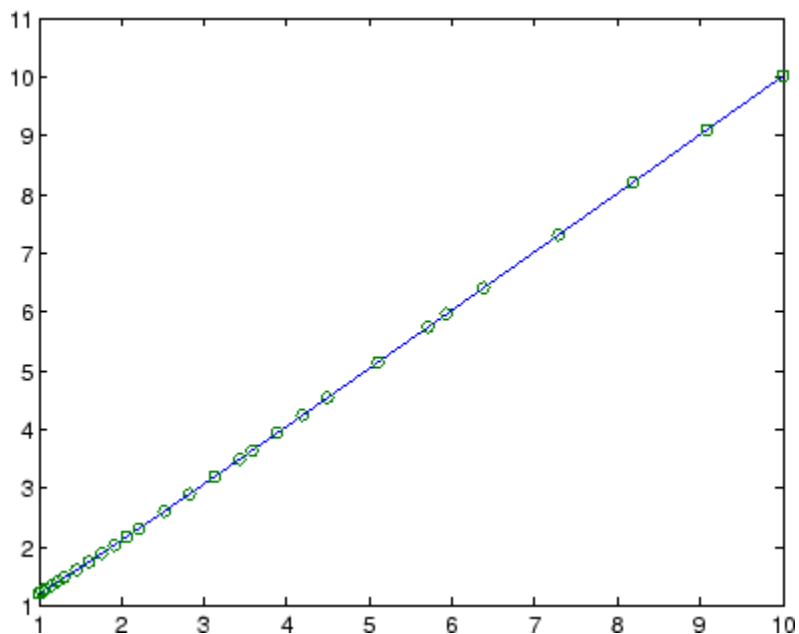
### Example 1

This example uses a helper function decic to hold fixed the initial value for  $y(t_0)$  and compute a consistent initial value for  $y'(t_0)$  for the Weissinger implicit ODE. The Weissinger function evaluates the residual of the implicit ODE.

```
t0 = 1;  
y0 = sqrt(3/2);  
yp0 = 0;  
[y0,yp0] = decic(@weissinger,t0,y0,1,yp0,0);
```

The example uses ode15i to solve the ODE, and then plots the numerical solution against the analytical solution.

```
[t,y] = ode15i(@weissinger,[1 10],y0,yp0);  
ytrue = sqrt(t.^2 + 0.5);  
plot(t,y,t,ytrue,'o');
```



### Other Examples

These demos provide examples of implicit ODEs: `ihb1dae`, `iburgersode`.

### See Also

`decic`, `deval`, `odeget`, `odeset`, `function_handle` (@)

Other ODE initial value problem solvers: `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`

# ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

---

## **Purpose**

Solve initial value problems for ordinary differential equations

## **Syntax**

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

where *solver* is one of ode45, ode23, ode113, ode15s, ode23s, ode23t, or ode23tb.

## Arguments

The following table describes the input arguments to the solvers.

odefun	A function handle that evaluates the right side of the differential equations. See “Function Handles” in the MATLAB Programming documentation for more information. All solvers solve systems of equations in the form $y' = f(t, y)$ or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$ . The ode23s solver can solve only equations with constant mass matrices. ode15s and ode23t can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
tspan	<p>A vector specifying the interval of integration, <math>[t_0, t_f]</math>. The solver imposes the initial conditions at <code>tspan(1)</code>, and integrates from <code>tspan(1)</code> to <code>tspan(end)</code>. To obtain solutions at specific times (all increasing or all decreasing), use <code>tspan = [t0, t1, ..., tf]</code>.</p> <p>For <code>tspan</code> vectors with two elements <math>[t_0 \ t_f]</math>, the solver returns the solution evaluated at every integration step. For <code>tspan</code> vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.</p>

# ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

---

Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points.

Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.

`y0`

A vector of initial conditions.

`options`

Structure of optional parameters that change the default integration properties. This is the fourth input argument.

```
[t,y] =  
solver(odefun,tspan,y0,options)
```

You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

T	Column vector of time points.
Y	Solution array. Each row in Y corresponds to the solution at a time returned in the corresponding row of T.
TE	The time at which an event occurs.
YE	The solution at the time of the event.



# ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

---

IE            The index *i* of the event function that vanishes.  
sol           Structure to evaluate the solution.

## Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations  $y' = f(t, y)$  from time `t0` to `tf` with initial conditions `y0`. `odefun` is a function handle. See Function Handles in the MATLAB Programming documentation for more information. Function `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to  $f(t, y)$ . Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0, t1, ..., tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` ( $1e-3$  by default) and a vector of absolute error tolerances `AbsTol` (all components are  $1e-6$  by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of  $(t,y)$ , called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the 'Events' property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the *i*th event function in `events`,

- `value(i)` is the value of the function.

## ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

---

- `isterminal(i)` = 1, if the integration is to terminate at a zero of this event function and 0 otherwise.
- `direction(i)` = 0 if all zeros are to be computed (the default), +1 if only the zeros where the event function increases, and -1 if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

<code>sol.x</code>	Steps chosen by the solver.
<code>sol.y</code>	Each column <code>sol.y(:,i)</code> contains the solution at <code>sol.x(i)</code> .
<code>sol.solver</code>	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

<code>sol.xe</code>	Points at which events, if any, occurred. <code>sol.xe(end)</code> contains the exact point of a terminal event, if any.
<code>sol.ye</code>	Solutions that correspond to events in <code>sol.xe</code> .
<code>sol.ie</code>	Indices into the vector returned by the function specified in the <code>Events</code> option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls

the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix  $\partial f / \partial y$  is critical to reliability and efficiency. Use `odeset` to set Jacobian to `@FJAC` if `FJAC(T,Y)` returns the Jacobian  $\partial f / \partial y$  or to the matrix  $\partial f / \partial y$  if the Jacobian is constant. If the `Jacobian` property is not set (the default),  $\partial f / \partial y$  is approximated by finite differences. Set the `Vectorized` property 'on' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If  $\partial f / \partial y$  is a sparse matrix, set the `JPattern` property to the sparsity pattern of  $\partial f / \partial y$ , i.e., a sparse matrix `S` with `S(i,j) = 1` if the *i*th component of  $f(t,y)$  depends on the *j*th component of  $y$ , and 0 otherwise.

The solvers of the ODE suite can solve problems of the form  $M(t,y)y' = f(t,y)$ , with time- and state-dependent mass matrix  $M$ . (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable  $y$  and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to 'none'.

# ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

- If the mass matrix depends weakly on  $y$ , set `MStateDependence` to 'weak' (the default); otherwise, set it to 'strong'. In either case, the function `MASS` is called with the two arguments  $(t,y)$ .

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse  $M(t, y)$ .
- Supply the sparsity pattern of  $\partial f / \partial y$  using the `JPattern` property or a sparse  $\partial f / \partial y$  using the `Jacobian` property.
- For strongly state-dependent  $M(t, y)$ , set `MvPattern` to a sparse matrix `S` with  $S(i, j) = 1$  if for any  $k$ , the  $(i, k)$  component of  $M(t, y)$  depends on component  $j$  of  $y$ , and 0 otherwise.

If the mass matrix  $M$  is singular, then  $M(t, y)y' = f(t, y)$  is a system of differential algebraic equations. DAEs have solutions only when  $y_0$  is consistent, that is, if there is a vector  $yp_0$  such that  $M(t_0, y_0)yp_0 = f(t_0, y_0)$ . The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that  $y_0$  is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to 'yes', 'no', or 'maybe'. The default value of 'maybe' causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and  $y_0$  and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that  $M$  is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
ode45	Nonstiff	Medium	Most of the time. This should be the first solver you try.

# ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

<b>Solver</b>	<b>Problem Type</b>	<b>Order of Accuracy</b>	<b>When to Use</b>
ode23	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
ode113	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
ode15s	Stiff	Low to medium	If ode45 is slow because the problem is stiff.
ode23s	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
ode23t	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
ode23tb	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 2-2517 for more details.

# ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

## Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
RelTol, AbsTol, NormControl	√	√	√	√	√	√	√
OutputFcn, OutputSel, Refine, Stats	√	√	√	√	√	√	√
NonNegative	√	√	√	√ *	—	√ *	√ *
Events	√	√	√	√	√	√	√
MaxStep, InitialStep	√	√	√	√	√	√	√
Jacobian, JPattern, Vectorized	—	—	—	√	√	√	√
Mass	√	√	√	√	√	√	√
MStateDependence	√	√	√	√	—	√	√
MvPattern	—	—	—	√	—	√	√
MassSingular	—	—	—	√	—	√	—
InitialSlope	—	—	—	√	—	√	—
MaxOrder, BDF	—	—	—	√	—	—	—

---

**Note** You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

---

## Examples

### Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned}y'_1 &= y_2 y_3 & y_1(0) &= 0 \\y'_2 &= -y_1 y_3 & y_2(0) &= 1 \\y'_3 &= -0.51 y_1 y_2 & y_3(0) &= 1\end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

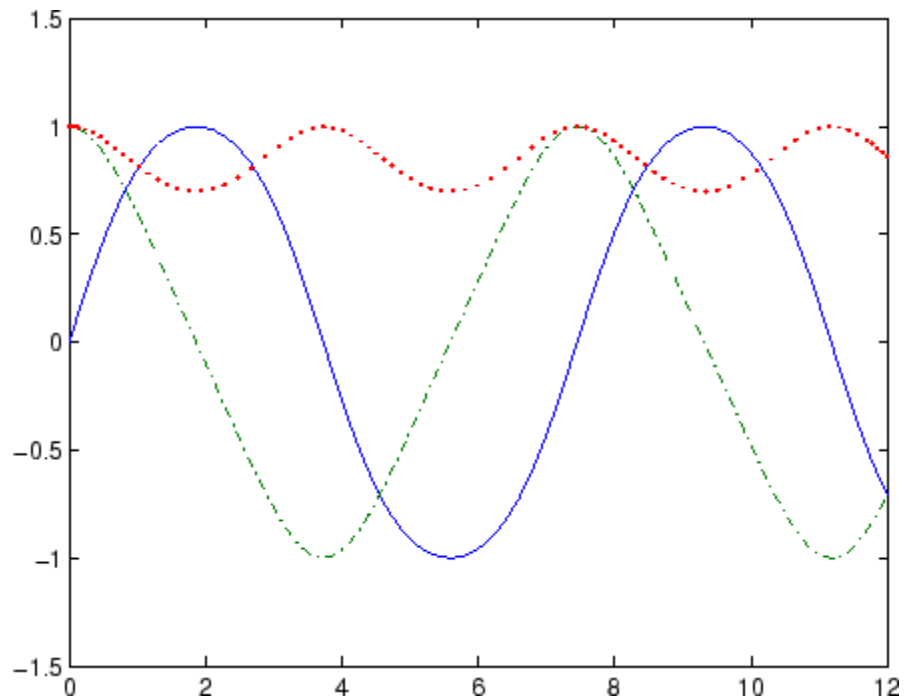
In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time 0.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'-.')
```

# ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb



## Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$\begin{aligned}y_1' &= y_2 & y_1(0) &= 2 \\ y_2' &= 1000(1 - y_1^2)y_2 - y_1 & y_2(0) &= 0\end{aligned}$$

To simulate this system, create a function `vdp1000` containing the equations

```
function dy = vdp1000(t,y)
```



# ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

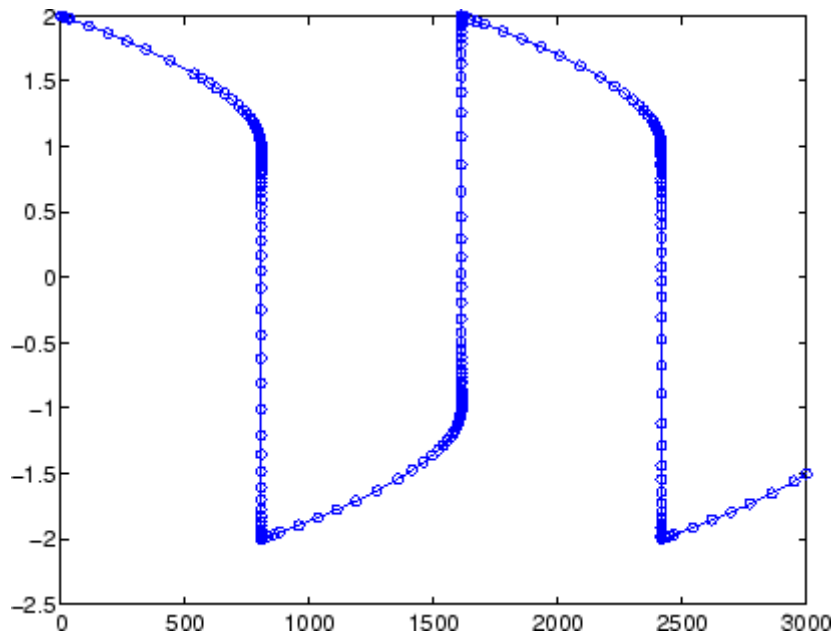
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances ( $1e-3$  and  $1e-6$ , respectively) and solve on a time interval of  $[0 \ 3000]$  with initial condition vector  $[2 \ 0]$  at time 0.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix Y versus T shows the solution

```
plot(T,Y(:,1),'-o')
```



# ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

---

## Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

$$y'(t) + f(t)y(t) = g(t)$$

The initial condition is  $y(0) = 0$ , where the function  $f(t)$  is defined through the  $n$ -by-1 vectors  $tf$  and  $f$ , and the function  $g(t)$  is defined through the  $m$ -by-1 vectors  $tg$  and  $g$ .

First, define the time-dependent parameters  $f(t)$  and  $g(t)$  as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write an M-file function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evaluate ODE at time t
```

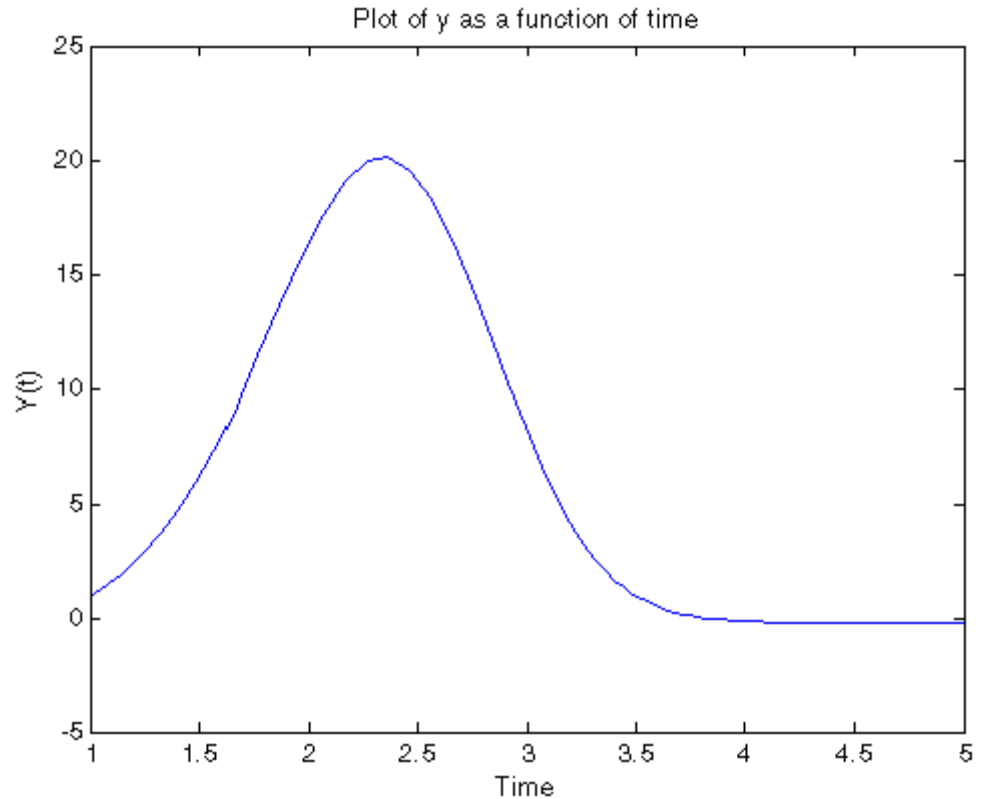
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution  $y(t)$  as a function of time:

```
plot(T, Y);
```

```
title('Plot of y as a function of time');  
xlabel('Time'); ylabel('Y(t)');
```



## Algorithms

ode45 is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing  $y(t_n)$ , it needs only the solution at the immediately preceding time point,  $y(t_{n-1})$ . In general, ode45 is the best function to apply as a *first try* for most problems. [3]

ode23 is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than ode45 at crude

## ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

---

tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear’s method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

### See Also

`deval`, `ode15i`, `odeget`, `odeset`, `function_handle` (@)

## References

- [1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp 436-451.
- [2] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp 1-9.
- [3] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp 19-26.
- [4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.
- [5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.
- [6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.
- [7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.
- [8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics* 20, 1996.
- [9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing*, Vol. 18, 1997, pp 1-22.
- [10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp 538-552.

# odefile

---

## Purpose

Define differential equation problem for ordinary differential equation solvers

---

**Note** This reference page describes the `odefile` and the syntax of the ODE solvers used in MATLAB, Version 5. MATLAB, Version 6, supports the `odefile` for backward compatibility, however the new solver syntax does not use an ODE file. New functionality is available only with the new syntax. For information about the new syntax, see `odeset` or any of the ODE solvers.

---

## Description

`odefile` is not a command or function. It is a help entry that describes how to create an M-file defining the system of equations to be solved. This definition is the first step in using any of the MATLAB ODE solvers. In MATLAB documentation, this M-file is referred to as an `odefile`, although you can give your M-file any name you like.

You can use the `odefile` M-file to define a system of differential equations in one of these forms

$$y' = f(t, y)$$

or

$$M(t, y)y' = f(t, y)v$$

where:

- $t$  is a scalar independent variable, typically representing time.
- $y$  is a vector of dependent variables.
- $f$  is a function of  $t$  and  $y$  returning a column vector the same length as  $y$ .
- $M(t, y)$  is a time-and-state-dependent mass matrix.

The ODE file must accept the arguments  $t$  and  $y$ , although it does not have to use them. By default, the ODE file must return a column vector the same length as  $y$ .

All of the solvers of the ODE suite can solve  $M(t, y)y' = f(t, y)$ , except `ode23s`, which can only solve problems with constant mass matrices. The `ode15s` and `ode23t` solvers can solve some differential-algebraic equations (DAEs) of the form  $M(t)y' = f(t, y)$ .

Beyond defining a system of differential equations, you can specify an entire initial value problem (IVP) within the ODE M-file, eliminating the need to supply time and initial value vectors at the command line (see “Examples” on page 2-2523).

### To Use the ODE File Template

- Enter the command `help odefile` to display the help entry.
- Cut and paste the ODE file text into a separate file.
- Edit the file to eliminate any cases not applicable to your IVP.
- Insert the appropriate information where indicated. The definition of the ODE system is required information.

```
switch flag
case '' % Return dy/dt = f(t,y).
    varargout{1} = f(t,y,p1,p2);
case 'init' % Return default [tspan,y0,options].
    [varargout{1:3}] = init(p1,p2);
case 'jacobian' % Return Jacobian matrix df/dy.
    varargout{1} = jacobian(t,y,p1,p2);
case 'jpattern' % Return sparsity pattern matrix S.
    varargout{1} = jpattern(t,y,p1,p2);
case 'mass' % Return mass matrix.
    varargout{1} = mass(t,y,p1,p2);
case 'events' % Return [value,isterminal,direction].
    [varargout{1:3}] = events(t,y,p1,p2);
otherwise
    error(['Unknown flag '' flag ''.']);
```

```
end
% -----
function dydt = f(t,y,p1,p2)
    dydt = Insert a function of t and/or y, p1, and p2 here.>
% -----
function [tspan,y0,options] = init(p1,p2)
    tspan = <Insert tspan here.>;
    y0 = <Insert y0 here.>;
    options = <Insert options = odeset(...) or [] here.>;
% -----
function dfdy = jacobian(t,y,p1,p2)
    dfdy = <Insert Jacobian matrix here.>;
% -----
function S = jpattern(t,y,p1,p2)
    S = <Insert Jacobian matrix sparsity pattern here.>;
% -----
function M = mass(t,y,p1,p2)
    M = <Insert mass matrix here.>;
% -----
function [value,isterminal,direction] = events(t,y,p1,p2)
    value = <Insert event function vector here.>
    isterminal = <Insert logical ISTERMINAL vector here.>;
    direction = <Insert DIRECTION vector here.>;
```

## Notes

- 1** The ODE file must accept  $t$  and  $y$  vectors from the ODE solvers and must return a column vector the same length as  $y$ . The optional input argument `flag` determines the type of output (mass matrix, Jacobian, etc.) returned by the ODE file.
- 2** The solvers repeatedly call the ODE file to evaluate the system of differential equations at various times. *This is required information* – you must define the ODE system to be solved.
- 3** The switch statement determines the type of output required, so that the ODE file can pass the appropriate information to the solver. (See notes 4 - 9.)



- 4 In the default *initial conditions* ('init') case, the ODE file returns basic information (time span, initial conditions, options) to the solver. If you omit this case, you must supply all the basic information on the command line.
- 5 In the 'jacobian' case, the ODE file returns a Jacobian matrix to the solver. You need only provide this case when you want to improve the performance of the stiff solvers ode15s, ode23s, ode23t, and ode23tb.
- 6 In the 'jpattern' case, the ODE file returns the Jacobian sparsity pattern matrix to the solver. You need to provide this case only when you want to generate sparse Jacobian matrices numerically for a stiff solver.
- 7 In the 'mass' case, the ODE file returns a mass matrix to the solver. You need to provide this case only when you want to solve a system in the form  $M(t, y)y' = f(t, y)$ .
- 8 In the 'events' case, the ODE file returns to the solver the values that it needs to perform event location. When the Events property is set to on, the ODE solvers examine any elements of the event vector for transitions to, from, or through zero. If the corresponding element of the logical isterminal vector is set to 1, integration will halt when a zero-crossing is detected. The elements of the direction vector are -1, 1, or 0, specifying that the corresponding event must be decreasing, increasing, or that any crossing is to be detected.
- 9 An unrecognized flag generates an error.

## Examples

The van der Pol equation,  $y''_1 - \mu(1 - y_1^2)y'_1 + y_1 = 0$ , is equivalent to a system of coupled first-order differential equations.

$$y'_1 = y_2$$

$$y'_2 = \mu(1 - y_1^2)y_2 - y_1$$

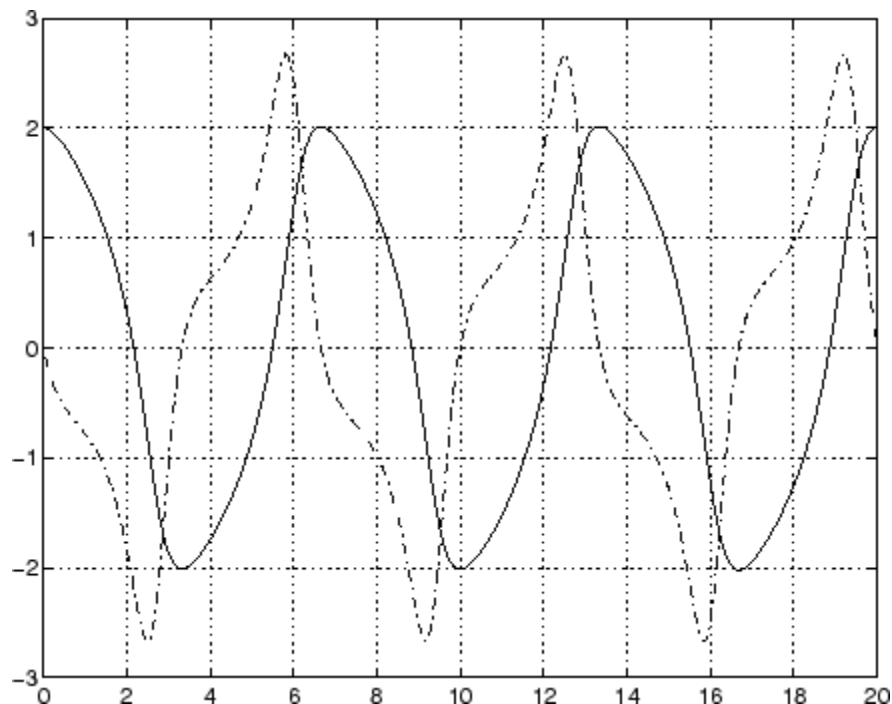
The M-file

```
function out1 = vdp1(t,y)
out1 = [y(2); (1-y(1)^2)*y(2) - y(1)];
```

defines this system of equations (with  $\mu = 1$ ).

To solve the van der Pol system on the time interval [0 20] with initial values (at time 0) of  $y(1) = 2$  and  $y(2) = 0$ , use

```
[t,y] = ode45('vdp1',[0 20],[2; 0]);
plot(t,y(:,1),'-',t,y(:,2),'-.')
```



To specify the entire initial value problem (IVP) within the M-file, rewrite vdp1 as follows.

```

function [out1,out2,out3] = vdp1(t,y,flag)
if nargin < 3 | isempty(flag)
    out1 = [y(1).*(1-y(2).^2)-y(2); y(1)];
else
    switch(flag)
        case 'init'
            % Return tspan, y0, and options.
            out1 = [0 20];
            out2 = [2; 0];
            out3 = [];
        otherwise
            error(['Unknown request '' flag ''.']);
    end
end
end

```

You can now solve the IVP without entering any arguments from the command line.

```
[t,Y] = ode23('vdp1')
```

In this example the `ode23` function looks to the `vdp1` M-file to supply the missing arguments. Note that, once you've called `odeset` to define options, the calling syntax

```
[t,Y] = ode23('vdp1',[],[],options)
```

also works, and that any options supplied via the command line override corresponding options specified in the M-file (see `odeset`).

## See Also

The MATLAB Version 5 help entries for the ODE solvers and their associated functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `odeget`, `odeset`

Type at the MATLAB command line:

`more on`, `type function`, `more off`. The Version 5 help follows the Version 6 help.

# odeget

---

**Purpose** Ordinary differential equation options parameters

**Syntax**  
`o = odeget(options,'name')`  
`o = odeget(options,'name',default)`

**Description** `o = odeget(options,'name')` extracts the value of the property specified by string 'name' from integrator options structure `options`, returning an empty matrix if the property value is not specified in `options`. It is only necessary to type the leading characters that uniquely identify the property name. Case is ignored for property names. The empty matrix `[]` is a valid `options` argument.

`o = odeget(options,'name',default)` returns `o = default` if the named property is not specified in `options`.

**Example** Having constructed an ODE options structure,

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-3 2e-3 3e-3]);
```

you can view these property settings with `odeget`.

```
odeget(options,'RelTol')
ans =

    1.0000e-04

odeget(options,'AbsTol')
ans =

    0.0010    0.0020    0.0030
```

**See Also** `odeset`

**Purpose**

Create or alter options structure for ordinary differential equation solvers

**Syntax**

```
options = odeset('name1',value1,'name2',value2,...)
options = odeset(olddopts,'name1',value1,...)
options = odeset(olddopts,newopts)
odeset
```

**Description**

The `odeset` function lets you adjust the integration parameters of the following ODE solvers.

For solving fully implicit differential equations:

```
ode15i
```

For solving initial value problems:

```
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
```

See below for information about the integration parameters.

`options = odeset('name1',value1,'name2',value2,...)` creates an options structure that you can pass as an argument to any of the ODE solvers. In the resulting structure, `options`, the named properties have the specified values. For example, `'name1'` has the value `value1`. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify a property name. Case is ignored for property names.

`options = odeset(olddopts,'name1',value1,...)` alters an existing options structure `olddopts`. This sets `options` equal to the existing structure `olddopts`, overwrites any values in `olddopts` that are respecified using name/value pairs, and adds any new pairs to the structure. The modified structure is returned as an output argument.

`options = odeset(olddopts,newopts)` alters an existing options structure `olddopts` by combining it with a new options structure `newopts`. Any new options not equal to the empty matrix overwrite corresponding options in `olddopts`.

odeset with no input arguments displays all property names as well as their possible and default values.

## ODE Properties

The following sections describe the properties that you can set using odeset. The available properties depend on the ODE solver you are using. There are several categories of properties:

- “Error Control Properties” on page 2-2528
- “Solver Output Properties” on page 2-2530
- “Step-Size Properties” on page 2-2534
- “Event Location Property” on page 2-2535
- “Jacobian Matrix Properties” on page 2-2537
- “Mass Matrix and DAE Properties” on page 2-2541
- “ode15s and ode15i-Specific Properties” on page 2-2543

---

**Note** This reference page describes the ODE properties for MATLAB, Version 7. The Version 5 properties are supported only for backward compatibility. For information on the Version 5 properties, type at the MATLAB command line: `more on`, type `odeset`, `more off`.

---

## Error Control Properties

At each step, the solver estimates the local error  $e$  in the  $i$ th component of the solution. This error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, `RelTol`, and the specified absolute tolerance, `AbsTol`.

$$|e(i)| \leq \max(\text{RelTol} * \text{abs}(y(i)), \text{AbsTol}(i))$$

For routine problems, the ODE solvers 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 `RelTol`. For

the absolute error tolerance, the scaling of the solution components is important: if  $|y|$  is somewhat smaller than `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 `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component  $y(i)$  when it is small, you may have to specify `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. Further information on each property is given following the table.

Property	Value	Description
<code>RelTol</code>	Positive scalar {1e-3}	Relative error tolerance that applies to all components of the solution vector $y$ .
<code>AbsTol</code>	Positive scalar or vector {1e-6}	Absolute error tolerances that apply to the individual components of the solution vector.
<code>NormControl</code>	on   {off}	Control error relative to norm of solution.

### Description of Error Control Properties

**RelTol** — This tolerance 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 `AbsTol(i)`.

The default, `1e-3`, corresponds to 0.1% accuracy.

**AbsTol** — `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.

If `AbsTol` is a vector, the length of `AbsTol` must be the same as the length of the solution vector `y`. If `AbsTol` is a scalar, the value applies to all components of `y`.

**NormControl** — Set this property on to request that the solvers control the error in each integration step with  $\text{norm}(e) \leq \max(\text{RelTol} \cdot \text{norm}(y), \text{AbsTol})$ . By default the solvers use a more stringent componentwise error control.

## Solver Output Properties

The following table lists the solver output properties that control the output that the solvers generate. Further information on each property is given following the table.

Property	Value	Description
NonNegative	Vector of integers	Specifies which components of the solution vector must be nonnegative. The default value is <code>[]</code> .
OutputFcn	Function handle	A function for the solver to call after every successful integration step.
OutputSel	Vector of indices	Specifies which components of the solution vector are to be passed to the output function.
Refine	Positive integer	Increases the number of output points by a factor of <code>Refine</code> .
Stats	on   {off}	Determines whether the solver should display statistics about its computations. By default, <code>Stats</code> is off.

### Description of Solver Output Properties

**NonNegative** — The `NonNegative` property is not available in `ode23s`, `ode15i`. In `ode15s`, `ode23t`, and `ode23tb`, `NonNegative` is not available for problems where there is a mass matrix.



**OutputFcn** — To specify an output function, set 'OutputFcn' to a function handle. For example,

```
options = odeset('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

```
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:

<b>Flag</b>	<b>Description</b>
init	The solver calls myfun(tspan,y0,'init') before beginning the integration to allow the output function to initialize. tspan and y0 are the input arguments to the ODE solver.

Flag	Description
{[]}	<p>The solver calls <code>status = myfun(t,y,[])</code> after each integration step on which output is requested. <code>t</code> contains points where output was generated during the step, and <code>y</code> is the numerical solution at the points in <code>t</code>. If <code>t</code> is a vector, the <code>ith</code> column of <code>y</code> corresponds to the <code>ith</code> element of <code>t</code>.</p> <p>When <code>length(tspan) &gt; 2</code> the output is produced at every point in <code>tspan</code>. When <code>length(tspan) = 2</code> the output is produced according to the <code>Refine</code> option.</p> <p><code>myfun</code> must return a <code>status</code> output value of 0 or 1. If <code>status = 1</code>, the solver halts integration. You can use this mechanism, for instance, to implement a <b>Stop</b> button.</p>
done	<p>The solver calls <code>myfun([],[],'done')</code> when integration is complete to allow the output function to perform any cleanup chores.</p>

You can use these general purpose output functions or you can edit them to create your own. Type `help function` at the command line for more information.

- `odeplot` — Time series plotting (default when you call the solver with no output arguments and you have not specified an output function)
- `odephas2` — Two-dimensional phase plane plotting
- `odephas3` — Three-dimensional phase plane plotting
- `odeprint` — Print solution as it is computed

---

**Note** If you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history.

---

**OutputSel** — Use `OutputSel` to specify which components of the solution vector you want passed 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

```
options = ...  
odeset('OutputFcn',@odeplot,'OutputSel',[1 3]);
```

By default, the solver passes all components of the solution to the output function.

**Refine** — If `Refine` is 1, the solver returns solutions only at the end of each time step. If `Refine` is  $n > 1$ , the solver subdivides each time step into  $n$  smaller intervals and returns solutions at each time point. `Refine` does not apply when `length(tspan) > 2`.

---

**Note** In all the solvers, the default value of `Refine` is 1. Within `ode45`, however, the default is 4 to compensate for the solver's large step sizes. To override this and see only the time steps chosen by `ode45`, set `Refine` to 1.

---

The extra values produced for `Refine` are computed by means of continuous extension formulas. These are specialized formulas used by the ODE solvers to obtain accurate solutions between computed time steps without significant increase in computation time.

**Stats** — By default, `Stats` is `off`. If it is on, after solving the problem the solver displays

- Number of successful steps
- Number of failed attempts
- Number of times the ODE function was called to evaluate  $f(t,y)$

Solvers based on implicit methods, including `ode23s`, `ode23t`, `ode23t`, `ode15s`, and `ode15i`, also display

- Number of times that the partial derivatives matrix  $\partial f / \partial x$  was formed
- Number of LU decompositions
- Number of solutions of linear systems

## Step-Size Properties

The step-size properties 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. Further information on each property is given following the table.

Property	Value	Description
InitialStep	Positive scalar	Suggested initial step size.
MaxStep	Positive scalar {0.1*abs(t0-tf)}	Upper bound on solver step size.

### Description of Step-Size Properties

**InitialStep** — InitialStep sets an upper bound on the magnitude of the first step size the solver tries. If you do not set InitialStep, the initial step size is based on the slope of the solution at the initial time `tspan(1)`, and 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.

**MaxStep** — If the differential equation has periodic coefficients or solutions, it might 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 for any of the following purposes:

- To produce more output points. This can significantly slow down solution time. Instead, use `Refine` to compute additional outputs by continuous extension at very low cost.
- 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 2-2528 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 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 ODE problems the times of specific events are important, such as the time at which a ball hits the ground, or the time at which a spaceship returns to the earth. While solving a problem, the ODE solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the `Events` property. Further information on each property is given following the table.

### ODE Events Property

String	Value	Description
Events	Function handle	Handle to a function that includes one or more event functions.

### Description of Event Location Properties

**Events** — The function is of the form

```
[value, isterminal, direction] = events(t,y)
```

`value`, `isterminal`, and `direction` are vectors for which the `ith` element corresponds to the `ith` event function:

- `value(i)` is the value of the `ith` event function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function, otherwise, 0.
- `direction(i) = 0` if all zeros are to be located (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 outputs:

- A column vector of times at which events occur
- Solution values corresponding to these times
- Indices into the vector returned by the events function. The values indicate which event the solver detected.

If you call the solver as

```
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
```

the solver returns these outputs as `TE`, `YE`, and `IE` respectively. If you call the solver as

```
sol = solver(odefun,tspan,y0,options)
```

the solver returns these outputs as `sol.xe`, `sol.ye`, and `sol.ie`, respectively.

For examples that use an event function, see “Event Location” and “Advanced Event Location” in the MATLAB Mathematics documentation.

## Jacobian Matrix Properties

The stiff ODE solvers often execute faster if you provide additional information about the Jacobian matrix  $\partial f / \partial y$ , a matrix of partial derivatives of the function that defines the differential equations.

$$\frac{\partial f}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

The Jacobian matrix properties pertain only to those solvers for stiff problems (`ode15s`, `ode23s`, `ode23t`, `ode23tb`, and `ode15i`) for which the Jacobian matrix  $\partial f / \partial y$  can be critical to reliability and efficiency. If you do not provide a function to calculate the Jacobian, these solvers approximate the Jacobian numerically using finite differences. In this case, you might want to use the `Vectorized` or `JPattern` properties.

The following table describes the Jacobian matrix properties for all implicit solvers except `ode15i`. Further information on each property is given following the table. See [Jacobian Properties for `ode15i`](#) on page 2-2540 for `ode15i`-specific information.

### Jacobian Properties for All Implicit Solvers Except `ode15i`

Property	Value	Description
Jacobian	Function   handle constant matrix	Matrix or function that evaluates the Jacobian.

### Jacobian Properties for All Implicit Solvers Except ode15i (Continued)

Property	Value	Description
JPattern	Sparse matrix of {0,1}	Generates a sparse Jacobian matrix numerically.
Vectorized	on   {off}	Allows the solver to reduce the number of function evaluations required.

#### Description of Jacobian Properties

**Jacobian** — Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function FJac, where  $FJac(t,y)$  computes  $\partial f / \partial y$ , or to the constant value of  $\partial f / \partial y$ .

The Jacobian for the “van der Pol Equation (Stiff)”, described in the MATLAB Mathematics documentation, can be coded as

```
function J = vdp1000jac(t,y)
J = [ 0 1
      (-2000*y(1)*y(2)-1) (1000*(1-y(1)^2)) ];
```

**JPattern** — JPattern is a sparsity pattern with 1s where there might be nonzero entries in the Jacobian.

---

**Note** If you specify Jacobian, the solver ignores any setting for JPattern.

---

Set this property to a sparse matrix  $S$  with  $S(i,j) = 1$  if component  $i$  of  $f(t,y)$  depends on component  $j$  of  $y$ , and 0 otherwise. The solver uses this sparsity pattern to generate a sparse Jacobian matrix numerically. If the Jacobian matrix is large and sparse, this can greatly



accelerate execution. For an example using the `JPattern` property, see [Example: Large, Stiff, Sparse Problem](#) in the MATLAB Mathematics documentation.

**Vectorized** — The `Vectorized` property allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time.

Set `on` to inform the solver that you have coded the ODE function `F` so that `F(t,[y1 y2 ...])` returns `[F(t,y1) F(t,y2) ...]`. This allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time.

---

**Note** If you specify `Jacobian`, the solver ignores a setting of `'on'` for `'Vectorized'`.

---

With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. For example, you can vectorize the “van der Pol Equation (Stiff)”, described in the MATLAB Mathematics documentation, by introducing colon notation into the subscripts and by using the array power (`.`<sup>`^`</sup>) and array multiplication (`.`<sup>`*`</sup>) operators.

```
function dydt = vdp1000(t,y)
    dydt = [y(2,:); 1000*(1-y(1,:).^2).*y(2,:)-y(1,:)];
```

---

**Note** Vectorization of the ODE function used by the ODE solvers differs from the vectorization used by the boundary value problem (BVP) solver, `bvp4c`. For the ODE solvers, the ODE function is vectorized only with respect to the second argument, while `bvp4c` requires vectorization with respect to the first and second arguments.

---

The following table describes the Jacobian matrix properties for `ode15i`.

## Jacobian Properties for ode15i

Property	Value	Description
Jacobian	Function handle   Cell array of constant values	Function that evaluates the Jacobian or a cell array of constant values.
JPattern	Sparse matrices of {0,1}	Generates a sparse Jacobian matrix numerically.
Vectorized	on   {off}	Vectorized ODE function

## Description of Jacobian Properties for ode15i

**Jacobian** — Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function

$$[dFdy, dFdp] = Fjac(t,y,yp)$$

or to a cell array of constant values  $\{\partial F/\partial y, (\partial F/\partial y)'\}$ .

**JPattern** — JPattern is a sparsity pattern with 1's where there might be nonzero entries in the Jacobian.

Set this property to {dFdyPattern, dFdypPattern}, the sparsity patterns of  $\partial F/\partial y$  and  $\partial F/\partial y'$ , respectively.

**Vectorized** —

Set this property to {yVect, ypVect}. Setting yVect to 'on' indicates that

$$F(t, [y1\ y2\ \dots], yp)$$

returns

$$[F(t,y1,yp), F(t,y2,yp)\ \dots]$$

Setting ypVect to 'on' indicates that

$$F(t, y, [yp1 \ yp2 \ \dots])$$

returns

$$[F(t, y, yp1) \ F(t, y, yp2) \ \dots]$$

## Mass Matrix and DAE Properties

This section describes mass matrix and differential-algebraic equation (DAE) properties, which apply to all the solvers except `ode15i`. These properties are not applicable to `ode15i` and their settings do not affect its behavior.

The solvers of the ODE suite can solve ODEs of the form

$$M(t, y)y' = f(t, y) \tag{2-1}$$

with a mass matrix  $M(t, y)$  that can be sparse.

When  $M(t, y)$  is nonsingular, the equation above is equivalent to

$y' = M^{-1}f(t, y)$  and the ODE has a solution for any initial values  $y_0$  at  $t_0$ . The more general form (Equation 2-1) is convenient when you express a model naturally in terms of a mass matrix. For large, sparse  $M(t, y)$ , solving Equation 2-1 directly reduces the storage and run-time needed to solve the problem.

When  $M(t, y)$  is singular, then  $M(t, y)$  times  $M(t, y)y' = f(t, y)$  is a DAE. A DAE has a solution only when  $y_0$  is consistent; that is, there exists an initial slope  $yp_0$  such that  $M(t_0, y_0)yp_0 = f(t_0, y_0)$ . If  $y_0$  and  $yp_0$  are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. For DAEs of index 1, solving an initial value problem with consistent initial conditions is much like solving an ODE.

The `ode15s` and `ode23t` solvers can solve DAEs of index 1. For examples of DAE problems, see Example: Differential-Algebraic Problem, in the MATLAB Mathematics documentation, and the examples `amp1dae` and `hb1dae`.

The following table describes the mass matrix and DAE properties. Further information on each property is given following the table.

## Mass Matrix and DAE Properties (Solvers Other Than ode15i)

Property	Value	Description
Mass	Matrix   function handle	Mass matrix or a function that evaluates the mass matrix $M(t, y)$ .
MStateDependence	none   {weak}   strong	Dependence of the mass matrix on $y$ .
MvPattern	Sparse matrix	$\partial(M(t, y)v) / \partial y$ sparsity pattern.
MassSingular	yes   no   {maybe}	Indicates whether the mass matrix is singular.
InitialSlope	Vector {zero vector}	Vector representing the consistent initial slope $yp_0$ .

## Description of Mass Matrix and DAE Properties

**Mass** — For problems of the form  $M(t)y' = f(t, y)$ , set 'Mass' to a mass matrix  $M$ . For problems of the form  $M(t)y' = f(t, y)$ , set 'Mass' to a function handle @Mfun, where Mfun(t, y) evaluates the mass matrix  $M(t, y)$ . The ode23s solver can only solve problems with a constant mass matrix  $M$ . When solving DAEs, using ode15s or ode23t, it is advantageous to formulate the problem so that  $M$  is a diagonal matrix (a semiexplicit DAE).

For example problems, see “Finite Element Discretization” in the MATLAB Mathematics documentation, or the examples fem2ode or batonode.

**MStateDependence** — Set this property to none for problems

$M(t)y' = f(t, y)$ . Both `weak` and `strong` indicate  $M(t, y)$ , but `weak` results in implicit solvers using approximations when solving algebraic equations.

**MvPattern** — Set this property to a sparse matrix  $S$  with  $S(i, j) = 1$  if, for any  $k$ , the  $(i, k)$  component of  $M(t, y)$  depends on component  $j$  of  $y$ , and 0 otherwise. For use with the `ode15s`, `ode23t`, and `ode23tb` solvers when `MStateDependence` is `strong`. See `burgersode` as an example.

**MassSingular** — Set this property to no if the mass matrix is not singular and you are using either the `ode15s` or `ode23t` solver. The default value of `maybe` causes the solver to test whether the problem is a DAE, by testing whether  $M(t_0, y_0)$  is singular.

**InitialSlope** — Vector representing the consistent initial slope  $yp_0$ , where  $yp_0$  satisfies  $M(t_0, y_0) \cdot y'_0 = f(t_0, y_0)$ . The default is the zero vector.

This property is for use with the `ode15s` and `ode23t` solvers when solving DAEs.

## ode15s and ode15i-Specific Properties

`ode15s` is a variable-order solver for stiff problems. It is based on the numerical differentiation formulas (NDFs). The NDFs are generally more efficient than the closely related family of backward differentiation formulas (BDFs), also known as Gear's methods. The `ode15s` properties let you choose among these formulas, as well as specifying the maximum order for the formula used.

`ode15i` solves fully implicit differential equations of the form

$$f(t, y, y') = 0$$

using the variable order BDF method.

The following table describes the `ode15s` and `ode15i`-specific properties. Further information on each property is given following the table. Use `odeset` to set these properties.

## ode15s and ode15i-Specific Properties

Property	Value	Description
MaxOrder	1   2   3   4   {5}	Maximum order formula used to compute the solution.
BDF (ode15s only)	on   {off}	Specifies whether you want to use the BDFs instead of the default NDFs.

### Description of ode15s and ode15i-Specific Properties

**MaxOrder** — Maximum order formula used to compute the solution.

**BDF** (ode15s only) — Set BDF on to have ode15s use the BDFs.

For both the NDFs and BDFs, the formulas of orders 1 and 2 are A-stable (the stability region includes the entire left half complex plane). The higher order formulas are not as stable, and the higher the order the worse the stability. There is a class of stiff problems (stiff oscillatory) that is solved more efficiently if MaxOrder is reduced (for example to 2) so that only the most stable formulas are used.

### See Also

deval, odeget, ode45, ode23, ode23t, ode23tb, ode113, ode15s, ode23s, function\_handle (@)

**Purpose** Extend solution of initial value problem for ordinary differential equation

**Syntax**

```

solext = odextend(sol, odefun, tfinal)
solext = odextend(sol, [], tfinal)
solext = odextend(sol, odefun, tfinal, yinit)
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
solext = odextend(sol, odefun, tfinal, yinit, options)

```

**Description** `solext = odextend(sol, odefun, tfinal)` extends the solution stored in `sol` to an interval with upper bound `tfinal` for the independent variable. `odefun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. `sol` is an ODE solution structure created using an ODE solver. The lower bound for the independent variable in `solext` is the same as in `sol`. If you created `sol` with an ODE solver other than `ode15i`, the function `odefun` computes the right-hand side of the ODE equation, which is of the form  $y' = f(t, y)$ . If you created `sol` using `ode15i`, the function `odefun` computes the left-hand side of the ODE equation, which is of the form  $f(t, y, y') = 0$ .

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `odefun`, if necessary.

`odextend` extends the solution by integrating `odefun` from the upper bound for the independent variable in `sol` to `tfinal`, using the same ODE solver that created `sol`. By default, `odextend` uses

- The initial conditions `y = sol.y(:, end)` for the subsequent integration
- The same integration properties and additional input arguments the ODE solver originally used to compute `sol`. This information is stored as part of the solution structure `sol` and is subsequently passed to `solext`. Unless you want to change these values, you do not need to pass them to `odextend`.

# odextend

---

`solext = odextend(sol, [], tfinal)` uses the same ODE function that the ODE solver uses to compute `sol` to extend the solution. It is not necessary to pass in `odefun` explicitly unless it differs from the original ODE function.

`solext = odextend(sol, odefun, tfinal, yinit)` uses the column vector `yinit` as new initial conditions for the subsequent integration, instead of the vector `sol.y(end)`.

---

**Note** To extend solutions obtained with `ode15i`, use the following syntax, in which the column vector `ypinit` is the initial derivative of the solution:

---

```
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
```

---

`solext = odextend(sol, odefun, tfinal, yinit, options)` uses the integration properties specified in `options` instead of the options the ODE solver originally used to compute `sol`. The new options are then stored within the structure `solext`. See `odeset` for details on setting options properties. Set `yinit = []` as a placeholder to specify the default initial conditions.

## Example

The following command

```
sol=ode45(@vdp1,[0 10],[2 0]);
```

uses `ode45` to solve the system  $y' = \text{vdp1}(t,y)$ , where `vdp1` is an example of an ODE function provided with MATLAB software, on the interval `[0 10]`. Then, the commands

```
sol=odextend(sol,@vdp1,20);  
plot(sol.x,sol.y(1,:));
```

extend the solution to the interval `[0 20]` and plot the first component of the solution on `[0 20]`.



**See Also**

deval, ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb, ode15i, odeset, odeget, deval, function\_handle (@)

# onCleanup

---

**Purpose** Cleanup tasks at function completion

**Syntax** `C = onCleanup(S)`

**Description** `C = onCleanup(S)` when called in function `F`, specifies any cleanup tasks that need to be performed when `F` completes. `S` is a handle to a function that performs necessary cleanup work when `F` exits. (For example, closing files that have been opened by `F`). `S` is called whether `F` exits normally or because of an error.

`onCleanup` is a MATLAB class and `C = onCleanup(S)` constructs an instance `C` of that class. Whenever an object of this class is explicitly or implicitly cleared from the workspace, it runs the cleanup function, `S`. Objects that are local variables in a function are implicitly cleared at the termination of that function.

**Examples** Use `onCleanup` to close a file in the first example, and to restore the current directory in the second:

```
function fileOpenSafely(fileName)
    fid = fopen(fileName, 'w');
    c = onCleanup(@()fclose(fid));

    functionThatMayError(fid);
end % c executes fclose(fid) here
```

MATLAB closes `fid` whether `functionThatMayError` returns an error or not.

```
function changeDirectorySafely(fileName)
    currentDir = pwd;
    c = onCleanup(@()cd(currentDir));

    functionThatMayError;
end % c executes cd(currentDir) here
```

The current directory is preserved whether `functionThatMayError` returns an error or not.

### See Also

`clear`, `clearvars`

**Purpose** Create array of all ones

**Syntax**

```
Y = ones(n)
Y = ones(m,n)
Y = ones([m n])
Y = ones(m,n,p,...)
Y = ones([m n p ...])
Y = ones(size(A))
ones(m, n,...,classname)
ones([m,n,...],classname)
```

**Description**

`Y = ones(n)` returns an  $n$ -by- $n$  matrix of 1s. An error message appears if  $n$  is not a scalar.

`Y = ones(m,n)` or `Y = ones([m n])` returns an  $m$ -by- $n$  matrix of ones.

`Y = ones(m,n,p,...)` or `Y = ones([m n p ...])` returns an  $m$ -by- $n$ -by- $p$ -by-... array of 1s.

---

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

---

`Y = ones(size(A))` returns an array of 1s that is the same size as  $A$ .

`ones(m, n,...,classname)` or `ones([m,n,...],classname)` is an  $m$ -by- $n$ -by-... array of ones of data type `classname`. `classname` is a string specifying the data type of the output. `classname` can have the following values: 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'.

**Example**

```
x = ones(2,3,'int8');
```

**See Also** eye, zeros, complex

**Purpose** Open files based on extension

**Syntax** `open('name')`

**Description** `open('name')` opens the object specified by the string `name`. The specific action taken upon opening depends on the type of object specified by `name`.

<b>name</b>	<b>Action</b>
DOC file (*.doc)	Open document in Microsoft Word application.
EXE file (*.exe)	Run Microsoft Windows executable file.
Figure file (*.fig)	Open figure in a MATLAB figure window.
HTML file (*.html, *.htm)	Open HTML document in a separate window.
M-file (name.m)	Open M-file <code>name</code> in M-file Editor.
MAT-file (name.mat)	Open MAT-file and store variables in a structure in the workspace.
Model (name.mdl)	Open model <code>name</code> in Simulink application.
P-file (name.p)	Open the corresponding M-file, <code>name.m</code> , if it exists, in the M-file Editor.
PDF file (*.pdf)	Open PDF document in Adobe® Acrobat® application.
PPT file (*.ppt)	Open document in Microsoft PowerPoint® application.
Project file (*.prj)	Open the project file in the MATLAB Compiler Deployment Tool. If the MATLAB Compiler or Deployment Tool is not installed, open the project file in a text editor.
URL file (*.url)	Open an Internet location in your default Web browser

# open

---

<b>name</b>	<b>Action</b>
Variable	Open array name in the Variable Editor (the array must be numeric).
Other extensions (name.xxx)	Open name.xxx by calling the helper function openxxx, where openxxx is a user-defined function.
No extension (name)	Open name in the default editor. If name does not exist, then open checks to see if name.mdl or name.m is on the path or in the current directory and, if so, opens the file returned by which('name').

If more than one file with the specified filename name exists on the MATLAB path, then open opens the file returned by which('name').

If no such file name exists, then open displays an error message.

You can create your own openxxx functions to set up handlers for new file types. This does not apply to the file types shown in the table above. open('filename.xxx') calls the openxxx function it finds on the path. For example, create a function openlog if you want a handler for opening files with file extension .log.

## Examples

### Example 1 – Opening a File on the Path

To open the M-file copyfile.m, type

```
open copyfile.m
```

MATLAB opens the copyfile.m file that resides in toolbox\matlab\general. If you have a copyfile.m file in a directory that is before toolbox\matlab\general on the MATLAB path, then open opens that file instead.

### Example 2 – Opening a File Not on the Path

To open a file that is not on the MATLAB path, enter the complete file specification. If no such file is found, then MATLAB displays an error message.

```
open('D:\temp\data.mat')
```

### Example 3 – Specifying a File Without a File Extension

When you specify a file without including its file extension, MATLAB determines which file to open for you. It does this by calling

```
which('filename')
```

In this example, `open matrixdemos` could open either an M-file or a Simulink model of the same name, since both exist on the path.

```
dir matrixdemos.*  
  
matrixdemos.m    matrixdemos.mdl
```

Because the call `which('matrixdemos')` returns the name of the Simulink model, `open` opens the `matrixdemos` model rather than the M-file of that name.

```
open matrixdemos           % Opens model matrixdemos.mdl
```

### Example 4 – Opening a MAT-File

This example opens a MAT-file containing MATLAB data and then keeps just one of the variables from that file. The others are overwritten when `ans` is reused by MATLAB.

```
% Open a MAT-file containing miscellaneous data.  
open D:\temp\data.mat
```

```
ans =  
  
    x: [3x2x2 double]  
    y: {4x5 cell}
```

```
k: 8
spArray: [5x5 double]
dblArray: [4x1 java.lang.Double[][]]
strArray: {2x5 cell}
```

```
% Keep the dblArray value by assigning it to a variable.
dbl = ans.dblArray
```

```
dbl =
```

```
java.lang.Double[][]:
 [ 5.7200] [ 6.7200] [ 7.7200]
 [10.4400] [11.4400] [12.4400]
 [15.1600] [16.1600] [17.1600]
 [19.8800] [20.8800] [21.8800]
```

## Example 5 – Using a User-Defined Handler Function

If you create an M-file function called `opencht` to handle files with extension `.cht`, and then issue the command

```
open myfigure.cht
```

`open` calls your handler function with the following syntax:

```
opencht('myfigure.cht')
```

## See Also

`edit`, `load`, `save`, `saveas`, `uiopen`, `which`, `file_formats`, `path`



**Purpose**

Open new copy or raise existing copy of saved figure

**Syntax**

```
openfig('filename.fig','new')  
openfig('filename.fig','new','visible')  
openfig('filename.fig','new','visible')  
openfig('filename.fig','reuse')  
openfig('filename.fig')  
openfig(...,'PropertyName',PropertyValue,...)  
figure_handle = openfig(...)
```

**Description**

openfig is designed for use with GUI figures. Use this function to:

- Open the FIG-file creating the GUI and ensure it is displayed on screen. This provides compatibility with different screen sizes and resolutions.
- Control whether the MATLAB software displays one or multiple instances of the GUI at any given time.
- Return the handle of the figure created, which is typically hidden for GUI figures.

openfig('filename.fig','new') opens the figure contained in the FIG-file, filename.fig, and ensures it is visible and positioned completely on screen. You do not have to specify the full path to the FIG-file as long as it is on your MATLAB path. The .fig extension is optional.

openfig('filename.fig','new','invisible') or  
openfig('filename.fig','reuse','invisible') opens the figure as in the preceding example, while forcing the figure to be invisible.

openfig('filename.fig','new','visible') or  
openfig('filename.fig','new','visible') opens the figure, while forcing the figure to be visible.

openfig('filename.fig','reuse') opens the figure contained in the FIG-file only if a copy is not currently open; otherwise openfig brings

# openfig

---

the existing copy forward, making sure it is still visible and completely on screen.

`openfig('filename.fig')` is the same as  
`openfig('filename.fig','new')`.

`openfig(...,'PropertyName',PropertyValue,...)` opens the FIG-file setting the specified figure properties before displaying the figure.

`figure_handle = openfig(...)` returns the handle to the figure.

## Remarks

If the FIG-file contains an invisible figure, `openfig` returns its handle and leaves it invisible. The caller should make the figure visible when appropriate.

## See Also

`guide`, `guihandles`, `movegui`, `open`, `hgload`, `save`

See *Deploying User Interfaces* in the MATLAB documentation for related functions

**Purpose** Control OpenGL rendering

**Syntax**

```
opengl info
s = opengl('data')
opengl software
opengl hardware
opengl verbose
opengl quiet
opengl DriverBugWorkaround
opengl('DriverBugWorkaround',WorkaroundState)
```

**Description** The OpenGL autoselection mode applies when the `RendererMode` of the figure is `auto`. Possible values for `selection_mode` are

- `autoselect` – allows OpenGL to be automatically selected if OpenGL is available and if there is graphics hardware on the host machine.
- `neverselect` – disables autoselection of OpenGL.
- `advise` – prints a message to the command window if OpenGL rendering is advised, but `RenderMode` is set to `manual`.

`opengl`, by itself, returns the current autoselection state.

Note that the autoselection state only specifies whether OpenGL should or should not be considered for rendering; it does not explicitly set the rendering to OpenGL. You can do this by setting the `Renderer` property of the figure to `OpenGL`. For example,

```
set(figure_handle, 'Renderer', 'OpenGL')
```

`opengl info` prints information with the version and vendor of the OpenGL on your system. Also indicates whether your system is currently using hardware or software OpenGL and the state of various driver bug workarounds. Note that calling `opengl info` loads the OpenGL Library.

For example, the following output is generated on a Windows XP computer that uses ATI Technologies graphics hardware:

```
>> opengl info
Version          = 1.3.4010 WinXP Release
Vendor           = ATI Technologies Inc.
Renderer        = RADEON 9600SE x86/SSE2
MaxTextureSize  = 2048
Visual          = 05 (RGB 16 bits(05 06 05 00) zdepth 16, Hardware
Accelerated, OpenGL, Double Buffered, Window)
Software        = false
# of Extensions = 85
Driver Bug Workarounds:
OpenGLBitmapZbufferBug   = 0
OpenGLWobbleTesselatorBug = 0
OpenGLLineSmoothingBug  = 0
OpenGLDockingBug        = 0
OpenGLClippedImageBug   = 0
```

Note that different computer systems may not list all OpenGL bugs.

`s = opengl('data')` returns a structure containing the same data that is displayed when you call `opengl info`, with the exception of the driver bug workaround state.

`opengl software` forces the MATLAB software to use software OpenGL rendering instead of hardware OpenGL. Note that Macintosh systems do not support software OpenGL.

`opengl hardware` reverses the `opengl software` command and enables MATLAB to use hardware OpenGL rendering if it is available. If your computer does not have OpenGL hardware acceleration, MATLAB automatically switches to software OpenGL rendering (except on Macintosh systems, which do not support software OpenGL).

Note that on UNIX systems, the `software` or `hardware` options with the `opengl` command works only if MATLAB has not yet used the OpenGL renderer or you have not issued the `opengl info` command (which attempts to load the OpenGL Library).

`opengl verbose` displays verbose messages about OpenGL initialization (if OpenGL is not already loaded) and other runtime messages.

`opengl quiet` disables verbose message setting.

`opengl DriverBugWorkaround` queries the state of the specified driver bug workaround. Use the command `opengl info` to see a list of all driver bug workarounds. See “Driver Bug Workarounds” on page 2-2559 for more information.

`opengl('DriverBugWorkaround',WorkaroundState)` sets the state of the specified driver bug workaround. You can set `WorkaroundState` to one of three values:

- 0 – Disable the specified *DriverBugWorkaround* (if enabled) and do not allow MATLAB to autoselect this workaround.
- 1 – Enable the specified *DriverBugWorkaround*.
- -1 – Set the specified *DriverBugWorkaround* to autoselection mode, which allows MATLAB to enable this workaround if the requisite conditions exist.

## Driver Bug Workarounds

The MATLAB software enables various OpenGL driver bug workarounds when it detects certain known problems with installed hardware. However, because there are many versions of graphics drivers, you might encounter situations when MATLAB does not enable a workaround that would solve a problem you are having with OpenGL rendering.

This section describes the symptoms that each workaround is designed to correct so you can decide if you want to try using one to fix an OpenGL rendering problem.

Use the `opengl info` command to see what driver bug workarounds are available on your computer.

---

**Note** These workarounds have not been tested under all driver combinations and therefore might produce undesirable results under certain conditions.

---

## **OpenGLBitmapZbufferBug**

Symptom: text with background color (including data tips) and text displayed on image, patch, or surface objects is not visible when using OpenGL renderer.

Possible side effect: text is always on top of other objects.

Command to enable:

```
opengl('OpenGLBitmapZbufferBug',1)
```

## **OpenGLWobbleTesselatorBug**

Symptom: Rendering complex patch object causes segmentation violation and returns a tessellator error message in the stack trace.

Command to enable:

```
opengl('OpenGLWobbleTesselatorBug',1)
```

## **OpenGLLineSmoothingBug**

Symptom: Lines with a LineWidth greater than 3 look bad.

Command to enable:

```
opengl('OpenGLLineSmoothingBug',1)
```

## **OpenGLDockingBug**

Symptom: MATLAB crashes when you dock a figure that has its Renderer property set to opengl.

Command to enable:

```
opengl('OpenGLDockingBug',1)
```

### **OpenGLClippedImageBug**

Symptom: Images (as well as colorbar displays) do not display when the `Renderer` property set to `opengl`.

Command to enable:

```
opengl('OpenGLClippedImageBug',1)
```

### **OpenGLERaseModeBug**

Symptom: Graphics objects with `EraseMode` property set to non-normal erase modes (`xor`, `none`, or `background`) do not draw when the figure `Renderer` property is set to `opengl`.

Command to enable:

```
opengl('OpenGLERaseModeBug',1)
```

### **See Also**

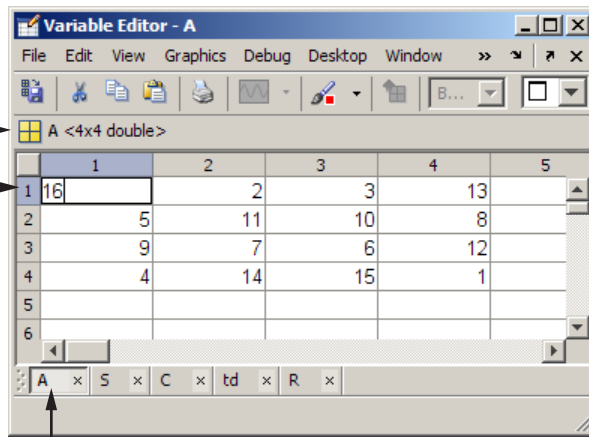
Figure `Renderer` property for information on autoselection.

# openvar

- Purpose** Open workspace variable in Variable Editor or other tool for graphical editing
- GUI Alternatives** As an alternative to the `openvar` function, double-click a variable in the Workspace browser.
- Syntax** `openvar('name')`
- Description** `openvar('name')` opens the workspace variable name in the Variable Editor for graphical editing, where `name` is a one or two-dimensional array, cell array, structure, or an object and its properties. You can also view the contents of a multidimensional array.
- The MATLAB software does not impose any limitation on the size of a variable that can be opened in the Variable Editor. Size is limited only by the operating system or the amount of physical memory installed on your system.
- For some toolboxes, `openvar` instead opens a tool appropriate for viewing or editing that type of object.

Size and class of the variable.

Click an element and then edit its value.



Select a tab to view a variable that you have open in the Variable Editor.



**See Also**

load, save, workspace

“Viewing and Editing Workspace Variables with the Variable Editor”  
in the MATLAB Desktop Tools and Development Environment  
Documentation.

# optimget

---

**Purpose** Optimization options values

**Syntax**

```
val = optimget(options,'param')  
val = optimget(options,'param',default)
```

**Description**

`val = optimget(options,'param')` returns the value of the specified parameter in the optimization options structure `options`. You need to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`val = optimget(options,'param',default)` returns `default` if the specified parameter is not defined in the optimization options structure `options`. Note that this form of the function is used primarily by other optimization functions.

**Examples**

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options`.

```
val = optimget(my_options,'Display')
```

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options` (as in the previous example) except that if the `Display` parameter is not defined, it returns the value `'final'`.

```
optnew = optimget(my_options,'Display','final');
```

**See Also** `optimset`, `fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`

**Purpose**

Create or edit optimization options structure

**Syntax**

```
options = optimset('param1',value1,'param2',value2,...)
optimset
options = optimset
options = optimset(optimfun)
options = optimset(olddopts,'param1',value1,...)
options = optimset(olddopts,newopts)
```

**Description**

The function `optimset` creates an `options` structure that you can pass as an input argument to the following four MATLAB optimization functions:

- `fminbnd`
- `fminsearch`
- `fzero`
- `lsqnonneg`

You can use the `options` structure to change the default parameters for these functions.

---

**Note** If you have purchased the Optimization Toolbox, you can also use `optimset` to create an expanded `options` structure containing additional options specifically designed for the functions provided in that toolbox. See the reference page for the enhanced `optimset` function in the Optimization Toolbox for more information about these additional options.

---

```
options = optimset('param1',value1,'param2',value2,...)
creates an optimization options structure called options, in which the
specified parameters (param) have specified values. Any unspecified
parameters are set to [] (parameters with value [] indicate to use
the default value for that parameter when options is passed to the
```

# optimset

---

optimization function). It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`optimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = optimset` (with no input arguments) creates an options structure `options` where all fields are set to `[]`.

`options = optimset(optimfun)` creates an options structure `options` with all parameter names and default values relevant to the optimization function `optimfun`.

`options = optimset(oldopts, 'param1', value1, ...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = optimset(oldopts, newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

## Options

The following table lists the available options for the MATLAB optimization functions.

Option	Value	Description
Display	'off'   'iter'   {'final'}   'notify'	Level of display. 'off' displays no output; 'iter' displays output at each iteration; 'final' displays just the final output; 'notify' displays output only if the function does not converge.

Option	Value	Description
FunValCheck	{'off'}   'on'	Check whether objective function values are valid. 'on' displays an error when the objective function returns a value that is complex or NaN. 'off' displays no error.
MaxFunEvals	positive integer	Maximum number of function evaluations allowed.
MaxIter	positive integer	Maximum number of iterations allowed.
OutputFcn	function   {}	User-defined function that an optimization function calls at each iteration.
PlotFcns	function   {}	User-defined plot function that an optimization function calls at each iteration.
TolFun	positive scalar	Termination tolerance on the function value.
TolX	positive scalar	Termination tolerance on $x$ .

### Examples

This statement creates an optimization options structure called `options` in which the `Display` parameter is set to 'iter' and the `TolFun` parameter is set to `1e-8`.

```
options = optimset('Display','iter','TolFun',1e-8)
```

# optimset

---

This statement makes a copy of the options structure called `options`, changing the value of the `TolX` parameter and storing new values in `optnew`.

```
optnew = optimset(options, 'TolX', 1e-4);
```

This statement returns an optimization options structure that contains all the parameter names and default values relevant to the function `fminbnd`.

```
optimset('fminbnd')
```

## See Also

`optimset` (Optimization Toolbox version), `optimget`, `fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`

**Purpose** Find logical OR of array or scalar inputs

**Syntax** `A | B | ...`  
`or(A, B)`

**Description** `A | B | ...` performs a logical OR 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 any 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.

`or(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 a MATLAB application. The element-wise OR operator described here is `|`. The short-circuit OR operator is `||`.

---

**Example** If matrix A is

0.4235	0.5798	0	0.7942	0
0.5155	0	0	0	0.8744
0	0	0	0.4451	0.0150
0.4329	0.6405	0.6808	0	0

and matrix B is

# or

---

0	1	0	1	0
1	1	0	0	1
0	0	0	1	0
0	1	0	0	1

then

A   B				
ans =				
1	1	0	1	0
1	1	0	0	1
0	0	0	1	1
1	1	1	0	1

## See Also

bitor, and, xor, not, any, all, logical operators, logical types, bitwise functions



**Purpose** Eigenvalues of quasitriangular matrices

**Syntax** E = ordeig(T)  
E = ordeig(AA,BB)

**Description** E = ordeig(T) takes a quasitriangular Schur matrix T, typically produced by schur, and returns the vector E of eigenvalues in their order of appearance down the diagonal of T.

E = ordeig(AA,BB) takes a quasitriangular matrix pair AA and BB, typically produced by qz, and returns the generalized eigenvalues in their order of appearance down the diagonal of  $AA - \lambda * BB$ .

ordeig is an order-preserving version of eig for use with ordschur and ordqz. It is also faster than eig for quasitriangular matrices.

## Examples

### Example 1

```
T=diag([1 -1 3 -5 2]);
```

ordeig(T) returns the eigenvalues of T in the same order they appear on the diagonal.

```
ordeig(T)
```

```
ans =
```

```
1  
-1  
3  
-5  
2
```

eig(T), on the other hand, returns the eigenvalues in order of increasing magnitude.

```
eig(T)
```

```
ans =
```

```
-5  
-1  
1  
2  
3
```

## Example 2

```
A = rand(10);  
[U, T] = schur(A);  
abs(ordeig(T))
```

```
ans =
```

```
5.3786  
0.7564  
0.7564  
0.7802  
0.7080  
0.7080  
0.5855  
0.5855  
0.1445  
0.0812
```

```
% Move eigenvalues with magnitude < 0.5 to the  
% upper-left corner of T.
```

```
[U,T] = ordschur(U,T,abs(E)<0.5);  
abs(ordeig(T))
```

```
ans =
```

```
0.1445  
0.0812  
5.3786  
0.7564  
0.7564  
0.7802
```

0.7080

0.7080

0.5855

0.5855

**See Also**

schur, qz, ordschur, ordqz, eig

# orderfields

---

**Purpose** Order fields of structure array

**Syntax**

```
s = orderfields(s1)
s = orderfields(s1, s2)
s = orderfields(s1, c)
s = orderfields(s1, perm)
[s, perm] = orderfields(...)
```

**Description** `s = orderfields(s1)` orders the fields in `s1` so that the new structure array `s` has field names in ASCII dictionary order.

`s = orderfields(s1, s2)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in `s2`. Structures `s1` and `s2` must have the same fields.

`s = orderfields(s1, c)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in the cell array of field name strings `c`. Structure `s1` and cell array `c` must contain the same field names.

`s = orderfields(s1, perm)` orders the fields in `s1` so that the new structure array `s` has fieldnames in the order specified by the indices in permutation vector `perm`.

If `s1` has `N` fieldnames, the elements of `perm` must be an arrangement of the numbers from 1 to `N`. This is particularly useful if you have more than one structure array that you would like to reorder in the same way.

`[s, perm] = orderfields(...)` returns a permutation vector representing the change in order performed on the fields of the structure array that results in `s`.

**Remarks** `orderfields` only orders top-level fields. It is not recursive.

**Examples** Create a structure `s`. Then create a new structure from `s`, but with the fields ordered alphabetically:

```
s = struct('b', 2, 'c', 3, 'a', 1)
s =
```

```
b: 2  
c: 3  
a: 1
```

```
snew = orderfields(s)  
snew =  
a: 1  
b: 2  
c: 3
```

Arrange the fields of `s` in the order specified by the second (cell array) argument of `orderfields`. Return the new structure in `snew` and the permutation vector used to create it in `perm`:

```
[snew, perm] = orderfields(s, {'b', 'a', 'c'})  
snew =  
b: 2  
a: 1  
c: 3  
perm =  
1  
3  
2
```

Now create a new structure, `s2`, having the same fieldnames as `s`. Reorder the fields using the permutation vector returned in the previous operation:

```
s2 = struct('b', 3, 'c', 7, 'a', 4)  
s2 =  
b: 3  
c: 7  
a: 4  
  
snew = orderfields(s2, perm)  
snew =  
b: 3  
a: 4
```

# orderfields

---

c: 7

## **See Also**

struct, fieldnames, setfield, getfield, isfield, rmfield, “Using Dynamic Field Names”

**Purpose** Reorder eigenvalues in QZ factorization

**Syntax**  
`[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)`  
`[...] = ordqz(AA,BB,Q,Z,keyword)`  
`[...] = ordqz(AA,BB,Q,Z,clusters)`

**Description** `[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)` reorders the QZ factorizations  $Q^*A^*Z = AA$  and  $Q^*B^*Z = BB$  produced by the `qz` function for a matrix pair  $(A,B)$ . It returns the reordered pair  $(AAS,BBS)$  and the cumulative orthogonal transformations `QS` and `ZS` such that  $QS^*A^*ZS = AAS$  and  $QS^*B^*ZS = BBS$ . In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular pair  $(AAS,BBS)$ , and the corresponding invariant subspace is spanned by the leading columns of `ZS`. The logical vector `select` specifies the selected cluster as  $E(\text{select})$  where  $E$  is the vector of eigenvalues as they appear along the diagonal of  $AA - \lambda^*BB$ .

---

**Note** To extract  $E$  from `AA` and `BB`, use `ordeig(BB)`, instead of `eig`. This ensures that the eigenvalues in  $E$  occur in the same order as they appear on the diagonal of  $AA - \lambda^*BB$ .

---

`[...] = ordqz(AA,BB,Q,Z,keyword)` sets the selected cluster to include all eigenvalues in the region specified by `keyword`:

keyword	Selected Region
'lhp'	Left-half plane ( $\text{real}(E) < 0$ )
'rhp'	Right-half plane ( $\text{real}(E) > 0$ )
'udi'	Interior of unit disk ( $\text{abs}(E) < 1$ )
'udo'	Exterior of unit disk ( $\text{abs}(E) > 1$ )

`[...] = ordqz(AA,BB,Q,Z,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices commensurate with  $E = \text{ordeig}(AA,BB)$ , such that all eigenvalues with the same `clusters`

# ordqz

---

value form one cluster, `ordqz` sorts the specified clusters in descending order along the diagonal of  $(AAS, BBS)$ . The cluster with highest index appears in the upper left corner.

## Algorithm

For full matrices `AA` and `BB`, `qz` uses the LAPACK routines listed in the following table.

	<b>AA and BB Real</b>	<b>AA or BB Complex</b>
A and B double	DTGSEN	ZTGSEN
A or B single	STGSEN	CTGSEN

## See Also

`ordeig`, `ordschur`, `qz`



**Purpose** Reorder eigenvalues in Schur factorization

**Syntax**

```
[US,TS] = ordschur(U,T,select)
[US,TS] = ordschur(U,T,keyword)
[US,TS] = ordschur(U,T,clusters)
```

**Description** `[US,TS] = ordschur(U,T,select)` reorders the Schur factorization  $X = U^*T^*U'$  produced by the `schur` function and returns the reordered Schur matrix `TS` and the cumulative orthogonal transformation `US` such that  $X = US^*TS^*US'$ . In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular Schur matrix `TS`, and the corresponding invariant subspace is spanned by the leading columns of `US`. The logical vector `select` specifies the selected cluster as `E(select)` where `E` is the vector of eigenvalues as they appear along `T`'s diagonal.

---

**Note** To extract `E` from `T`, use `E = ordeig(T)`, instead of `eig`. This ensures that the eigenvalues in `E` occur in the same order as they appear on the diagonal of `TS`.

---

`[US,TS] = ordschur(U,T,keyword)` sets the selected cluster to include all eigenvalues in one of the following regions:

keyword	Selected Region
'lhp'	Left-half plane ( $\text{real}(E) < 0$ )
'rhp'	Right-half plane ( $\text{real}(E) > 0$ )
'udi'	Interior of unit disk ( $\text{abs}(E) < 1$ )
'udo'	Exterior of unit disk ( $\text{abs}(E) > 1$ )

`[US,TS] = ordschur(U,T,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices, commensurate with `E = ordeig(T)`, and such that all eigenvalues with the same `clusters` value form one cluster, `ordschur` sorts the specified clusters

in descending order along the diagonal of  $TS$ , the cluster with highest index appearing in the upper left corner.

## Algorithm **Input of Type Double**

If  $U$  and  $T$  have type `double`, `ordschur` uses the LAPACK routines listed in the following table to compute the Schur form of a matrix:

Matrix Type	Routine
Real	DTRSEN
Complex	ZTRSEN

## Input of Type Single

If  $U$  and  $T$  have type `single`, `ordschur` uses the LAPACK routines listed in the following table to reorder the Schur form of a matrix:

Matrix Type	Routine
Real	STRSEN
Complex	CTRSEN

## See Also

`ordeig`, `ordqz`, `schur`

---

<b>Purpose</b>	Hardcopy paper orientation
<b>GUI Alternative</b>	Use <b>File</b> → <b>Print Preview</b> on the figure window menu to directly manipulate print layout, paper size, headers, fonts and other properties when printing figures. For details, see Using Print Preview in the MATLAB Graphics documentation.
<b>Syntax</b>	<pre>orient orient landscape orient portrait orient tall orient(fig_handle), orient(simulink_model) orient(fig_handle,orientation), orient(simulink_model,     orientation)</pre>
<b>Description</b>	<p><code>orient</code> returns a string with the current paper orientation: <code>portrait</code>, <code>landscape</code>, or <code>tall</code>.</p> <p><code>orient landscape</code> sets the paper orientation of the current figure to full-page landscape, orienting the longest page dimension horizontally. The figure is centered on the page and scaled to fit the page with a 0.25 inch border.</p> <p><code>orient portrait</code> sets the paper orientation of the current figure to portrait, orienting the longest page dimension vertically. The <code>portrait</code> option returns the page orientation to the MATLAB default. (Note that the result of using the <code>portrait</code> option is affected by changes you make to figure properties. See the "Algorithm" section for more specific information.)</p> <p><code>orient tall</code> maps the current figure to the entire page in portrait orientation, leaving a 0.25 inch border.</p> <p><code>orient(fig_handle)</code>, <code>orient(simulink_model)</code> returns the current orientation of the specified figure or Simulink model.</p> <p><code>orient(fig_handle,orientation)</code>, <code>orient(simulink_model,orientation)</code> sets the orientation for</p>

the specified figure or Simulink model to the specified orientation (landscape, portrait, or tall).

## Algorithm

`orient` sets the `PaperOrientation`, `PaperPosition`, and `PaperUnits` properties of the current figure. Subsequent print operations use these properties. The result of using the `portrait` option can be affected by default property values as follows:

- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the current values of `PaperOrientation` and `PaperPosition` to place the figure on the page.
- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the default figure `PaperPosition` with the `x`, `y` and `width`, `height` values reversed (i.e., `[y,x,height,width]`) to position the figure on the page.
- If the current figure `PaperType` is different from the default figure `PaperType`, then the `orient portrait` command uses the current figure `PaperPosition` with the `x`, `y` and `width`, `height` values reversed (i.e., `[y,x,height,width]`) to position the figure on the page.

## See Also

`print`, `printpreview`, `set`

`PaperOrientation`, `PaperPosition`, `PaperSize`, `PaperType`, and `PaperUnits` properties of figure graphics objects

“Printing” on page 1-96 for related functions

---

<b>Purpose</b>	Range space of matrix
<b>Syntax</b>	$B = \text{orth}(A)$
<b>Description</b>	$B = \text{orth}(A)$ returns an orthonormal basis for the range of $A$ . The columns of $B$ span the same space as the columns of $A$ , and the columns of $B$ are orthogonal, so that $B' * B = \text{eye}(\text{rank}(A))$ . The number of columns of $B$ is the rank of $A$ .
<b>See Also</b>	<code>null</code> , <code>svd</code> , <code>rank</code>

# otherwise

---

**Purpose** Default part of switch statement

**Syntax**

```
switch switch_expr
  case case_expr
    statement, ..., statement
  case {case_expr1, case_expr2, case_expr3, ...}
    statement, ..., statement
  otherwise
    statement, ..., statement
end
```

**Description** `otherwise` is part of the `switch` statement syntax, which allows for conditional execution. The statements following `otherwise` are executed only if none of the preceding case expressions (`case_expr`) matches the switch expression (`sw_expr`).

**Examples** The general form of the `switch` statement is

```
switch sw_expr
  case case_expr
    statement
    statement
  case {case_expr1,case_expr2,case_expr3}
    statement
    statement
  otherwise
    statement
    statement
end
```

See `switch` for more details.

**See Also** `switch`, `case`, `end`, `if`, `else`, `elseif`, `while`

## Symbols and Numerics

' 2-39  
 & 2-51 2-58  
 \* 2-39  
 + 2-39  
 - 2-39  
 / 2-39  
 : 2-65  
 < 2-49  
 > 2-49  
 @ 2-1389  
 \ 2-39  
 ^ 2-39  
 | 2-51 2-58  
 ~ 2-51 2-58  
 && 2-58  
 == 2-49  
 ]) 2-64  
 || 2-58  
 ~= 2-49  
 1-norm 2-2480 2-2918  
 2-norm (estimate of) 2-2482

## A

abs 2-68  
 absolute accuracy  
   BVP 2-465  
   DDE 2-865  
   ODE 2-2529  
 absolute value 2-68  
 Accelerator  
   Uimenu property 2-3773  
 accumularray 2-69  
 accuracy  
   of linear equation solution 2-660  
   of matrix inversion 2-660  
 acos 2-76  
 acosd 2-78  
 acosh 2-79

acot 2-81  
 acotd 2-83  
 acoth 2-84  
 acsc 2-86  
 acscd 2-88  
 acsch 2-89  
 activelegend 1-91 2-2714  
 actxcontrol 2-91  
 actxcontrollist 2-99  
 actxcontrolselect 2-100  
 actxserver 2-105  
 Adams-Bashforth-Moulton ODE solver 2-2518  
 addCause, MException method 2-109  
 addevent 2-113  
 addframe  
   AVI files 2-115  
 addition (arithmetic operator) 2-39  
 addlistener 2-117  
 addOptional method  
   of inputParser object 2-119  
 addParamValue method  
   of inputParser object 2-122  
 addpath 2-125  
 addpref function 2-127  
 addprop dynamicprops method 2-128  
 addproperty 2-129  
 addRequired method  
   of inputParser object 2-131  
 addressing selected array elements 2-65  
 addsample 2-134  
 addsampletocollection 2-136  
 addtodate 2-138  
 addts 2-140  
 adjacency graph 2-973  
 airy 2-142  
 Airy functions  
   relationship to modified Bessel  
     functions 2-142  
 align function 2-144

- aligning scattered data
  - multi-dimensional 2-2416
  - two-dimensional 2-1530
- ALim, Axes property 2-291
- all 2-152
- allchild function 2-154
- allocation of storage (automatic) 2-4098
- AlphaData
  - image property 2-1712
  - surface property 2-3456
  - surfaceplot property 2-3479
- AlphaDataMapping
  - image property 2-1713
  - patch property 2-2616
  - surface property 2-3457
  - surfaceplot property 2-3479
- AmbientLightColor, Axes property 2-292
- AmbientStrength
  - Patch property 2-2617
  - Surface property 2-3457
  - surfaceplot property 2-3480
- amd 2-160 2-1985
- analytical partial derivatives (BVP) 2-466
- analyzer
  - code 2-2339
- and 2-165
- and (M-file function equivalent for &) 2-55
- AND, logical
  - bit-wise 2-412
- angle 2-167
- annotating graphs
  - deleting annotations 2-170
  - in plot edit mode 2-2715
- Annotation
  - areaseries property 2-221
  - contourgroup property 2-686
  - errorbarseries property 2-1045
  - hggroup property 2-1617 2-1646
  - image property 2-1713
  - line property 2-351 2-2045
  - lineseries property 2-2060
  - Patch property 2-2617
  - quivergroup property 2-2866
  - rectangle property 2-2937
  - scattergroup property 2-3093
  - stairsseries property 2-3276
  - stemsseries property 2-3310
  - Surface property 2-3458
  - surfaceplot property 2-3480
  - text property 2-3563
- annotationfunction 2-168
- ans 2-211
- anti-diagonal 2-1562
- any 2-212
- arccosecant 2-86
- arccosine 2-76
- arccotangent 2-81
- arcsecant 2-244
- arcsine 2-249
- arctangent 2-258
  - four-quadrant 2-260
- arguments, M-file
  - checking number of inputs 2-2407
  - checking number of outputs 2-2411
  - number of input 2-2409
  - number of output 2-2409
  - passing variable numbers of 2-3961
- arithmetic operations, matrix and array
  - distinguished 2-39
- arithmetic operators
  - reference 2-39



- array
  - addressing selected elements of 2-65
  - dimension
    - rearrange 2-1302
  - displaying 2-952
  - flip dimension of 2-1302
  - left division (arithmetic operator) 2-41
  - maximum elements of 2-2225
  - mean elements of 2-2231
  - median elements of 2-2234
  - minimum elements of 2-2311
  - multiplication (arithmetic operator) 2-40
  - of all ones 2-2550
  - of all zeros 2-4098
  - power (arithmetic operator) 2-41
  - product of elements 2-2787
  - rearrange
    - dimension 2-1302
  - removing first n singleton dimensions
    - of 2-3165
  - removing singleton dimensions of 2-3263
  - reshaping 2-3010
  - reverse dimension of 2-1302
  - right division (arithmetic operator) 2-40
  - shift circularly 2-577
  - shifting dimensions of 2-3165
  - size of 2-3179
  - sorting elements of 2-3200
  - structure 2-1480 2-3033 2-3151
  - sum of elements 2-3436
  - swapping dimensions of 2-1857 2-2689
  - transpose (arithmetic operator) 2-41
- arrayfun 2-237
- arrays
  - detecting empty 2-1871
  - maximum size of 2-658
  - opening 2-2551
- arrays, structure
  - field names of 2-1178
- arrowhead matrix 2-645
- ASCII
  - delimited files
    - writing 2-968
- ASCII data
  - converting sparse matrix after loading
    - from 2-3213
  - reading 2-964
  - reading from disk 2-2111
  - saving to disk 2-3069
- ascii function 2-243
- asec 2-244
- asecd 2-246
- asech 2-247
- asin 2-249
- asind 2-251
- asinh 2-252
- aspect ratio of axes 2-786 2-2651
- assert 2-254
- assignin 2-256
- atan 2-258
- atan2 2-260
- atand 2-262
- atanh 2-263
- .au files
  - reading 2-276
  - writing 2-277
- audio
  - saving in AVI format 2-278
  - signal conversion 2-2038 2-2390
- audioplayer 1-86 2-265
- audiorecorder 1-86 2-270
- aufinfo 2-275
- auread 2-276
- AutoScale
  - quivergroup property 2-2867
- AutoScaleFactor
  - quivergroup property 2-2867
- autoselection of OpenGL 2-1216
- auwrite 2-277
- average of array elements 2-2231

- average,running 2-1268
- avi 2-278
- avifile 2-278
- aviinfo 2-282
- aviread 2-284
- axes 2-285
  - editing 2-2715
  - setting and querying data aspect ratio 2-786
  - setting and querying limits 2-4070
  - setting and querying plot box aspect ratio 2-2651
- Axes
  - creating 2-285
  - defining default properties 2-289
  - fixed-width font 2-308
  - property descriptions 2-291
- axis 2-329
- axis crossing. *See* zero of a function
- azimuth (spherical coordinates) 2-3229
- azimuth of viewpoint 2-3978
  
- B**
- BackFaceLighting
  - Surface property 2-3459
  - surfaceplot property 2-3482
- BackFaceLightingpatch property 2-2619
- BackgroundColor
  - annotation textbox property 2-201
  - Text property 2-3564
  - Uitable property 2-3844
- BackgroundColor
  - Uicontrol property 2-3726
- badly conditioned 2-2918
- balance 2-335
- BarLayout
  - barseries property 2-352
- BarWidth
  - barseries property 2-352
- base to decimal conversion 2-368
- base two operations
  - conversion from decimal to binary 2-881
  - logarithm 2-2131
  - next power of two 2-2476
- base2dec 2-368
- BaseLine
  - barseries property 2-352
  - stem property 2-3311
- BaseValue
  - areaseries property 2-222
  - barseries property 2-353
  - stem property 2-3311
- beep 2-369
- BeingDeleted
  - areaseries property 2-222
  - barseries property 2-353
  - contour property 2-687
  - errorbar property 2-1046
  - group property 2-1183 2-1714 2-3565
  - hggroup property 2-1618
  - hgtransform property 2-1647
  - light property 2-2028
  - line property 2-2046
  - lineseries property 2-2061
  - quivergroup property 2-2867
  - rectangle property 2-2938
  - scatter property 2-3094
  - stairs series property 2-3277
  - stem property 2-3311
  - surface property 2-3459
  - surfaceplot property 2-3482
  - transform property 2-2619
  - Uipushtool property 2-3808
  - Uitable property 2-3845
  - Uitoggletool property 2-3874
  - Uitoolbar property 2-3887
- bench 2-370
- benchmark 2-370

- Bessel functions
  - first kind 2-379
  - modified, first kind 2-376
  - modified, second kind 2-382
  - second kind 2-385
- Bessel functions, modified
  - relationship to Airy functions 2-142
- besseli 2-376
- besselj 2-379
- besselk 2-382
- Bessel's equation
  - (defined) 2-379
  - modified (defined) 2-376
- bessely 2-385
- beta 2-389
- beta function
  - (defined) 2-389
  - incomplete (defined) 2-391
  - natural logarithm 2-393
- betainc 2-391
- betaln 2-393
- bicg 2-394
- bicgstab 2-403
- BiConjugate Gradients method 2-394
- BiConjugate Gradients Stabilized method 2-403
- big endian formats 2-1320
- bin2dec 2-409
- binary
  - data
    - writing to file 2-1401
  - files
    - reading 2-1352
    - mode for opened files 2-1319
- binary data
  - reading from disk 2-2111
  - saving to disk 2-3069
- binary function 2-410
- binary to decimal conversion 2-409
- bisection search 2-1411
- bit depth
  - querying 2-1732
- bit-wise operations
  - AND 2-412
  - get 2-415
  - OR 2-418
  - set bit 2-419
  - shift 2-420
  - XOR 2-422
- bitand 2-412
- bitcmp 2-413
- bitget 2-415
- bitmaps
  - writing 2-1757
- bitmax 2-416
- bitor 2-418
- bitset 2-419
- bitshift 2-420
- bitxor 2-422
- blanks 2-423
  - removing trailing 2-877
- blkdiag 2-424
- BMP files
  - writing 2-1757
- bold font
  - TeX characters 2-3587
- boundary value problems 2-472
- box 2-425
- Box, Axes property 2-293
- braces, curly (special characters) 2-61
- brackets (special characters) 2-61
- break 2-426
- breakpoints
  - listing 2-835
  - removing 2-823
  - resuming execution from 2-826
  - setting in M-files 2-839
- brighten 2-427
- browser
  - for help 2-1601

- brush 2-429
- bsxfun 2-439
- bubble plot (scatter function) 2-3088
- Buckminster Fuller 2-3533
- builtin 1-69 2-442
- BusyAction
  - areaserie property 2-222
  - Axes property 2-293
  - barseries property 2-353
  - contour property 2-687
  - errorbar property 2-1047
  - Figure property 2-1184
  - hggroup property 2-1619
  - hgtransform property 2-1648
  - Image property 2-1715
  - Light property 2-2028
  - line property 2-2047
  - Line property 2-2061
  - patch property 2-2619
  - quivergroup property 2-2868
  - rectangle property 2-2939
  - Root property 2-3037
  - scatter property 2-3095
  - stairs series property 2-3278
  - stem property 2-3312
  - Surface property 2-3459
  - surfaceplot property 2-3482
  - Text property 2-3566
  - Uicontextmenu property 2-3711
  - Uicontrol property 2-3727
  - Uimenu property 2-3774
  - Uipushtool property 2-3809
  - Uitable property 2-3845
  - Uitoggletool property 2-3875
  - Uitoolbar property 2-3887
- ButtonDownFcn
  - area series property 2-223
  - Axes property 2-294
  - barseries property 2-354
  - contour property 2-688
  - errorbar property 2-1047
  - Figure property 2-1184
  - hggroup property 2-1619
  - hgtransform property 2-1648
  - Image property 2-1715
  - Light property 2-2029
  - Line property 2-2047
  - lineseries property 2-2062
  - patch property 2-2620
  - quivergroup property 2-2868
  - rectangle property 2-2939
  - Root property 2-3037
  - scatter property 2-3095
  - stairs series property 2-3278
  - stem property 2-3312
  - Surface property 2-3460
  - surfaceplot property 2-3483
  - Text property 2-3566
  - Uicontrol property 2-3727
  - Uitable property 2-3846
- BVP solver properties
  - analytical partial derivatives 2-466
  - error tolerance 2-464
  - Jacobian matrix 2-466
  - mesh 2-469
  - singular BVPs 2-469
  - solution statistics 2-470
  - vectorization 2-465
- bvp4c 2-443
- bvp5c 2-454
- bvpget 2-459
- bvpinit 2-460
- bvpset 2-463
- bvpxtend 2-472

**C**

- calendar 2-473
- call history 2-2794
- CallBack
  - Uicontextmenu property 2-3712
  - Uicontrol property 2-3728
  - Uimenu property 2-3775
- CallbackObject, Root property 2-3037
- calllib 2-474
- callSoapService 2-476
- camdolly 2-477
- camera
  - dollying position 2-477
  - moving camera and target postions 2-477
  - placing a light at 2-481
  - positioning to view objects 2-483
  - rotating around camera target 1-103 2-485 2-487
  - rotating around viewing axis 2-491
  - setting and querying position 2-488
  - setting and querying projection type 2-490
  - setting and querying target 2-492
  - setting and querying up vector 2-494
  - setting and querying view angle 2-496
- CameraPosition, Axes property 2-295
- CameraPositionMode, Axes property 2-295
- CameraTarget, Axes property 2-296
- CameraTargetMode, Axes property 2-296
- CameraUpVector, Axes property 2-296
- CameraUpVectorMode, Axes property 2-296
- CameraViewAngle, Axes property 2-297
- CameraViewAngleMode, Axes property 2-297
- camlight 2-481
- camlookat 2-483
- camorbit 2-485
- campan 2-487
- campos 2-488
- camproj 2-490
- camroll 2-491
- camtarget 2-492
- camup 2-494
- camva 2-496
- camzoom 2-498
- CaptureMatrix, Root property 2-3037
- CaptureRect, Root property 2-3038
- cart2pol 2-499
- cart2sph 2-501
- Cartesian coordinates 2-499 2-501 2-2725 2-3229
- case 2-502
  - in switch statement (defined) 2-3520
  - lower to upper 2-3924
  - upper to lower 2-2143
- cast 2-504
- cat 2-505
- catch 2-507
- caxis 2-510
- Cayley-Hamilton theorem 2-2745
- cd 2-515
- cd (ftp) function 2-518
- CData
  - Image property 2-1716
  - scatter property 2-3096
  - Surface property 2-3461
  - surfaceplot property 2-3484
  - Uicontrol property 2-3729
  - Uipushtool property 2-3809
  - Uitoggletool property 2-3875
- CDataMapping
  - Image property 2-1718
  - patch property 2-2622
  - Surface property 2-3462
  - surfaceplot property 2-3484
- CDataMode
  - surfaceplot property 2-3485
- CDatapatch property 2-2620
- CDataSource
  - scatter property 2-3096
  - surfaceplot property 2-3485
- cdf2rdf 2-519
- cdfepoch 2-521

- cdfinfo 2-522
- cdfread 2-526
- cdfwrite 2-530
- ceil 2-533
- cell 2-534
- cell array
  - conversion to from numeric array 2-2490
  - creating 2-534
  - structure of, displaying 2-547
- cell2mat 2-536
- cell2struct 2-538
- celldisp 2-540
- CellEditCallback
  - Uitable property 2-3847
- cellfun 2-541
- cellplot 2-547
- CellSelectionCallback
  - Uitable property 2-3849
- cgs 2-550
- char 1-51 1-58 1-62 2-555
- characters
  - conversion, in format specification
    - string 2-1341 2-3252
  - escape, in format specification string 2-1342
    - 2-3252
- check boxes 2-3719
- Checked, Uimenu property 2-3775
- checkerboard pattern (example) 2-2999
- checkin 2-556
  - examples 2-557
  - options 2-556
- checkout 2-559
  - examples 2-560
  - options 2-559
- child functions 2-2789
- Children
  - areaseries property 2-224
  - Axes property 2-298
  - barseries property 2-355
  - contour property 2-688
  - errorbar property 2-1048
  - Figure property 2-1185
  - hggroup property 2-1620
  - hgtransform property 2-1649
  - Image property 2-1718
  - Light property 2-2029
  - Line property 2-2048
  - lineseries property 2-2062
  - patch property 2-2623
  - quivergroup property 2-2869
  - rectangle property 2-2940
  - Root property 2-3038
  - scatter property 2-3097
  - stairs series property 2-3279
  - stem property 2-3313
  - Surface property 2-3462
  - surfaceplot property 2-3486
  - Text property 2-3568
  - Uicontextmenu property 2-3712
  - Uicontrol property 2-3730
  - Uimenu property 2-3776
  - Uitable property 2-3849
  - Uitoolbar property 2-3888
- chol 2-562
- Cholesky factorization 2-562
  - (as algorithm for solving linear equations) 2-2335
  - lower triangular factor 2-2607
  - preordering for 2-645
- cholinc 2-566
- cholupdate 2-574
- circle
  - rectangle function 2-2932
- circshift 2-577
- cla 2-578

- clabel 2-579
- class 2-585
- class, object. *See* object classes
- classes
  - field names 2-1178
  - loaded 2-1783
- clc 2-590 2-599 2-3164
- clear 2-591
  - serial port I/O 2-598
- clearing
  - Command Window 2-590
  - items from workspace 2-591
  - Java import list 2-593
- clf 2-599
- ClickedCallback
  - Uipushtool property 2-3810
  - Uitoggletool property 2-3876
- CLim, Axes property 2-299
- CLimMode, Axes property 2-299
- clipboard 2-600
- Clipping
  - areaseries property 2-224
  - Axes property 2-300
  - barseries property 2-355
  - contour property 2-689
  - errobar property 2-1048
  - Figure property 2-1186
  - hggroup property 2-1620
  - hgtransform property 2-1649
  - Image property 2-1719
  - Light property 2-2029
  - Line property 2-2048
  - lineseries property 2-2063
  - quivergroup property 2-2869
  - rectangle property 2-2940
  - Root property 2-3038
  - scatter property 2-3097
  - stairseries property 2-3279
  - stem property 2-3313
  - Surface property 2-3463
  - surfaceplot property 2-3486
  - Text property 2-3568
  - Uicontrol property 2-3730
  - Uitable property 2-3849
- Clippingpatch property 2-2623
- clock 2-601
- close 2-602
  - AVI files 2-604
- close (ftp) function 2-605
- CloseRequestFcn, Figure property 2-1186
- closest point search 2-990
- closest triangle search 2-3672
- closing
  - files 2-1140
  - MATLAB 2-2856
- cmapeditor 2-625
- cmopts 2-607
- code
  - analyzer 2-2339
- colamd 2-609

- colon operator 2-65
- Color
  - annotation arrow property 2-172
  - annotation doublearrow property 2-176
  - annotation line property 2-184
  - annotation textbox property 2-201
  - Axes property 2-300
  - errorbar property 2-1048
  - Figure property 2-1188
  - Light property 2-2029
  - Line property 2-2049
  - lineseries property 2-2063
  - quivergroup property 2-2870
  - stairs series property 2-3279
  - stem property 2-3314
  - Text property 2-3568
  - textarrow property 2-190
- color of fonts, see also `FontColor` property 2-3587
- colorbar 2-613
- colormap 2-620
  - editor 2-625
- Colormap, Figure property 2-1189
- colormaps
  - converting from RGB to HSV 1-102 2-3022
  - plotting RGB components 1-102 2-3023
- ColorOrder, Axes property 2-300
- ColorSpec 2-643
- colperm 2-645
- ColumnEditable
  - Uitable property 2-3849
- ColumnFormat
  - Uitable property 2-3850
- ColumnName
  - Uitable property 2-3856
- ColumnWidth
  - Uitable property 2-3856
- COM
  - object methods
    - `actxcontrol` 2-91
    - `actxcontrollist` 2-99
    - `actxcontrolselect` 2-100
    - `actxserver` 2-105
    - `addproperty` 2-129
    - `delete` 2-907
    - `deleteproperty` 2-914
    - `eventlisteners` 2-1081
    - `events` 2-1085
    - `get` 1-116 2-1456
    - `inspect` 2-1799
    - `invoke` 2-1854
    - `iscom` 2-1869
    - `isevent` 2-1880
    - `isinterface` 2-1892
    - `ismethod` 2-1903
    - `load` 2-2116
    - `move` 2-2368
    - `propedit` 2-2798
    - `registerevent` 2-2984
    - `release` 2-2989
    - `save` 2-3077
    - `set` 1-117 2-3133
    - `unregisterallev events` 2-3908
    - `unregisterevent` 2-3911
- object properties
  - `isprop` 2-1926
- server methods
  - `Execute` 2-1087
  - `Feval` 2-1149
- combinations of n elements 2-2415
- combs 2-2415
- comet 2-647
- comet3 2-649
- comma (special characters) 2-63
- command syntax 2-1598 2-3538
- Command Window
  - clearing 2-590
  - cursor position 1-4 2-1669
  - get width 2-652
- commandhistory 2-651



- commands
  - help for 2-1597 2-1607
  - system 1-4 1-11 2-3541
  - UNIX 2-3903
- commandwindow 2-652
- comments
  - block of 2-63
- common elements. *See* set operations, intersection
- compan 2-653
- companion matrix 2-653
- compass 2-654
- CompilerConfiguration 2-2295
- CompilerConfigurationDetails 2-2295
- complementary error function
  - (defined) 2-1034
  - scaled (defined) 2-1034
- complete elliptic integral
  - (defined) 2-1017
  - modulus of 2-1015 2-1017
- complex 2-656 2-1703
  - exponential (defined) 2-1095
  - logarithm 2-2128 to 2-2129
  - numbers 2-1679
  - numbers, sorting 2-3200 2-3204
  - phase angle 2-167
  - sine 2-3173
  - unitary matrix 2-2827
  - See also* imaginary
- complex conjugate 2-670
  - sorting pairs of 2-747
- complex data
  - creating 2-656
- complex numbers, magnitude 2-68
- complex Schur form 2-3111
- compression
  - lossy 2-1761
- computer 2-658
- computer MATLAB is running on 2-658
- concatenation
  - of arrays 2-505
- cond 2-660
- condeig 2-661
- condest 2-662
- condition number of matrix 2-660 2-2918
  - improving 2-335
- coneplot 2-664
- conj 2-670
- conjugate, complex 2-670
  - sorting pairs of 2-747
- connecting to FTP server 2-1381
- containers
  - Map 2-1894 2-1969 2-2008 2-2180 2-2994 2-3182 2-3954
- contents.m file 2-1598
- context menu 2-3707
- continuation (*...*, special characters) 2-63
- continue 2-671
- continued fraction expansion 2-2912
- contour
  - and mesh plot 2-1115
  - filled plot 2-1107
  - functions 2-1103
  - of mathematical expression 2-1104
  - with surface plot 2-1133
- contour3 2-677
- contourc 2-680
- contourf 2-682
- ContourMatrix
  - contour property 2-689
- contours
  - in slice planes 2-707
- contourslice 2-707
- contrast 2-711
- conv 2-712
- conv2 2-714

- conversion
  - base to decimal 2-368
  - binary to decimal 2-409
  - Cartesian to cylindrical 2-499
  - Cartesian to polar 2-499
  - complex diagonal to real block diagonal 2-519
  - cylindrical to Cartesian 2-2725
  - decimal number to base 2-874 2-880
  - decimal to binary 2-881
  - decimal to hexadecimal 2-882
  - full to sparse 2-3210
  - hexadecimal to decimal 2-1611
  - integer to string 2-1813
  - lowercase to uppercase 2-3924
  - matrix to string 2-2190
  - numeric array to cell array 2-2490
  - numeric array to logical array 2-2132
  - numeric array to string 2-2493
  - partial fraction expansion to
    - pole-residue 2-3012
  - polar to Cartesian 2-2725
  - pole-residue to partial fraction expansion 2-3012
  - real to complex Schur form 2-3066
  - spherical to Cartesian 2-3229
  - string matrix to cell array 2-549
  - string to numeric array 2-3335
  - uppercase to lowercase 2-2143
  - vector to character string 2-555
- conversion characters in format specification
  - string 2-1341 2-3252
- convex hulls
  - multidimensional vizualization 2-723
  - two-dimensional visualization 2-720
- convhull 2-720
- convhulln 2-723
- convn 2-726
- convolution 2-712
  - inverse. *See* deconvolution
  - two-dimensional 2-714
- coordinate system and viewpoint 2-3978
- coordinates
  - Cartesian 2-499 2-501 2-2725 2-3229
  - cylindrical 2-499 2-501 2-2725
  - polar 2-499 2-501 2-2725
  - spherical 2-3229
- coordinates. 2-499
  - See also* conversion
- copyfile 2-727
- copyobj 2-730
- corrcoef 2-732
- cos 2-735
- cosd 2-737
- cosecant
  - hyperbolic 2-759
  - inverse 2-86
  - inverse hyperbolic 2-89
- cosh 2-738
- cosine 2-735
  - hyperbolic 2-738
  - inverse 2-76
  - inverse hyperbolic 2-79
- cot 2-740
- cotangent 2-740
  - hyperbolic 2-743
  - inverse 2-81
  - inverse hyperbolic 2-84
- cotd 2-742
- coth 2-743
- cov 2-745
- cplxpair 2-747
- cputime 2-748
- create, RandStream method 2-749
- createClassFromWsd1 2-751
- createCopy method
  - of inputParser object 2-752

**CreateFcn**

- areaserie property 2-224
- Axes property 2-301
- barseries property 2-355
- contour property 2-690
- errorbar property 2-1049
- Figure property 2-1189
- group property 2-1649
- hggroup property 2-1620
- Image property 2-1719
- Light property 2-2030
- Line property 2-2049
- lineseries property 2-2063
- patch property 2-2623
- quivergroup property 2-2870
- rectangle property 2-2941
- Root property 2-3038
- scatter property 2-3097
- stairs series property 2-3280
- stemseries property 2-3314
- Surface property 2-3463
- surfaceplot property 2-3486
- Text property 2-3568
- Uicontextmenu property 2-3712
- Uicontrol property 2-3730
- Uimenu property 2-3776
- Uipushtool property 2-3810
- Uitable property 2-3857
- Uitoggletool property 2-3876
- Uitoolbar property 2-3888

createSoapMessage 2-754

creating your own MATLAB functions 2-1387

cross 2-755

cross product 2-755

csc 2-756

cscd 2-758

csch 2-759

csvread 2-761

csvwrite 2-764

ctranspose (M-file function equivalent for \q) 2-45

ctranspose (timeseries) 2-766

cubic interpolation 2-1830 2-1833 2-1836 2-2661

- piecewise Hermite 2-1820

cubic spline interpolation

- one-dimensional 2-1820 2-1830 2-1833 2-1836

cumprod 2-768

cumsum 2-770

cumtrapz 2-772

cumulative

- product 2-768
- sum 2-770

curl 2-774

curly braces (special characters) 2-61

current directory 2-2820

- changing 2-515

CurrentAxes 2-1190

CurrentAxes, Figure property 2-1190

CurrentCharacter, Figure property 2-1191

CurrentFigure, Root property 2-3038

CurrentObject, Figure property 2-1191

CurrentPoint

- Axes property 2-301
- Figure property 2-1191

cursor images

- reading 2-1745

cursor position 1-4 2-1669

Curvature, rectangle property 2-2942

curve fitting (polynomial) 2-2737

customverctrl 2-777

Cuthill-McKee ordering, reverse 2-3523 2-3533

cylinder 2-778

cylindrical coordinates 2-499 2-501 2-2725

**D**

daqread 2-781

daspect 2-786

- data
  - ASCII
    - reading from disk 2-2111
  - ASCII, saving to disk 2-3069
  - binary
    - writing to file 2-1401
  - binary, saving to disk 2-3069
  - computing 2-D stream lines 1-106 2-3345
  - computing 3-D stream lines 1-106 2-3347
  - formatted
    - reading from files 2-1367
    - writing to file 2-1340
  - formatting 2-1340 2-3250
  - isosurface from volume data 2-1918
  - reading binary from disk 2-2111
  - reading from files 2-3593
  - reducing number of elements in 1-106 2-2958
  - smoothing 3-D 1-106 2-3193
  - writing to strings 2-3250
- Data
  - Uitable property 2-3858
- data aspect ratio of axes 2-786
- data brushing
  - different plot types 2-430
  - gestures for 2-435
  - restrictions on 2-432
- data types
  - complex 2-656
- data, aligning scattered
  - multi-dimensional 2-2416
  - two-dimensional 2-1530
- data, ASCII
  - converting sparse matrix after loading from 2-3213
- DataAspectRatio, Axes property 2-303
- DataAspectRatioMode, Axes property 2-306
- datatipinfo 2-797
- date 2-798
- date and time functions 2-1028
- date string
  - format of 2-803
- date vector 2-820
- datenum 2-799
- datestr 2-803
- datevec 2-819
- dbclear 2-823
- dbcont 2-826
- dbdown 2-827
- dblquad 2-828
- dbmex 2-830
- dbquit 2-831
- dbstack 2-833
- dbstatus 2-835
- dbstep 2-837
- dbstop 2-839
- dbtype 2-850
- dbup 2-851
- DDE solver properties
  - error tolerance 2-864
  - event location 2-870
  - solver output 2-866
  - step size 2-868
- dde23 2-852
- ddeget 2-857
- ddephas2 output function 2-867
- ddephas3 output function 2-867
- ddeplot output function 2-867
- ddeprint output function 2-867
- ddesd 2-858
- ddeset 2-863
- deal 2-874
- deblank 2-877

- debugging
  - changing workspace context 2-827
  - changing workspace to calling M-file 2-851
  - displaying function call stack 2-833
  - M-files 2-1968 2-2789
  - MEX-files on UNIX 2-830
  - removing breakpoints 2-823
  - resuming execution from breakpoint 2-837
  - setting breakpoints in 2-839
  - stepping through lines 2-837
- dec2base 2-874 2-880
- dec2bin 2-881
- dec2hex 2-882
- decic function 2-883
- decimal number to base conversion 2-874 2-880
- decimal point (.)
  - (special characters) 2-62
  - to distinguish matrix and array operations 2-39
- decomposition
  - Dulmage-Mendelsohn 2-972
  - "economy-size" 2-2827 2-3512
  - orthogonal-triangular (QR) 2-2827
  - Schur 2-3111
  - singular value 2-2911 2-3512
- deconv 2-885
- deconvolution 2-885
- definite integral 2-2839
- del operator 2-886
- del2 2-886
- delaunay 2-889
- Delaunay tessellation
  - 3-dimensional visualization 2-896
  - multidimensional visualization 2-900
- Delaunay triangulation
  - visualization 2-889
- delaunay3 2-896
- delaunayn 2-900
- delete 2-905 2-907
  - serial port I/O 2-911
  - timer object 2-913
- delete (ftp) function 2-909
- delete handle method 2-910
- DeleteFcn
  - areaserie property 2-225
  - Axes property 2-307
  - barseries property 2-356
  - contour property 2-690
  - errorbar property 2-1049
  - Figure property 2-1193
  - hggroup property 2-1621
  - hgtransform property 2-1650
  - Image property 2-1719
  - Light property 2-2031
  - lineseries property 2-2064
  - quivergroup property 2-2870
  - Root property 2-3039
  - scatter property 2-3098
  - stairs series property 2-3280
  - stem property 2-3315
  - Surface property 2-3463
  - surfaceplot property 2-3487
  - Text property 2-3569 2-3572
  - Uicontextmenu property 2-3713 2-3731
  - Uimenu property 2-3777
  - Uipushtool property 2-3811
  - Uitable property 2-3859
  - Uitoggletool property 2-3877
  - Uitoolbar property 2-3890
- DeleteFcn, line property 2-2050
- DeleteFcn, rectangle property 2-2942
- DeleteFcnpatch property 2-2624
- deleteproperty 2-914
- deleting
  - files 2-905
  - items from workspace 2-591
- delevent 2-916
- delimiters in ASCII files 2-964 2-968

- delsample 2-917
- delsamplefromcollection 2-918
- demo 2-919
- demos
  - in Command Window 2-994
- density
  - of sparse matrix 2-2477
- depdir 2-925
- dependence, linear 2-3428
- dependent functions 2-2789
- depfun 2-926
- derivative
  - approximate 2-941
  - polynomial 2-2734
- desktop
  - starting without 2-2207
- det 2-930
- detecting
  - alphabetic characters 2-1898
  - empty arrays 2-1871
  - global variables 2-1886
  - logical arrays 2-1899
  - members of a set 2-1901
  - objects of a given class 2-1863
  - positive, negative, and zero array elements 2-3172
  - sparse matrix 2-1935
- determinant of a matrix 2-930
- detrend 2-931
- detrend (timeseries) 2-933
- deval 2-934
- diag 2-936
- diagonal 2-936
  - anti- 2-1562
  - k-th (illustration) 2-3655
  - main 2-936
  - sparse 2-3215
- dialog 2-938
- dialog box
  - error 2-1063
  - help 2-1605
  - input 2-1788
  - list 2-2106
  - message 2-2383
  - print 1-96 1-108 2-2777
  - question 1-108 2-2854
  - warning 2-4007
- diary 2-939
- Diary, Root property 2-3039
- DiaryFile, Root property 2-3039
- diff 2-941
- differences
  - between adjacent array elements 2-941
  - between sets 2-3147
- differential equation solvers
  - defining an ODE problem 2-2520
  - ODE boundary value problems 2-443 2-454
    - adjusting parameters 2-463
    - extracting properties 2-459
    - extracting properties of 2-1067 to 2-1068 2-3652 to 2-3653
    - forming initial guess 2-460
  - ODE initial value problems 2-2507
    - adjusting parameters of 2-2527
    - extracting properties of 2-2526
  - parabolic-elliptic PDE problems 2-2670
- diffuse 2-943
- DiffuseStrength
  - Surface property 2-3464
  - surfaceplot property 2-3487
- DiffuseStrengthpatch property 2-2624
- digamma function 2-2802
- dimension statement (lack of in MATLAB) 2-4098
- dimensions
  - size of 2-3179
- Diophantine equations 2-1441
- dir 2-944

- dir (ftp) function 2-949
- direct term of a partial fraction expansion 2-3012
- directive
  - %#eml 2-2342
  - %#ok 2-2342
- directories 2-515
  - adding to search path 2-125
  - checking existence of 2-1090
  - copying 2-727
  - creating 2-2322
  - listing 2-2144
  - listing contents of 2-944
  - listing MATLAB files in 2-4035
  - removing 2-3029
  - removing from search path 2-3034
  - See also* directory, search path
- directory 2-944
  - changing on FTP server 2-518
  - listing for FTP server 2-949
  - making on FTP server 2-2325
  - MATLAB location 2-2200
  - root 2-2200
  - temporary system 2-3551
  - See also* directories
- directory, changing 2-515
- directory, current 2-2820
- disconnect 2-605
- discontinuities, eliminating (in arrays of phase angles) 2-3920
- discontinuities, plotting functions with 2-1131
- discontinuous problems 2-1317
- disp 2-952
  - memmapfile object 2-954
  - serial port I/O 2-957
  - timer object 2-958
- disp, MException method 2-955
- display 2-960
- display format 2-1328
- displaying output in Command Window 2-2366
- DisplayName
  - areaseries property 2-225
  - barseries property 2-356
  - contourgroup property 2-691
  - errorbarseries property 2-1049
  - hggroup property 2-1621
  - hgtransform property 2-1651
  - image property 2-1720
  - Line property 2-2051
  - lineseries property 2-2064
  - Patch property 2-2624
  - quivergroup property 2-2871
  - rectangle property 2-2943
  - scattergroup property 2-3098
  - stairs series property 2-3281
  - stemseries property 2-3315
  - surface property 2-3465
  - surfaceplot property 2-3488
  - text property 2-3570
- distribution
  - Gaussian 2-1034
- division
  - array, left (arithmetic operator) 2-41
  - array, right (arithmetic operator) 2-40
  - by zero 2-1775
  - matrix, left (arithmetic operator) 2-40
  - matrix, right (arithmetic operator) 2-40
  - of polynomials 2-885
- divisor
  - greatest common 2-1441
- dll libraries
  - MATLAB functions
    - calllib 2-474
    - libfunctions 2-2013
    - libfunctionsview 2-2014
    - libisloaded 2-2015
    - libpointer 2-2017
    - libstruct 2-2019
    - loadlibrary 2-2120
    - unloadlibrary 2-3906

- dlmread 2-964
- dlmwrite 2-968
- dmperm 2-972
- Dockable, Figure property 2-1194
- docsearch 2-979
- documentation
  - displaying online 2-1601
- dolly camera 2-477
- dos 2-981
  - UNC pathname error 2-982
- dot 2-983
- dot product 2-755 2-983
- dot-parentheses (special characters 2-63
- double 1-57 2-984
- double click, detecting 2-1218
- double integral
  - numerical evaluation 2-828
- DoubleBuffer, Figure property 2-1194
- downloading files from FTP server 2-2310
- dragrect 2-985
- drawing shapes
  - circles and rectangles 2-2932
- DrawMode, Axes property 2-307
- drawnow 2-987
- dsearch 2-989
- dsearchn 2-990
- Dulmage-Mendelsohn decomposition 2-972
- dynamic fields 2-63
- dynamicprops class 2-991
- dynamicprops.addprop 2-128

## E

- echo 2-992
- Echo, Root property 2-3039
- echodemo 2-994
- edge finding, Sobel technique 2-716
- EdgeAlpha
  - patch property 2-2625
  - surface property 2-3465
  - surfaceplot property 2-3488
- EdgeColor
  - annotation ellipse property 2-181
  - annotation rectangle property 2-187
  - annotation textbox property 2-201
  - areaserie property 2-226
  - barseries property 2-357
  - patch property 2-2626
  - Surface property 2-3466
  - surfaceplot property 2-3489
  - Text property 2-3571
- EdgeColor, rectangle property 2-2944
- EdgeLighting
  - patch property 2-2626
  - Surface property 2-3467
  - surfaceplot property 2-3490
- editable text 2-3719
- editing
  - M-files 2-996
- eig 2-999
- eigensystem
  - transforming 2-519
- eigenvalue
  - accuracy of 2-999
  - complex 2-519
  - matrix logarithm and 2-2137
  - modern approach to computation of 2-2730
  - of companion matrix 2-653
  - problem 2-1000 2-2735
  - problem, generalized 2-1000 2-2735
  - problem, polynomial 2-2735
  - repeated 2-1001
  - Wilkinson test matrix and 2-4056
- eigenvalues
  - effect of roundoff error 2-335
  - improving accuracy 2-335



- eigenvector
  - left 2-1000
  - matrix, generalized 2-2887
  - right 2-1000
- eigs 2-1005
- elevation (spherical coordinates) 2-3229
- elevation of viewpoint 2-3978
- ellipj 2-1015
- ellipke 2-1017
- ellipsoid 1-94 2-1019
- elliptic functions, Jacobian
  - (defined) 2-1015
- elliptic integral
  - complete (defined) 2-1017
  - modulus of 2-1015 2-1017
- else 2-1021
- elseif 2-1022
- %#eml 2-2342
- Enable
  - Uicontrol property 2-3732
  - Uimenu property 2-3778
  - Uipushtool property 2-3812
  - Uitable property 2-3859
  - Uitogglehtool property 2-3878
- end 2-1026
- end caps for isosurfaces 2-1908
- end of line, indicating 2-63
- end-of-file indicator 2-1145
- eomday 2-1028
- eps 2-1029
- eq 2-1031
- eq, MException method 2-1033
- equal arrays
  - detecting 2-1874 2-1878
- equal sign (special characters) 2-62
- equations, linear
  - accuracy of solution 2-660
- EraseMode
  - areaseries property 2-226
  - barseries property 2-357
  - contour property 2-691
  - errorbar property 2-1050
  - hggroup property 2-1622
  - hgtransform property 2-1651
  - Image property 2-1721
  - Line property 2-2052
  - lineseries property 2-2065
  - quivergroup property 2-2872
  - rectangle property 2-2944
  - scatter property 2-3099
  - stairs series property 2-3282
  - stem property 2-3316
  - Surface property 2-3467
  - surfaceplot property 2-3490
  - Text property 2-3572
- EraseModepatch property 2-2627
- error 2-1036
  - roundoff. *See* roundoff error
- error function
  - complementary 2-1034
  - (defined) 2-1034
  - scaled complementary 2-1034
- error message
  - displaying 2-1036
  - Index into matrix is negative or zero 2-2133
  - retrieving last generated 2-1974 2-1982
- error messages
  - Out of memory 2-2585
- error tolerance
  - BVP problems 2-464
  - DDE problems 2-864
  - ODE problems 2-2528
- errorbars, confidence interval 2-1041
- errordlg 2-1063
- ErrorMessage, Root property 2-3039

## errors

- in file input/output 2-1146
- MException class 2-1033
  - addCause 2-109
  - constructor 2-2301
  - disp 2-955
  - eq 2-1033
  - getReport 2-1494
  - isequal 2-1877
  - last 2-1972
  - ne 2-2421
  - rethrow 2-3019
  - throw 2-3620
  - throwAsCaller 2-3623
- ErrorType, Root property 2-3040
- escape characters in format specification
  - string 2-1342 2-3252
- etime 2-1066
- etree 2-1067
- etreeplot 2-1068
- eval 2-1069
- evalc 2-1072
- evalin 2-1073
- event location (DDE) 2-870
- event location (ODE) 2-2535
- event.EventData 2-1075
- event.listener 2-1077
- event.PropertyEvent 2-1076
- event.propListener 2-1079
- eventlisteners 2-1081
- events 2-1084 to 2-1085

## examples

- calculating isosurface normals 2-1915
- contouring mathematical expressions 2-1104
- isosurface end caps 2-1908
- isosurfaces 2-1919
- mesh plot of mathematical function 2-1113
- mesh/contour plot 2-1117
- plotting filled contours 2-1108
- plotting function of two variables 2-1121
- plotting parametric curves 2-1124
- polar plot of function 2-1127
- reducing number of patch faces 2-2955
- reducing volume data 2-2958
- subsampling volume data 2-3433
- surface plot of mathematical function 2-1131
- surface/contour plot 2-1135

Excel spreadsheets

- loading 2-4075

exclamation point (special characters) 2-64

Execute 2-1087

executing statements repeatedly 2-1325 2-4043

executing statements repeatedly in

- parallel 2-2601

execution

- improving speed of by setting aside
  - storage 2-4098
- pausing M-file 2-2649
- resuming from breakpoint 2-826
- time for M-files 2-2789

exifread 2-1089

exist 2-1090

exit 2-1094

exp 2-1095

expint 2-1096

expm 2-1097

expm1 2-1099

exponential 2-1095

- complex (defined) 2-1095
- integral 2-1096
- matrix 2-1097

- exponentiation
  - array (arithmetic operator) 2-41
  - matrix (arithmetic operator) 2-41
- export2wsdlg 2-1100
- extension, filename
  - .m 2-1387
  - .mat 2-3069
- Extent
  - Text property 2-3573
  - Uicontrol property 2-3732
  - Uitable property 2-3860
- eye 2-1102
- ezcontour 2-1103
- ezcontourf 2-1107
- ezmesh 2-1111
- ezmeshc 2-1115
- ezplot 2-1119
- ezplot3 2-1123
- ezpolar 2-1126
- ezsurf 2-1129
- ezsurfz 2-1133
  
- F**
- F-norm 2-2480
- FaceAlpha
  - annotation textbox property 2-202
- FaceAlphapatch property 2-2628
- FaceAlphasurface property 2-3468
- FaceAlphasurfaceplot property 2-3491
- FaceColor
  - annotation ellipse property 2-181
  - annotation rectangle property 2-187
  - areaseries property 2-228
  - barseries property 2-359
  - Surface property 2-3469
  - surfaceplot property 2-3492
- FaceColor, rectangle property 2-2945
- FaceColorpatch property 2-2629
- FaceLighting
  - Surface property 2-3469
  - surfaceplot property 2-3493
- FaceLightingpatch property 2-2629
- faces, reducing number in patches 1-106 2-2954
- Faces,patch property 2-2630
- FaceVertexAlphaData, patch property 2-2631
- FaceVertexCData,patch property 2-2631
- factor 2-1137
- factorial 2-1138
- factorization 2-2827
  - LU 2-2160
  - QZ 2-2736 2-2887
  - See also* decomposition
- factorization, Cholesky 2-562
  - (as algorithm for solving linear equations) 2-2335
  - preordering for 2-645
- factors, prime 2-1137
- false 2-1139
- fclose 2-1140
  - serial port I/O 2-1141
- feather 2-1143
- feof 2-1145
- ferror 2-1146
- feval 2-1147
- Feval 2-1149
- fft 2-1154
- FFT. *See* Fourier transform
- fft2 2-1159
- fftn 2-1160
- fftshift 2-1162
- fftw 2-1165
- FFTW 2-1157
- fgetl 2-1170
  - serial port I/O 2-1171
- fgets 2-1174
  - serial port I/O 2-1175
- field names of a structure, obtaining 2-1178
- fieldnames 2-1178

- fields, noncontiguous, inserting data into 2-1401
- fields, of structures
  - dynamic 2-63
- figure 2-1180
- Figure
  - creating 2-1180
  - defining default properties 2-1182
  - properties 2-1183
  - redrawing 1-100 2-2961
- figure windows
  - moving in front of MATLAB® desktop 2-3164
- figure windows, displaying 2-1281
- figurepalette 1-91 2-1237
- figures
  - annotating 2-2715
  - opening 2-2551
  - saving 2-3081
- Figures
  - updating from M-file 2-987
- file
  - extension, getting 2-1256
  - modification date 2-944
  - position indicator
    - finding 2-1258 2-1380
    - setting 2-1378
    - setting to start of file 2-1366
- file formats
  - getting list of supported formats 2-1735
  - reading 2-781 2-1743
  - writing 2-1755
- file size
  - querying 2-1732
- fileattrib 2-1239
- filebrowser 2-1246
- filehandle 2-1253
- filemarker 2-1254
- filename
  - building from parts 2-1384
  - parts 2-1256
  - temporary 2-3552
- filename extension
  - .m 2-1387
  - .mat 2-3069
- fileparts 2-1256
- fileread 2-1258
- files 2-1140
  - ASCII delimited
    - reading 2-964
    - writing 2-968
  - beginning of, rewinding to 2-1366 2-1740
  - checking existence of 2-1090
  - closing 2-1140
  - contents, listing 2-3680
  - copying 2-727
  - deleting 2-905
  - deleting on FTP server 2-909
  - end of, testing for 2-1145
  - errors in input or output 2-1146
  - Excel spreadsheets
    - loading 2-4075
  - fig 2-3081
  - figure, saving 2-3081
  - finding position within 2-1258 2-1380
  - getting next line 2-1170
  - getting next line (with line terminator) 2-1174
  - listing
    - in directory 2-4035
    - names in a directory 2-944
  - listing contents of 2-3680
  - locating 2-4040
  - md1 2-3081
  - mode when opened 2-1319
  - model, saving 2-3081
  - opening 2-1320 2-2551
    - in Web browser 1-5 2-4028
  - opening in Windows applications 2-4057
  - path, getting 2-1256
  - pathname for 2-4040
  - reading

- binary 2-1352
  - data from 2-3593
  - formatted 2-1367
- reading data from 2-781
- reading image data from 2-1743
- rewinding to beginning of 2-1366 2-1740
- setting position within 2-1378
- size, determining 2-947
- sound
  - reading 2-276 2-4021
  - writing 2-277 to 2-278 2-4026
- startup 2-2199
- version, getting 2-1256
- .wav
  - reading 2-4021
  - writing 2-4026
- WK1
  - loading 2-4061
  - writing to 2-4063
- writing binary data to 2-1401
- writing formatted data to 2-1340
- writing image data to 2-1755
- See also* file
- filesep 2-1259
- fill 2-1261
- Fill
  - contour property 2-693
- fill3 2-1264
- filter 2-1267
  - digital 2-1267
  - finite impulse response (FIR) 2-1267
  - infinite impulse response (IIR) 2-1267
  - two-dimensional 2-714
- filter (timeseries) 2-1270
- filter2 2-1273
- find 2-1275
- findall function 2-1280
- findfigs 2-1281
- finding 2-1275
  - sign of array elements 2-3172
  - zero of a function 2-1407
  - See also* detecting
- findobj 2-1282
- findobj handle method 2-1285
- findprop handle method 2-1286
- findstr 2-1287
- finish 2-1288
- finish.m 2-2856
- FIR filter 2-1267
- FitBoxToText, annotation textbox
  - property 2-202
- FitHeightToText
  - annotation textbox property 2-202
- fitsinfo 2-1290
- fitsread 2-1299
- fix 2-1301
- fixed-width font
  - axes 2-308
  - text 2-3574
  - uicontrols 2-3734
  - uitables 2-3861
- FixedColors, Figure property 2-1195
- FixedWidthFontName, Root property 2-3040
- flints 2-2390
- flip
  - array dimension 2-1302
- flip array
  - along dimension 2-1302
- flip matrix
  - on horizontal axis 2-1304
  - on vertical axis 2-1303
- flipdim 2-1302
- fliplr 2-1303
- flipud 2-1304
- floating-point
  - integer, maximum 2-416
- floating-point arithmetic, IEEE
  - smallest positive number 2-2927

- floor 2-1306
- flow control
  - break 2-426
  - case 2-502
  - end 2-1026
  - error 2-1037
  - for 2-1325
  - keyboard 2-1968
  - otherwise 2-2584
  - parfor 2-2601
  - return 2-3021
  - switch 2-3520
  - while 2-4043
- fminbnd 2-1308
- fminsearch 2-1313
- font
  - fixed-width, axes 2-308
  - fixed-width, text 2-3574
  - fixed-width, uicontrols 2-3734
  - fixed-width, uitable 2-3861
- FontAngle
  - annotation textbox property 2-204
  - Axes property 2-308
  - Text property 2-191 2-3574
  - Uicontrol property 2-3733
  - Uitable property 2-3861
- FontName
  - annotation textbox property 2-204
  - Axes property 2-308
  - Text property 2-3574
  - textarrow property 2-191
  - Uicontrol property 2-3733
  - Uitable property 2-3861
- fonts
  - bold 2-191 2-205 2-3575
  - italic 2-191 2-204 2-3574
  - specifying size 2-3575
  - TeX characters
    - bold 2-3587
    - italics 2-3587
    - specifying family 2-3587
    - specifying size 2-3587
  - units 2-191 2-205 2-3575
- FontSize
  - annotation textbox property 2-205
  - Axes property 2-309
  - Text property 2-3575
  - textarrow property 2-191
  - Uicontrol property 2-3734
  - Uitable property 2-3862
- FontUnits
  - Axes property 2-309
  - Text property 2-3575
  - Uicontrol property 2-3735
  - Uitable property 2-3862
- FontWeight
  - annotation textbox property 2-205
  - Axes property 2-309
  - Text property 2-3575
  - textarrow property 2-191
  - Uicontrol property 2-3735
  - Uitable property 2-3862
- fopen 2-1318
  - serial port I/O 2-1323
- for 2-1325
- ForegroundColor
  - Uicontrol property 2-3735
  - Uimenu property 2-3778
  - Uitable property 2-3862
- format 2-1328
  - precision when writing 2-1352
  - reading files 2-1368
  - specification string, matching file data to 2-3267
- Format 2-3040
- formats
  - big endian 2-1320
  - little endian 2-1320
- FormatSpacing, Root property 2-3041

- formatted data
    - reading from file 2-1367
    - writing to file 2-1340
  - formatting data 2-3250
  - Fourier transform
    - algorithm, optimal performance of 2-1157
      - 2-1689 2-1691 2-2476
    - as method of interpolation 2-1835
    - convolution theorem and 2-712
    - discrete, n-dimensional 2-1160
    - discrete, one-dimensional 2-1154
    - discrete, two-dimensional 2-1159
    - fast 2-1154
    - inverse, n-dimensional 2-1693
    - inverse, one-dimensional 2-1689
    - inverse, two-dimensional 2-1691
    - shifting the zero-frequency component
      - of 2-1163
  - fplot 2-1335 2-1351
  - fprintf 2-1340
    - displaying hyperlinks with 2-1345
    - serial port I/O 2-1347
  - fraction, continued 2-2912
  - fragmented memory 2-2585
  - frame2im 2-1351
  - frames 2-3719
  - fread 2-1352
    - serial port I/O 2-1362
  - freqspace 2-1365
  - frequency response
    - desired response matrix
      - frequency spacing 2-1365
  - frequency vector 2-2140
  - frewind 2-1366
  - fromName meta.class method 2-2265
  - fromName meta.package method 2-2276
  - fscanf 2-1367
    - serial port I/O 2-1374
  - fseek 2-1378
  - ftell 2-1380
  - FTP
    - connecting to server 2-1381
  - ftp function 2-1381
  - full 2-1383
  - fullfile 2-1384
  - func2str 2-1385
  - function 2-1387
  - function handle 2-1389
  - function handles
    - overview of 2-1389
  - function syntax 2-1598 2-3538
  - functions 2-1392
    - call history 2-2794
    - call stack for 2-833
    - checking existence of 2-1090
    - clearing from workspace 2-591
    - finding using keywords 2-2141
    - help for 2-1597 2-1607
    - in memory 2-1783
    - locating 2-4040
    - pathname for 2-4040
    - that work down the first non-singleton
      - dimension 2-3165
  - funm 2-1396
  - fwrite 2-1401
    - serial port I/O 2-1403
  - fzero 2-1407
- G**
- gallery 2-1413
  - gamma function
    - (defined) 2-1436
    - incomplete 2-1436
    - logarithm of 2-1436
    - logarithmic derivative 2-2802
  - Gauss-Kronrod quadrature 2-2847
  - Gaussian distribution function 2-1034

- Gaussian elimination
  - (as algorithm for solving linear equations) 2-1850 2-2336
  - Gauss Jordan elimination with partial pivoting 2-3064
  - LU factorization 2-2160
- gca 2-1438
- gcbf function 2-1439
- gcbo function 2-1440
- gcd 2-1441
- gcf 2-1443
- gco 2-1444
- ge 2-1445
- generalized eigenvalue problem 2-1000 2-2735
- generating a sequence of matrix names (M1 through M12) 2-1070
- genpath 2-1447
- genvarname 2-1449
- geodesic dome 2-3533
- get 1-116 2-1453 2-1456
  - memmapfile object 2-1459
  - serial port I/O 2-1463
  - timer object 2-1465
- get (timeseries) 2-1467
- get (tscollection) 2-1468
- get hgsetget class method 2-1458
- get, RandStream method 2-1462
- getabstime (timeseries) 2-1469
- getabstime (tscollection) 2-1471
- getAllPackages meta.package method 2-2277
- getappdata function 2-1473
- getCompilerConfigurations 2-2295
- getdatasamplesize 2-1476
- getDefaultStream, RandStream method 2-1477
- getdisp hgsetget class method 2-1478
- getenv 2-1479
- getfield 2-1480
- getframe 2-1482
  - image resolution and 2-1483
- getinterpmethod 2-1488
- getpixelposition 2-1489
- getpref function 2-1491
- getqualitydesc 2-1493
- getReport, MException method 2-1494
- getsamplusingtime (timeseries) 2-1497
- getsamplusingtime (tscollection) 2-1498
- gettimeseriesnames 2-1499
- gettsafteratevent 2-1500
- gettsafterevent 2-1501
- gettsatevent 2-1502
- gettsbeforeatevent 2-1503
- gettsbeforeevent 2-1504
- gettsbetweenevents 2-1505
- GIF files
  - writing 2-1757
- ginput function 2-1510
- global 2-1512
- global variable
  - defining 2-1512
- global variables, clearing from workspace 2-591
- gmres 2-1514
- golden section search 2-1311
- Goup
  - defining default properties 2-1645
- gplot 2-1520
- grabcode function 2-1522
- gradient 2-1524
- gradient, numerical 2-1524
- graph
  - adjacency 2-973



- graphics objects
    - Axes 2-285
    - Figure 2-1180
    - getting properties 1-97 1-100 2-1453
    - Image 2-1704
    - Light 2-2026
    - Line 2-2039
    - Patch 2-2608
    - resetting properties 1-104 2-3007
    - Root 1-98 2-3036
    - setting properties 1-98 1-100 2-3129
    - Surface 1-98 1-101 2-3451
    - Text 1-98 2-3558
    - uicontextmenu 2-3707
    - Uicontrol 2-3718
    - Uimenu 1-111 2-3770
  - graphics objects, deleting 2-905
  - graphs
    - editing 2-2715
  - graymon 2-1527
  - greatest common divisor 2-1441
  - Greek letters and mathematical symbols 2-195
    - 2-207 2-3585
  - grid 2-1528
    - aligning data to a 2-1530
  - grid arrays
    - for volumetric plots 2-2259
    - multi-dimensional 2-2416
  - griddata 2-1530
  - griddata3 2-1534
  - griddatan 2-1537
  - GridLineStyle, Axes property 2-310
  - group
    - hggroup function 2-1614
  - gsvd 2-1540
  - gt 2-1546
  - gtext 2-1548
  - guidata function 2-1549
  - GUIDE
    - object methods
      - inspect 2-1799
  - guihandles function 2-1552
  - GUIs, printing 2-2771
  - gunzip 2-1553
  - gzip 2-1555
- ## H
- H1 line 2-1599
  - hadamard 2-1557
  - Hadamard matrix 2-1557
    - subspaces of 2-3428
  - handle class 2-1558
  - handle graphics
    - hgtransform 2-1634
  - handle graphicshggroup 2-1614
  - handle relational operators 2-2991
  - handle.addlistener 2-117
  - handle.delete 2-910
  - handle.findobj 2-1285
  - handle.findprop 2-1286
  - handle.isvalid 2-1943
  - handle.notify 2-2485

- HandleVisibility
  - areaserie property 2-228
  - Axes property 2-310
  - barseries property 2-359
  - contour property 2-693
  - errorbar property 2-1051
  - Figure property 2-1195
  - hggroup property 2-1624
  - hgtransform property 2-1653
  - Image property 2-1722
  - Light property 2-2031
  - Line property 2-2053
  - lineseries property 2-2066
  - patch property 2-2633
  - quivergroup property 2-2873
  - rectangle property 2-2945
  - Root property 2-3041
  - stairs series property 2-3283
  - stem property 2-3317
  - Surface property 2-3470
  - surfaceplot property 2-3493
  - Text property 2-3576
  - Uicontextmenu property 2-3714
  - Uicontrol property 2-3735
  - Uimenu property 2-3778
  - Uipushtool property 2-3812
  - Uitable property 2-3863
  - Uitoggletool property 2-3879
  - Uitoolbar property 2-3890
- hankel 2-1562
- Hankel matrix 2-1562
- HDF
  - appending to when saving (WriteMode) 2-1760
  - compression 2-1760
  - setting JPEG quality when writing 2-1760
- HDF files
  - writing images 2-1757
- HDF4
  - summary of capabilities 2-1563
- HDF5
  - high-level access 2-1565
  - summary of capabilities 2-1565
- HDF5 class
  - low-level access 2-1565
- hdf5info 2-1568
- hdf5read 2-1570
- hdf5write 2-1572
- hdfinfo 2-1576
- hdfread 2-1584
- hdfstool 2-1596
- Head1Length
  - annotation doublearrow property 2-176
- Head1Style
  - annotation doublearrow property 2-177
- Head1Width
  - annotation doublearrow property 2-178
- Head2Length
  - annotation doublearrow property 2-176
- Head2Style
  - annotation doublearrow property 2-177
- Head2Width
  - annotation doublearrow property 2-178
- HeadLength
  - annotation arrow property 2-172
  - textarrow property 2-192
- HeadStyle
  - annotation arrow property 2-172
  - textarrow property 2-192
- HeadWidth
  - annotation arrow property 2-173
  - textarrow property 2-193
- Height
  - annotation ellipse property 2-182
- help 2-1597
  - contents file 2-1598
  - keyword search in functions 2-2141
  - online 2-1597
- Help browser 2-1601
  - accessing from doc 2-975

- Help Window 2-1607
- helpbrowser 2-1601
- helpdesk 2-1604
- helpdlg 2-1605
- helpwin 2-1607
- Hermite transformations, elementary 2-1441
- hess 2-1608
- Hessenberg form of a matrix 2-1608
- hex2dec 2-1611
- hex2num 2-1612
- hgsetget class 2-1633
- hgsetget.get 2-1458
- hgsetget.getdisp 2-1478
- hgsetget.set 2-3134
- hidden 2-1658
- Hierarchical Data Format (HDF) files
  - writing images 2-1757
- hilb 2-1659
- Hilbert matrix 2-1659
  - inverse 2-1853
- hist 2-1660
- histc 2-1664
- HitTest
  - areaseries property 2-230
  - Axes property 2-311
  - barseries property 2-361
  - contour property 2-695
  - errorbar property 2-1053
  - Figure property 2-1197
  - hggroup property 2-1625
  - hgtransform property 2-1654
  - Image property 2-1724
  - Light property 2-2033
  - Line property 2-2053
  - lineseries property 2-2068
  - Patch property 2-2634
  - quivergroup property 2-2875
  - rectangle property 2-2946
  - Root property 2-3041
  - scatter property 2-3102
  - stairservices property 2-3285
  - stem property 2-3319
  - Surface property 2-3471
  - surfaceplot property 2-3495
  - Text property 2-3577
  - Uicontrol property 2-3736
  - Uipushtool property 2-3813
  - Uitable property 2-3864
  - Uitoggletool property 2-3879
  - Uitoolbarl property 2-3891
- HitTestArea
  - areaseries property 2-230
  - barseries property 2-361
  - contour property 2-695
  - errorbar property 2-1053
  - quivergroup property 2-2875
  - scatter property 2-3102
  - stairservices property 2-3285
  - stem property 2-3319
- hold 2-1667
- home 2-1669

- HorizontalAlignment
    - Text property 2-3578
    - textarrow property 2-193
    - textbox property 2-205
    - Uicontrol property 2-3737
  - horzcat 2-1670
  - horzcat (M-file function equivalent for [,]) 2-64
  - horzcat (tscollection) 2-1672
  - hostid 2-1673
  - Householder reflections (as algorithm for solving linear equations) 2-2337
  - hsv2rgb 2-1675
  - HTML
    - in Command Window 2-2194
    - save M-file as 2-2805
  - HTML browser
    - in MATLAB 2-1601
  - HTML files
    - opening 1-5 2-4028
  - hyperbolic
    - cosecant 2-759
    - cosecant, inverse 2-89
    - cosine 2-738
    - cosine, inverse 2-79
    - cotangent 2-743
    - cotangent, inverse 2-84
    - secant 2-3118
    - secant, inverse 2-247
    - sine 2-3177
    - sine, inverse 2-252
    - tangent 2-3547
    - tangent, inverse 2-263
  - hyperlink
    - displaying in Command Window 2-952
  - hyperlinks
    - in Command Window 2-2194
  - hyperplanes, angle between 2-3428
  - hypot 2-1676
- I**
- i 2-1679
  - icon images
    - reading 2-1745
  - idealfilter (timeseries) 2-1680
  - identity matrix 2-1102
    - sparse 2-3226
  - idivide 2-1683
  - IEEE floating-point arithmetic
    - smallest positive number 2-2927
  - if 2-1685
  - ifft 2-1689
  - ifft2 2-1691
  - ifftn 2-1693
  - ifftshift 2-1695
  - IIR filter 2-1267
  - ilu 2-1696
  - im2java 2-1701
  - imag 2-1703
  - image 2-1704
  - Image
    - creating 2-1704
    - properties 2-1712
  - image types
    - querying 2-1732
  - images
    - file formats 2-1743 2-1755
    - reading data from files 2-1743
    - returning information about 2-1731
    - writing to files 2-1755
  - Images
    - converting MATLAB image to Java Image 2-1701
  - imagesc 2-1728
  - imaginary 2-1703
    - part of complex number 2-1703
    - unit (sqrt(\(\times d 0 1)) 2-1679 2-1948
    - See also* complex
  - imfinfo
    - returning file information 2-1731

- imformats 2-1735
- import 2-1738
- importdata 2-1740
- importing
  - Java class and package names 2-1738
- imread 2-1743
- imwrite 2-1755
- incomplete beta function
  - (defined) 2-391
- incomplete gamma function
  - (defined) 2-1436
- ind2sub 2-1771
- Index into matrix is negative or zero (error message) 2-2133
- indexing
  - logical 2-2132
- indicator of file position 2-1366
- indices, array
  - of sorted elements 2-3201
- Inf 2-1775
- inferiorto 2-1777
- infinity 2-1775
  - norm 2-2480
- info 2-1779
- information
  - returning file information 2-1731
- inheritance, of objects 2-586
- inline 2-1780
- inmem 2-1783
- inpolygon 2-1785
- input 2-1787
  - checking number of M-file arguments 2-2407
  - name of array passed as 2-1792
  - number of M-file arguments 2-2409
  - prompting users for 2-1787 2-2252
- inputdlg 2-1788
- inputname 2-1792
- inputParser 2-1793
- inspect 2-1799
- installation, root directory of 2-2200
- instance properties 2-128
- instrcallback 2-1807
- instrfind 2-1808
- instrfindall 2-1810
  - example of 2-1811
- int2str 2-1813
- integer
  - floating-point, maximum 2-416
- IntegerHandle
  - Figure property 2-1197
- integration
  - polynomial 2-2741
  - quadrature 2-2839 2-2842
- interfaces 2-1817
- interp1 2-1819
- interp1q 2-1827
- interp2 2-1829
- interp3 2-1833
- interpft 2-1835
- interpn 2-1836
- interpolated shading and printing 2-2772
- interpolation
  - cubic method 2-1530 2-1819 2-1829 2-1833 2-1836
  - cubic spline method 2-1819 2-1829 2-1833 2-1836
  - FFT method 2-1835
  - linear method 2-1819 2-1829 2-1833 2-1836
  - multidimensional 2-1836
  - nearest neighbor method 2-1530 2-1819 2-1829 2-1833 2-1836
  - one-dimensional 2-1819
  - three-dimensional 2-1833
  - trilinear method 2-1530
  - two-dimensional 2-1829
- Interpreter
  - Text property 2-3578
  - textarrow property 2-193
  - textbox property 2-206
- interpstreamspeed 2-1839

**Interruptible**

- areaseries property 2-230
- Axes property 2-311
- barseries property 2-361
- contour property 2-695
- errorbar property 2-1054
- Figure property 2-1197
- hggroup property 2-1625
- hgtransform property 2-1654
- Image property 2-1724
- Light property 2-2033
- Line property 2-2054
- lineseries property 2-2068
- patch property 2-2634
- quivergroup property 2-2875
- rectangle property 2-2947
- Root property 2-3041
- scatter property 2-3103
- stairs series property 2-3285
- stem property 2-3320
- Surface property 2-3471 2-3495
- Text property 2-3580
- Uicontextmenu property 2-3715
- Uicontrol property 2-3737
- Uimenu property 2-3779
- Uipushtool property 2-3813
- Uitable property 2-3864
- Uitoggletool property 2-3880
- Uitoolbar property 2-3891

intersect 2-1843

intmax 2-1844

intmin 2-1845

intwarning 2-1846

inv 2-1850

**inverse**

- cosecant 2-86
- cosine 2-76
- cotangent 2-81
- Fourier transform 2-1689 2-1691 2-1693
- Hilbert matrix 2-1853
- hyperbolic cosecant 2-89
- hyperbolic cosine 2-79
- hyperbolic cotangent 2-84
- hyperbolic secant 2-247
- hyperbolic sine 2-252
- hyperbolic tangent 2-263
- of a matrix 2-1850
- secant 2-244
- sine 2-249
- tangent 2-258
- tangent, four-quadrant 2-260

inversion, matrix

- accuracy of 2-660

InvertHardCopy, Figure property 2-1198

invhilb 2-1853

invoke 2-1854

involutary matrix 2-2607

ipermute 2-1857

iqr (timeseries) 2-1858

is\* 2-1860

isa 2-1863

isappdata function 2-1865

iscell 2-1866

iscellstr 2-1867

ischar 2-1868

iscom 2-1869

isdir 2-1870

isempty 2-1871

isempty (timeseries) 2-1872

isempty (tscollection) 2-1873

isequal 2-1874

isequal, MException method 2-1877

isequalwithequalnans 2-1878

isevent 2-1880

isfield 2-1882  
isfinite 2-1884  
isfloat 2-1885  
isglobal 2-1886  
ishandle 2-1888  
isinf 2-1890  
isinteger 2-1891  
isinterface 2-1892  
isjava 2-1893  
iskeyword 2-1896  
isletter 2-1898  
islogical 2-1899  
ismac 2-1900  
ismember 2-1901  
ismethod 2-1903  
isnan 2-1904  
isnumeric 2-1905  
isobject 2-1906  
isocap 2-1908  
isonormals 2-1915  
isosurface 2-1918  
    calculate data from volume 2-1918  
    end caps 2-1908  
    vertex normals 2-1915  
ispc 2-1923  
ispref function 2-1924  
isprime 2-1925  
isprop 2-1926  
isreal 2-1927  
isscalar 2-1930  
issorted 2-1931  
isspace 2-1934 2-1937  
issparse 2-1935  
isstr 2-1936  
isstruct 2-1940  
isstudent 2-1941  
isunix 2-1942  
isvalid 2-1944  
    timer object 2-1945  
isvalid handle method 2-1943

isvarname 2-1946  
isvector 2-1947  
italics font  
    TeX characters 2-3587

## J

j 2-1948  
Jacobi rotations 2-3248  
Jacobian elliptic functions  
    (defined) 2-1015  
Jacobian matrix (BVP) 2-466  
Jacobian matrix (ODE) 2-2537  
    generating sparse numerically 2-2538  
    2-2540  
    specifying 2-2537 2-2540  
    vectorizing ODE function 2-2538 to 2-2540  
Java  
    class names 2-593 2-1738  
    object methods  
        inspect 2-1799  
    objects 2-1893  
Java Image class  
    creating instance of 2-1701  
Java import list  
    adding to 2-1738  
    clearing 2-593  
Java version used by MATLAB 2-3971  
java\_method 2-1953 2-1960  
java\_object 2-1962  
javaaddath 2-1949  
javachk 2-1954  
javaclasspath 2-1956  
javaMethod 2-1960  
javaObject 2-1962  
javarmpath 2-1964  
joining arrays. *See* concatenation  
Joint Photographic Experts Group (JPEG)  
    writing 2-1757

- JPEG
  - setting Bitdepth 2-1761
  - specifying mode 2-1761
- JPEG comment
  - setting when writing a JPEG image 2-1761
- JPEG files
  - parameters that can be set when writing 2-1761
  - writing 2-1757
- JPEG quality
  - setting when writing a JPEG image 2-1761 2-1765
  - setting when writing an HDF image 2-1760
- jvm
  - version used by MATLAB 2-3971
  
- K**
- K>> prompt
  - keyboard function 2-1968
- keep
  - some variables when clearing 2-596
- keyboard 2-1968
- keyboard mode 2-1968
  - terminating 2-3021
- KeyPressFcn
  - Uicontrol property 2-3738
  - Uitable property 2-3865
- KeyPressFcn, Figure property 2-1198
- KeyReleaseFcn, Figure property 2-1200
- keyword search in functions 2-2141
- keywords
  - iskeyword function 2-1896
- kron 2-1970
- Kronecker tensor product 2-1970
  
- L**
- Label, Uimenu property 2-3780
  
- labeling
  - axes 2-4068
  - matrix columns 2-952
  - plots (with numeric values) 2-2493
- LabelSpacing
  - contour property 2-696
- Laplacian 2-886
- largest array elements 2-2225
- last, MException method 2-1972
- lasterr 2-1974
- lasterror 2-1977
- lastwarn 2-1982
- LaTeX, see TeX 2-195 2-207 2-3585
- Layer, Axes property 2-312
- Layout Editor
  - starting 2-1551
- lcm 2-1984
- LData
  - errorbar property 2-1054
- LDataSource
  - errorbar property 2-1054
- ldivide (M-file function equivalent for .\ ) 2-44
- le 2-1992
- least common multiple 2-1984
- least squares
  - polynomial curve fitting 2-2737
  - problem, overdetermined 2-2698
- legend 2-1994
  - properties 2-1999
  - setting text properties 2-1999
- legendre 2-2003
- Legendre functions
  - (defined) 2-2003
  - Schmidt semi-normalized 2-2003
- length 2-2007
  - serial port I/O 2-2010
- length (timeseries) 2-2011
- length (tscollection) 2-2012
- LevelList
  - contour property 2-696



- LevelListMode
  - contour property 2-696
- LevelStep
  - contour property 2-697
- LevelStepMode
  - contour property 2-697
- libfunctions 2-2013
- libfunctionsview 2-2014
- libisloaded 2-2015
- libpointer 2-2017
- libstruct 2-2019
- license 2-2022
- light 2-2026
- Light
  - creating 2-2026
  - defining default properties 2-1710 2-2027
  - positioning in camera coordinates 2-481
  - properties 2-2028
- Light object
  - positioning in spherical coordinates 2-2036
- lightangle 2-2036
- lighting 2-2037
- limits of axes, setting and querying 2-4070
- line 2-2039
  - editing 2-2715
- Line
  - creating 2-2039
  - defining default properties 2-2044
  - properties 2-2045 2-2060
- line numbers in M-files 2-850
- linear audio signal 2-2038 2-2390
- linear dependence (of data) 2-3428
- linear equation systems
  - accuracy of solution 2-660
  - solving overdetermined 2-2829 to 2-2830
- linear equation systems, methods for solving
  - Cholesky factorization 2-2335
  - Gaussian elimination 2-2336
  - Householder reflections 2-2337
  - matrix inversion (inaccuracy of) 2-1850
- linear interpolation 2-1819 2-1829 2-1833 2-1836
- linear regression 2-2737
- linearly spaced vectors, creating 2-2102
- LineColor
  - contour property 2-697
- lines
  - computing 2-D stream 1-106 2-3345
  - computing 3-D stream 1-106 2-3347
  - drawing stream lines 1-106 2-3349
- LineStyle 1-90 2-2077
- LineStyle
  - annotation arrow property 2-173
  - annotation doublearrow property 2-178
  - annotation ellipse property 2-182
  - annotation line property 2-184
  - annotation rectangle property 2-188
  - annotation textbox property 2-206
  - areaseries property 2-231
  - barseries property 2-362
  - contour property 2-698
  - errorbar property 2-1055
  - Line property 2-2055
  - lineseries property 2-2069
  - patch property 2-2635
  - quivergroup property 2-2876
  - rectangle property 2-2947
  - stairsseries property 2-3286
  - stem property 2-3320
  - surface object 2-3472
  - surfaceplot object 2-3495
  - text object 2-3580
  - textarrow property 2-194
- LineStyleOrder
  - Axes property 2-312

- LineWidth
  - annotation arrow property 2-174
  - annotation doublearrow property 2-179
  - annotation ellipse property 2-182
  - annotation line property 2-185
  - annotation rectangle property 2-188
  - annotation textbox property 2-206
  - areaseries property 2-231
  - Axes property 2-313
  - barseries property 2-362
  - contour property 2-698
  - errorbar property 2-1055
  - Line property 2-2055
  - lineseries property 2-2069
  - Patch property 2-2635
  - quivergroup property 2-2876
  - rectangle property 2-2947
  - scatter property 2-3103
  - stairs series property 2-3286
  - stem property 2-3321
  - Surface property 2-3472
  - surfaceplot property 2-3496
  - text object 2-3581
  - textarrow property 2-194
- linkaxes 2-2083
- linkdata 2-2087
- linkprop 2-2095
- links
  - in Command Window 2-2194
- linsolve 2-2099
- linspace 2-2102
- lint tool for checking problems 2-2339
- list boxes 2-3720
  - defining items 2-3744
- list, RandStream method 2-2103
- ListboxTop, Uicontrol property 2-3739
- listdlg 2-2106
- listfonts 2-2109
- little endian formats 2-1320
- load 2-2111 2-2116
  - serial port I/O 2-2118
- loadlibrary 2-2120
- loadobj 2-2127
- Lobatto IIIa ODE solver 2-452 2-458
- local variables 2-1387 2-1512
- locking M-files 2-2353
- log 2-2128
  - saving session to file 2-939
- log10 [log010] 2-2129
- log1p 2-2130
- log2 2-2131
- logarithm
  - base ten 2-2129
  - base two 2-2131
  - complex 2-2128 to 2-2129
  - natural 2-2128
  - of beta function (natural) 2-393
  - of gamma function (natural) 2-1437
  - of real numbers 2-2925
  - plotting 2-2134
- logarithmic derivative
  - gamma function 2-2802
- logarithmically spaced vectors, creating 2-2140
- logical 2-2132
- logical array
  - converting numeric array to 2-2132
  - detecting 2-1899
- logical indexing 2-2132
- logical operations
  - AND, bit-wise 2-412
  - OR, bit-wise 2-418
  - XOR 2-4095
  - XOR, bit-wise 2-422
- logical operators 2-51 2-58
- logical OR
  - bit-wise 2-418
- logical tests 2-1863
  - all 2-152
  - any 2-212

*See also* detecting  
 logical XOR 2-4095  
     bit-wise 2-422  
 loglog 2-2134  
 logm 2-2137  
 logspace 2-2140  
 lookfor 2-2141  
 lossy compression  
     writing JPEG files with 2-1761  
 Lotus WK1 files  
     loading 2-4061  
     writing 2-4063  
 lower 2-2143  
 lower triangular matrix 2-3655  
 lowercase to uppercase 2-3924  
 ls 2-2144  
 lscov 2-2145  
 lsqnonneg 2-2150  
 lsqr 2-2153  
 lt 2-2158  
 lu 2-2160  
 LU factorization 2-2160  
     storage requirements of (sparse) 2-2497  
 luinc 2-2168

## M

M-file  
     debugging 2-1968  
     displaying during execution 2-992  
     function 2-1387  
     function file, echoing 2-992  
     naming conventions 2-1387  
     pausing execution of 2-2649  
     programming 2-1387  
     script 2-1387  
     script file, echoing 2-992

M-file execution  
     resuming after suspending 2-3826  
     suspending from GUI 2-3894  
 M-files  
     checking existence of 2-1090  
     checking for problems 2-2339  
     clearing from workspace 2-591  
     cyclomatic complexity of 2-2339  
     debugging with profile 2-2789  
     deleting 2-905  
     editing 2-996  
     line numbers, listing 2-850  
     lint tool 2-2339  
     listing names of in a directory 2-4035  
     locking (preventing clearing) 2-2353  
     McCabe complexity of 2-2339  
     opening 2-2551  
     optimizing 2-2789  
     problems, checking for 2-2339  
     save to HTML 2-2805  
     setting breakpoints 2-839  
     unlocking (allowing clearing) 2-2402  
 M-Lint  
     function 2-2339  
     function for entire directory 2-2349  
     HTML report 2-2349  
 machine epsilon 2-4045  
 magic 2-2175  
 magic squares 2-2175  
 Map containers  
     constructor 2-2180 2-3182  
     methods 2-2008 2-2994 2-3954  
 Map methods  
     constructor 2-1894 2-1969  
 Margin  
     annotation textbox property 2-207  
     text object 2-3583

**Marker**

- Line property 2-2055
- lineseries property 2-2069
- marker property 2-1056
- Patch property 2-2635
- quivergroup property 2-2876
- scatter property 2-3104
- stairs series property 2-3286
- stem property 2-3321
- Surface property 2-3472
- surfaceplot property 2-3496

**MarkerEdgeColor**

- errorbar property 2-1056
- Line property 2-2056
- lineseries property 2-2070
- Patch property 2-2636
- quivergroup property 2-2877
- scatter property 2-3104
- stairs series property 2-3287
- stem property 2-3322
- Surface property 2-3473
- surfaceplot property 2-3497

**MarkerFaceColor**

- errorbar property 2-1057
- Line property 2-2056
- lineseries property 2-2070
- Patch property 2-2637
- quivergroup property 2-2877
- scatter property 2-3105
- stairs series property 2-3287
- stem property 2-3322
- Surface property 2-3474
- surfaceplot property 2-3497

**MarkerSize**

- errorbar property 2-1057
- Line property 2-2057
- lineseries property 2-2071
- Patch property 2-2637
- quivergroup property 2-2878
- stairs series property 2-3288
- stem property 2-3322
- Surface property 2-3474
- surfaceplot property 2-3498
- mass matrix (ODE) 2-2541
  - initial slope 2-2542 to 2-2543
  - singular 2-2542
  - sparsity pattern 2-2542
  - specifying 2-2542
  - state dependence 2-2542

**MAT-file 2-3069**

- converting sparse matrix after loading
  - from 2-3213

**MAT-files 2-2111**

- listing for directory 2-4035

**mat2cell 2-2187****mat2str 2-2190****material 2-2192****MATLAB**

- directory location 2-2200
- installation directory 2-2200
- quitting 2-2856
- startup 2-2199
- version number, comparing 2-3969
- version number, displaying 2-3963

**matlab : function 2-2194****matlab (UNIX command) 2-2203****matlab (Windows command) 2-2218****matlab function for UNIX 2-2203****matlab function for Windows 2-2218****MATLAB startup file 2-3296****MATLAB® desktop**

- moving figure windows in front of 2-3164

**matlab.mat 2-2111 2-3069**

- matlabcolon function 2-2194
- matlabrc 2-2199
- matlabroot* 2-2200
- `$matlabroot` 2-2200
- matrices
  - preallocation 2-4098
- matrix 2-39
  - addressing selected rows and columns
    - of 2-65
  - arrowhead 2-645
  - columns
    - rearrange 2-1303
  - companion 2-653
  - complex unitary 2-2827
  - condition number of 2-660 2-2918
  - condition number, improving 2-335
  - converting to formatted data file 2-1340
  - converting to from string 2-3266
  - converting to vector 2-65
  - decomposition 2-2827
  - defective (defined) 2-1001
  - detecting sparse 2-1935
  - determinant of 2-930
  - diagonal of 2-936
  - Dulmage-Mendelsohn decomposition 2-972
  - evaluating functions of 2-1396
  - exponential 2-1097
  - Hadamard 2-1557 2-3428
  - Hankel 2-1562
  - Hermitian Toeplitz 2-3645
  - Hessenberg form of 2-1608
  - Hilbert 2-1659
  - identity 2-1102
  - inverse 2-1850
  - inverse Hilbert 2-1853
  - inversion, accuracy of 2-660
  - involutary 2-2607
  - left division (arithmetic operator) 2-40
  - lower triangular 2-3655
  - magic squares 2-2175 2-3436
  - maximum size of 2-658
  - modal 2-999
  - multiplication (defined) 2-40
  - orthonormal 2-2827
  - Pascal 2-2607 2-2744
  - permutation 2-2160 2-2827
  - poorly conditioned 2-1659
  - power (arithmetic operator) 2-41
  - pseudoinverse 2-2698
  - reading files into 2-964
  - rearrange

- columns 2-1303
- rows 2-1304
- reduced row echelon form of 2-3064
- replicating 2-2999
- right division (arithmetic operator) 2-40
- rotating 90\textbackslash\textbackslashfb 2-3053
- rows
  - rearrange 2-1304
- Schur form of 2-3066 2-3111
- singularity, test for 2-930
- sorting rows of 2-3204
- sparse. *See* sparse matrix
- specialized 2-1413
- square root of 2-3260
- subspaces of 2-3428
- test 2-1413
- Toeplitz 2-3645
- trace of 2-936 2-3647
- transpose (arithmetic operator) 2-41
- transposing 2-62
- unimodular 2-1441
- unitary 2-3512
- upper triangular 2-3662
- Vandermonde 2-2739
- Wilkinson 2-3219 2-4056
- writing as binary data 2-1401
- writing formatted data to 2-1367
- writing to ASCII delimited file 2-968
- writing to spreadsheet 2-4063
- See also* array
- Matrix
  - hgtransform property 2-1655
- matrix functions
  - evaluating 2-1396
- matrix names, (M1 through M12) generating a sequence of 2-1070
- matrix power. *See* matrix, exponential
- max 2-2225
- max (timeseries) 2-2226
- Max, Uicontrol property 2-3739
- MaxHeadSize
  - quivergroup property 2-2878
- maximum matching 2-972
- MDL-files
  - checking existence of 2-1090
- mean 2-2231
- mean (timeseries) 2-2232
- median 2-2234
- median (timeseries) 2-2235
- median value of array elements 2-2234
- memmapfile 2-2237
- memory 2-2243
  - clearing 2-591
  - minimizing use of 2-2585
  - variables in 2-4049
- menu (of user input choices) 2-2252
- menu function 2-2252
- MenuBar, Figure property 2-1203
- Mersenne twister 2-2905 2-2909
- mesh plot
  - tetrahedron 2-3553
- mesh size (BVP) 2-469
- meshc 1-101 2-2254
- meshgrid 2-2259
- MeshStyle, Surface property 2-3474
- MeshStyle, surfaceplot property 2-3498
- meshz 1-101 2-2254
- message
  - error *See* error message 2-4010
  - warning *See* warning message 2-4010
- meta.class 2-2261
- meta.DynamicProperty 2-2266
- meta.event 2-2270
- meta.method 2-2272
- meta.package class 2-2275
- meta.property 2-2278
- methods 2-2283
  - inheritance of 2-586
  - locating 2-4040
- methodsview 2-2285

- mex 2-2287
- mex build script
  - switches 2-2288
    - ada sfcn.ads 2-2289
    - arch 2-2289
    - argcheck 2-2289
    - c 2-2289
    - compatibleArrayDims 2-2289
    - cxx 2-2290
    - Dname 2-2290
    - Dname=value 2-2290
    - f optionsfile 2-2290
    - fortran 2-2290
    - g 2-2290
    - h[elp] 2-2290
    - inline 2-2291
    - Ipathname 2-2290
    - largeArrayDims 2-2291
    - Ldirectory 2-2291
    - lname 2-2291
    - n 2-2291
    - name=value 2-2292
    - O 2-2291
    - outdir dirname 2-2291
    - output resultname 2-2291
    - @rsp\_file 2-2288
    - setup 2-2292
    - Uname 2-2292
    - v 2-2292
- mex.CompilerConfiguration 2-2295
- mex.CompilerConfigurationDetails 2-2295
- MEX-files
  - clearing from workspace 2-591
  - debugging on UNIX 2-830
  - listing for directory 2-4035
- mex.getCompilerConfigurations 2-2295
- MException
  - constructor 2-1033 2-2301
  - methods
    - addCause 2-109
- disp 2-955
- eq 2-1033
- getReport 2-1494
- isequal 2-1877
- last 2-1972
- ne 2-2421
- rethrow 2-3019
- throw 2-3620
- throwAsCaller 2-3623
- mexext 2-2308
- mfilename 2-2309
- mget function 2-2310
- Microsoft Excel files
  - loading 2-4075
- min 2-2311
- min (timeseries) 2-2312
- Min, Uicontrol property 2-3740
- MinColorMap, Figure property 2-1203
- MinorGridLineStyle, Axes property 2-314
- minres 2-2316
- minus (M-file function equivalent for -) 2-44
- mislocked 2-2321
- mkdir 2-2322
- mkdir (ftp) 2-2325
- mkpp 2-2326
- mldivide (M-file function equivalent for \) 2-44
- mlint 2-2339
- mlintrpt 2-2349
  - suppressing messages 2-2352
- mlock 2-2353
- mmfileinfo 2-2354
- mmreader 2-2357
- mod 2-2361
- modal matrix 2-999
- mode 2-2363
- mode objects
  - pan, using 2-2590
  - rotate3d, using 2-3057
  - zoom, using 2-4103

- models
    - opening 2-2551
    - saving 2-3081
  - modification date
    - of a file 2-944
  - modified Bessel functions
    - relationship to Airy functions 2-142
  - modulo arithmetic 2-2361
  - MonitorPosition
    - Root property 2-3041
  - Moore-Penrose pseudoinverse 2-2698
  - more 2-2366 2-2390
  - move 2-2368
  - movefile 2-2370
  - movegui function 2-2373
  - movie 2-2375
  - movie2avi 2-2379
  - movies
    - exporting in AVI format 2-278
  - mpower (M-file function equivalent for  $\wedge$ ) 2-45
  - mput function 2-2382
  - mrdivide (M-file function equivalent for  $/$ ) 2-44
  - msgbox 2-2383
  - mtimes 2-2386
  - mtimes (M-file function equivalent for  $*$ ) 2-44
  - mu-law encoded audio signals 2-2038 2-2390
  - multibandread 2-2391
  - multibandwrite 2-2396
  - multidimensional arrays 2-2007
    - concatenating 2-505
    - interpolation of 2-1836
    - longest dimension of 2-2007
    - number of dimensions of 2-2418
    - rearranging dimensions of 2-1857 2-2689
    - removing singleton dimensions of 2-3263
    - reshaping 2-3010
    - size of 2-3179
    - sorting elements of 2-3200
    - See also* array
  - multiple
    - least common 2-1984
  - multiplication
    - array (arithmetic operator) 2-40
    - matrix (defined) 2-40
    - of polynomials 2-712
  - multistep ODE solver 2-2518
  - munlock 2-2402
- ## N
- Name, Figure property 2-1204
  - namelengthmax 2-2404
  - naming conventions
    - M-file 2-1387
  - NaN 2-2405
  - NaN (Not-a-Number) 2-2405
    - returned by rem 2-2993
  - nargchk 2-2407
  - nargoutchk 2-2411
  - native2unicode 2-2413
  - ndgrid 2-2416
  - ndims 2-2418
  - ne 2-2419
  - ne, MException method 2-2421
  - nearest neighbor interpolation 2-1530 2-1819
    - 2-1829 2-1833 2-1836
  - netcdf
    - summary of capabilities 2-2422 2-2455
  - netcdf.abort
    - revert recent netCDF file definitions 2-2425
  - netcdf.close
    - close netCDF file 2-2427
  - netcdf.copyAtt
    - copy attribute to new location 2-2428
  - netcdf.create
    - create netCDF file 2-2430
  - netcdf.defDim
    - create dimension in netCDF file 2-2432



- netcdf.defVar
  - define variable in netCDF dataset 2-2433
- netcdf.delAtt
  - delete netCDF attribute 2-2435
- netcdf.endDef
  - takes a netCDF file out of define mode 2-2437
- netcdf.getAtt
  - return data from netCDF attribute 2-2439
- netcdf.getConstant
  - get numeric value of netCDF constant 2-2441
- netcdf.getConstantNames
  - get list of netCDF constants 2-2442
- netcdf.getVar
  - return data from netCDF variable 2-2443
- netcdf.inq
  - return information about netCDF file 2-2445
- netcdf.inqAtt
  - return information about a netCDF attribute 2-2447
- netcdf.inqAttID
  - return identifier of netCDF attribute 2-2449
- netcdf.inqAttName
  - return name of netCDF attribute 2-2450
- netcdf.inqDim
  - return information about netCDF dimension 2-2452
- netcdf.inqDimID
  - return dimension ID for netCDF file 2-2453
- netcdf.inqLibVers
  - return version of netCDF library 2-2454
- netcdf.inqVarID
  - return netCDF variable identifier 2-2457
- netcdf.open
  - open an existing netCDF file 2-2458
- netcdf.putAtt
  - write a netCDF attribute 2-2459
- netcdf.putVar
  - write data to netCDF variable 2-2461
- netcdf.reDef
  - put netCDF file into define mode 2-2463
- netcdf.renameAtt
  - netCDF function to change the name of an attribute 2-2464
- netcdf.renameDim
  - netCDF function to change the name of a dimension 2-2466
- netcdf.renameVar
  - change the name of a netCDF variable 2-2468
- netcdf.setDefaultFormat
  - change the default netCDF file format 2-2470
- netcdf.setFill
  - set netCDF fill behavior 2-2471
- netcdf.sync
  - synchronize netCDF dataset to disk 2-2472
- newplot 2-2473
- NextPlot
  - Axes property 2-314
  - Figure property 2-1204
- nextpow2 2-2476
- nnz 2-2477
- no derivative method 2-1317
- nodesktop startup option 2-2207
- noncontiguous fields, inserting data into 2-1401
- nonzero entries
  - specifying maximum number of in sparse matrix 2-3210
- nonzero entries (in sparse matrix)
  - allocated storage for 2-2497
  - number of 2-2477
  - replacing with ones 2-3240
  - vector of 2-2479
- nonzeros 2-2479

- norm 2-2480
    - 1-norm 2-2480 2-2918
    - 2-norm (estimate of) 2-2482
    - F-norm 2-2480
    - infinity 2-2480
    - matrix 2-2480
    - pseudoinverse and 2-2698 2-2700
    - vector 2-2480
  - normal vectors, computing for volumes 2-1915
  - NormalMode
    - Patch property 2-2637
    - Surface property 2-3475
    - surfaceplot property 2-3498
  - normest 2-2482
  - not 2-2483
  - not (M-file function equivalent for ~) 2-55
  - notebook 2-2484
  - notify 2-2485
  - now 2-2486
  - nthroot 2-2487
  - null 2-2488
  - null space 2-2488
  - num2cell 2-2490
  - num2hex 2-2492
  - num2str 2-2493
  - number
    - of array dimensions 2-2418
  - numbers
    - imaginary 2-1703
    - NaN 2-2405
    - plus infinity 2-1775
    - prime 2-2755
    - real 2-2924
    - smallest positive 2-2927
  - NumberTitle, Figure property 2-1205
  - numel 2-2495
  - numeric format 2-1328
  - numeric precision
    - format reading binary data 2-1352
  - numerical differentiation formula ODE solvers 2-2518
  - numerical evaluation
    - double integral 2-828
    - triple integral 2-3657
  - nzmax 2-2497
- O**
- object
    - determining class of 2-1863
    - inheritance 2-586
  - object classes, list of predefined 2-585 2-1863
  - objects
    - Java 2-1893
  - ODE file template 2-2521
  - ODE solver properties
    - error tolerance 2-2528
    - event location 2-2535
    - Jacobian matrix 2-2537
    - mass matrix 2-2541
    - ode15s 2-2543
    - solver output 2-2530
    - step size 2-2534
  - ODE solvers
    - backward differentiation formulas 2-2543
    - numerical differentiation formulas 2-2543
    - obtaining solutions at specific times 2-2505
    - variable order solver 2-2543
  - ode15i function 2-2498
  - odefile 2-2520
  - odeget 2-2526
  - odephas2 output function 2-2532
  - odephas3 output function 2-2532
  - odeplot output function 2-2532
  - odeprint output function 2-2532
  - odeset 2-2527
  - odextend 2-2545
  - off-screen figures, displaying 2-1281

- OffCallback
    - Uitoggletool property 2-3881
  - %#ok 2-2342
  - OnCallback
    - Uitoggletool property 2-3881
  - one-step ODE solver 2-2517
  - ones 2-2550
  - online documentation, displaying 2-1601
  - online help 2-1597
  - open 2-2551
  - openfig 2-2555
  - OpenGL 2-1212
    - autoselection criteria 2-1216
  - opening
    - files in Windows applications 2-4057
  - opening files 2-1320
  - openvar 2-2562
  - operating system
    - MATLAB is running on 2-658
  - operating system command 1-4 1-11 2-3541
  - operating system command, issuing 2-64
  - operators
    - arithmetic 2-39
    - logical 2-51 2-58
    - overloading arithmetic 2-45
    - overloading relational 2-49
    - relational 2-49 2-2132
    - symbols 2-1597
  - optimget 2-2564
  - optimization parameters structure 2-2564 to 2-2565
  - optimizing M-file execution 2-2789
  - optimset 2-2565
  - or 2-2569
  - or (M-file function equivalent for |) 2-55
  - ordeig 2-2571
  - orderfields 2-2574
  - ordering
    - reverse Cuthill-McKee 2-3523 2-3533
  - ordqz 2-2577
  - ordschur 2-2579
  - orient 2-2581
  - orth 2-2583
  - orthogonal-triangular decomposition 2-2827
  - orthographic projection, setting and querying 2-490
  - orthonormal matrix 2-2827
  - otherwise 2-2584
  - Out of memory (error message) 2-2585
  - OuterPosition
    - Axes property 2-314
    - Figure property 2-1205
  - output
    - checking number of M-file arguments 2-2411
    - controlling display format 2-1328
    - in Command Window 2-2366
    - number of M-file arguments 2-2409
  - output points (ODE)
    - increasing number of 2-2530
  - output properties (DDE) 2-866
  - output properties (ODE) 2-2530
    - increasing number of output points 2-2530
  - overdetermined equation systems, solving 2-2829 to 2-2830
  - overflow 2-1775
  - overloading
    - arithmetic operators 2-45
    - relational operators 2-49
    - special characters 2-64
- P**
- P-files
    - checking existence of 2-1090
  - pack 2-2585
  - padecoeff 2-2587
  - pagesetupdlg 2-2588
  - paging
    - of screen 2-1599
  - paging in the Command Window 2-2366

- pan mode objects 2-2590
- PaperOrientation, Figure property 2-1206
- PaperPosition, Figure property 2-1206
- PaperPositionMode, Figure property 2-1207
- PaperSize, Figure property 2-1207
- PaperType, Figure property 2-1207
- PaperUnits, Figure property 2-1208
- parametric curve, plotting 2-1123
- Parent
  - areaseries property 2-232
  - Axes property 2-316
  - barseries property 2-363
  - contour property 2-698
  - errorbar property 2-1057
  - Figure property 2-1209
  - hggroup property 2-1626
  - hgtransform property 2-1655
  - Image property 2-1724
  - Light property 2-2033
  - Line property 2-2057
  - lineseries property 2-2071
  - Patch property 2-2637
  - quivergroup property 2-2878
  - rectangle property 2-2947
  - Root property 2-3042
  - scatter property 2-3105
  - stairs series property 2-3288
  - stem property 2-3322
  - Surface property 2-3475
  - surfaceplot property 2-3499
  - Text property 2-3584
  - Uicontextmenu property 2-3716
  - Uicontrol property 2-3741
  - Uimenu property 2-3781
  - Uipushtool property 2-3815
  - Uitable property 2-3866
  - Uitoggletool property 2-3882
  - Uitoolbar property 2-3892
- parentheses (special characters) 2-62
- parfor 2-2600
- parse method
  - of inputParser object 2-2602
- parseSoapResponse 2-2604
- partial fraction expansion 2-3012
- partialpath 2-2605
- pascal 2-2607
- Pascal matrix 2-2607 2-2744
- patch 2-2608
- Patch
  - converting a surface to 1-107 2-3449
  - creating 2-2608
  - defining default properties 2-2614
  - properties 2-2616
  - reducing number of faces 1-106 2-2954
  - reducing size of face 1-106 2-3168
- path 2-2642
  - adding directories to 2-125
  - building from parts 2-1384
  - current 2-2642
  - removing directories from 2-3034
  - toolbox directory 1-8 2-3646
  - viewing 2-2647
- path2rc 2-2645
- pathname
  - partial 2-2605
- pathnames
  - of functions or files 2-4040
  - relative 2-2605
- pathsep 2-2646
- pathstool 2-2647
- pause 2-2649
- pauses, removing 2-823
- pausing M-file execution 2-2649
- pbaspect 2-2651
- PBM
  - parameters that can be set when writing 2-1761
- PBM files
  - writing 2-1757
- pcg 2-2657

- pchip 2-2661
- pcode 2-2664
- pcolor 2-2666
- PCX files
  - writing 2-1757
- PDE. *See* Partial Differential Equations
- pdepe 2-2670
- pdeval 2-2682
- percent sign (special characters) 2-63
- percent-brace (special characters) 2-63
- perfect matching 2-972
- performance 2-370
- period (.), to distinguish matrix and array
  - operations 2-39
- period (special characters) 2-62
- perl 2-2685
- perl function 2-2685
- Perl scripts in MATLAB 1-4 1-11 2-2685
- perms 2-2688
- permutation
  - matrix 2-2160 2-2827
  - of array dimensions 2-2689
  - random 2-2903
- permutations of n elements 2-2688
- permute 2-2689
- persistent 2-2690
- persistent variable 2-2690
- perspective projection, setting and querying 2-490
- PGM
  - parameters that can be set when writing 2-1761
- PGM files
  - writing 2-1758
- phase angle, complex 2-167
- phase, complex
  - correcting angles 2-3917
- pi 2-2693
- pie 2-2694
- pie3 2-2696
- pinv 2-2698
- planerot 2-2701
- platform MATLAB is running on 2-658
- playshow function 2-2702
- plot 2-2703
  - editing 2-2715
- plot (timeseries) 2-2710
- plot box aspect ratio of axes 2-2651
- plot editing mode
  - overview 2-2716
- Plot Editor
  - interface 2-2716 2-2797
- plot, volumetric
  - generating grid arrays for 2-2259
  - slice plot 1-95 1-106 2-3187
- PlotBoxAspectRatio, Axes property 2-316
- PlotBoxAspectRatioMode, Axes property 2-316
- plottedit 2-2715

- plotting
  - 2-D plot 2-2703
  - 3-D plot 1-90 2-2711
  - contours (a) 2-1103
  - contours (ez function) 2-1103
  - ez-function mesh plot 2-1111
  - feather plots 2-1143
  - filled contours 2-1107
  - function plots 2-1335
  - functions with discontinuities 2-1131
  - histogram plots 2-1660
  - in polar coordinates 2-1126
  - isosurfaces 2-1918
  - loglog plot 2-2134
  - mathematical function 2-1119
  - mesh contour plot 2-1115
  - mesh plot 1-101 2-2254
  - parametric curve 2-1123
  - plot with two y-axes 2-2722
  - ribbon plot 1-95 2-3025
  - rose plot 1-94 2-3049
  - scatter plot 2-2718
  - scatter plot, 3-D 1-95 2-3090
  - semilogarithmic plot 1-90 2-3121
  - stem plot, 3-D 1-93 2-3307
  - surface plot 1-101 2-3443
  - surfaces 1-94 2-1129
  - velocity vectors 2-664
  - volumetric slice plot 1-95 1-106 2-3187
  - . *See* visualizing
- plus (M-file function equivalent for +) 2-44
- PNG
  - writing options for 2-1762
    - alpha 2-1762
    - background color 2-1762
    - chromaticities 2-1763
    - gamma 2-1763
    - interlace type 2-1763
    - resolution 2-1764
    - significant bits 2-1763
    - transparency 2-1764
- PNG files
  - writing 2-1758
- PNM files
  - writing 2-1758
- Pointer, Figure property 2-1209
- PointerLocation, Root property 2-3042
- PointerShapeCData, Figure property 2-1209
- PointerShapeHotSpot, Figure property 2-1210
- PointerWindow, Root property 2-3043
- pol2cart 2-2725
- polar 2-2727
- polar coordinates 2-2725
  - computing the angle 2-167
  - converting from Cartesian 2-499
  - converting to cylindrical or Cartesian 2-2725
  - plotting in 2-1126
- poles of transfer function 2-3012
- poly 2-2729
- polyarea 2-2732
- polyder 2-2734
- polyeig 2-2735
- polyfit 2-2737
- polygamma function 2-2802
- polygon
  - area of 2-2732
  - creating with patch 2-2608
  - detecting points inside 2-1785
- polyint 2-2741

- polynomial
  - analytic integration 2-2741
  - characteristic 2-2729 to 2-2730 2-3047
  - coefficients (transfer function) 2-3012
  - curve fitting with 2-2737
  - derivative of 2-2734
  - division 2-885
  - eigenvalue problem 2-2735
  - evaluation 2-2742
  - evaluation (matrix sense) 2-2744
  - make piecewise 2-2326
  - multiplication 2-712
- polyval 2-2742
- polyvalm 2-2744
- poorly conditioned
  - matrix 2-1659
- poorly conditioned eigenvalues 2-335
- pop-up menus 2-3720
  - defining choices 2-3744
- Portable Anymap files
  - writing 2-1758
- Portable Bitmap (PBM) files
  - writing 2-1757
- Portable Graymap files
  - writing 2-1758
- Portable Network Graphics files
  - writing 2-1758
- Portable pixmap format
  - writing 2-1758
- Position
  - annotation ellipse property 2-182
  - annotation line property 2-185
  - annotation rectangle property 2-189
  - arrow property 2-174
  - Axes property 2-317
  - doubletarrow property 2-179
  - Figure property 2-1210
  - Light property 2-2033
  - Text property 2-3584
  - textarrow property 2-194
  - textbox property 2-207
  - Uicontextmenu property 2-3716
  - Uicontrol property 2-3741
  - Uimenu property 2-3781
  - Uitable property 2-3866
- position indicator in file 2-1258 2-1380
- position of camera
  - dollyng 2-477
- position of camera, setting and querying 2-488
- Position, rectangle property 2-2948
- PostScript
  - default printer 2-2762
  - levels 1 and 2 2-2762
  - printing interpolated shading 2-2772
- pow2 2-2746
- power 2-2747
  - matrix. *See* matrix exponential
  - of real numbers 2-2928
  - of two, next 2-2476
- power (M-file function equivalent for .^) 2-45
- PPM
  - parameters that can be set when writing 2-1761
- PPM files
  - writing 2-1758
- ppval 2-2748
- preallocation
  - matrix 2-4098

- precision 2-1328
    - reading binary data writing 2-1352
  - prefdir 2-2750
  - preferences 2-2754
    - opening the dialog box 2-2754
  - prime factors 2-1137
    - dependence of Fourier transform on 2-1157
      - 2-1159 to 2-1160
  - prime numbers 2-2755
  - primes 2-2755
  - printdlg 1-96 1-108 2-2777
  - printdlg function 2-2777
  - printer
    - default for linux and unix 2-2762
  - printer drivers
    - GhostScript drivers 2-2758
    - interploated shading 2-2772
    - MATLAB printer drivers 2-2758
  - printing
    - GUIs 2-2771
    - interpolated shading 2-2772
    - on MS-Windows 2-2770
    - with a variable file name 2-2774
    - with nodisplay 2-2765
    - with noFigureWindows 2-2765
    - with non-normal EraseMode 2-2053 2-2628
      - 2-2945 2-3468 2-3573
  - printing figures
    - preview 1-96 1-108 2-2778
  - printing tips 2-2770
  - printing, suppressing 2-63
  - printpreview 1-96 1-108 2-2778
  - prod 2-2787
  - product
    - cumulative 2-768
    - Kronecker tensor 2-1970
    - of array elements 2-2787
    - of vectors (cross) 2-755
    - scalar (dot) 2-755
  - profile 2-2789
  - profsave 2-2796
  - projection type, setting and querying 2-490
  - ProjectionType, Axes property 2-317
  - prompting users for input 2-1787 2-2252
  - propedit 2-2797 to 2-2798
  - properties 2-2799
  - proppanel 1-91 2-2801
  - pseudoinverse 2-2698
  - psi 2-2802
  - publish function 2-2804
  - push buttons 2-3721
  - PutFullMatrix 2-2813
  - pwd 2-2820
- Q**
- qmr 2-2821
  - qr 2-2827
  - QR decomposition 2-2827
    - deleting column from 2-2832
  - qrdelete 2-2832
  - qrinsert 2-2834
  - grupdate 2-2836
  - quad 2-2839
  - quadgk 2-2842
  - quadl 2-2848
  - quadrature 2-2839 2-2842
  - quadv 2-2851
  - questdlg 1-108 2-2854
  - questdlg function 2-2854
  - quit 2-2856
  - quitting MATLAB 2-2856
  - quiver 2-2859
  - quiver3 2-2863
  - quotation mark
    - inserting in a string 2-1345
  - qz 2-2887
  - QZ factorization 2-2736 2-2887



**R**

- radio buttons 2-3721
- rand, RandStream method 2-2892
- randi, RandStream method 2-2897
- randn, RandStream method 2-2902
- random
  - permutation 2-2903
  - sparse matrix 2-3246 to 2-3247
  - symmetric sparse matrix 2-3248
- random number generators 2-2103 2-2892
  - 2-2897 2-2902 2-2905 2-2909
- randperm 2-2903
- randStream
  - constructor 2-2909
- RandStream 2-2905 2-2909
  - constructor 2-2905
  - methods
    - create 2-749
    - get 2-1462
    - getDefaultStream 2-1477
    - list 2-2103
    - rand 2-2892
    - randi 2-2897
    - randn 2-2902
    - setDefaultStream 2-3146
- range space 2-2583
- rank 2-2911
- rank of a matrix 2-2911
- RAS files
  - parameters that can be set when writing 2-1765
  - writing 2-1758
- RAS image format
  - specifying color order 2-1765
  - writing alpha data 2-1765
- Raster image files
  - writing 2-1758
- rational fraction approximation 2-2912
- rbbox 1-105 2-2916 2-2961
- rcond 2-2918
- rdivide (M-file function equivalent for ./) 2-44
- read 2-2919
- readasync 2-2921
- reading
  - binary files 2-1352
  - data from files 2-3593
  - formatted data from file 2-1367
  - formatted data from strings 2-3266
- readme files, displaying 1-5 2-1870 2-4039
- real 2-2924
- real numbers 2-2924
- realloc 2-2925
- realmax 2-2926
- realmin 2-2927
- realpow 2-2928
- realsqrt 2-2929
- rearrange array
  - flip along dimension 2-1302
  - reverse along dimension 2-1302
- rearrange matrix
  - flip left-right 2-1303
  - flip up-down 2-1304
  - reverse column order 2-1303
  - reverse row order 2-1304
- RearrangeableColumn
  - Uitable property 2-3867
- rearranging arrays
  - converting to vector 2-65
  - removing first n singleton dimensions 2-3165
  - removing singleton dimensions 2-3263
  - reshaping 2-3010
  - shifting dimensions 2-3165
  - swapping dimensions 2-1857 2-2689
- rearranging matrices
  - converting to vector 2-65
  - rotating 90\xfb 2-3053
  - transposing 2-62
- record 2-2930

- rectangle
  - properties 2-2937
  - rectangle function 2-2932
- rectint 2-2950
- RecursionLimit
  - Root property 2-3043
- recycle 2-2951
- reduced row echelon form 2-3064
- reducepatch 2-2954
- reducevolume 2-2958
- reference page
  - accessing from doc 2-975
- refresh 2-2961
- regexprep 2-2977
- regexptranslate 2-2981
- registerevent 2-2984
- regression
  - linear 2-2737
- regularly spaced vectors, creating 2-65 2-2102
- rehash 2-2987
- relational operators 2-49 2-2132
- relational operators for handle objects 2-2991
- relative accuracy
  - BVP 2-465
  - DDE 2-865
  - norm of DDE solution 2-865
  - norm of ODE solution 2-2529
  - ODE 2-2529
- release 2-2989
- rem 2-2993
- removets 2-2996
- rename function 2-2998
- renderer
  - OpenGL 2-1212
  - painters 2-1211
  - zbuffer 2-1212
- Renderer, Figure property 2-1211
- RendererMode, Figure property 2-1215
- repeatedly executing statements 2-1325 2-4043
- repeatedly executing statements in
  - parallel 2-2601
- replicating a matrix 2-2999
- repmat 2-2999
- resample (timeseries) 2-3001
- resample (tscollection) 2-3004
- reset 2-3007
- reshape 2-3010
- residue 2-3012
- residues of transfer function 2-3012
- Resize, Figure property 2-1217
- ResizeFcn, Figure property 2-1217
- restoredefaultpath 2-3016
- rethrow 2-3017
- rethrow, MException method 2-3019
- return 2-3021
- reverse
  - array along dimension 2-1302
  - array dimension 2-1302
  - matrix column order 2-1303
  - matrix row order 2-1304
- reverse Cuthill-McKee ordering 2-3523 2-3533
- rewinding files to beginning of 2-1366 2-1740
- RGB, converting to HSV 1-102 2-3022
- rgb2hsv 2-3022
- rgbplot 2-3023
- ribbon 2-3025
- right-click and context menus 2-3707
- rmappdata function 2-3028
- rmdir 2-3029
- rmdir (ftp) function 2-3032
- rmfield 2-3033
- rmpath 2-3034
- rmpref function 2-3035
- RMS. *See* root-mean-square
- rolling camera 2-491
- root 1-98 2-3036
- root directory 2-2200
- root directory for MATLAB 2-2200
- Root graphics object 1-98 2-3036

root object 2-3036  
root, see rootobject 1-98 2-3036  
root-mean-square  
    of vector 2-2480  
roots 2-3047  
roots of a polynomial 2-2729 to 2-2730 2-3047  
rose 2-3049  
Rosenbrock  
    banana function 2-1315  
    ODE solver 2-2518  
rosser 2-3052  
rot90 2-3053  
rotate 2-3054  
rotate3d 2-3057  
rotate3d mode objects 2-3057  
rotating camera 2-485  
rotating camera target 1-103 2-487  
Rotation, Text property 2-3584  
rotations  
    Jacobi 2-3248  
round 2-3063  
    to nearest integer 2-3063  
    towards infinity 2-533  
    towards minus infinity 2-1306  
    towards zero 2-1301  
roundoff error  
    characteristic polynomial and 2-2730  
    convolution theorem and 2-712  
    effect on eigenvalues 2-335  
    evaluating matrix functions 2-1399  
    in inverse Hilbert matrix 2-1853  
    partial fraction expansion and 2-3013  
    polynomial roots and 2-3047  
    sparse matrix conversion and 2-3214  
RowName  
    Uitable property 2-3867  
RowStripping  
    Uitable property 2-3868  
rref 2-3064

rrefmovie 2-3064  
rsf2csf 2-3066  
rubberband box 1-105 2-2916  
run 2-3068  
Runge-Kutta ODE solvers 2-2517  
running average 2-1268

## S

save 2-3069 2-3077  
    serial port I/O 2-3079  
saveas 2-3081  
saveobj 2-3085  
savepath 2-3086  
saving  
    ASCII data 2-3069  
    session to a file 2-939  
    workspace variables 2-3069  
scalar product (of vectors) 2-755  
scaled complementary error function  
    (defined) 2-1034  
scatter 2-3087  
scatter3 2-3090  
scattered data, aligning  
    multi-dimensional 2-2416  
    two-dimensional 2-1530  
scattergroup  
    properties 2-3093  
Schmidt semi-normalized Legendre  
    functions 2-2003  
schur 2-3111  
Schur decomposition 2-3111  
Schur form of matrix 2-3066 2-3111  
screen, paging 2-1599  
ScreenDepth, Root property 2-3043  
ScreenPixelsPerInch, Root property 2-3044  
ScreenSize, Root property 2-3044  
script 2-3114  
scrolling screen 2-1599

- search path 2-3034
  - adding directories to 2-125
  - MATLAB 2-2642
  - modifying 2-2647
  - user directory 1-4 1-7 2-3931
  - viewing 2-2647
- search, string 2-1287
- sec 2-3115
- secant 2-3115
  - hyperbolic 2-3118
  - inverse 2-244
  - inverse hyperbolic 2-247
- secd 2-3117
- sech 2-3118
- Selected
  - areaseries property 2-232
  - Axes property 2-318
  - barseries property 2-363
  - contour property 2-698
  - errorbar property 2-1057
  - Figure property 2-1218
  - hggroup property 2-1626
  - hgtransform property 2-1655
  - Image property 2-1725
  - Light property 2-2034
  - Line property 2-2057
  - lineseries property 2-2071
  - Patch property 2-2638
  - quivergroup property 2-2878
  - rectangle property 2-2948
  - Root property 2-3045
  - scatter property 2-3105
  - stairs series property 2-3288
  - stem property 2-3323
  - Surface property 2-3475
  - surfaceplot property 2-3499
  - Text property 2-3585
  - Uicontrol property 2-3742
  - Uitable property 2-3868
- selecting areas 1-105 2-2916
- SelectionHighlight
  - areaseries property 2-232
  - Axes property 2-318
  - barseries property 2-363
  - contour property 2-699
  - errorbar property 2-1058
  - Figure property 2-1218
  - hggroup property 2-1626
  - hgtransform property 2-1655
  - Image property 2-1725
  - Light property 2-2034
  - Line property 2-2057
  - lineseries property 2-2071
  - Patch property 2-2638
  - quivergroup property 2-2879
  - rectangle property 2-2948
  - scatter property 2-3105
  - stairs series property 2-3288
  - stem property 2-3323
  - Surface property 2-3475
  - surfaceplot property 2-3499
  - Text property 2-3585
  - Uicontrol property 2-3742
  - Uitable property 2-3868
- SelectionType, Figure property 2-1218
- selectmoveresize 2-3120
- semicolon (special characters) 2-63
- sendmail 2-3124
- Separator
  - Uipushtool property 2-3815
  - Uitoggletool property 2-3882
- Separator, Uimenu property 2-3782
- sequence of matrix names (M1 through M12)
  - generating 2-1070
- serial 2-3126
- serialbreak 2-3128
- server (FTP)
  - connecting to 2-1381
- server variable 2-1149

- session
  - saving 2-939
- set 1-117 2-3129 2-3133
  - serial port I/O 2-3136
  - timer object 2-3138
- set (timeseries) 2-3141
- set (tscollection) 2-3142
- set hgsetget class method 2-3134
- set operations
  - difference 2-3147
  - exclusive or 2-3161
  - intersection 2-1843
  - membership 2-1901
  - union 2-3898
  - unique 2-3900
- setabstime (timeseries) 2-3143
- setabstime (tscollection) 2-3144
- setappdata 2-3145
- setDefaultStream, RandStream method 2-3146
- setdiff 2-3147
- setdisp hgsetget class method 2-3148
- setenv 2-3149
- setfield 2-3151
- setinterpmethod 2-3153
- setpixelposition 2-3155
- setpref function 2-3158
- setstr 2-3159
- settimeseriesnames 2-3160
- setxor 2-3161
- shading 2-3162
- shading colors in surface plots 1-102 2-3162
- shared libraries
  - MATLAB functions
    - calllib 2-474
    - libfunctions 2-2013
    - libfunctionsview 2-2014
    - libisloaded 2-2015
    - libpointer 2-2017
    - libstruct 2-2019
    - loadlibrary 2-2120
    - unloadlibrary 2-3906
- shell script 1-4 1-11 2-3541 2-3903
- shiftdim 2-3165
- shifting array
  - circular 2-577
- ShowArrowHead
  - quivergroup property 2-2879
- ShowBaseLine
  - barseries property 2-363
- ShowHiddenHandles, Root property 2-3045
- showplottool 2-3166
- ShowText
  - contour property 2-699
- shrinkfaces 2-3168
- shutdown 2-2856
- sign 2-3172
- signum function 2-3172
- simplex search 2-1317
- Simpson's rule, adaptive recursive 2-2841
- Simulink
  - version number, comparing 2-3969
  - version number, displaying 2-3963
- sin 2-3173
- sind 2-3175
- sine 2-3173
  - hyperbolic 2-3177
  - inverse 2-249
  - inverse hyperbolic 2-252
- single 2-3176
- single quote (special characters) 2-62
- singular value
  - decomposition 2-2911 2-3512
  - largest 2-2480
  - rank and 2-2911
- sinh 2-3177
- size
  - array dimesions 2-3179
  - serial port I/O 2-3184
- size (timeseries) 2-3185
- size (tscollection) 2-3186

- size of array dimensions 2-3179
- size of fonts, see also `FontSize` property 2-3587
- size vector 2-3010
- `SizeData`
  - scatter property 2-3106
- skipping bytes (during file I/O) 2-1401
- slice 2-3187
- slice planes, contouring 2-707
- sliders 2-3721
- `SliderStep`, `Uicontrol` property 2-3742
- smallest array elements 2-2311
- `smooth3` 2-3193
- smoothing 3-D data 1-106 2-3193
- soccer ball (example) 2-3533
- solution statistics (BVP) 2-470
- sort 2-3200
- sorting
  - array elements 2-3200
  - complex conjugate pairs 2-747
  - matrix rows 2-3204
- `sortrows` 2-3204
- sound 2-3207 to 2-3208
  - converting vector into 2-3207 to 2-3208
  - files
    - reading 2-276 2-4021
    - writing 2-277 2-4026
  - playing 1-87 2-4019
  - recording 1-87 2-4024
  - resampling 1-87 2-4019
  - sampling 1-87 2-4024
- source control on UNIX platforms
  - checking out files
    - function 2-559
- source control system
  - viewing current system 2-607
- source control systems
  - checking in files 2-556
  - undo checkout 1-10 2-3896
- `spalloc` 2-3209
- `sparse` 2-3210
- sparse matrix
  - allocating space for 2-3209
  - applying function only to nonzero elements of 2-3227
  - density of 2-2477
  - detecting 2-1935
  - diagonal 2-3215
  - finding indices of nonzero elements of 2-1275
  - identity 2-3226
  - number of nonzero elements in 2-2477
  - permuting columns of 2-645
  - random 2-3246 to 2-3247
  - random symmetric 2-3248
  - replacing nonzero elements of with ones 2-3240
  - results of mixed operations on 2-3211
  - solving least squares linear system 2-2828
  - specifying maximum number of nonzero elements 2-3210
  - vector of nonzero elements 2-2479
  - visualizing sparsity pattern of 2-3257
- sparse storage
  - criterion for using 2-1383
- `spaugment` 2-3212
- `spconvert` 2-3213
- `spdiags` 2-3215
- special characters
  - descriptions 2-1597
  - overloading 2-64
- specular 2-3225
- `SpecularColorReflectance`
  - Patch property 2-2638
  - Surface property 2-3475
  - surfaceplot property 2-3499
- `SpecularExponent`
  - Patch property 2-2639
  - Surface property 2-3476
  - surfaceplot property 2-3500

- SpecularStrength
  - Patch property 2-2639
  - Surface property 2-3476
  - surfaceplot property 2-3500
- speye 2-3226
- spfun 2-3227
- sph2cart 2-3229
- sphere 2-3230
- spherical coordinates
  - defining a Light position in 2-2036
- spherical coordinates 2-3229
- spinmap 2-3232
- spline 2-3233
- spline interpolation (cubic)
  - one-dimensional 2-1820 2-1830 2-1833 2-1836
- Spline Toolbox 2-1825
- spones 2-3240
- spparms 2-3241
- sprand 2-3246
- sprandn 2-3247
- sprandsym 2-3248
- sprank 2-3249
- spreadsheets
  - loading WK1 files 2-4061
  - loading XLS files 2-4075
  - reading into a matrix 2-964
  - writing from matrix 2-4063
  - writing matrices into 2-968
- sprintf 2-3250
- sqrt 2-3259
- sqrtn 2-3260
- square root
  - of a matrix 2-3260
  - of array elements 2-3259
  - of real numbers 2-2929
- squeeze 2-3263
- sscanf 2-3266
- stack, displaying 2-833
- standard deviation 2-3297
- start
  - timer object 2-3293
- startat
  - timer object 2-3294
- startup 2-3296
  - directory and path 1-4 1-7 2-3931
- startup file 2-3296
- startup files 2-2199
- State
  - Uitoggletool property 2-3882
- static text 2-3721
- std 2-3297
- std (timeseries) 2-3299
- stem 2-3301
- stem3 2-3307
- step size (DDE)
  - initial step size 2-869
  - upper bound 2-870
- step size (ODE) 2-868 2-2534
  - initial step size 2-2534
  - upper bound 2-2534
- stop
  - timer object 2-3329
- stopasync 2-3330
- stopwatch timer 2-3625
- storage
  - allocated for nonzero entries (sparse) 2-2497
  - sparse 2-3210
- storage allocation 2-4098
- str2cell 2-549
- str2double 2-3331
- str2func 2-3332
- str2mat 2-3334
- str2num 2-3335
- strcat 2-3339
- stream lines
  - computing 2-D 1-106 2-3345
  - computing 3-D 1-106 2-3347
  - drawing 1-106 2-3349
- stream2 2-3345

- stream3 2-3347
- stretch-to-fill 2-286
- strfind 2-3377
- string
  - comparing one to another 2-3341 2-3383
  - converting from vector to 2-555
  - converting matrix into 2-2190 2-2493
  - converting to lowercase 2-2143
  - converting to numeric array 2-3335
  - converting to uppercase 2-3924
  - dictionary sort of 2-3204
  - finding first token in 2-3395
  - searching and replacing 2-3394
  - searching for 2-1287
- String
  - Text property 2-3585
  - textarrow property 2-195
  - textbox property 2-207
  - Uicontrol property 2-3743
- string matrix to cell array conversion 2-549
- strings 2-3379
  - converting to matrix (formatted) 2-3266
  - inserting a quotation mark in 2-1345
  - writing data to 2-3250
- strjust 1-52 1-63 2-3381
- strmatch 2-3382
- stread 2-3386
- strep 1-52 1-63 2-3394
- strtok 2-3395
- strtrim 2-3398
- struct 2-3399
- struct2cell 2-3404
- structfun 2-3405
- structure array
  - getting contents of field of 2-1480
  - remove field from 2-3033
  - setting contents of a field of 2-3151
- structure arrays
  - field names of 2-1178
- structures
  - dynamic fields 2-63
- strvcat 2-3408
- Style
  - Light property 2-2034
  - Uicontrol property 2-3746
- sub2ind 2-3410
- subfunction 2-1387
- subplot 2-3412
- subplots
  - assymetrical 2-3417
  - suppressing ticks in 2-3420
- subsasgn 1-75 2-3425
- subscripts
  - in axis title 2-3643
  - in text strings 2-3589
- subsindex 2-3427
- subspace 1-21 2-3428
- subsref 1-75 2-3429
- subsref (M-file function equivalent for  $A(i, j, k, \dots)$ ) 2-64
- substruct 2-3431
- subtraction (arithmetic operator) 2-39
- subvolume 2-3433
- sum 2-3436
  - cumulative 2-770
  - of array elements 2-3436
- sum (timeseries) 2-3439
- superiorto 2-3441
- superscripts
  - in axis title 2-3643
  - in text strings 2-3589
- support 2-3442
- surf2patch 2-3449
- surface 2-3451



## Surface

- and contour plotter 2-1133
- converting to a patch 1-107 2-3449
- creating 1-98 1-101 2-3451
- defining default properties 2-2935 2-3455
- plotting mathematical functions 2-1129
- properties 2-3456 2-3479
- surface normals, computing for volumes 2-1915
- surf1 2-3506
- surfnorm 2-3510
- svd 2-3512
- svds 2-3515
- swapbytes 2-3518
- switch 2-3520
- symamd 2-3522
- symbfact 2-3526
- symbols
  - operators 2-1597
- symbols in text 2-195 2-207 2-3585
- symmlq 2-3528
- symrcm 2-3533
- synchronize 2-3536
- syntax 2-1598
- syntax, command 2-3538
- syntax, function 2-3538
- syntaxes
  - of M-file functions, defining 2-1387
- system 2-3541
  - UNC pathname error 2-3542
- system directory, temporary 2-3551

**T**

table lookup. *See* interpolation

## Tag

- areaseries property 2-232
- Axes property 2-318
- barseries property 2-364
- contour property 2-699
- errorbar property 2-1058
- Figure property 2-1220
- hggroup property 2-1626
- hgtransform property 2-1656
- Image property 2-1725
- Light property 2-2034
- Line property 2-2058
- lineseries property 2-2072
- Patch property 2-2639
- quivergroup property 2-2879
- rectangle property 2-2948
- Root property 2-3045
- scatter property 2-3106
- stairs series property 2-3289
- stem property 2-3323
- Surface property 2-3476
- surfaceplot property 2-3500
- Text property 2-3590
- Uicontextmenu property 2-3716
- Uicontrol property 2-3746
- Uimenu property 2-3782
- Uipushtool property 2-3815
- Uitable property 2-3868
- Uitoggletool property 2-3882
- Uitoolbar property 2-3892
- Tagged Image File Format (TIFF)
  - writing 2-1758
- tan 2-3544
- tand 2-3546
- tangent 2-3544
  - four-quadrant, inverse 2-260
  - hyperbolic 2-3547
  - inverse 2-258
  - inverse hyperbolic 2-263
- tanh 2-3547

- tar 2-3549
- target, of camera 2-492
- tcpip 2-3926
- tempdir 2-3551
- tempname 2-3552
- temporary
  - files 2-3552
  - system directory 2-3551
- tensor, Kronecker product 2-1970
- terminating MATLAB 2-2856
- test matrices 2-1413
- test, logical. *See* logical tests *and* detecting
- tetrahedron
  - mesh plot 2-3553
- tetramesh 2-3553
- TeX commands in text 2-195 2-207 2-3585
- text 2-3558
  - editing 2-2715
  - subscripts 2-3589
  - superscripts 2-3589
- Text
  - creating 1-98 2-3558
  - defining default properties 2-3561
  - fixed-width font 2-3574
  - properties 2-3563
- text mode for opened files 2-1319
- TextBackgroundColor
  - textarrow property 2-197
- TextColor
  - textarrow property 2-197
- TextEdgeColor
  - textarrow property 2-197
- TextLineWidth
  - textarrow property 2-198
- TextList
  - contour property 2-700
- TextListMode
  - contour property 2-700
- TextMargin
  - textarrow property 2-198
- textread 1-81 2-3593
- TextRotation, textarrow property 2-198
- textscan 1-81 2-3599
- TextStep
  - contour property 2-701
- TextStepMode
  - contour property 2-701
- textwrap 2-3619
- throw, MException method 2-3620
- throwAsCaller, MException method 2-3623
- TickDir, Axes property 2-319
- TickDirMode, Axes property 2-319
- TickLength, Axes property 2-319
- TIFF
  - compression 2-1766
  - encoding 2-1761
  - ImageDescription field 2-1766
  - maxvalue 2-1761
  - parameters that can be set when writing 2-1765
  - resolution 2-1766
  - writemode 2-1766
  - writing 2-1758
- TIFF image format
  - specifying color space 2-1765
- tiling (copies of a matrix) 2-2999
- time
  - CPU 2-748
  - elapsed (stopwatch timer) 2-3625
  - required to execute commands 2-1066
- time and date functions 2-1028
- timer
  - properties 2-3628
  - timer object 2-3628
- timerfind
  - timer object 2-3635
- timerfindall
  - timer object 2-3637
- times (M-file function equivalent for .\*) 2-44
- timeseries 2-3639

- timestamp 2-944
- title 2-3642
  - with superscript 2-3643
- Title, Axes property 2-320
- todatetime 2-3644
- toeplitz 2-3645
- Toeplitz matrix 2-3645
- toggle buttons 2-3721
- token 2-3395
  - See also* string
- Toolbar
  - Figure property 2-1220
- Toolbox
  - Spline 2-1825
- toolbox directory, path 1-8 2-3646
- toolboxdir 2-3646
- TooltipString
  - Uicontrol property 2-3746
  - Uipushtool property 2-3815
  - Uitable property 2-3869
  - Uitoggletool property 2-3882
- trace 2-3647
- trace of a matrix 2-936 2-3647
- trailing blanks
  - removing 2-877
- transform
  - hgtransform function 2-1634
- transform, Fourier
  - discrete, n-dimensional 2-1160
  - discrete, one-dimensional 2-1154
  - discrete, two-dimensional 2-1159
  - inverse, n-dimensional 2-1693
  - inverse, one-dimensional 2-1689
  - inverse, two-dimensional 2-1691
  - shifting the zero-frequency component of 2-1163
- transformation
  - See also* conversion 2-519
- transformations
  - elementary Hermite 2-1441
- transmitting file to FTP server 1-89 2-2382
- transpose
  - array (arithmetic operator) 2-41
  - matrix (arithmetic operator) 2-41
- transpose (M-file function equivalent for `.\q`) 2-45
- transpose (timeseries) 2-3648
- trapz 2-3650
- treelayout 2-3652
- treeplot 2-3653
- triangulation
  - 2-D plot 2-3659
- tricubic interpolation 2-1530
- tril 2-3655
- trilinear interpolation 2-1530
- trimesh 2-3656
- triple integral
  - numerical evaluation 2-3657
- triplequad 2-3657
- triplet 2-3659
- trisurf 2-3661
- triu 2-3662
- true 2-3663
- truth tables (for logical operations) 2-51
- try 2-3664
- tscollection 2-3667
- tsdata.event 2-3670
- tsearch 2-3671
- tsearchn 2-3672
- tsprops 2-3673
- tstool 2-3679
- type 2-3680

**Type**

- areaserie property 2-233
- Axes property 2-320
- barseries property 2-364
- contour property 2-701
- errorbar property 2-1058
- Figure property 2-1221
- hggroup property 2-1627
- hgtransform property 2-1656
- Image property 2-1726
- Light property 2-2034
- Line property 2-2058
- lineseries property 2-2072
- Patch property 2-2640
- quivergroup property 2-2880
- rectangle property 2-2949
- Root property 2-3045
- scatter property 2-3106
- stairsereis property 2-3289
- stem property 2-3324
- Surface property 2-3476
- surfaceplot property 2-3501
- Text property 2-3590
- Uicontextmenu property 2-3717
- Uicontrol property 2-3746
- Uimenu property 2-3782
- Uipushtool property 2-3815
- Uitable property 2-3869
- Uitoggletool property 2-3883
- Uitoolbar property 2-3893

typecast 2-3681

**U****UData**

- errorbar property 2-1059
- quivergroup property 2-2881

**UDataSource**

- errorbar property 2-1059
- quivergroup property 2-2881

**Uibuttongroup**

- defining default properties 2-3689

uibuttongroup function 2-3685

Uibuttongroup Properties 2-3689

uicontextmenu 2-3707

**UiContextMenu**

- Uicontrol property 2-3747

- Uipushtool property 2-3816

- Uitoggletool property 2-3883

- Uitoolbar property 2-3893

**UIContextMenu**

- areaserie property 2-233

- Axes property 2-321

- barseries property 2-364

- contour property 2-702

- errorbar property 2-1059

- Figure property 2-1221

- hggroup property 2-1627

- hgtransform property 2-1656

- Image property 2-1726

- Light property 2-2035

- Line property 2-2058

- lineseries property 2-2072

- Patch property 2-2640

- quivergroup property 2-2880

- rectangle property 2-2949

- scatter property 2-3107

- stairsereis property 2-3290

- stem property 2-3324

- Surface property 2-3477

- surfaceplot property 2-3501

- Text property 2-3591

- Uitable property 2-3869

Uicontextmenu Properties 2-3710

uicontrol 2-3718

**Uicontrol**

- defining default properties 2-3724

- fixed-width font 2-3734

- types of 2-3718

Uicontrol Properties 2-3724

- uicontrols
  - printing 2-2771
- uigetdir 2-3750
- uigetfile 2-3755
- uigetpref function 2-3765
- uiimport 2-3769
- uimenu 2-3770
- Uimenu
  - creating 1-111 2-3770
  - defining default properties 2-3772
  - Properties 2-3772
- Uimenu Properties 2-3772
- uint16 2-3783
- uint32 2-3783
- uint64 2-3783
- uint8 2-1814 2-3783
- uiopen 2-3785
- Uipanel
  - defining default properties 2-3789
- uipanel function 2-3787
- Uipanel Properties 2-3789
- uipushtool 2-3805
- Uipushtool
  - defining default properties 2-3807
- Uipushtool Properties 2-3807
- uiputfile 2-3817
- uiresume 2-3826
- uisave 2-3828
- uisetcolor function 2-3831
- uisetfont 2-3832
- uisetpref function 2-3834
- uistack 2-3835
- Uitable
  - defining default properties 2-3842
  - fixed-width font 2-3861
- uitable function 2-3836
- Uitable Properties 2-3842
- uitoggletool 2-3871
- Uitoggletool
  - defining default properties 2-3873
- Uitoggletool Properties 2-3873
- uitoolbar 2-3884
- Uitoolbar
  - defining default properties 2-3886
- Uitoolbar Properties 2-3886
- uiwait 2-3894
- uminus (M-file function equivalent for unary  $\backslash \times d0$  ) 2-44
- UNC pathname error and dos 2-982
- UNC pathname error and system 2-3542
- unconstrained minimization 2-1313
- undefined numerical results 2-2405
- undocheckout 2-3896
- unicode2native 2-3897
- unimodular matrix 2-1441
- union 2-3898
- unique 2-3900
- unitary matrix (complex) 2-2827
- Units
  - annotation ellipse property 2-182
  - annotation rectangle property 2-189
  - arrow property 2-174
  - Axes property 2-321
  - doublearrow property 2-179
  - Figure property 2-1221
  - line property 2-185
  - Root property 2-3046
  - Text property 2-3590
  - textarrow property 2-198
  - textbox property 2-209
  - Uicontrol property 2-3747
  - Uitable property 2-3869
- unix 2-3903
- UNIX
  - Web browser 2-978
- unloadlibrary 2-3906
- unlocking M-files 2-2402
- unmkpp 2-3907
- unregisterallevents 2-3908
- unregisterevent 2-3911

- untar 2-3915
- unwrap 2-3917
- unzip 2-3922
- up vector, of camera 2-494
- updating figure during M-file execution 2-987
- uplus (M-file function equivalent for unary  
+) 2-44
- upper 2-3924
- upper triangular matrix 2-3662
- uppercase to lowercase 2-2143
- url
  - opening in Web browser 1-5 2-4028
- urlread 2-3925
- urlwrite 2-3927
- usejava 2-3929

- UserData
  - areaserie property 2-233
  - Axes property 2-322
  - barseries property 2-365
  - contour property 2-702
  - errorbar property 2-1060
  - Figure property 2-1222
  - hggroup property 2-1627
  - hgtransform property 2-1657
  - Image property 2-1726
  - Light property 2-2035
  - Line property 2-2058
  - lineseries property 2-2073
  - Patch property 2-2640
  - quivergroup property 2-2880
  - rectangle property 2-2949
  - Root property 2-3046
  - scatter property 2-3107
  - stairs series property 2-3290
  - stem property 2-3324
  - Surface property 2-3477
  - surfaceplot property 2-3501
  - Text property 2-3591
  - Uicontextmenu property 2-3717
  - Uicontrol property 2-3747
  - Uimenu property 2-3782
  - Uipushtool property 2-3816
  - Uitable property 2-3870
  - Uitoggletool property 2-3883
  - Uitoolbar property 2-3893
- userpath 2-3931

## **V**

- validateattributes 2-3941
- validatestring 2-3948
- Value, Uicontrol property 2-3748
- vander 2-3955
- Vandermonde matrix 2-2739
- var 2-3956

- var (timeseries) 2-3957
- varargin 2-3959
- varargout 2-3961
- variable numbers of M-file arguments 2-3961
- variable-order solver (ODE) 2-2543
- variables
  - checking existence of 2-1090
  - clearing from workspace 2-591
  - global 2-1512
  - in workspace 2-4065
  - keeping some when clearing 2-596
  - linking to graphs with linkdata 2-2087
  - listing 2-4049
  - local 2-1387 2-1512
  - name of passed 2-1792
  - opening 2-2551 2-2562
  - persistent 2-2690
  - saving 2-3069
  - sizes of 2-4049
- VData
  - quivergroup property 2-2881
- VDataSource
  - quivergroup property 2-2882
- vector
  - dot product 2-983
  - frequency 2-2140
  - length of 2-2007
  - product (cross) 2-755
- vector field, plotting 2-664
- vectorize 2-3962
- vectorizing ODE function (BVP) 2-466
- vectors, creating
  - logarithmically spaced 2-2140
  - regularly spaced 2-65 2-2102
- velocity vectors, plotting 2-664
- ver 2-3963
- verctrl function (Windows) 2-3965
- verLessThan 2-3969
- version 2-3971
- version numbers
  - comparing 2-3969
  - displaying 2-3963
- vertcat 2-3973
- vertcat (M-file function equivalent for [ 2-64
- vertcat (timeseries) 2-3975
- vertcat (tscollection) 2-3976
- VertexNormals
  - Patch property 2-2640
  - Surface property 2-3477
  - surfaceplot property 2-3501
- VerticalAlignment, Text property 2-3591
- VerticalAlignment, textarrow property 2-199
- VerticalAlignment, textbox property 2-209
- Vertices, Patch property 2-2640
- video
  - saving in AVI format 2-278
- view 2-3977
  - azimuth of viewpoint 2-3978
  - coordinate system defining 2-3978
  - elevation of viewpoint 2-3978
- view angle, of camera 2-496
- View, Axes property (obsolete) 2-322
- viewing
  - a group of object 2-483
  - a specific object in a scene 2-483
- viewmtx 2-3980

**Visible**

- areaserie property 2-234
- Axes property 2-322
- barseries property 2-365
- contour property 2-702
- errorbar property 2-1060
- Figure property 2-1222
- hggroup property 2-1628
- hgtransform property 2-1657
- Image property 2-1726
- Light property 2-2035
- Line property 2-2058
- lineseries property 2-2073
- Patch property 2-2640
- quivergroup property 2-2880
- rectangle property 2-2949
- Root property 2-3046
- scatter property 2-3107
- stairs series property 2-3290
- stem property 2-3324
- Surface property 2-3477
- surfaceplot property 2-3502
- Text property 2-3592
- Uicontextmenu property 2-3717
- Uicontrol property 2-3748
- Uimenu property 2-3782
- Uipushtool property 2-3816
- Uitable property 2-3870
- Uitoggletool property 2-3883
- Uitoolbar property 2-3893

**visualizing**

- cell array structure 2-547
- sparse matrices 2-3257

**volumes**

- calculating isosurface data 2-1918
- computing 2-D stream lines 1-106 2-3345
- computing 3-D stream lines 1-106 2-3347
- computing isosurface normals 2-1915
- contouring slice planes 2-707
- drawing stream lines 1-106 2-3349
- end caps 2-1908
- reducing face size in isosurfaces 1-106 2-3168
- reducing number of elements in 1-106 2-2958

**voronoi 2-3992****Voronoi diagrams**

- multidimensional vizualization 2-3998
- two-dimensional vizualization 2-3992

**voronoin 2-3998****W****wait**

- timer object 2-4002

**waitbar 2-4003****waitfor 2-4005****waitforbuttonpress 2-4006****warndlg 2-4007****warning 2-4010**

- warning message (enabling, suppressing, and displaying) 2-4010

**waterfall 2-4014****.wav files**

- reading 2-4021

- writing 2-4026

**waverecord 2-4024****wavfinfo 2-4018****wavplay 1-87 2-4019****wavread 2-4018 2-4021****wavrecord 1-87 2-4024****wavwrite 2-4026****WData**

- quivergroup property 2-2882



- WDataSource
    - quivergroup property 2-2883
  - web 2-4028
  - Web browser
    - displaying help in 2-1601
    - pointing to file or url 1-5 2-4028
    - specifying for UNIX 2-978
  - weekday 2-4033
  - well conditioned 2-2918
  - what 2-4035
  - whatsnew 2-4039
  - which 2-4040
  - while 2-4043
  - white space characters, ASCII 2-1934 2-3395
  - whitebg 2-4047
  - who, whos
    - who 2-4049
  - wilkinson 2-4056
  - Wilkinson matrix 2-3219 2-4056
  - WindowButtonDownFcn, Figure property 2-1223
  - WindowButtonMotionFcn, Figure
    - property 2-1223
  - WindowButtonUpFcn, Figure property 2-1224
  - WindowKeyPressFcn , Figure property 2-1224
  - WindowKeyReleaseFcn , Figure property 2-1225
  - Windows Paintbrush files
    - writing 2-1757
  - WindowScrollWheelFcn, Figure property 2-1226
  - WindowStyle, Figure property 2-1229
  - winopen 2-4057
  - winqueryreg 2-4058
  - WK1 files
    - loading 2-4061
    - writing from matrix 2-4063
  - wk1finfo 2-4060
  - wk1read 2-4061
  - wk1write 2-4063
  - workspace 2-4065
    - changing context while debugging 2-827
      - 2-851
    - clearing items from 2-591
    - consolidating memory 2-2585
    - predefining variables 2-3296
    - saving 2-3069
    - variables in 2-4049
    - viewing contents of 2-4065
  - workspace variables
    - reading from disk 2-2111
  - writing
    - binary data to file 2-1401
    - formatted data to file 2-1340
  - WVisual, Figure property 2-1231
  - WVisualMode, Figure property 2-1233
- X**
- X
    - annotation arrow property 2-175 2-179
    - annotation line property 2-185
    - textarrow property 2-200
  - X Windows Dump files
    - writing 2-1758
  - x-axis limits, setting and querying 2-4070
  - XAxisLocation, Axes property 2-322
  - XColor, Axes property 2-323

**XData**

- areaserie property 2-234
- barseries property 2-365
- contour property 2-702
- errorbar property 2-1060
- Image property 2-1726
- Line property 2-2059
- lineseries property 2-2073
- Patch property 2-2641
- quivergroup property 2-2883
- scatter property 2-3107
- stairsereis property 2-3290
- stem property 2-3325
- Surface property 2-3477
- surfaceplot property 2-3502

**XDataMode**

- areaserie property 2-234
- barseries property 2-365
- contour property 2-703
- errorbar property 2-1060
- lineseries property 2-2073
- quivergroup property 2-2884
- stairsereis property 2-3291
- stem property 2-3325
- surfaceplot property 2-3502

**XDataSource**

- areaserie property 2-235
- barseries property 2-366
- contour property 2-703
- errorbar property 2-1061
- lineseries property 2-2074
- quivergroup property 2-2884
- scatter property 2-3108
- stairsereis property 2-3291
- stem property 2-3325
- surfaceplot property 2-3502

XDir, Axes property 2-323

XDisplay, Figure property 2-1233

XGrid, Axes property 2-324

xlabel 1-91 2-4068

XLabel, Axes property 2-324

xlim 2-4070

XLim, Axes property 2-325

XLimMode, Axes property 2-325

XLS files

- loading 2-4075

xlsfinfo 2-4073

xlsread 2-4075

xlswrite 2-4085

XMinorGrid, Axes property 2-326

xmlread 2-4089

xmlwrite 2-4094

xor 2-4095

XOR, printing 2-227 2-358 2-692 2-1051 2-1652  
2-1722 2-2053 2-2066 2-2628 2-2873 2-2945  
2-3100 2-3283 2-3317 2-3468 2-3491 2-3573

XScale, Axes property 2-326

xslt 2-4096

XTick, Axes property 2-326

XTickLabel, Axes property 2-327

XTickLabelMode, Axes property 2-328

XTickMode, Axes property 2-327

XVisual, Figure property 2-1234

XVisualMode, Figure property 2-1236

XWD files

- writing 2-1758

xyz coordinates . *See* Cartesian coordinates

**Y****Y**

- annotation arrow property 2-175 2-179 2-186
- textarrow property 2-200

y-axis limits, setting and querying 2-4070

YAxisLocation, Axes property 2-322

YColor, Axes property 2-323

**YData**

- areaserie property 2-235
- barseries property 2-366
- contour property 2-704
- errorbar property 2-1061
- Image property 2-1727
- Line property 2-2059
- lineseries property 2-2074
- Patch property 2-2641
- quivergroup property 2-2885
- scatter property 2-3108
- stairsereis property 2-3292
- stem property 2-3326
- Surface property 2-3478
- surfaceplot property 2-3503

**YDataMode**

- contour property 2-704
- quivergroup property 2-2885
- surfaceplot property 2-3503

**YDataSource**

- areaserie property 2-236
- barseries property 2-367
- contour property 2-704
- errorbar property 2-1062
- lineseries property 2-2075
- quivergroup property 2-2885
- scatter property 2-3109
- stairsereis property 2-3292
- stem property 2-3326
- surfaceplot property 2-3503

**YDir, Axes property 2-323****YGrid, Axes property 2-324****ylabel 1-91 2-4068****YLabel, Axes property 2-324****ylim 2-4070****YLim, Axes property 2-325****YLimMode, Axes property 2-325****YMinorGrid, Axes property 2-326****YScale, Axes property 2-326****YTick, Axes property 2-326****YTickLabel, Axes property 2-327****YTickLabelMode, Axes property 2-328****YTickMode, Axes property 2-327****Z****z-axis limits, setting and querying 2-4070****ZColor, Axes property 2-323****ZData**

- contour property 2-705
- Line property 2-2059
- lineseries property 2-2075
- Patch property 2-2641
- quivergroup property 2-2886
- scatter property 2-3109
- stemsereis property 2-3327
- Surface property 2-3478
- surfaceplot property 2-3504

**ZDataSource**

- contour property 2-705
- lineseries property 2-2075 2-3327
- scatter property 2-3109
- surfaceplot property 2-3504

**ZDir, Axes property 2-323****zero of a function, finding 2-1407****zeros 2-4098****ZGrid, Axes property 2-324****Ziggurat 2-2905 2-2909****zip 2-4100****zlabel 1-91 2-4068****zlim 2-4070****ZLim, Axes property 2-325****ZLimMode, Axes property 2-325****ZMinorGrid, Axes property 2-326****zoom 2-4102****zoom mode objects 2-4103****ZScale, Axes property 2-326****ZTick, Axes property 2-326****ZTickLabel, Axes property 2-327****ZTickLabelMode, Axes property 2-328**

ZTickMode, Axes property 2-327