

MATLAB

The Language of Technical Computing

- Computation
- Visualization
- Programming

Function Reference
Volume 3: P-Z

Version 7



How to Contact The MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
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 1984–2006 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

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

Patents

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

Revision History

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)

Functions — By Category

1

Desktop Tools and Development Environment	1-3
Startup and Shutdown	1-3
Command Window and History	1-4
Help for Using MATLAB	1-4
Workspace, Search Path, and File Operations	1-5
Programming Tools	1-8
System	1-10
Mathematics	1-13
Arrays and Matrices	1-14
Linear Algebra	1-19
Elementary Math	1-23
Polynomials	1-27
Interpolation and Computational Geometry	1-28
Cartesian Coordinate System Conversion	1-30
Nonlinear Numerical Methods	1-31
Specialized Math	1-34
Sparse Matrices	1-35
Math Constants	1-39
Data Analysis	1-40
Basic Operations	1-40
Descriptive Statistics	1-40
Filtering and Convolution	1-41
Interpolation and Regression	1-41
Fourier Transforms	1-42
Derivatives and Integrals	1-42
Time Series Objects	1-43
Time Series Collections	1-46
Programming and Data Types	1-48
Data Types	1-48
Data Type Conversion	1-56
Operators and Special Characters	1-59

String Functions	1-61
Bit-wise Functions	1-64
Logical Functions	1-65
Set Functions	1-65
Date and Time Functions	1-66
Programming in MATLAB	1-66
File I/O	1-73
File Name Construction	1-73
Opening, Loading, Saving Files	1-74
Memory Mapping	1-74
Low-Level File I/O	1-74
Text Files	1-75
XML Documents	1-75
Spreadsheets	1-76
Scientific Data	1-76
Audio and Audio/Video	1-78
Images	1-80
Internet Exchange	1-80
Graphics	1-82
Basic Plots and Graphs	1-82
Plotting Tools	1-83
Annotating Plots	1-83
Specialized Plotting	1-84
Bit-Mapped Images	1-88
Printing	1-88
Handle Graphics	1-89
3-D Visualization	1-94
Surface and Mesh Plots	1-94
View Control	1-96
Lighting	1-98
Transparency	1-98
Volume Visualization	1-98
Creating Graphical User Interfaces	1-100
Predefined Dialog Boxes	1-100
Deploying User Interfaces	1-101
Developing User Interfaces	1-101
User Interface Objects	1-102
Finding Objects from Callbacks	1-103

GUI Utility Functions	1-103
Controlling Program Execution	1-103
External Interfaces	1-104
Dynamic Link Libraries	1-104
Java	1-105
Component Object Model and ActiveX	1-106
Dynamic Data Exchange	1-108
Web Services	1-109
Serial Port Devices	1-109

Functions — Alphabetical List

2

Index

Functions — By Category

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. 1-40)

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. 1-48)

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

File I/O (p. 1-73)

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

Graphics (p. 1-82)

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

3-D Visualization (p. 1-94)

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

Creating Graphical User Interfaces
(p. 1-100)

GUIDE, programming graphical
user interfaces

External Interfaces (p. 1-104)

Interfaces to DLLs, Java, COM
and ActiveX, DDE, 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. 1-4)	Command line help, online documentation in the Help browser, demos
Workspace, Search Path, and File Operations (p. 1-5)	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. 1-10)	Identify current computer, license, product version, and more

Startup and Shutdown

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

quit	Terminate MATLAB
startup	MATLAB startup M-file for user-defined options

Command Window and History

clc	Clear Command Window
commandhistory	Open Command History window, or select it if already open
commandwindow	Open Command Window, or select it if already open
diary	Save session to file
dos	Execute DOS command and return result
format	Set display format for output
home	Move cursor to upper-left corner of Command Window
more	Control paged output for Command Window
perl	Call Perl script using appropriate operating system executable
system	Execute operating system command and return result
unix	Execute UNIX command and return result

Help for Using MATLAB

demo	Access product demos via Help browser
doc	Reference page in Help browser

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

Search Path

addpath	Add directories to MATLAB search path
genpath	Generate path string
partialpath	Partial pathname description
path	View or change MATLAB directory search path
path2rc	Save current MATLAB search path to pathdef.m file
pathdef	Directories in MATLAB search path
pathsep	Path separator for current platform

pathtool	Open Set Path dialog box to view and change MATLAB path
restoredefaultpath	Restore default MATLAB search path
rmpath	Remove directories from MATLAB search path
savepath	Save current MATLAB search path to pathdef.m file

File Operations

See also “File I/O” on page 1-73 functions.

cd	Change working directory
copyfile	Copy file or directory
delete	Remove files or graphics objects
dir	Directory listing
exist	Check existence of variable, function, directory, or Java class
fileattrib	Set or get attributes of file or directory
filebrowser	Current Directory browser
isdir	Determine whether input is a directory
lookfor	Search for keyword in all help entries
ls	Directory contents on UNIX system
matlabroot	Root directory of MATLAB installation
mkdir	Make new directory
movefile	Move file or directory
pwd	Identify current directory

recycle	Set option to move deleted files to recycle folder
rehash	Refresh function and file system path caches
rmdir	Remove directory
type	Display contents of file
web	Open Web site or file in Web browser or Help browser
what	List MATLAB files in current directory
which	Locate functions and files

Programming Tools

Edit and Debug M-Files (p. 1-8)	Edit and debug M-files
Improve Performance and Tune M-Files (p. 1-9)	Improve performance and find potential problems in M-files
Source Control (p. 1-10)	Interface MATLAB with source control system
Publishing (p. 1-10)	Publish M-file code and results

Edit and Debug M-Files

clipboard	Copy and paste strings to and from system clipboard
datatipinfo	Produce short description of input variable
dbclear	Clear breakpoints
dbcont	Resume execution
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

Improve Performance and Tune M-Files

memory	Help for memory limitations
mlint	Check M-files for possible problems
mlintrpt	Run <code>mlint</code> for file or directory, reporting results in a browser
pack	Consolidate workspace memory
profile	Profile execution time for function
profsave	Save profile report in HTML format
rehash	Refresh function and file system path caches
sparse	Create sparse matrix
zeros	Create array of all zeros

Source Control

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

Publishing

grabcode	MATLAB code from M-files published to HTML
notebook	Open M-book in Microsoft Word (Windows)
publish	Run M-file script containing cells, saving results to file of specified type

System

Operating System Interface (p. 1-11)	Exchange operating system information and commands with MATLAB
MATLAB Version and License (p. 1-11)	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 is running
dos	Execute DOS command and return result
getenv	Environment variable
hostid	MATLAB server host identification number
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 Microsoft Windows registry

MATLAB Version and License

ispc	Determine whether PC (Windows) version of MATLAB
isstudent	Determine whether Student Version of MATLAB
isunix	Determine whether UNIX version of MATLAB
javachk	Generate error message based on Java feature support
license	Return license number or perform licensing task

<code>prefdir</code>	Directory containing preferences, history, and layout files
<code>usejava</code>	Determine whether Java feature is supported in MATLAB
<code>ver</code>	Version information for MathWorks products
<code>version</code>	Version number for MATLAB

Mathematics

Arrays and Matrices (p. 1-14)	Basic array operators and operations, creation of elementary and specialized arrays and matrices
Linear Algebra (p. 1-19)	Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization
Elementary Math (p. 1-23)	Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math
Polynomials (p. 1-27)	Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion
Interpolation and Computational Geometry (p. 1-28)	Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation
Cartesian Coordinate System Conversion (p. 1-30)	Conversions between Cartesian and polar or spherical coordinates
Nonlinear Numerical Methods (p. 1-31)	Differential equations, optimization, integration
Specialized Math (p. 1-34)	Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions
Sparse Matrices (p. 1-35)	Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations
Math Constants (p. 1-39)	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. 1-17)

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

Specialized Matrices (p. 1-18)

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

Basic Information

disp

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

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

Operators

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

<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>diag</code>	Diagonal matrices and diagonals of matrix
<code>eye</code>	Identity matrix
<code>freqspace</code>	Frequency spacing for frequency response
<code>ind2sub</code>	Subscripts from linear index
<code>linspace</code>	Generate linearly spaced vectors
<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>randn</code>	Normally distributed random numbers
<code>sub2ind</code>	Single index from subscripts
<code>zeros</code>	Create array of all zeros

Array Operations

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

accumarray	Construct array with accumulation
arrayfun	Apply function to each element of array
cast	Cast variable to different data type
cross	Vector cross product
cumprod	Cumulative product
cumsum	Cumulative sum
dot	Vector dot product
idivide	Integer division with rounding option
kron	Kronecker tensor product
prod	Product of array elements
sum	Sum of array elements
tril	Lower triangular part of matrix
triu	Upper triangular part of matrix

Array Manipulation

blkdiag	Construct block diagonal matrix from input arguments
cat	Concatenate arrays along specified dimension
circshift	Shift array circularly
diag	Diagonal matrices and diagonals of matrix

end	Terminate block of code, or indicate last array index
flipdim	Flip array along specified dimension
fliplr	Flip matrix left to right
flipud	Flip matrix up to down
horzcat	Concatenate arrays horizontally
ipermute	Inverse permute dimensions of N-D array
permute	Rearrange dimensions of N-D array
repmat	Replicate and tile array
reshape	Reshape array
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
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. 1-19)	Compute norm, rank, determinant, condition number, etc.
Linear Equations (p. 1-20)	Solve linear systems, least squares, LU factorization, Cholesky factorization, etc.
Eigenvalues and Singular Values (p. 1-21)	Eigenvalues, eigenvectors, Schur decomposition, Hessenburg matrices, etc.
Matrix Logarithms and Exponentials (p. 1-22)	Matrix logarithms, exponentials, square root
Factorization (p. 1-22)	Cholesky, LU, and QR factorizations, diagonal forms, singular value decomposition

Matrix Analysis

cond	Condition number with respect to inversion
condeig	Condition number with respect to eigenvalues
det	Matrix determinant
norm	Vector and matrix norms
normest	2-norm estimate
null	Null space

orth	Range space of matrix
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
rref	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
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

balance	Diagonal scaling to improve eigenvalue accuracy
cdf2rdf	Convert complex diagonal form to real block diagonal form
condeig	Condition number with respect to eigenvalues
eig	Find eigenvalues and eigenvectors
eigs	Find largest eigenvalues and eigenvectors of sparse matrix
gsvd	Generalized singular value decomposition
hess	Hessenberg form of matrix
ordeig	Eigenvalues of quasitriangular matrices
ordqz	Reorder eigenvalues in QZ factorization
ordschur	Reorder eigenvalues in Schur factorization
poly	Polynomial with specified roots
polyeig	Polynomial eigenvalue problem
rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
sqrtm	Matrix square root

svd	Singular value decomposition
svds	Find singular values and vectors

Matrix Logarithms and Exponentials

expm	Matrix exponential
logm	Matrix logarithm
sqrtn	Matrix square root

Factorization

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

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

Elementary Math

Trigonometric (p. 1-23)	Trigonometric functions with results in radians or degrees
Exponential (p. 1-25)	Exponential, logarithm, power, and root functions
Complex (p. 1-26)	Numbers with real and imaginary components, phase angles
Rounding and Remainder (p. 1-26)	Rounding, modulus, and remainder
Discrete Math (e.g., Prime Factors) (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
tand	Tangent of argument in degrees
tanh	Hyperbolic tangent

Exponential

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

Complex

abs	Absolute value and complex magnitude
angle	Phase angle
complex	Construct complex data from real and imaginary components
conj	Complex conjugate
cplxpair	Sort complex numbers into complex conjugate pairs
i	Imaginary unit
imag	Imaginary part of complex number
isreal	Determine whether input is real array
j	Imaginary unit
real	Real part of complex number
sign	Signum function
unwrap	Correct phase angles to produce smoother phase plots

Rounding and Remainder

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

Discrete Math (e.g., Prime Factors)

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

Polynomials

conv	Convolution and polynomial multiplication
deconv	Deconvolution and polynomial division
poly	Polynomial with specified roots
polyder	Polynomial derivative
polyeig	Polynomial eigenvalue problem
polyfit	Polynomial curve fitting
polyint	Integrate polynomial analytically
polyval	Polynomial evaluation
polyvalm	Matrix polynomial evaluation

residue	Convert between partial fraction expansion and polynomial coefficients
roots	Polynomial roots

Interpolation and Computational Geometry

Interpolation (p. 1-28)	Data interpolation, data gridding, polynomial evaluation, nearest point search
Delaunay Triangulation and Tessellation (p. 1-29)	Delaunay triangulation and tessellation, triangular surface and mesh plots
Convex Hull (p. 1-30)	Plot convex hull, plotting functions
Voronoi Diagrams (p. 1-30)	Plot Voronoi diagram, patch graphics object, plotting functions
Domain Generation (p. 1-30)	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 ≥ 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
interp	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
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

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

Voronoi Diagrams

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

Domain Generation

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

Cartesian Coordinate System Conversion

cart2pol	Transform Cartesian coordinates to polar or cylindrical
cart2sph	Transform Cartesian coordinates to spherical

pol2cart	Transform polar or cylindrical coordinates to Cartesian
sph2cart	Transform spherical coordinates to Cartesian

Nonlinear Numerical Methods

Ordinary Differential Equations (IVP) (p. 1-31)	Solve stiff and nonstiff differential equations, define the problem, set solver options, evaluate solution
Delay Differential Equations (p. 1-32)	Solve delay differential equations with constant and general delays, set solver options, evaluate solution
Boundary Value Problems (p. 1-33)	Solve boundary value problems for ordinary differential equations, set solver options, evaluate solution
Partial Differential Equations (p. 1-33)	Solve initial-boundary value problems for parabolic-elliptic PDEs, evaluate solution
Optimization (p. 1-33)	Find minimum of single and multivariable functions, solve nonnegative least-squares constraint problem
Numerical Integration (Quadrature) (p. 1-34)	Evaluate Simpson, Lobatto, and vectorized quadratures, evaluate double and triple integrals

Ordinary Differential Equations (IVP)

decic	Compute consistent initial conditions for ode15i
deval	Evaluate solution of differential equation problem

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

Delay Differential Equations

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

Boundary Value Problems

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

Partial Differential Equations

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

Optimization

fminbnd	Find minimum of single-variable function on fixed interval
fminsearch	Find minimum of unconstrained multivariable function using derivative-free method
fzero	Find root of continuous function of one variable
lsqnonneg	Solve nonnegative least-squares constraints problem

optimget	Optimization options values
optimset	Create or edit optimization options structure

Numerical Integration (Quadrature)

dblquad	Numerically evaluate double integral
quad	Numerically evaluate integral, adaptive Simpson quadrature
quadl	Numerically evaluate integral, adaptive Lobatto quadrature
quadv	Vectorized quadrature
triplequad	Numerically evaluate triple integral

Specialized Math

airy	Airy functions
besselh	Bessel function of third kind (Hankel function)
besseli	Modified Bessel function of first kind
besselj	Bessel function of first kind
besselk	Modified Bessel function of second kind
bessely	Bessel function of second kind
beta	Beta function
betainc	Incomplete beta function
betaln	Logarithm of beta function
ellipj	Jacobi elliptic functions

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

Sparse Matrices

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

Elementary Sparse Matrices

<code>spdiags</code>	Extract and create sparse band and diagonal matrices
<code>speye</code>	Sparse identity matrix
<code>sprand</code>	Sparse uniformly distributed random matrix
<code>sprandn</code>	Sparse normally distributed random matrix
<code>sprandsym</code>	Sparse symmetric random matrix

Full to Sparse Conversion

<code>find</code>	Find indices and values of nonzero elements
<code>full</code>	Convert sparse matrix to full matrix
<code>sparse</code>	Create sparse matrix
<code>spconvert</code>	Import matrix from sparse matrix external format

Working with Sparse Matrices

<code>issparse</code>	Determine whether input is sparse
<code>nnz</code>	Number of nonzero matrix elements
<code>nonzeros</code>	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

spparms	Set parameters for sparse matrix routines
spy	Visualize sparsity pattern

Reordering Algorithms

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

Linear Algebra

cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
condst	1-norm condition number estimate
eigs	Find largest eigenvalues and eigenvectors of sparse matrix
luinc	Sparse incomplete LU factorization
normest	2-norm estimate

spaugment	Form least squares augmented system
sprank	Structural rank
svds	Find singular values and vectors

Linear Equations (Iterative Methods)

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

Tree Operations

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

Math Constants

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

Data Analysis

Basic Operations (p. 1-40)	Sums, products, sorting
Descriptive Statistics (p. 1-40)	Statistical summaries of data
Filtering and Convolution (p. 1-41)	Data preprocessing
Interpolation and Regression (p. 1-41)	Data fitting
Fourier Transforms (p. 1-42)	Frequency content of data
Derivatives and Integrals (p. 1-42)	Data rates and accumulations
Time Series Objects (p. 1-43)	Methods for timeseries objects
Time Series Collections (p. 1-46)	Methods for tscollection objects

Basic Operations

cumprod	Cumulative product
cumsum	Cumulative sum
prod	Product of array elements
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
sum	Sum of array elements

Descriptive Statistics

corrcoef	Correlation coefficients
cov	Covariance matrix
max	Largest elements in array
mean	Average or mean value of array
median	Median value of array

<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>interp</code>	N-D data interpolation (table lookup)
<code>mldivide \</code> , <code>mrdivide /</code>	Left or right matrix division
<code>polyfit</code>	Polynomial curve fitting
<code>polyval</code>	Polynomial evaluation

Fourier Transforms

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

Derivatives and Integrals

cumtrapz	Cumulative trapezoidal numerical integration
del2	Discrete Laplacian
diff	Differences and approximate derivatives
gradient	Numerical gradient
polyder	Polynomial derivative

polyint	Integrate polynomial analytically
trapz	Trapezoidal numerical integration

Time Series Objects

General Purpose (p. 1-43)	Combine timeseries objects, query and set timeseries object properties, plot timeseries objects
Data Manipulation (p. 1-44)	Add or delete data, manipulate timeseries objects
Event Data (p. 1-45)	Add or delete events, create new timeseries objects based on event data
Descriptive Statistics (p. 1-45)	Descriptive statistics for timeseries objects

General Purpose

get (timeseries)	Query timeseries object property values
getdatasamplesize	Size of data sample in timeseries object
getqualitydesc	Data quality descriptions
isempty (timeseries)	Determine whether timeseries object is empty
length (timeseries)	Length of time vector
plot (timeseries)	Plot time series
set (timeseries)	Set properties of timeseries object
size (timeseries)	Size of timeseries object
timeseries	Create timeseries 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>getsampleusingtime (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>gettsafterevent</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>gettsbeforeevent</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 <code>timeseries</code> data
<code>std (timeseries)</code>	Standard deviation of <code>timeseries</code> data
<code>sum (timeseries)</code>	Sum of <code>timeseries</code> data
<code>var (timeseries)</code>	Variance of <code>timeseries</code> data

Time Series Collections

General Purpose (p. 1-46)	Query and set <code>tscollection</code> object properties, plot <code>tscollection</code> objects
Data Manipulation (p. 1-47)	Add or delete data, manipulate <code>tscollection</code> objects

General Purpose

<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>getsamplingsusingtime (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. 1-48)	Numeric, character, structures, cell arrays, and data type conversion
Data Type Conversion (p. 1-56)	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
String Functions (p. 1-61)	Create, identify, manipulate, parse, evaluate, and compare strings
Bit-wise Functions (p. 1-64)	Perform set, shift, and, or, compare, etc. on specific bit fields
Logical Functions (p. 1-65)	Evaluate conditions, testing for true or false
Set Functions (p. 1-65)	Find set members, unions, intersections, etc.
Date and Time Functions (p. 1-66)	Obtain information about dates and times
Programming in MATLAB (p. 1-66)	M-files, function/expression evaluation, program control, function handles, object oriented programming, error handling

Data Types

Numeric Types (p. 1-49)	Integer and floating-point data
Characters and Strings (p. 1-50)	Characters and arrays of characters
Structures (p. 1-51)	Data of varying types and sizes stored in fields of a structure
Cell Arrays (p. 1-52)	Data of varying types and sizes stored in cells of array

Function Handles (p. 1-53)	Invoke a function indirectly via handle
MATLAB Classes and Objects (p. 1-53)	MATLAB object-oriented class system
Java Classes and Objects (p. 1-54)	Access Java classes through MATLAB interface
Data Type Identification (p. 1-55)	Determine data type of a variable

Numeric Types

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

<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>	Determine whether 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 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 “String Functions” 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>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>	MATLAB 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

orderfields	Order fields of structure array
rmfield	Remove fields from structure
setfield	Set value of structure array field
struct	Create structure array
struct2cell	Convert structure to cell array
structfun	Apply function to each field of scalar structure

Cell Arrays

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

isscalar	Determine whether input is scalar
isvector	Determine whether input is vector
mat2cell	Divide matrix into cell array of matrices
num2cell	Convert numeric array to cell array
struct2cell	Convert structure to cell array

Function Handles

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

MATLAB Classes and Objects

class	Create object or return class of object
fieldnames	Field names of structure, or public fields of object
inferiorto	Establish inferior class relationship
isa	Determine whether input is object of given class

isobject	Determine whether input is MATLAB OOPs object
loadobj	User-defined extension of load function for user objects
methods	Information on class methods
methodsview	Information on class methods in separate window
saveobj	User-defined extension of save function for user objects
subsasgn	Subscripted assignment for objects
subsindex	Subscripted indexing for objects
subsref	Subscripted reference for objects
substruct	Create structure argument for subsasgn or subsref
superiorto	Establish superior class relationship

Java Classes and Objects

cell	Construct cell array
class	Create object or return class of object
clear	Remove items from workspace, freeing up system memory
depfun	List dependencies of M-file or P-file
exist	Check existence of variable, function, directory, or Java class
fieldnames	Field names of structure, or public fields of object
im2java	Convert image to Java image
import	Add package or class to current Java import list

<code>inmem</code>	Names of M-files, MEX-files, Java classes in memory
<code>isa</code>	Determine whether input is object of given class
<code>isjava</code>	Determine whether input is Java object
<code>javaaddpath</code>	Add entries to dynamic Java class path
<code>javaArray</code>	Construct Java array
<code>javachk</code>	Generate error message based on Java feature support
<code>javaclasspath</code>	Set and get dynamic Java class path
<code>javaMethod</code>	Invoke Java method
<code>javaObject</code>	Construct Java object
<code>javarmpath</code>	Remove entries from dynamic Java class path
<code>methods</code>	Information on class methods
<code>methodsviw</code>	Information on class methods in separate window
<code>usejava</code>	Determine whether Java feature is supported in MATLAB
<code>which</code>	Locate functions and files

Data Type Identification

<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

ischar	Determine whether item is character array
isfield	Determine whether input is structure array field
isfloat	Determine whether input is floating-point array
isinteger	Determine whether input is integer array
isjava	Determine whether input is Java object
islogical	Determine whether input is logical array
isnumeric	Determine whether input is numeric array
isobject	Determine whether input is MATLAB OOPs object
isreal	Determine whether input is real array
isstruct	Determine whether input is structure array

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. 1-57)	Convert numeric to character equivalent
Other Conversions (p. 1-58)	Convert to structure, cell array, function handle, etc.

Numeric

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

String to Numeric

base2dec	Convert base N number string to decimal number
bin2dec	Convert binary number string to decimal number
cast	Cast variable to different data type
hex2dec	Convert hexadecimal number string to decimal number
hex2num	Convert hexadecimal number string to double-precision number
str2double	Convert string to double-precision value
str2num	Convert string to number
unicode2native	Convert Unicode characters to numeric bytes

Numeric to String

cast	Cast variable to different data type
char	Convert to character array (string)

dec2base	Convert decimal to base N number in string
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
int2str	Convert integer to string
mat2str	Convert matrix to string
native2unicode	Convert numeric bytes to Unicode characters
num2str	Convert number to string

Other Conversions

cell2mat	Convert cell array of matrices to single matrix
cell2struct	Convert cell array to structure array
datestr	Convert date and time to string format
func2str	Construct function name string from function handle
logical	Convert numeric values to logical
mat2cell	Divide matrix into cell array of matrices
num2cell	Convert numeric array to cell array
num2hex	Convert singles and doubles to IEEE hexadecimal strings
str2func	Construct function handle from function name string

<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. 1-60)	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><</code>	Less than
<code><=</code>	Less than or equal to
<code>></code>	Greater than
<code>>=</code>	Greater than or equal to
<code>==</code>	Equal to
<code>~=</code>	Not equal to

Logical Operators

See also “Logical Functions” on page 1-65 for functions like `xor`, `all`, `any`, etc.

<code>&&</code>	Logical AND
<code> </code>	Logical OR
<code>&</code>	Logical AND for arrays
<code> </code>	Logical OR for arrays
<code>~</code>	Logical NOT

Special Characters

<code>:</code>	Create vectors, subscript arrays, specify for-loop iterations
<code>()</code>	Pass function arguments, prioritize operators
<code>[]</code>	Construct array, concatenate elements, specify multiple outputs from function
<code>{}</code>	Construct cell array, index into cell array

.	Insert decimal point, define structure field, reference methods of object
.()	Reference dynamic field of structure
..	Reference parent directory
...	Continue statement to next line
,	Separate rows of array, separate function input/output arguments, separate commands
;	Separate columns of array, suppress output from current command
%	Insert comment line into code
%{ %}	Insert block of comments into code
!	Issue command to operating system
' '	Construct character array
@	Construct function handle, reference class directory

String Functions

Description of Strings in MATLAB (p. 1-62)	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 MATLAB string handling

String Creation

blanks	Create string of blank characters
cellstr	Create cell array of strings from character array
char	Convert to character array (string)
sprintf	Write formatted data to string
strcat	Concatenate strings horizontally
strvcat	Concatenate strings vertically

String Identification

class	Create object or return class of object
isa	Determine whether input is object of given class
iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array
isletter	Array elements that are alphabetic letters
isscalar	Determine whether input is scalar
isspace	Array elements that are space characters
isstrprop	Determine whether string is of specified category
isvector	Determine whether input is vector

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

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

String Comparison

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

Bit-wise Functions

bitand	Bitwise AND
bitcmp	Bitwise complement
bitget	Bit at specified position
bitmax	Maximum double-precision floating-point integer
bitor	Bitwise OR
bitset	Set bit at specified position
bitshift	Shift bits specified number of places
bitxor	Bitwise XOR
swapbytes	Swap byte ordering

Logical Functions

all	Determine whether all array elements are nonzero
any	Determine whether any array elements are nonzero
false	Logical 0 (false)
find	Find indices and values of nonzero elements
isa	Determine whether input is object of given class
iskeyword	Determine whether input is MATLAB keyword
isvarname	Determine whether input is valid variable name
logical	Convert numeric values to logical
true	Logical 1 (true)
xor	Logical exclusive-OR

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

Set Functions

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

Date and Time Functions

addtodate	Modify date number by field
calendar	Calendar for specified month
clock	Current time as date vector
cputime	Elapsed CPU time
date	Current date string
datenum	Convert date and time to serial date number
datestr	Convert date and time to string format
datevec	Convert date and time to vector of components
eomday	Last day of month
etime	Time elapsed between date vectors
now	Current date and time
weekday	Day of week

Programming in MATLAB

M-File Functions and Scripts (p. 1-67)	Declare functions, handle arguments, identify dependencies, etc.
Evaluation of Expressions and Functions (p. 1-68)	Evaluate expression in string, apply function to array, run script file, etc.
Timer Functions (p. 1-69)	Schedule execution of MATLAB commands
Variables and Functions in Memory (p. 1-69)	List files in memory, clear M-files in memory, assign to variable in nondefault workspace, refresh caches

Control Flow (p. 1-70)	if-then-else, for loops, switch-case, try-catch
Error Handling (p. 1-71)	Generate warnings and errors, test for and catch errors, retrieve most recent error message
MEX Programming (p. 1-72)	Compile MEX function from C or Fortran code, list MEX-files in memory, debug MEX-files

M-File Functions and Scripts

depdir	List dependent directories of M-file or P-file
depfun	List dependencies of M-file or P-file
echo	Echo M-files during execution
end	Terminate block of code, or indicate last array index
function	Declare M-file function
input	Request user input
inputname	Variable name of function input
mfilename	Name of currently running M-file
namelengthmax	Maximum identifier length
nargchk	Validate number of input arguments
nargin, nargout	Number of function arguments
nargoutchk	Validate number of output arguments
pcode	Create prepared pseudocode file (P-file)
script	Script M-file description
varargin	Variable length input argument list
varargout	Variable length output argument list

Evaluation of Expressions and Functions

arrayfun	Apply function to each element of array
builtin	Execute built-in function from overloaded method
cellfun	Apply function to each cell in cell array
echo	Echo M-files during execution
eval	Execute string containing MATLAB expression
evalc	Evaluate MATLAB expression with capture
evalin	Execute MATLAB expression in specified workspace
feval	Evaluate function
iskeyword	Determine whether input is MATLAB keyword
isvarname	Determine whether input is valid variable name
pause	Halt execution temporarily
run	Run script that is not on current path
script	Script M-file description
structfun	Apply function to each field of scalar structure
symvar	Determine symbolic variables in expression

Timer Functions

delete	Remove files or graphics objects
disp	Text or array
get	Query object properties
isvalid	Determine whether serial port objects are valid
set	Set object properties
start	Start timer(s) running
startat	Start timer(s) running at specified time
stop	Stop timer(s)
timer	Construct timer object
timerfind	Find timer objects
timerfindall	Find timer objects, including invisible objects
wait	Wait until timer stops running

Variables and Functions in Memory

assignin	Assign value to variable in specified workspace
datatipinfo	Produce short description of input variable
genvarname	Construct valid variable name from string
global	Declare global variables
inmem	Names of M-files, MEX-files, Java classes in memory

mislocked	Determine whether M-file or MEX-file cannot be cleared from memory
mlock	Prevent clearing M-file or MEX-file from memory
munlock	Allow clearing M-file or MEX-file from memory
namelengthmax	Maximum identifier length
pack	Consolidate workspace memory
persistent	Define persistent variable
rehash	Refresh function and file system path caches

Control Flow

break	Terminate execution of for or while loop
case	Execute block of code if condition is true
catch	Specify how to respond to error in try statement
continue	Pass control to next iteration of for or while loop
else	Execute statements if condition is false
elseif	Execute statements if additional condition is true
end	Terminate block of code, or indicate last array index
error	Display message and abort function
for	Execute block of code specified number of times

if	Execute statements if condition is true
otherwise	Default part of switch statement
return	Return to invoking function
switch	Switch among several cases, based on expression
try	Attempt to execute block of code, and catch errors
while	Repeatedly execute statements while condition is true

Error Handling

catch	Specify how to respond to error in try statement
error	Display message and abort function
ferror	Query MATLAB about errors in file input or output
intwarning	Control state of integer warnings
lasterr	Last error message
lasterror	Last error message and related information
lastwarn	Last warning message
rethrow	Reissue error
try	Attempt to execute block of code, and catch errors
warning	Warning message

MEX Programming

dbmex	Enable MEX-file debugging
inmem	Names of M-files, MEX-files, Java classes in memory
mex	Compile MEX-function from C or Fortran source code
mexext	MEX-filename extension

File I/O

File Name Construction (p. 1-73)	Get path, directory, filename information; construct filenames
Opening, Loading, Saving Files (p. 1-74)	Open files; transfer data between files and MATLAB workspace
Memory Mapping (p. 1-74)	Access file data via memory map using MATLAB array indexing
Low-Level File I/O (p. 1-74)	Low-level operations that use a file identifier
Text Files (p. 1-75)	Delimited or formatted I/O to text files
XML Documents (p. 1-75)	Documents written in Extensible Markup Language
Spreadsheets (p. 1-76)	Excel and Lotus 1-2-3 files
Scientific Data (p. 1-76)	CDF, FITS, HDF formats
Audio and Audio/Video (p. 1-78)	General audio functions; SparcStation, WAVE, AVI files
Images (p. 1-80)	Graphics files
Internet Exchange (p. 1-80)	URL, FTP, zip, tar, and e-mail

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

File Name Construction

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

tempdir	Name of system's temporary directory
tempname	Unique name for temporary file

Opening, Loading, Saving Files

importdata	Load data from disk file
load	Load workspace variables from disk
open	Open files based on extension
save	Save workspace variables to disk
uiimport	Open Import Wizard to import data
winopen	Open file in appropriate application (Windows)

Memory Mapping

disp	Text or array
get	Query object properties
memmapfile	Construct memmapfile object

Low-Level File I/O

fclose	Close one or more open files
feof	Test for end-of-file
ferror	Query MATLAB about errors in file input or output
fgetl	Read line from file, discarding newline character
fgets	Read line from file, keeping newline character

<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>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 Functions (p. 1-76)	Read and write Microsoft Excel spreadsheet
Lotus 1-2-3 Functions (p. 1-76)	Read and write Lotus WK1 spreadsheet

Microsoft Excel Functions

xlsfinfo	Determine whether file contains Microsoft Excel (.xls) spreadsheet
xlsread	Read Microsoft Excel spreadsheet file (.xls)
xlswrite	Write Microsoft Excel spreadsheet file (.xls)

Lotus 1-2-3 Functions

Scientific Data

Common Data Format (CDF) (p. 1-77)	Work with CDF files
Flexible Image Transport System (p. 1-77)	Work with FITS files
Hierarchical Data Format (HDF) (p. 1-77)	Work with HDF files
Band-Interleaved Data (p. 1-78)	Work with band-interleaved files

Common Data Format (CDF)

<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>

Flexible Image Transport System

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

Hierarchical Data Format (HDF)

<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

hdfread	Read data from HDF4 or HDF-EOS file
hdftool	Browse and import data from HDF4 or HDF-EOS files

Band-Interleaved Data

multibandread	Read band-interleaved data from binary file
multibandwrite	Write band-interleaved data to file

Audio and Audio/Video

General (p. 1-78)	Create audio player object, obtain information about multimedia files, convert to/from audio signal
SPARCstation-Specific Sound Functions (p. 1-79)	Access NeXT/SUN (.au) sound files
Microsoft WAVE Sound Functions (p. 1-79)	Access Microsoft WAVE (.wav) sound files
Audio/Video Interleaved (AVI) Functions (p. 1-79)	Access Audio/Video interleaved (.avi) sound files

General

audioplayer	Create audio player object
audiorecorder	Create audio recorder object
beep	Produce beep sound
lin2mu	Convert linear audio signal to mu-law
mmfileinfo	Information about multimedia file

mu2lin	Convert mu-law audio signal to linear
sound	Convert vector into sound
soundsc	Scale data and play as sound

SPARCstation-Specific Sound Functions

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

Microsoft WAVE Sound Functions

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

Audio/Video Interleaved (AVI) Functions

addframe	Add frame to Audio/Video Interleaved (AVI) file
avifile	Create new Audio/Video Interleaved (AVI) file
aviinfo	Information about Audio/Video Interleaved (AVI) file

aviread	Read Audio/Video Interleaved (AVI) file
close	Remove specified figure
movie2avi	Create Audio/Video Interleaved (AVI) movie from MATLAB movie

Images

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

Internet Exchange

URL, Zip, Tar, E-Mail (p. 1-80)	Send e-mail, read from given URL, extract from tar or zip file, compress and decompress files
FTP Functions (p. 1-81)	Connect to FTP server, download from server, manage FTP files, close server connection

URL, Zip, Tar, E-Mail

gunzip	Uncompress GNU zip files
gzip	Compress files into GNU zip files
sendmail	Send e-mail message to address list
tar	Compress files into tar file
untar	Extract contents of tar file

unzip	Extract contents of zip file
urlread	Read content at URL
urlwrite	Save contents of URL to file
zip	Compress files into zip file

FTP Functions

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

Graphics

Basic Plots and Graphs (p. 1-82)	Linear line plots, log and semilog plots
Plotting Tools (p. 1-83)	GUIs for interacting with plots
Annotating Plots (p. 1-83)	Functions for and properties of titles, axes labels, legends, mathematical symbols
Specialized Plotting (p. 1-84)	Bar graphs, histograms, pie charts, contour plots, function plotters
Bit-Mapped Images (p. 1-88)	Display image object, read and write graphics file, convert to movie frames
Printing (p. 1-88)	Printing and exporting figures to standard formats
Handle Graphics (p. 1-89)	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 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
subplot

Semilogarithmic plots
Create axes in tiled positions

Plotting Tools

figurepalette
pan
plotbrowser
plottools
propertyeditor
zoom

Show or hide figure palette
Pan view of a graph interactively
Show or hide figure plot browser
Show or hide plot tools
Show or hide property editor
Turn zooming on or off or magnify
by factor

Annotating Plots

annotation
clabel
datetick
gtext
legend
line
rectangle
texlabel

title
xlabel, ylabel, zlabel

Create annotation objects
Contour plot elevation labels
Date formatted tick labels
Mouse placement of text in 2-D view
Graph legend for lines and patches
Create line object
Create a 2-D rectangle object
Produce TeX format from character
string
Add title to current axes
Label x -, y -, and z -axis

Specialized Plotting

Area, Bar, and Pie Plots (p. 1-84)	1-D, 2-D, and 3-D graphs and charts
Contour Plots (p. 1-85)	Unfilled and filled contours in 2-D and 3-D
Direction and Velocity Plots (p. 1-85)	Comet, compass, feather and quiver plots
Discrete Data Plots (p. 1-85)	Stair, step, and stem plots
Function Plots (p. 1-85)	Easy-to-use plotting utilities for graphing functions
Histograms (p. 1-86)	Plots for showing distributions of data
Polygons and Surfaces (p. 1-86)	Functions to generate and plot surface patches in two or more dimensions
Scatter/Bubble Plots (p. 1-87)	Plots of point distributions
Animation (p. 1-87)	Functions to create and play movies of plots

Area, Bar, and Pie Plots

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

Contour Plots

<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

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

Scatter/Bubble Plots

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

Animation

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

movie	Play recorded movie frames
noanimate	Change EraseMode of all objects to normal

Bit-Mapped Images

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

Printing

frameedit	Edit print frames for Simulink and Stateflow block diagrams
orient	Hardcopy paper orientation
pagesetupdlg	Page setup dialog box
print, printopt	Print figure or save to file and configure printer defaults
printdlg	Print dialog box

printpreview	Preview figure to print
savesas	Save figure or Simulink block diagram using specified format

Handle Graphics

Finding and Identifying Graphics Objects (p. 1-89)	Find and manipulate graphics objects via their handles
Object Creation Functions (p. 1-90)	Constructors for core graphics objects
Plot Objects (p. 1-91)	Property descriptions for plot objects
Figure Windows (p. 1-92)	Control and save figures
Axes Operations (p. 1-92)	Operate on axes objects
Operating on Object Properties (p. 1-93)	Query, set, and link object properties

Finding and Identifying Graphics Objects

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

gcbo	Handle of object whose callback is executing
gco	Handle of current object
get	Query object properties
ishandle	Is object handle valid
propedit	Open Property Editor
set	Set object properties

Object Creation Functions

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

Figure Windows

clf	Clear current figure window
close	Remove specified figure
closereq	Default figure close request function
drawnow	Complete pending drawing events
gcf	Current figure handle
hglload	Load Handle Graphics object hierarchy from file
hgsave	Save Handle Graphics object hierarchy to file
newplot	Determine where to draw graphics objects
opengl	Control OpenGL rendering
refresh	Redraw current figure
saveas	Save figure or Simulink block diagram using specified format

Axes Operations

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

Operating on Object Properties

get	Query object properties
linkaxes	Synchronize limits of specified 2-D axes
linkprop	Keep same value for corresponding properties
set	Set object properties

3-D Visualization

Surface and Mesh Plots (p. 1-94)	Plot matrices, visualize functions of two variables, specify colormap
View Control (p. 1-96)	Control the camera viewpoint, zooming, rotation, aspect ratio, set axis limits
Lighting (p. 1-98)	Add and control scene lighting
Transparency (p. 1-98)	Specify and control object transparency
Volume Visualization (p. 1-98)	Visualize gridded volume data

Surface and Mesh Plots

Creating Surfaces and Meshes (p. 1-94)	Visualizing gridded and triangulated data as lines and surfaces
Domain Generation (p. 1-95)	Gridding data and creating arrays
Color Operations (p. 1-95)	Specifying, converting, and manipulating color spaces, colormaps, colorbars, and backgrounds
Colormaps (p. 1-96)	Built-in colormaps you can use

Creating Surfaces and Meshes

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

tetramesh	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
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

surfnorm	Compute and display 3-D surface normals
whitebg	Change axes background color

Colormaps

contrast	Grayscale colormap for contrast enhancement
----------	---

View Control

Controlling the Camera Viewpoint (p. 1-96)	Orbiting, dollying, pointing, rotating camera positions and setting fields of view
Setting the Aspect Ratio and Axis Limits (p. 1-97)	Specifying what portions of axes to view and how to scale them
Object Manipulation (p. 1-97)	Panning, rotating, and zooming views
Selecting Region of Interest (p. 1-98)	Interactively identifying rectangular regions

Controlling the Camera Viewpoint

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

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

Setting the Aspect Ratio and Axis Limits

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

Object Manipulation

pan	Pan view of a graph interactively
reset	Reset graphics object properties to their defaults
rotate	Rotate object in specified direction
selectmoveresize	Select, move, resize, or copy axes and uicontrol graphics objects
zoom	Turn zooming on or off or magnify by factor

Selecting Region of Interest

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

Lighting

camlight	Create or move light object in camera coordinates
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

Transparency

alim	Set or query axes alpha limits
alpha	Set transparency properties for objects in current axes
alphamap	Specify figure alphamap (transparency)

Volume Visualization

coneplot	Plot velocity vectors as cones in 3-D vector field
contourslice	Draw contours in volume slice planes
curl	Compute curl and angular velocity of vector field

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

Creating Graphical User Interfaces

Predefined Dialog Boxes (p. 1-100)	Dialog boxes for error, user input, waiting, etc.
Deploying User Interfaces (p. 1-101)	Launch GUIs, create the handles structure
Developing User Interfaces (p. 1-101)	Start GUIDE, manage application data, get user input
User Interface Objects (p. 1-102)	Create GUI components
Finding Objects from Callbacks (p. 1-103)	Find object handles from within callbacks functions
GUI Utility Functions (p. 1-103)	Move objects, wrap text
Controlling Program Execution (p. 1-103)	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 the 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>pagesetupdlg</code>	Page setup dialog box
<code>printdlg</code>	Print dialog box
<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

Deploying User Interfaces

<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

Developing User Interfaces

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

<code>ginput</code>	Graphical input from a 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>uibbuttongroup</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>uitoggletool</code>	Create toggle button on toolbar
<code>uitoolbar</code>	Create toolbar on figure

Finding 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 Utility Functions

<code>align</code>	Align user interface controls (uicontrols) and axes.
<code>selectmoveresize</code>	Select, move, resize, or copy axes and uicontrol graphics objects
<code>textwrap</code>	Wrapped string matrix for given uicontrol
<code>uistack</code>	Reorder visual stacking order of objects

Controlling Program Execution

<code>uiresume</code> , <code>uiwait</code>	Control program execution
---	---------------------------

External Interfaces

Dynamic Link Libraries (p. 1-104)	Access functions stored in external shared library (.dll) files
Java (p. 1-105)	Work with objects constructed from Java API and third-party class packages
Component Object Model and ActiveX (p. 1-106)	Integrate COM components into your application
Dynamic Data Exchange (p. 1-108)	Communicate between applications by establishing a DDE conversation
Web Services (p. 1-109)	Communicate between applications over a network using SOAP and WSDL
Serial Port Devices (p. 1-109)	Read and write to devices connected to your computer's serial port

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

Dynamic Link Libraries

calllib	Call function in external library
libfunctions	Information on functions in external library
libfunctionsview	Create window displaying information on functions in external library
libisloaded	Determine whether external library is loaded
libpointer	Create pointer object for use with external libraries

libstruct	Construct structure as defined in external library
loadlibrary	Load external library into MATLAB
unloadlibrary	Unload external library from memory

Java

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

javarmpath	Remove entries from dynamic Java class path
methods	Information on class methods
methodsview	Information on class methods in separate window
usejava	Determine whether Java feature is supported in MATLAB

Component Object Model and ActiveX

actxcontrol	Create ActiveX control in figure window
actxcontrollist	List all currently installed ActiveX controls
actxcontrolselect	Open GUI to create ActiveX control
actxserver	Create COM Automation server
addproperty	Add custom property to object
class	Create object or return class of object
delete (COM)	Remove COM control or server
deleteproperty	Remove custom property from object
enableservice	Enable DDE or Automation server
eventlisteners	List of events attached to listeners
events	List of events control can trigger
Execute	Execute MATLAB command in 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 object or interface, or display methods
isa	Determine whether input is object of given class
iscom	Is input COM object
isevent	Is input event
isinterface	Is input COM interface
ismethod	Determine whether input is object method
isprop	Determine whether input is object property
load (COM)	Initialize control object from file
MaximizeCommandWindow	Open server window on Windows desktop
methods	Information on class methods
methodsview	Information on class methods in separate window
MinimizeCommandWindow	Minimize size of server window
move	Move or resize control in parent window
propedit (COM)	Open built-in property page for control
PutCharArray	Store character array in server
PutFullMatrix	Store matrix in server

PutWorkspaceData	Store data in server workspace
Quit (COM)	Terminate MATLAB server
registerevent	Register event handler with control's event
release	Release interface
save (COM)	Serialize control object to file
send	Return list of events control can trigger
set	Set object properties
set (COM)	Set object or interface property to specified value
unregisterallevents	Unregister all events for control
unregisterevent	Unregister event handler with control's event

Dynamic Data Exchange

ddeadv	Set up advisory link
ddeexec	Send string for execution
ddeinit	Initiate Dynamic Data Exchange (DDE) conversation
ddepoke	Send data to application
ddereq	Request data from application
ddeterm	Terminate Dynamic Data Exchange (DDE) conversation
ddeunadv	Release advisory link

Web Services

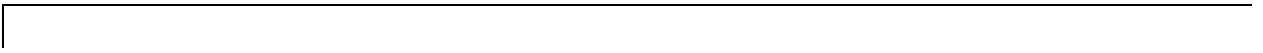
<code>callSoapService</code>	Send SOAP message off to endpoint
<code>createClassFromWsd</code>	Create MATLAB object based on WSDL file
<code>createSoapMessage</code>	Create SOAP message to send to server
<code>parseSoapResponse</code>	Convert response string from SOAP server into MATLAB data types

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>disp (serial)</code>	Serial port object summary information
<code>fclose (serial)</code>	Disconnect serial port object from device
<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</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</code>	Set object properties
<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



Purpose

Consolidate workspace memory

Syntax

```
pack
pack filename
pack('filename')
```

Description

`pack` frees up needed space by reorganizing information so that it only uses the minimum memory required. All variables from your base and global workspaces are preserved. Any persistent variables that are defined at the time are set to their default value (the empty matrix, []).

MATLAB temporarily stores your workspace data in a file called `tp#####.mat` (where ##### is a numeric value) that is located in your temporary directory. (You can use the command `dir(tempdir)` to see the files in this directory).

`pack filename` frees space in memory, temporarily storing workspace data in a file specified by `filename`. This file resides in your current working directory and, unless specified otherwise, has a `.mat` file extension.

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

Remarks

You can only run `pack` from the MATLAB command line.

If you specify a `filename` argument, that file must reside in a directory for which you have write permission.

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

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

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

The `pack` function frees space by

pack

- Saving all variables in the base and global workspaces to a temporary file.
- Clearing all variables and functions from memory.
- Reloading the base and global workspace variables back from the temporary file and then deleting the file.

If you use `pack` and there is still not enough free memory to proceed, you must clear some variables. If you run out of memory often, you can allocate larger matrices earlier in the MATLAB session and use these system-specific tips:

- UNIX: Ask your system manager to increase your swap space.
- Windows: Increase virtual memory using the Windows Control Panel.

To maintain persistent variables when you run `pack`, use `mlock` in the function.

Examples

Change the current directory to one that is writable, run `pack`, and return to the previous directory.

```
cwd = pwd;  
cd(tempdir);  
pack  
cd(cwd)
```

See Also

`clear`, `memory`

Purpose Page setup dialog box

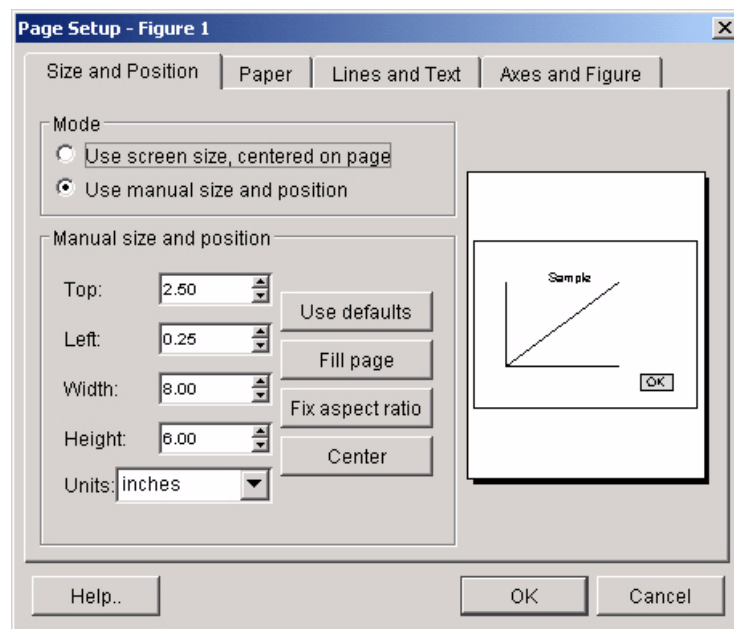
Syntax `dlg = pagesetupdlg(fig)`

Note This function is obsolete. Use `printpreview` instead.

Description `dlg = pagesetupdlg(fig)` creates a dialog box from which a set of pagelayout properties for the figure window, `fig`, can be set.

`pagesetupdlg` implements the Page Setup... option in the **Figure File Menu**.

`pagesetupdlg` supports setting the layout for a single figure. `fig` must be a single figure handle, not a vector of figures or a simulink diagram.



pagesetupdlg


See Also

printdlg, printpreview, printopt

Purpose

Pan view of a graph interactively

GUI Alternatives

Use the Pan tool  on the figure toolbar to enable and disable pan mode on a plot, or select **Pan** from the figure's **Tools** menu. For details, see “Panning — Moving Your View of the Graph” in the MATLAB Graphics documentation.

Syntax

```
pan on
pan xon
pan yon
pan off
pan
pan(figure_handle,...)
h = pan(figure_handle)
```

Description

`pan on` turns on mouse-based panning in the current figure.

`pan xon` turns on panning only in the *x* direction in the current figure.

`pan yon` turns on panning only in the *y* direction in the current figure.

`pan off` turns panning off in the current figure.

`pan` toggles the pan state in the current figure on or off.

`pan(figure_handle,...)` sets the pan state in the specified figure.

`h = pan(figure_handle)` returns the figure's *mode object* for figure *figure_handle* for you to customize the mode's behavior.

Using Pan Mode Objects

You access the following properties of pan mode objects via `get` and modify some of them using `set`:

Enable 'on' | 'off'

Specifies whether this figure mode is currently enabled on the figure.

Motion 'horizontal' | 'vertical' | 'both'

The type of panning enabled for the figure.

FigureHandle <handle>

The associated figure handle. This read-only property cannot be set.

ButtonDownFilter <function_handle>

The application can inhibit the panning operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

```
function [res] = myfunction(obj,event_obj)
% obj          handle to the object that has been clicked on.
% event_obj    handle to event object (empty in this release).
% res          a logical flag to determine whether the pan
%              operation should take place or the 'ButtonDownFcn'
%              property of the object should take precedence.
```

ActionPreCallback <function_handle>

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

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

The event object has the following read-only property:

Axes	The handle of the axes that is being panned.
------	--

ActionPostCallback <function_handle>

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

```
function myfunction(obj,event_obj)
```

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

```
flags = isAllowAxesPan(h,axes)
```

Calling the function `isAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `axes`, as input will return a logical array of the same dimension as the axes handle vector which indicate whether a pan operation is permitted on the axes objects.

```
setAllowAxesPan(h,axes,flag)
```

Calling the function `setAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, will either allow or disallow a pan operation on the axes objects.

```
info = getAxesPanMotion(h,axes)
```

Calling the function `getAxesPanMotion` on the pan object, `h`, with a vector of axes handles, `axes`, as input will return a character cell array of the same dimension as the axes handle vector which indicates the type of pan operation for each axes. Possible values for the type of operation are 'horizontal', 'vertical' or 'both'.

```
setAxesPanMotion(h,axes,style)
```

Calling the function `setAxesPanMotion` on the pan object, `h`, with a vector of axes handles, `axes`, and a character array, `style`, will set the style of panning on each axes.

Examples

Example 1

Simple pan

```
plot(1:10);  
pan on  
% pan on the plot
```

Example 2

Constrain pan to x -axis using set:

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

Example 3

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

```
ax1 = subplot(2,2,1);
plot(1:10);
h = pan;
ax2 = subplot(2,2,2);
plot(rand(3));
setAllowAxesPan(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesPanMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
contour(peaks);
setAxesPanMotion(h,ax4,'vertical');
% pan on the plots.
```

Example 4

Create a `ButtonDown` callback for pan mode objects to trigger. Copy the following code to a new M-file, execute it, and observe panning behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
h = pan;
```



```

set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
    flag = true;
else
    flag = false;
end

```

Example 5

Create callbacks for pre- and post-buttonDown events for pan mode objects to trigger. Copy the following code to a new M-file, execute it, and observe panning behavior:

```

function demo
% Listen to pan events
plot(1:10);
h = pan;
set(h,'ActionPreCallback',@myprecallback);
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');
%
function myprecallback(obj,evd)
disp('A pan is about to occur.');
```

```

%
function mypostcallback(obj,evd)
newLim = get(evd.Axes,'XLim');
msgbox(sprintf('The new X-Limits are [%0.2f %0.2f].',newLim));

```

Remarks

You can create a pan mode object once and use it to customize the behavior of different axes, as example 3 illustrates. You can also change its callback functions on the fly.

pan

When you assign different pan behaviors to different subplot axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse will carry over to the linked axes, regardless of the behavior you previously set for the other axes.

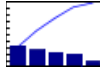
See Also


`zoom`, `linkaxes`, `rotate3d`

“Object Manipulation” on page 1-97 for related functions

Purpose

Pareto chart

**GUI Alternatives**

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
pareto(Y)
pareto(Y, names)
pareto(Y, X)
H = pareto(...)
```

Description

Pareto charts display the values in the vector Y as bars drawn in descending order. Values in Y must be nonnegative and not include NaNs. Only the first 95% of the cumulative distribution is displayed.

`pareto(Y)` labels each bar with its element index in Y and also plots a line displaying the cumulative sum of Y .

`pareto(Y, names)` labels each bar with the associated name in the string matrix or cell array `names`.

`pareto(Y, X)` labels each bar with the associated value from X .

`pareto(ax, ...)` plots a Pareto chart in existing axes `ax` rather than GCA.

`H = pareto(...)` returns a combination of patch and line object handles.

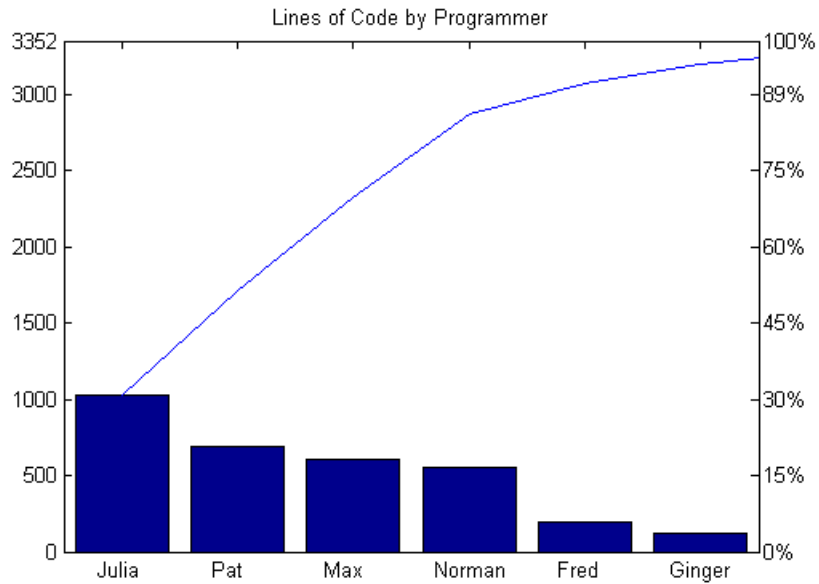
Examples

Example 1:

Examine the cumulative productivity of a group of programmers to see how normal its distribution is:

pareto

```
codelines = [200 120 555 608 1024 101 57 687];
coders =
{'Fred', 'Ginger', 'Norman', 'Max', 'Julia', 'Wally', 'Heidi', 'Pat'};
pareto(codelines, coders)
title('Lines of Code by Programmer')
```



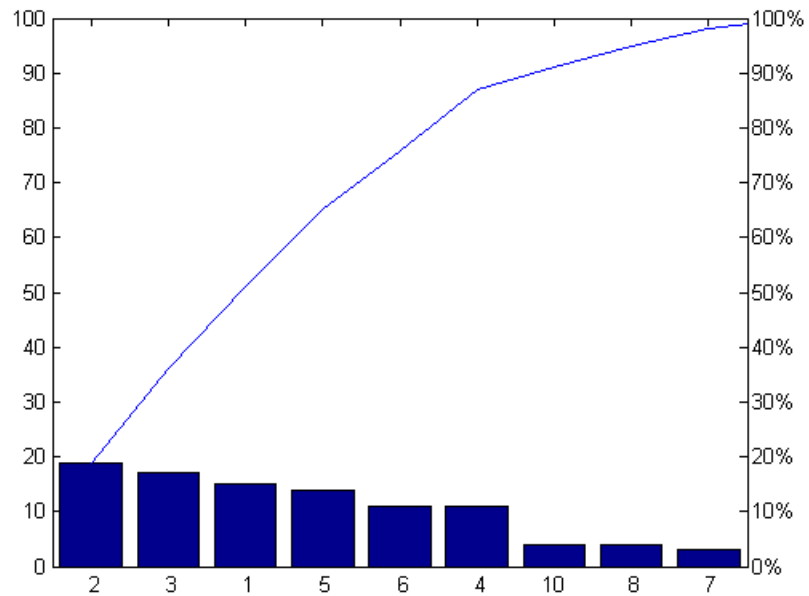
Example 2:

Generate a vector, X , representing diagnostic codes with values from 1 to 10 indicating various faults on devices emerging from a production line:

```
X = min(round(abs(randn(100,1)*4))+1,10);
```

Plot a Pareto chart showing the frequency of failure for each diagnostic code from the most to the least common:

```
pareto(hist(X))
```



Remarks

You can use `pareto` to display the output of `hist`, even for vectors that include negative numbers. Because only the first 95 percent of values are displayed, one or more of the smallest bars may not appear. If you extend the `Xlim` of your chart, you can display all the values, but the new bars will not be labeled.

See Also

`hist`, `bar`

parseSoapResponse

Purpose Convert response string from SOAP server into MATLAB data types

Syntax `parseSoapMessage(response)`

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

Example

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

See Also `callSoapService`, `createClassFromWsd1`, `createSoapMessage`

Purpose Partial pathname description

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

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

Partial pathnames make it easy to find a toolbox or MATLAB relative files on your path, independent of the location where MATLAB is installed.

Many commands accept partial pathnames instead of a full pathname. Some of these commands are

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

Examples The following example uses a partial pathname:

```
what graph2d/@figobj
```

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

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

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

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

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

partialpath

what graph2d/figobj

See Also fileparts, matlabroot, path

Purpose Pascal matrix

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

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

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

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

Examples pascal(4) returns

1	1	1	1
1	2	3	4
1	3	6	10
1	4	10	20

A = pascal(3,2) produces

A =	1	1	1
	-2	-1	0
	1	0	0

See Also chol

patch

Purpose Create patch graphics object

Syntax

```
patch(X,Y,C)
patch(X,Y,Z,C)
patch(FV)
patch(... 'PropertyName',propertyvalue...)
patch('PropertyName',propertyvalue,...)
handle = patch(...)
```

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

`patch(X,Y,C)` adds the filled two-dimensional patch to the current axes. The elements of `X` and `Y` specify the vertices of a polygon. If `X` and `Y` are matrices, MATLAB draws one polygon per column. `C` determines the color of the patch. It can be a single `ColorSpec`, one color per face, or one color per vertex (see “Remarks” on page 2-2219). If `C` is a 1-by-3 vector, it is assumed to be an RGB triplet, specifying a color directly.

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

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

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

`patch('PropertyName',propertyvalue,...)` specifies all properties using property name/property value pairs. This form enables you to omit the color specification because MATLAB uses the default face color and edge color unless you explicitly assign a value to the `FaceColor` and `EdgeColor` properties. This form also allows you to specify the patch using the `Faces` and `Vertices` properties instead of `x`-, `y`-, and

z -coordinates. See the “Examples” on page 2-2222 section for more information.

`handle = patch(...)` returns the handle of the patch object it creates.

Remarks

Unlike high-level area creation functions, such as `fill` or `area`, `patch` does not check the settings of the figure and axes `NextPlot` properties. It simply adds the patch object to the current axes.

If the coordinate data does not define closed polygons, `patch` closes the polygons. The data can define concave or intersecting polygons. However, if the edges of an individual patch face intersect themselves, the resulting face may or may not be completely filled. In that case, it is better to break up the face into smaller polygons.

Specifying Patch Properties

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

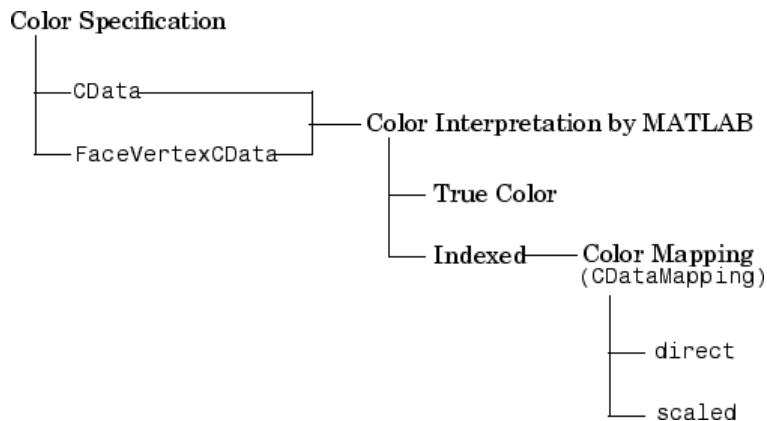
There are two patch properties that specify color:

- `CData` — Use when specifying x -, y -, and z -coordinates (`XData`, `YData`, `ZData`).
- `FaceVertexCData` — Use when specifying vertices and connection matrix (`Vertices` and `Faces`).

The `CData` and `FaceVertexCData` properties accept color data as indexed or true color (RGB) values. See the `CData` and `FaceVertexCData` property descriptions for information on how to specify color.

Indexed color data can represent either direct indices into the colormap or scaled values that map the data linearly to the entire colormap (see the `caxis` function for more information on this scaling). The `CDataMapping` property determines how MATLAB interprets indexed color data.

patch



Color Data Interpretation

You can specify patch colors as

- A single color for all faces
- One color for each face, enabling flat coloring
- One color for each vertex, enabling interpolated coloring

The following tables summarize how MATLAB interprets color data defined by the CData and FaceVertexCData properties.

Interpretation of the CData Property

[X,Y,Z]Data	CData Required for		Results Obtained
Dimensions	Indexed	True Color	
m-by-n	scalar	1-by-1-by-3	Use the single color specified for all patch faces. Edges can be only a single color.
m-by-n	1-by-n (n >= 4)	1-by-n-by-3	Use one color for each patch face. Edges can be only a single color.
m-by-n	m-by-n	m-by-n-3	Assign a color to each vertex. Patch faces can be flat (a single color) or interpolated. Edges can be flat or interpolated.

Interpretation of the FaceVertexCData Property

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

Examples

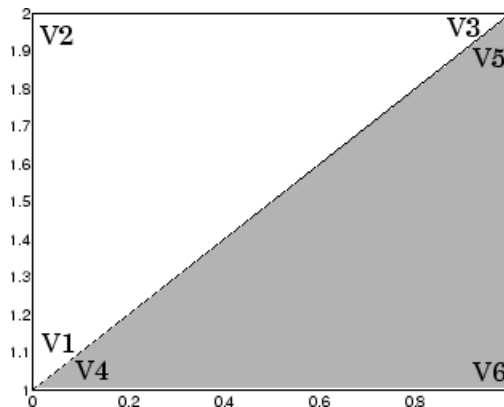
This example creates a patch object using two different methods:

- Specifying x -, y -, and z -coordinates and color data (XData, YData, ZData, and CData properties)
- Specifying vertices, the connection matrix, and color data (Vertices, Faces, FaceVertexCData, and FaceColor properties)

Specifying X, Y, and Z Coordinates

The first approach specifies the coordinates of each vertex. In this example, the coordinate data defines two triangular faces, each having three vertices. Using true color, the top face is set to white and the bottom face to gray.

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



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

Specifying Vertices and Faces

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

```
vert = [0 1 1;0 2 1;1 2 1;1 1 1];
```

There are only two faces, defined by connecting the vertices in the order indicated.

```
fac = [1 2 3;1 3 4];
```

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

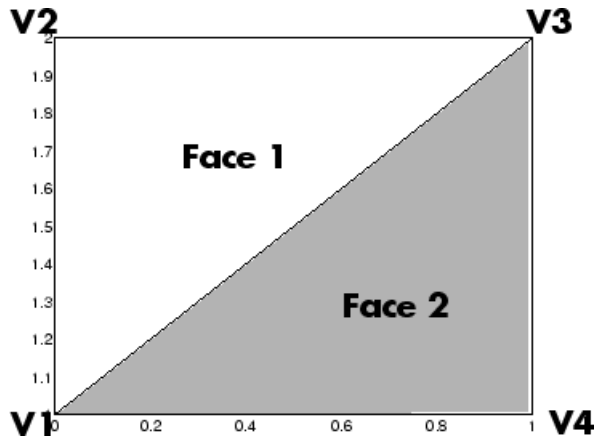
```
tcolor = [1 1 1;.7 .7 .7];
```

With two faces and two colors, MATLAB can color each face with flat shading. This means you must set the FaceColor property to flat, since the faces/vertices technique is available only as a low-level function call (i.e., only by specifying property name/property value pairs).

Create the patch by specifying the Faces, Vertices, and FaceVertexCData properties as well as the FaceColor property.

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

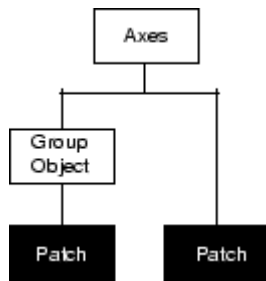
patch



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

MATLAB does not require each face to have the same number of vertices. In cases where they do not, pad the Faces matrix with NaNs. To define a patch with faces that do not close, add one or more NaNs to the row in the Vertices matrix that defines the vertex you do not want connected.

Object Hierarchy



Setting Default Properties

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


```
set(0, 'DefaultPatchPropertyName', PropertyValue...)  
set(gcf, 'DefaultPatchPropertyName', PropertyValue...)  
set(gca, 'DefaultPatchPropertyName', PropertyValue...)
```

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

See Also

`area`, `caxis`, `fill`, `fill3`, `isosurface`, `surface`

“Object Creation Functions” on page 1-90 for related functions

Patch Properties for property descriptions

“Creating 3-D Models with Patches” for examples that use patches

Patch Properties

Modifying Properties

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

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

To change the default values of properties, see “Setting Default Property Values”.

See “Core Graphics Objects” for general information about this type of object.

Patch Property Descriptions

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

AlphaDataMapping
none | {scaled} | direct

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

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

is an array of `uint8` integers, then the indexing begins at 0 (i.e., MATLAB maps a value of 0 to the first alpha value in the `alphamap`).

AmbientStrength

scalar ≥ 0 and ≤ 1

Strength of ambient light. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The `AmbientColor` property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of light objects. See the `DiffuseStrength` and `SpecularStrength` properties.

BackFaceLighting

`unlit` | `lit` | `{reverselit}`

Face lighting control. This property determines how faces are lit when their vertex normals point away from the camera:

- `unlit` — Face is not lit.
- `lit` — Face is lit in normal way.
- `reverselit` — Face is lit as if the vertex pointed towards the camera.

This property is useful for discriminating between the internal and external surfaces of an object. See the `Using MATLAB Graphics` manual for an example.

BeingDeleted

`on` | `{off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in

Patch Properties

the process of being deleted. MATLAB sets the `BeingDeleted` property to on when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to on while the delete function executes, after which the object no longer exists.

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

`BusyAction`

`cancel` | `{queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is off, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

string or function handle

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

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

Set this property to a function handle that references the callback. You can also use a string that is a valid MATLAB expression or the name of an M-file. The expressions execute in the MATLAB workspace.

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

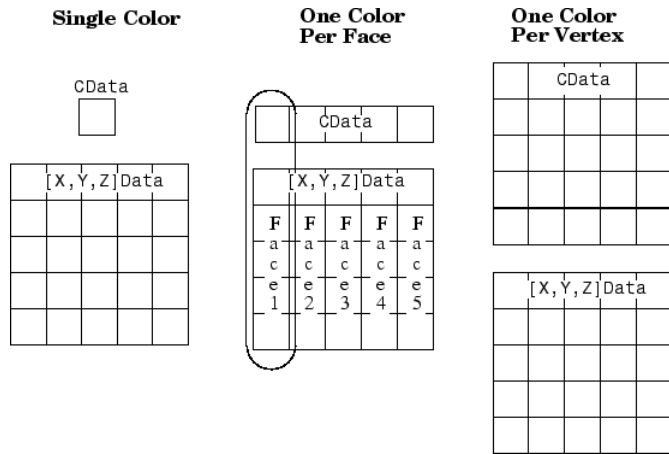
CData

scalar, vector, or matrix

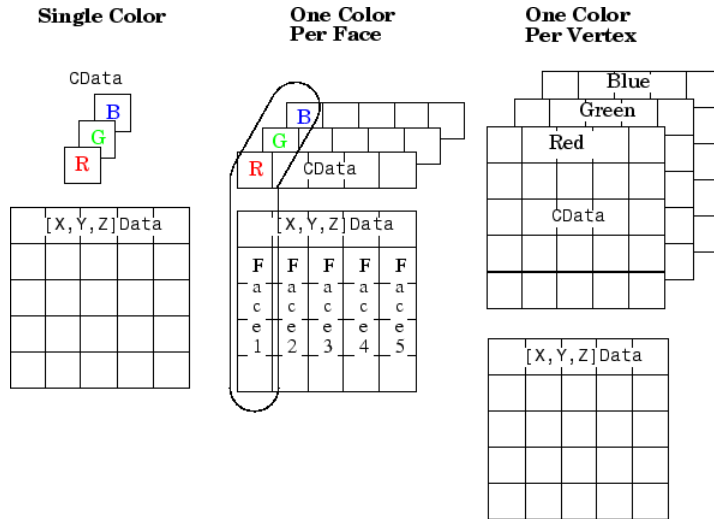
Patch colors. This property specifies the color of the patch. You can specify color for each vertex, each face, or a single color for the entire patch. The way MATLAB interprets CData depends on the type of data supplied. The data can be numeric values that are scaled to map linearly into the current colormap, integer values that are used directly as indices into the current colormap, or arrays of RGB values. RGB values are not mapped into the current colormap, but interpreted as the colors defined. On true color systems, MATLAB uses the actual colors defined by the RGB triples.

The following two diagrams illustrate the dimensions of CData with respect to the coordinate data arrays, XData, YData, and ZData. The first diagram illustrates the use of indexed color.

Patch Properties



The second diagram illustrates the use of true color. True color requires m -by- n -by-3 arrays to define red, green, and blue components for each color.



Note that if CData contains NaNs, MATLAB does not color the faces.

See also the Faces, Vertices, and FaceVertexCData properties for an alternative method of patch definition.

CDataMapping
{scaled} | direct

Direct or scaled color mapping. This property determines how MATLAB interprets indexed color data used to color the patch. (If you use true color specification for CData or FaceVertexCData, this property has no effect.)

- **scaled** — Transform the color data to span the portion of the colormap indicated by the axes CLim property, linearly mapping data values to colors. See the caxis command for more information on this mapping.
- **direct** — Use the color data as indices directly into the colormap. When not scaled, the data are usually integer values ranging from 1 to length(colormap). MATLAB maps values less than 1 to the first color in the colormap, and values greater than length(colormap) to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

Children
matrix of handles

Always the empty matrix; patch objects have no children.

Clipping
{on} | off

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

CreateFcn
string or function handle

Patch Properties

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

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

```
patch(x,y,z,c,'CreateFcn',@myCreateFcn)
```

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

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

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

`DeleteFcn`
string or function handle

Delete patch callback routine. A callback routine that executes when you delete the patch object (e.g., when you issue a `delete` command or clear the axes (`cla`) or figure (`clf`) containing the patch). MATLAB executes the routine before deleting the object’s properties so these values are available to the callback routine.

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

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

DiffuseStrength

scalar ≥ 0 and ≤ 1

Intensity of diffuse light. This property sets the intensity of the diffuse component of the light falling on the patch. Diffuse light comes from light objects in the axes.

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

EdgeAlpha

{scalar = 1} | flat | interp

Transparency of the edges of patch faces. This property can be any of the following:

- **scalar** — A single non-NaN scalar value between 0 and 1 that controls the transparency of all the edges of the object. 1 (the default) means fully opaque and 0 means completely transparent.
- **flat** — The alpha data (FaceVertexAlphaData) of each vertex controls the transparency of the edge that follows it.
- **interp** — Linear interpolation of the alpha data (FaceVertexAlphaData) at each vertex determines the transparency of the edge.

Note that you cannot specify flat or interp EdgeAlpha without first setting FaceVertexAlphaData to a matrix containing one alpha value per face (flat) or one alpha value per vertex (interp).

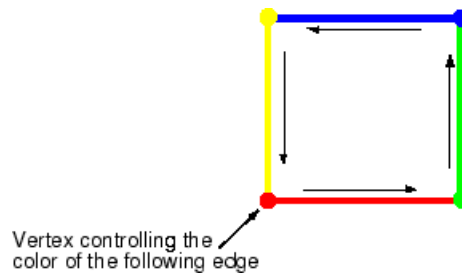
EdgeColor

{ColorSpec} | none | flat | interp

Color of the patch edge. This property determines how MATLAB colors the edges of the individual faces that make up the patch.

Patch Properties

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default edge color is black. See `ColorSpec` for more information on specifying color.
- `none` — Edges are not drawn.
- `flat` — The color of each vertex controls the color of the edge that follows it. This means `flat` edge coloring is dependent on the order in which you specify the vertices:



- `interp` — Linear interpolation of the `CData` or `FaceVertexCData` values at the vertices determines the edge color.

EdgeLighting

{none} | flat | gouraud | phong

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on patch edges. Choices are

- `none` — Lights do not affect the edges of this object.
- `flat` — The effect of light objects is uniform across each edge of the patch.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each edge line and

calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

EraseMode

{normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase patch objects. Alternative erase modes are useful in creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- **none** — Do not erase the patch when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- **xor** — Draw and erase the patch by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the patch does not damage the color of the objects behind it. However, patch color depends on the color of the screen behind it and is correctly colored only when over the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`.
- **background** — Erase the patch by drawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`. This damages objects that are behind the erased patch, but the patch is always properly colored.

Printing with Nonnormal Erase Modes

Patch 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 may mathematically combine layers of colors (e.g., perform an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

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

FaceAlpha

{scalar = 1} | flat | interp

Transparency of the patch face. This property can be any of the following:

- **A scalar** — A single non-NaN value between 0 and 1 that controls the transparency of all the faces of the object. 1 (the default) means fully opaque and 0 means completely transparent (invisible).
- **flat** — The values of the alpha data (`FaceVertexAlphaData`) determine the transparency for each face. The alpha data at the first vertex determines the transparency of the entire face.
- **interp** — Bilinear interpolation of the alpha data (`FaceVertexAlphaData`) at each vertex determines the transparency of each face.

Note that you cannot specify `flat` or `interp` `FaceAlpha` without first setting `FaceVertexAlphaData` to a matrix containing one alpha value per face (`flat`) or one alpha value per vertex (`interp`).

FaceColor

{ColorSpec} | none | flat | interp

Color of the patch face. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See `ColorSpec` for more information on specifying color.
- `none` — Do not draw faces. Note that edges are drawn independently of faces.
- `flat` — The `CData` or `FaceVertexCData` property must contain one value per face and determines the color for each face in the patch. The color data at the first vertex determines the color of the entire face.
- `interp` — Bilinear interpolation of the color at each vertex determines the coloring of each face. The `CData` or `FaceVertexCData` property must contain one value per vertex.

FaceLighting

`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on patch faces. Choices are

- `none` — Lights do not affect the faces of this object.
- `flat` — The effect of light objects is uniform across the faces of the patch. Select this choice to view faceted objects.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

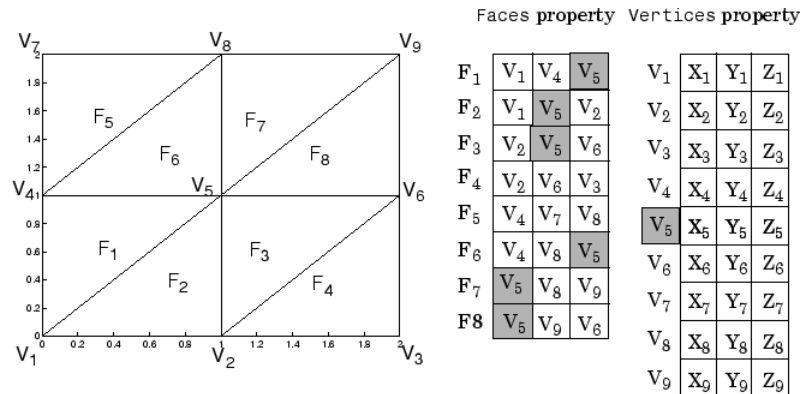
Faces

m-by-n matrix

Patch Properties

Vertex connection defining each face. This property is the connection matrix specifying which vertices in the Vertices property are connected. The Faces matrix defines m faces with up to n vertices each. Each row designates the connections for a single face, and the number of elements in that row that are not NaN defines the number of vertices for that face.

The Faces and Vertices properties provide an alternative way to specify a patch that can be more efficient than using x , y , and z coordinates in most cases. For example, consider the following patch. It is composed of eight triangular faces defined by nine vertices.



The corresponding Faces and Vertices properties are shown to the right of the patch. Note how some faces share vertices with other faces. For example, the fifth vertex (V5) is used six times, once each by faces one, two, and three and six, seven, and eight. Without sharing vertices, this same patch requires 24 vertex definitions.

FaceVertexAlphaData
m-by-1 matrix

Face and vertex transparency data. The `FaceVertexAlphaData` property specifies the transparency of patches that have been defined by the `Faces` and `Vertices` properties. The interpretation of the values specified for `FaceVertexAlphaData` depends on the dimensions of the data.

`FaceVertexAlphaData` can be one of the following:

- A single value, which applies the same transparency to the entire patch. The `FaceAlpha` property must be set to `flat`.
- An m -by-1 matrix (where m is the number of rows in the `Faces` property), which specifies one transparency value per face. The `FaceAlpha` property must be set to `flat`.
- An m -by-1 matrix (where m is the number of rows in the `Vertices` property), which specifies one transparency value per vertex. The `FaceAlpha` property must be set to `interp`.

The `AlphaDataMapping` property determines how MATLAB interprets the `FaceVertexAlphaData` property values.

`FaceVertexCData`
matrix

Face and vertex colors. The `FaceVertexCData` property specifies the color of patches defined by the `Faces` and `Vertices` properties. You must also set the values of the `FaceColor`, `EdgeColor`, `MarkerFaceColor`, or `MarkerEdgeColor` appropriately. The interpretation of the values specified for `FaceVertexCData` depends on the dimensions of the data.

For indexed colors, `FaceVertexCData` can be

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

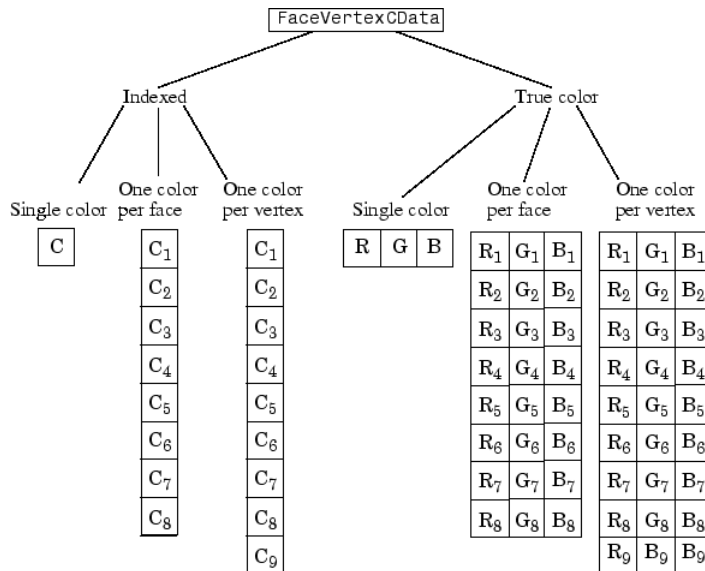
Patch Properties

- An n -by-1 matrix, where n is the number of rows in the Vertices property, which specifies one color per vertex

For true colors, FaceVertexCData can be

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

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



HandleVisibility
{on} | callback | off

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. HandleVisibility is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when HandleVisibility is on.

Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

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

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

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

Patch Properties

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

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

`HitTest`
{on} | off

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

`Interruptible`
{on} | off

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

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

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

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

Edge line width. The width, in points, of the patch edges (1 point = $\frac{1}{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`
character (see table)

Marker symbol. The `Marker` property specifies marks that locate vertices. You can set values for the `Marker` property independently from the `LineStyle` property. The following table lists the available markers.

Marker Specifier	Description
+	Plus sign
o	Circle
*	Asterisk
.	Point
x	Cross
s	Square

Patch Properties

Marker Specifier	Description
d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Five-pointed star (pentagram)
h	Six-pointed star (hexagram)
none	No marker (default)

MarkerEdgeColor

ColorSpec | none | {auto} | flat

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- ColorSpec — Defines the color to use.
- none — Specifies no color, which makes nonfilled markers invisible.
- auto — Sets MarkerEdgeColor to the same color as the EdgeColor property.

MarkerFaceColor

ColorSpec | {none} | auto | flat

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- ColorSpec — Defines the color to use.
- none — Makes the interior of the marker transparent, allowing the background to show through.

- `auto` — Sets the fill color to the axes color, or the figure color, if the axes `Color` property is set to `none`.

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker, in points. The default value for `MarkerSize` is 6 points (1 point = $\frac{1}{72}$ inch). Note that MATLAB draws the point marker at $\frac{1}{3}$ of the specified size.

`NormalMode`
{`auto`} | `manual`

MATLAB generated or user-specified normal vectors. When this property is `auto`, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB sets this property to `manual` and does not generate its own data. See also the `VertexNormals` property.

`Parent`
handle of axes, `hggroup`, or `hgtransform`

Parent of patch object. This property contains the handle of the patch object's parent. The parent of a patch object is the axes, `hggroup`, or `hgtransform` object that contains it.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

`Selected`
`on` | {`off`}

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

Patch Properties

SelectionHighlight
{on} | off

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

- Drawing handles at each vertex for a single-faced patch
- Drawing a dashed bounding box for a multifaced patch

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

SpecularColorReflectance
scalar in the range 0 to 1

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

SpecularExponent
scalar ≥ 1

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

SpecularStrength
scalar ≥ 0 and ≤ 1

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

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

Tag

string

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

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

```
patch(X,Y,'k','Tag','PatchBorder')
```

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

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

Type

string (read only)

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

UIContextMenu

handle of a `uicontextmenu` object

Associate a context menu with the patch. Assign this property the handle of a `uicontextmenu` object created in the same figure as the patch. Use the `uicontextmenu` function to create the

Patch Properties

context menu. MATLAB displays the context menu whenever you right-click over the patch.

UserData
matrix

User-specified data. Any matrix you want to associate with the patch object. MATLAB does not use this data, but you can access it using `set` and `get`.

VertexNormals
matrix

Surface normal vectors. This property contains the vertex normals for the patch. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

Vertices
matrix

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

Visible
{on} | off

Patch object visibility. By default, all patches are visible. When set to `off`, the patch is not visible, but still exists, and you can query and set its properties.

XData
vector or matrix

X-coordinates. The x -coordinates of the patch vertices. If `XData` is a matrix, each column represents the x -coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

YData

vector or matrix

Y-coordinates. The y -coordinates of the patch vertices. If YData is a matrix, each column represents the y -coordinates of a single face of the patch. In this case, XData, YData, and ZData must have the same dimensions.

ZData

vector or matrix

Z-coordinates. The z -coordinates of the patch vertices. If ZData is a matrix, each column represents the z -coordinates of a single face of the patch. In this case, XData, YData, and ZData must have the same dimensions.

See Also

patch

path

Purpose	View or change MATLAB directory search path
GUI Alternatives	As an alternative to the path function, select File > Set Path to use the Set Path dialog box.
Syntax	<pre>path path('newpath') path(path,'newpath') path('newpath',path) p = path(...)</pre>
Description	<p>path displays the current MATLAB search path. The initial search path list is defined by toolbox/local/pathdef.m.</p> <p>path('newpath') changes the search path to newpath, where newpath is a string array of directories.</p> <p>path(path,'newpath') adds the newpath directory to the bottom of the current search path. If newpath is already on the path, then path(path,'newpath') moves newpath to the end of the path.</p> <p>path('newpath',path) adds the newpath directory to the top of the current search path. If newpath is already on the path, then path('newpath', path) moves newpath to the beginning of the path.</p> <p>p = path(...) returns the specified path in string variable p.</p>

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

Examples

Add a new directory to the search path on Windows.

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

Add a new directory to the search path on UNIX.

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

See Also

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

Search Path topic in the MATLAB Desktop Tools and Development Environment documentation

path2rc

Purpose Save current MATLAB search path to pathdef.m file

Syntax path2rc

Description path2rc runs savepath. The savepath function is replacing path2rc. Use savepath instead of path2rc and replace instances of path2rc with savepath.

Purpose	Directories in MATLAB search path
GUI Alternatives	As an alternative to the pathdef function, select File > Set Path to use the Set Path dialog box .
Syntax	pathdef
Description	<p>pathdef returns a string listing of the directories in the MATLAB search path. Use path to view each directory in pathdef.m on a separate line.</p> <p>When you start a new session, MATLAB creates the search path defined in the pathdef.m file located in the MATLAB startup directory. If that directory does not contain a pathdef.m file, MATLAB uses the search path defined in <i>matlabroot/toolbox/local/pathdef.m</i>. It modifies the search path using any path statements contained in the startup.m file.</p> <p>Make changes to the path using the Set Path dialog box and addpath and rmpath. While you can edit pathdef.m directly, use caution so you do not accidentally make MATLAB supplied directories unusable. Use savepath to save pathdef.m, and to use that path in future sessions, specify the MATLAB startup directory as its location.</p>
See Also	<p>addpath, cd, dir, genpath, matlabroot, partialpath, path, pathsep, pathtool, rehash, restoredefaultpath, rmpath, savepath, startup, what</p> <p>MATLAB Desktop Tools and Development Environment documentation topics</p> <ul style="list-style-type: none">• How MATLAB Finds the Search Path, pathdef.m• Saving Settings to the Path• Using the Path in Future Sessions• Recovering from Problems with the Search Path

pathsep

Purpose Path separator for current platform

Syntax `c = pathsep`

Description `c = pathsep` returns the path separator character for this platform. The path separator is the character that separates directories in the string returned by the `matlabpath` function.

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

```
s = matlabpath;
p = 1;

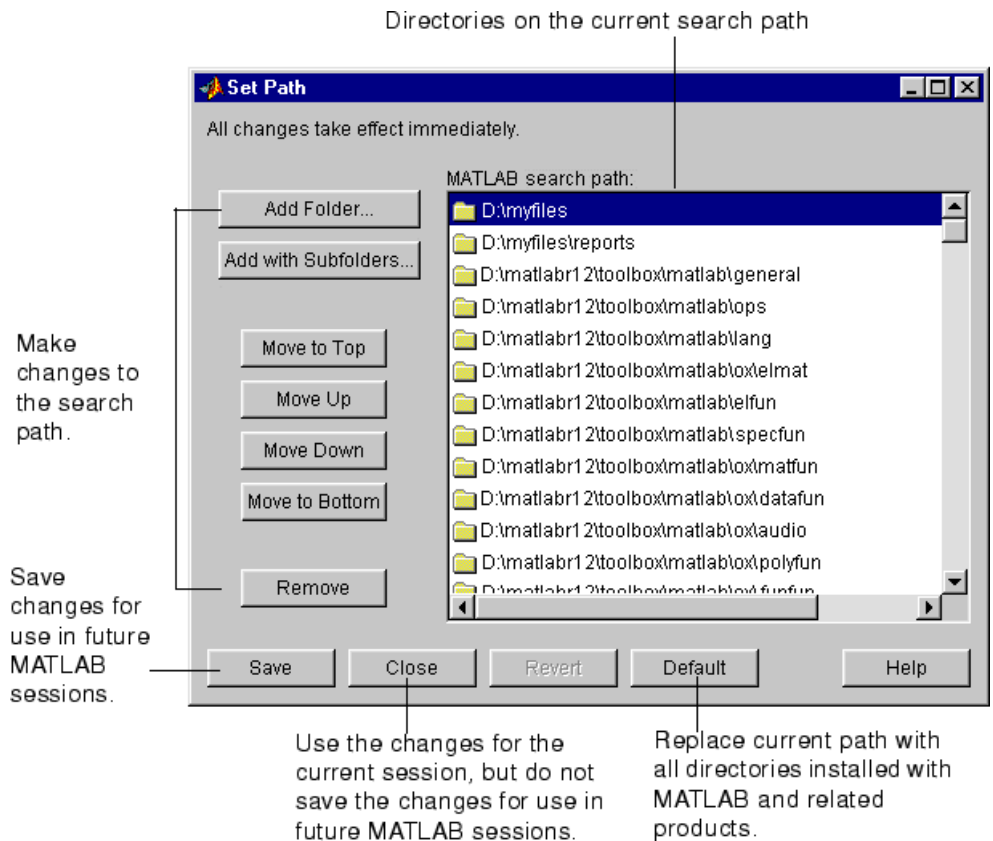
while true
    t = strtok(s(p:end), pathsep);
    disp(sprintf('%s', t))
    p = p + length(t) + 1;
    if isempty(strfind(s(p:end),';')) break, end;
end
```

Here is the output:

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

See Also `filesep`, `fullfile`, `fileparts`

- Purpose** Open Set Path dialog box to view and change MATLAB path
- GUI Alternatives** As an alternative to the pathtool function, select **File > Set Path** in the MATLAB desktop.
- Syntax** `pathtool`
- Description** `pathtool` opens the **Set Path** dialog box, a graphical user interface you use to view and modify the MATLAB search path.



pathtool

See Also

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

Search Path topics, including Setting the Search Path, in the MATLAB Desktop Tools and Development Environment documentation

Purpose Halt execution temporarily

Syntax

```
pause
pause(n)
pause on
pause off
```

Description

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

pause(n) pauses execution for n seconds before continuing, where n can be any nonnegative real number. The resolution of the clock is platform specific. A fractional pause of 0.01 seconds should be supported on most platforms.

Typing pause(inf) puts you into an infinite loop. To return to the MATLAB prompt, type **Ctrl+C**.

pause on allows subsequent pause commands to pause execution.

pause off ensures that any subsequent pause or pause(n) statements do not pause execution. This allows normally interactive scripts to run unattended.

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

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

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

pause

See Also

drawnow

Purpose Set or query plot box aspect ratio

Syntax

```
pbaspect
pbaspect([aspect_ratio])
pbaspect('mode')
pbaspect('auto')
pbaspect('manual')
pbaspect(axes_handle,...)
```

Description The plot box aspect ratio determines the relative size of the x -, y -, and z -axes.

`pbaspect` with no arguments returns the plot box aspect ratio of the current axes.

`pbaspect([aspect_ratio])` sets the plot box aspect ratio in the current axes to the specified value. Specify the aspect ratio as three relative values representing the ratio of the x -, y -, and z -axes size. For example, a value of `[1 1 1]` (the default) means the plot box is a cube (although with stretch-to-fill enabled, it may not appear as a cube). See Remarks.

`pbaspect('mode')` returns the current value of the plot box aspect ratio mode, which can be either `auto` (the default) or `manual`. See Remarks.

`pbaspect('auto')` sets the plot box aspect ratio mode to `auto`.

`pbaspect('manual')` sets the plot box aspect ratio mode to `manual`.

`pbaspect(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. If you do not specify an axes handle, `pbaspect` operates on the current axes.

Remarks `pbaspect` sets or queries values of the axes object `PlotBoxAspectRatio` and `PlotBoxAspectRatioMode` properties.

When the plot box aspect ratio mode is `auto`, MATLAB sets the ratio to `[1 1 1]`, but may change it to accommodate manual settings of the data aspect ratio, camera view angle, or axis limits. See the axes `DataAspectRatio` property for a table listing the interactions between various properties.

Setting a value for the plot box aspect ratio or setting the plot box aspect ratio mode to manual disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the plot box aspect ratio to its current value,

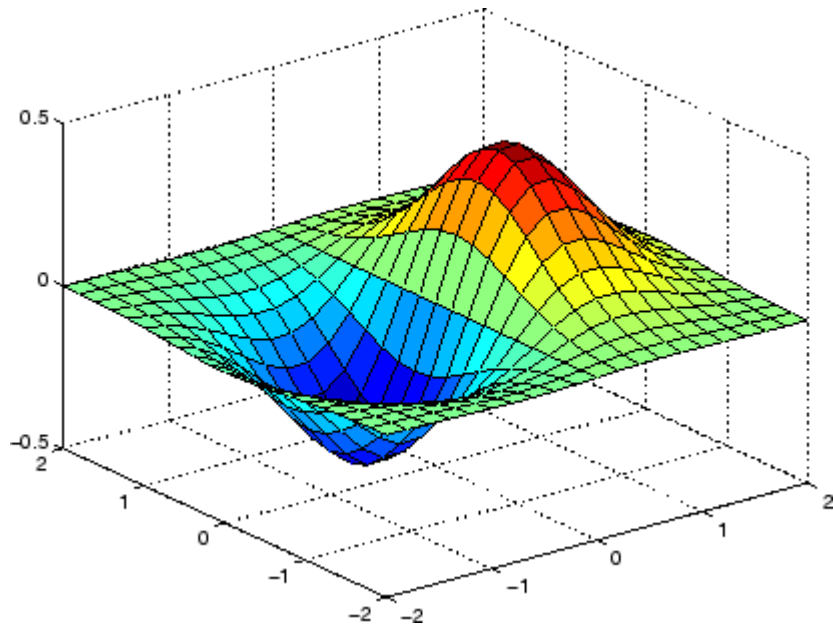
```
pbaspect(pbaspect)
```

can cause a change in the way the graphs look. See the Remarks section of the axes reference description, “Axes Aspect Ratio Properties” in the 3-D Visualization manual, and “Setting Aspect Ratio” in the MATLAB Graphics manual for a discussion of stretch-to-fill.

Examples

The following surface plot of the function $z = xe^{-x^2 - y^2}$ is useful to illustrate the plot box aspect ratio. First plot the function over the range $-2 \leq x \leq 2$, $-2 \leq y \leq 2$,

```
[x,y] = meshgrid([-2:.2:2]);  
z = x.*exp(-x.^2 - y.^2);  
surf(x,y,z)
```



Querying the plot box aspect ratio shows that the plot box is square.

```
pbaspect
ans =
    1    1    1
```

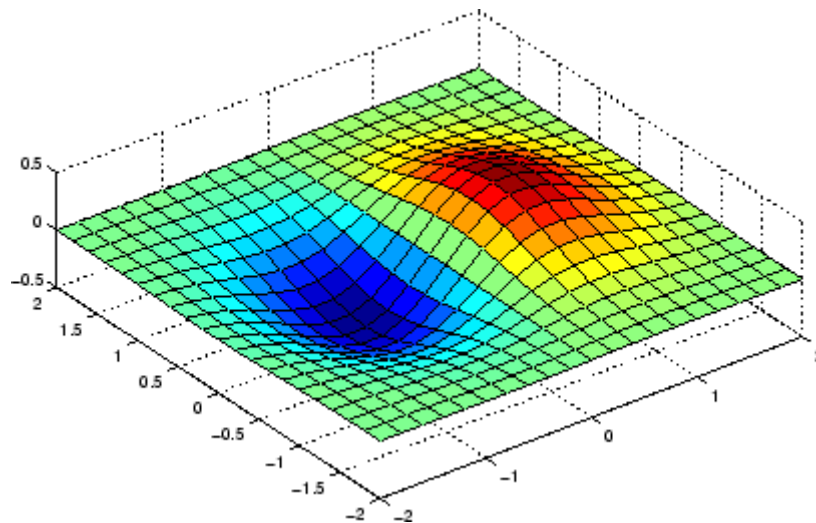
It is also interesting to look at the data aspect ratio selected by MATLAB.

```
daspect
ans =
    4    4    1
```

To illustrate the interaction between the plot box and data aspect ratios, set the data aspect ratio to [1 1 1] and again query the plot box aspect ratio.

```
daspect([1 1 1])
```

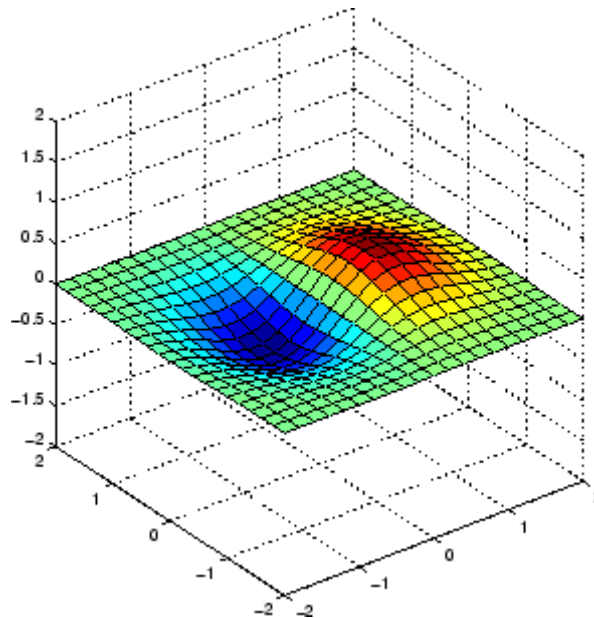
pbaspect



```
pbaspect
ans =
     4     4     1
```

The plot box aspect ratio has changed to accommodate the specified data aspect ratio. Now suppose you want the plot box aspect ratio to be [1 1 1] as well.

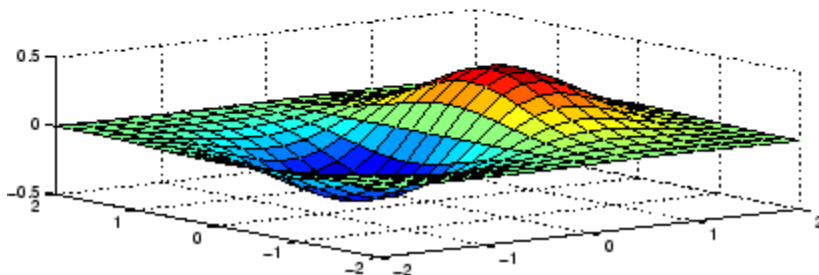
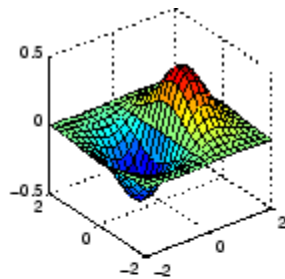
```
pbaspect([1 1 1])
```



Notice how MATLAB changed the axes limits because of the constraints introduced by specifying both the plot box and data aspect ratios.

You can also use `pbaspect` to disable stretch-to-fill. For example, displaying two subplots in one figure can give surface plots a squashed appearance. Disabling stretch-to-fill,

```
upper_plot = subplot(211);
surf(x,y,z)
lower_plot = subplot(212);
surf(x,y,z)
pbaspect(upper_plot, 'manual')
```



See Also

`axis`, `daspect`, `xlim`, `ylim`, `zlim`

The axes properties `DataAspectRatio`, `PlotBoxAspectRatio`, `XLim`, `YLim`, `ZLim`

Setting Aspect Ratio in the MATLAB Graphics manual

Axes Aspect Ratio Properties in the 3-D Visualization manual

Purpose

Preconditioned conjugate gradients method

Syntax

```
x = pcg(A,b)
pcg(A,b,tol)
pcg(A,b,tol,maxit)
pcg(A,b,tol,maxit,M)
pcg(A,b,tol,maxit,M1,M2,x0)
[x,flag] = pcg(A,b,...)
[x,flag,relres] = pcg(A,b,...)
[x,flag,relres,iter] = pcg(A,b,...)
[x,flag,relres,iter,resvec] = pcg(A,b,...)
```

Description

`x = pcg(A,b)` attempts to solve the system of linear equations $A*x=b$ for x . The n -by- n coefficient matrix A must be symmetric and positive definite, and should also be large and sparse. The column vector b must have length n . A can be a function handle `afun` such that `afun(x)` returns $A*x$. See [Function Handles in the MATLAB Programming documentation](#) for more information.

[Parameterizing Functions Called by Function Functions](#), in the [MATLAB Mathematics documentation](#), explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

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

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

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

`pcg(A,b,tol,maxit,M)` and `pcg(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner M or $M = M1*M2$ and effectively solve the system $\text{inv}(M)*A*x = \text{inv}(M)*b$ for x . If M is `[]` then `pcg` applies

no preconditioner. M can be a function handle `mfun` such that `mfun(x)` returns $M \backslash x$.

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

`[x,flag] = pcg(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	<code>pcg</code> converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.
1	<code>pcg</code> iterated <code>maxit</code> times but did not converge.
2	Preconditioner M was ill-conditioned.
3	<code>pcg</code> stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during <code>pcg</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] = pcg(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] = pcg(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] = pcg(A,b,...)` also returns a vector of the residual norms at each iteration including $\text{norm}(b-A*x_0)$.

Examples

Example 1.

```
n1 = 21;  
A = gallery('moler',n1);  
b1 = A*ones(n1,1);  
tol = 1e-6;  
maxit = 15;
```

```
M = diag([10:-1:1 1 1:10]);
[x1,flag1,rr1,iter1,rv1] = pcg(A,b1,tol,maxit,M);
```

Alternatively, you can use the following parameterized matrix-vector product function `afun` in place of the matrix `A`:

```
afun = @(x,n)gallery('moler',n)*x;
n2 = 21;
b2 = afun(ones(n2,1),n2);
[x2,flag2,rr2,iter2,rv2] = pcg(@(x)afun(x,n2),b2,tol,maxit,M);
```

Example 2.

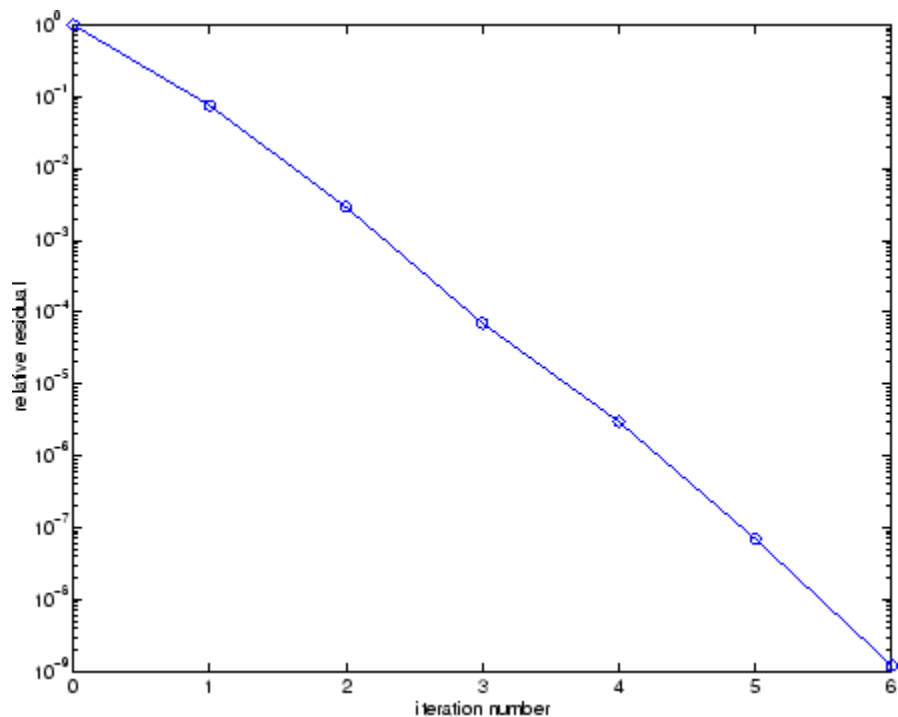
```
A = delsq(numgrid('C',25));
b = ones(length(A),1);
[x,flag] = pcg(A,b)
```

`flag` is 1 because `pcg` does not converge to the default tolerance of $1e-6$ within the default 20 iterations.

```
R = cholinc(A,1e-3);
[x2,flag2,relres2,iter2,resvec2] = pcg(A,b,1e-8,10,R',R)
```

`flag2` is 0 because `pcg` converges to the tolerance of $1.2e-9$ (the value of `relres2`) at the sixth iteration (the value of `iter2`) when preconditioned by the incomplete Cholesky factorization with a drop tolerance of $1e-3$. `resvec2(1) = norm(b)` and `resvec2(7) = norm(b-A*x2)`. You can follow the progress of `pcg` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
```



See Also

bicg, bicgstab, cgs, cholinc, gmres, lsqr, minres, qmr, symmlq
function_handle (@), mldivide (\)

References

[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.

Purpose

Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)

Syntax

```
yi = pchip(x,y,xi)
pp = pchip(x,y)
```

Description

`yi = pchip(x,y,xi)` returns vector `yi` containing elements corresponding to the elements of `xi` and determined by piecewise cubic interpolation within vectors `x` and `y`. The vector `x` specifies the points at which the data `y` is given. If `y` is a matrix, then the interpolation is performed for each column of `y` and `yi` is `length(xi)-by-size(y,2)`.

`pp = pchip(x,y)` returns a piecewise polynomial structure for use by `ppval`. `x` can be a row or column vector. `y` is a row or column vector of the same length as `x`, or a matrix with `length(x)` columns.

`pchip` finds values of an underlying interpolating function $P(x)$ at intermediate points, such that:

- On each subinterval $x_k \leq x \leq x_{k+1}$, $P(x)$ is the cubic Hermite interpolant to the given values and certain slopes at the two endpoints.
- $P(x)$ interpolates y , i.e., $P(x_j) = y_j$, and the first derivative $P'(x)$ is continuous. $P''(x)$ is probably not continuous; there may be jumps at the x_j .
- The slopes at the x_j are chosen in such a way that $P(x)$ preserves the shape of the data and respects monotonicity. This means that, on intervals where the data are monotonic, so is $P(x)$; at points where the data has a local extremum, so does $P(x)$.

Note If y is a matrix, $P(x)$ satisfies the above for each column of y .

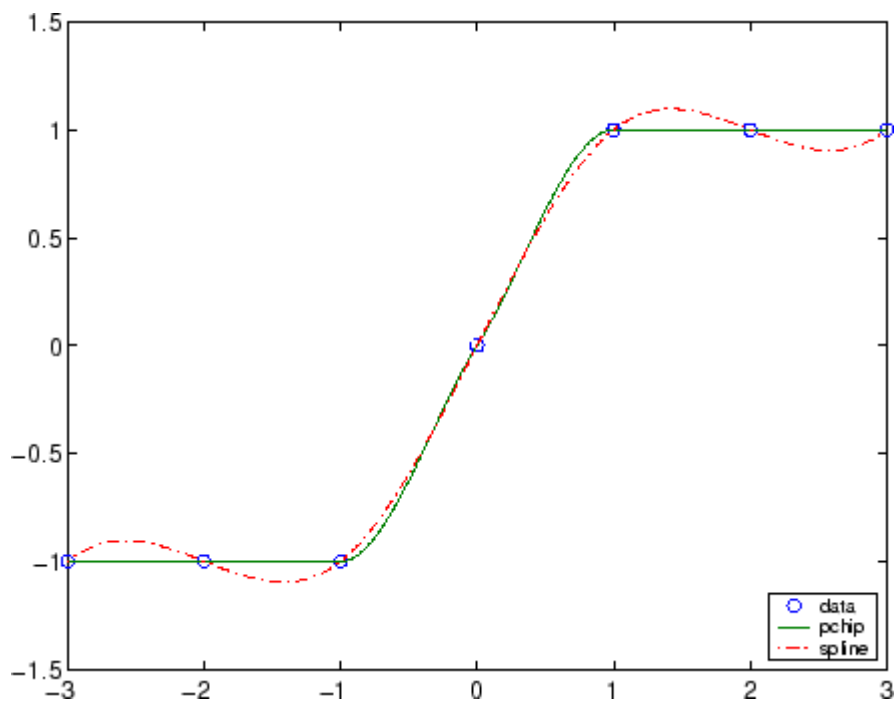
Remarks

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

- spline produces a smoother result, i.e. $S''(x)$ is continuous.
- spline produces a more accurate result if the data consists of values of a smooth function.
- pchip has no overshoots and less oscillation if the data are not smooth.
- pchip is less expensive to set up.
- The two are equally expensive to evaluate.

Examples

```
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
t = -3:.01:3;
p = pchip(x,y,t);
s = spline(x,y,t);
plot(x,y,'o',t,p,'-',t,s,'-.-')
legend('data','pchip','spline',4)
```



See Also

`interp1`, `spline`, `ppval`

References

- [1] Fritsch, F. N. and R. E. Carlson, Monotone Piecewise Cubic Interpolation, *SIAM J. Numerical Analysis*, Vol. 17, 1980, pp.238-246.
- [2] Kahaner, David, Cleve Moler, Stephen Nash, *Numerical Methods and Software*, Prentice Hall, 1988.

pcode

Purpose Create prepared pseudocode file (P-file)

Syntax

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

Description

`pcode fun` parses the M-file `fun.m` into the P-file `fun.p` and puts it into the current directory. The original M-file can be anywhere on the search path.

`pcode *.m` creates P-files for all the M-files in the current directory.


`pcode fun1 fun2 ...` creates P-files for the listed functions.

`pcode... -inplace` creates P-files in the same directory as the M-files. An error occurs if the files can't be created.

Purpose Pseudocolor (checkerboard) plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

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

Description

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

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

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

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

`pcolor(X,Y,C)` draws a pseudocolor plot of the elements of C at the locations specified by X and Y . The plot is a logically rectangular, two-dimensional grid with vertices at the points $[X(i,j), Y(i,j)]$. X and Y are vectors or matrices that specify the spacing of the grid lines. If

X and Y are vectors, X corresponds to the columns of C and Y corresponds to the rows. If X and Y are matrices, they must be the same size as C.

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

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

Remarks

A pseudocolor plot is a flat surface plot viewed from above. `pcolor(X,Y,C)` is the same as viewing `surf(X,Y,zeros(size(X)),C)` using `view([0 90])`.

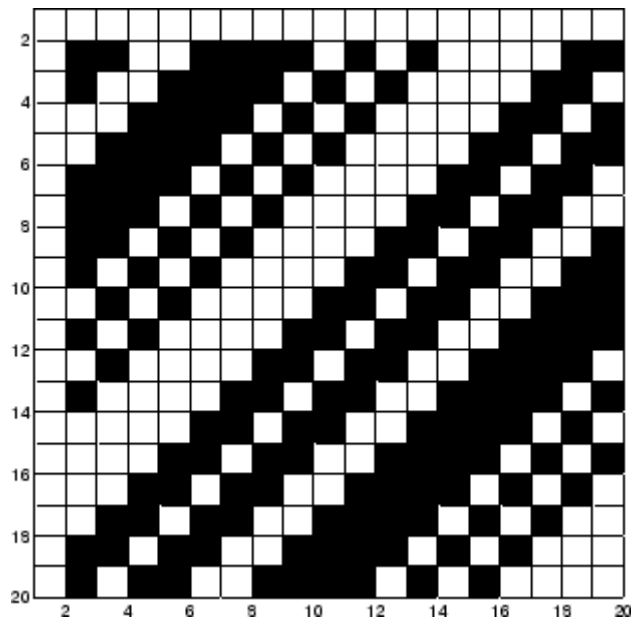
When you use shading `faceted` or shading `flat`, the constant color of each cell is the color associated with the corner having the smallest *x-y* coordinates. Therefore, `C(i,j)` determines the color of the cell in the *i*th row and *j*th column. The last row and column of C are not used.

When you use shading `interp`, each cell's color results from a bilinear interpolation of the colors at its four vertices, and all elements of C are used.

Examples

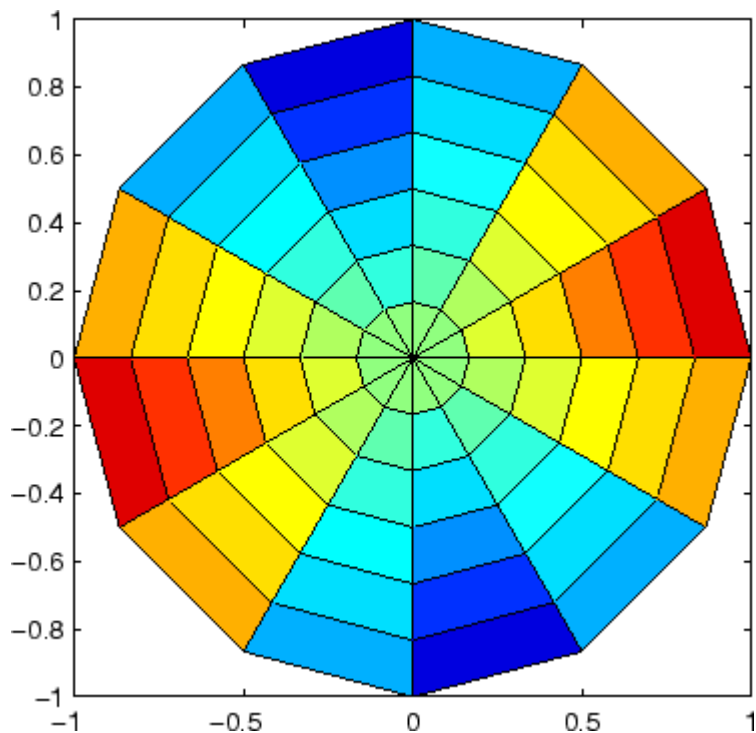
A Hadamard matrix has elements that are +1 and -1. A colormap with only two entries is appropriate when displaying a pseudocolor plot of this matrix.

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



A simple color wheel illustrates a polar coordinate system.

```
n = 6;  
r = (0:n)'/n;  
theta = pi*( n:n)/n;  
X = r*cos(theta);  
Y = r*sin(theta);  
C = r*cos(2*theta);  
pcolor(X,Y,C)  
axis equal tight
```



Algorithm

The number of vertex colors for `pcolor(C)` is the same as the number of cells for `image(C)`. `pcolor` differs from `image` in that `pcolor(C)` specifies the colors of vertices, which are scaled to fit the colormap; changing the axes `clim` property changes this color mapping. `image(C)` specifies the colors of cells and directly indexes into the colormap without scaling. Additionally, `pcolor(X,Y,C)` can produce parametric grids, which is not possible with `image`.

See Also

`caxis`, `image`, `mesh`, `shading`, `surf`, `view`

Purpose Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D

Syntax

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
```

Arguments

<code>m</code>	A parameter corresponding to the symmetry of the problem. <code>m</code> can be <code>slab = 0</code> , <code>cylindrical = 1</code> , or <code>spherical = 2</code> .
<code>pdefun</code>	A handle to a function that defines the components of the PDE.
<code>icfun</code>	A handle to a function that defines the initial conditions.
<code>bcfun</code>	A handle to a function that defines the boundary conditions.
<code>xmesh</code>	A vector [<code>x0</code> , <code>x1</code> , ..., <code>xn</code>] specifying the points at which a numerical solution is requested for every value in <code>tspan</code> . The elements of <code>xmesh</code> must satisfy $x_0 < x_1 < \dots < x_n$. The length of <code>xmesh</code> must be ≥ 3 .
<code>tspan</code>	A vector [<code>t0</code> , <code>t1</code> , ..., <code>tf</code>] specifying the points at which a solution is requested for every value in <code>xmesh</code> . The elements of <code>tspan</code> must satisfy $t_0 < t_1 < \dots < t_f$. The length of <code>tspan</code> must be ≥ 3 .
<code>options</code>	Some options of the underlying ODE solver are available in <code>pdepe</code> : <code>RelTol</code> , <code>AbsTol</code> , <code>NormControl</code> , <code>InitialStep</code> , and <code>MaxStep</code> . In most cases, default values for these options provide satisfactory solutions. See <code>odeset</code> for details.

Description `sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)` solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable x and time t . `pdefun`, `icfun`, and `bcfun` are function handles. See [Function Handles in the MATLAB](#)

Programming documentation for more information. The ordinary differential equations (ODEs) resulting from discretization in space are integrated to obtain approximate solutions at times specified in `tspan`. The `pdepe` function returns values of the solution on a mesh provided in `xmesh`.

Parameterizing Functions Called by Function Functions, in the MATLAB mathematics documentation, explains how to provide additional parameters to the functions `pdefun`, `icfun`, or `bcfun`, if necessary.

`pdepe` solves PDEs of the form:

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right) \frac{\partial u}{\partial t} = x^{-m} \frac{\partial}{\partial x} \left(x^m f\left(x, t, u, \frac{\partial u}{\partial x}\right) \right) + s\left(x, t, u, \frac{\partial u}{\partial x}\right) \quad (2-2)$$

The PDEs hold for $t_0 \leq t \leq t_f$ and $a \leq x \leq b$. The interval $[a, b]$ must be finite. m can be 0, 1, or 2, corresponding to slab, cylindrical, or spherical symmetry, respectively. If $m > 0$, then a must be $>= 0$.

In Equation 2-2, $f(x, t, u, \partial u/\partial x)$ is a flux term and $s(x, t, u, \partial u/\partial x)$ is a source term. The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix $c(x, t, u, \partial u/\partial x)$. The diagonal elements of this matrix are either identically zero or positive. An element that is identically zero corresponds to an elliptic equation and otherwise to a parabolic equation. There must be at least one parabolic equation. An element of c that corresponds to a parabolic equation can vanish at isolated values of x if those values of x are mesh points. Discontinuities in c and/or s due to material interfaces are permitted provided that a mesh point is placed at each interface.

For $t = t_0$ and all x , the solution components satisfy initial conditions of the form

$$u(x, t_0) = u_0(x) \quad (2-3)$$

For all t and either $x = a$ or $x = b$, the solution components satisfy a boundary condition of the form

$$p(x, t, u) + q(x, t) f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0 \quad (2-4)$$

Elements of \mathbf{q} are either identically zero or never zero. Note that the boundary conditions are expressed in terms of the flux \mathbf{f} rather than $\partial u / \partial x$. Also, of the two coefficients, only \mathbf{p} can depend on \mathbf{u} .

In the call `sol = pdepe(m, pdefun, icfun, bcfun, xmesh, tspan)`:

- `m` corresponds to m .
- `xmesh(1)` and `xmesh(end)` correspond to \mathbf{a} and \mathbf{b} .
- `tspan(1)` and `tspan(end)` correspond to t_0 and t_f .
- `pdefun` computes the terms \mathbf{c} , \mathbf{f} , and \mathbf{s} (Equation 2-2). It has the form

$$[\mathbf{c}, \mathbf{f}, \mathbf{s}] = \text{pdefun}(x, t, u, \text{dudx})$$

The input arguments are scalars x and t and vectors u and dudx that approximate the solution \mathbf{u} and its partial derivative with respect to \mathbf{x} , respectively. \mathbf{c} , \mathbf{f} , and \mathbf{s} are column vectors. \mathbf{c} stores the diagonal elements of the matrix \mathbf{c} (Equation 2-2).

- `icfun` evaluates the initial conditions. It has the form

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

When called with an argument x , `icfun` evaluates and returns the initial values of the solution components at x in the column vector u .

- `bcfun` evaluates the terms \mathbf{p} and \mathbf{q} of the boundary conditions (Equation 2-4). It has the form

$$[\mathbf{p}_l, \mathbf{q}_l, \mathbf{p}_r, \mathbf{q}_r] = \text{bcfun}(x_l, u_l, x_r, u_r, t)$$

u_l is the approximate solution at the left boundary $x_l = \mathbf{a}$ and u_r is the approximate solution at the right boundary $x_r = \mathbf{b}$. \mathbf{p}_l and \mathbf{q}_l are column vectors corresponding to \mathbf{p} and \mathbf{q} evaluated at x_l , similarly \mathbf{p}_r and \mathbf{q}_r correspond to x_r . When $m > 0$ and $\mathbf{a} = \mathbf{0}$, boundedness

of the solution near $x = 0$ requires that the flux f vanish at $a = 0$. pdepe imposes this boundary condition automatically and it ignores values returned in p1 and q1.

pdepe returns the solution as a multidimensional array sol. $u_i = ui = sol(:, :, i)$ is an approximation to the i th component of the solution vector u . The element $ui(j, k) = sol(j, k, i)$ approximates u_i at $(t, x) = (tspan(j), xmesh(k))$.

$ui = sol(j, :, i)$ approximates component i of the solution at time $tspan(j)$ and mesh points $xmesh(:)$. Use pdeval to compute the approximation and its partial derivative $\partial u_i / \partial x$ at points not included in $xmesh$. See pdeval for details.

`sol = pdepe(m, pdefun, icfun, bcfun, xmesh, tspan, options)` solves as above with default integration parameters replaced by values in options, an argument created with the odeset function. Only some of the options of the underlying ODE solver are available in pdepe: RelTol, AbsTol, NormControl, InitialStep, and MaxStep. The defaults obtained by leaving off the input argument options will generally be satisfactory. See odeset for details.

Remarks

- The arrays `xmesh` and `tspan` play different roles in pdepe.
 - tspan** The pdepe function performs the time integration with an ODE solver that selects both the time step and formula dynamically. The elements of `tspan` merely specify where you want answers and the cost depends weakly on the length of `tspan`.
 - xmesh** Second order approximations to the solution are made on the mesh specified in `xmesh`. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. pdepe does *not* select the mesh in x automatically. You must provide an appropriate fixed mesh in `xmesh`. The cost depends strongly on the length of `xmesh`. When $m > 0$, it is not necessary to use a fine mesh near $x = 0$ to account for the coordinate singularity.
- The time integration is done with ode15s. pdepe exploits the capabilities of ode15s for solving the differential-algebraic equations

that arise when Equation 2-2 contains elliptic equations, and for handling Jacobians with a specified sparsity pattern.

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

No adjustment is necessary for elements of the initial conditions vector that correspond to parabolic equations.

Examples

Example 1. This example illustrates the straightforward formulation, computation, and plotting of the solution of a single PDE.

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

This equation holds on an interval $0 \leq x \leq 1$ for times $t \geq 0$.

The PDE satisfies the initial condition

$$u(x, 0) = \sin \pi x$$

and boundary conditions

$$u(0, t) \equiv 0$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1, t) = 0$$

It is convenient to use subfunctions to place all the functions required by pdepe in a single M-file.

```
function pdex1
```

```
m = 0;
x = linspace(0,1,20);
t = linspace(0,2,5);

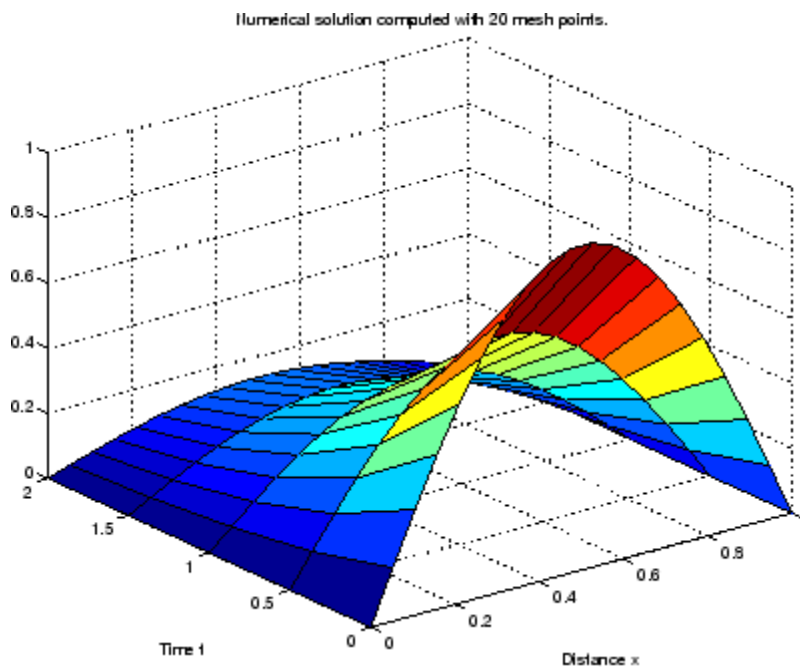
sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
% Extract the first solution component as u.
u = sol(:,:,1);

% A surface plot is often a good way to study a solution.
surf(x,t,u)
title('Numerical solution computed with 20 mesh points.')
xlabel('Distance x')
ylabel('Time t')

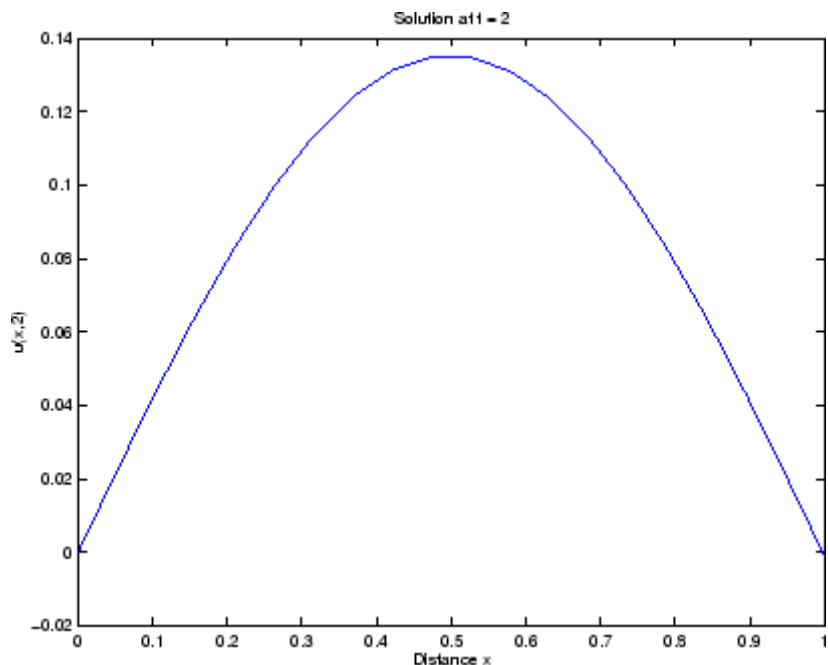
% A solution profile can also be illuminating.
figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
ylabel('u(x,2)')
% -----
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
s = 0;
% -----
function u0 = pdex1ic(x)
u0 = sin(pi*x);
% -----
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
```

In this example, the PDE, initial condition, and boundary conditions are coded in subfunctions `pdex1pde`, `pdex1ic`, and `pdex1bc`.

The surface plot shows the behavior of the solution.



The following plot shows the solution profile at the final value of t (i.e., $t = 2$).



Example 2. This example illustrates the solution of a system of PDEs. The problem has boundary layers at both ends of the interval. The solution changes rapidly for small t .

The PDEs are

$$\frac{\partial u_1}{\partial t} = 0.024 \frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2)$$

$$\frac{\partial u_2}{\partial t} = 0.170 \frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2)$$

where $F(y) = \exp(5.73y) - \exp(-11.46y)$.

This equation holds on an interval $0 \leq x \leq 1$ for times $t \geq 0$.

The PDE satisfies the initial conditions

$$u_1(x, 0) \equiv 1$$

$$u_2(x, 0) \equiv 0$$

and boundary conditions

$$\frac{\partial u_1}{\partial x}(0, t) \equiv 0$$

$$u_2(0, t) \equiv 0$$

$$u_1(1, t) \equiv 1$$

$$\frac{\partial u_2}{\partial x}(1, t) \equiv 0$$

In the form expected by pdepe, the equations are

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

The boundary conditions on the partial derivatives of \mathbf{u} have to be written in terms of the flux. In the form expected by pdepe, the left boundary condition is

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

and the right boundary condition is

$$\begin{bmatrix} u_1 - 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} .* \begin{bmatrix} 0.024(\partial u_1 / \partial x) \\ 0.170(\partial u_2 / \partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The solution changes rapidly for small t . The program selects the step size in time to resolve this sharp change, but to see this behavior in the plots, the example must select the output times accordingly. There are boundary layers in the solution at both ends of [0,1], so the example places mesh points near 0 and 1 to resolve these sharp changes. Often some experimentation is needed to select a mesh that reveals the behavior of the solution.

```
function pdex4
m = 0;
x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];

sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
u1 = sol(:,:,1);
u2 = sol(:,:,2);

figure
surf(x,t,u1)
title('u1(x,t)')
xlabel('Distance x')
ylabel('Time t')

figure
surf(x,t,u2)
title('u2(x,t)')
xlabel('Distance x')
ylabel('Time t')
% -----
function [c,f,s] = pdex4pde(x,t,u,DuDx)
c = [1; 1];
f = [0.024; 0.17] .* DuDx;
y = u(1) - u(2);
```

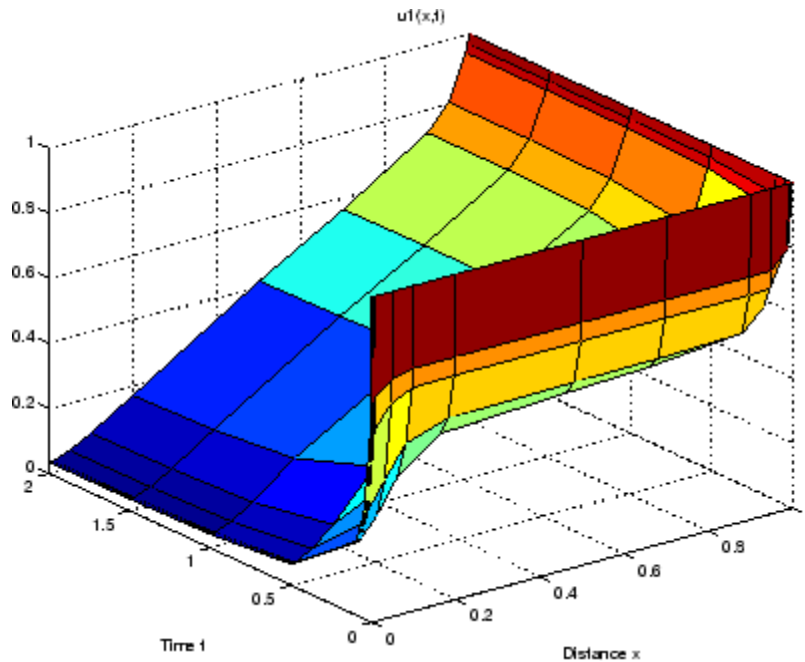
```

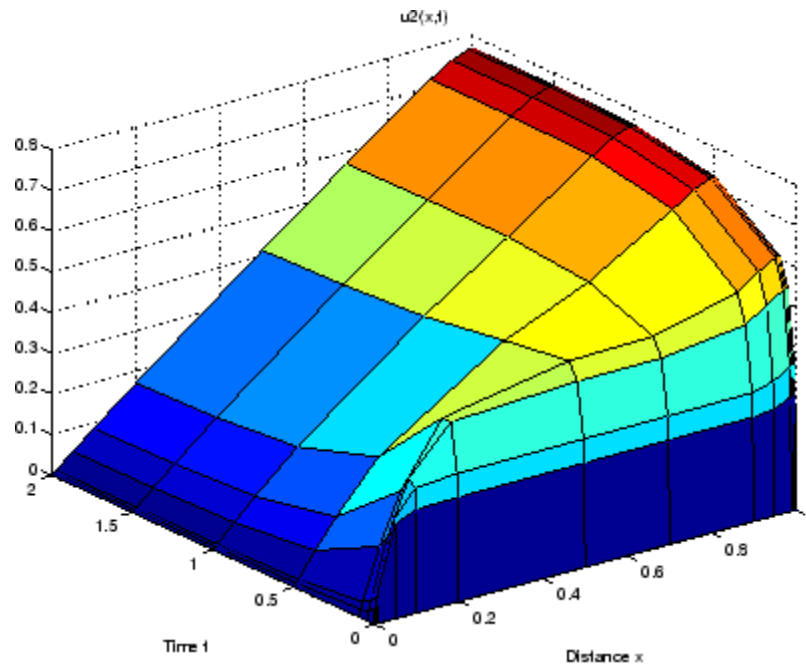
F = exp(5.73*y)-exp(-11.47*y);
s = [-F; F];
% -----
function u0 = pdex4ic(x);
u0 = [1; 0];
% -----
function [pl,q1,pr,qr] = pdex4bc(xl,u1,xr,ur,t)
pl = [0; u1(2)];
q1 = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];

```

In this example, the PDEs, initial conditions, and boundary conditions are coded in subfunctions `pdex4pde`, `pdex4ic`, and `pdex4bc`.

The surface plots show the behavior of the solution components.





See Also

`function_handle (@)`, `pdeval`, `ode15s`, `odeset`, `odeget`

References

[1] Skeel, R. D. and M. Berzins, A Method for the Spatial Discretization of Parabolic Equations in One Space Variable, *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, 1990, pp.1-32.

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

Description `[uout,duoutdx] = pdeval(m,x,ui,xout)` approximates the solution u_i and its partial derivative $\frac{\partial u_i}{\partial x}$ at points from the interval `[x0,xn]`. The pdeval function returns the computed values in `uout` and `duoutdx`, respectively.

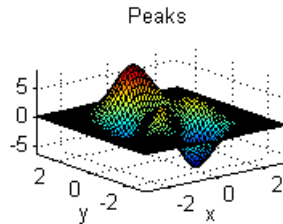
Note pdeval evaluates the partial derivative $\frac{\partial u_i}{\partial x}$ rather than the flux f . Although the flux is continuous, the partial derivative may have a jump at a material interface.

See Also pdepe

peaks

Purpose

Example function of two variables



Syntax

```
Z = peaks;  
Z = peaks(n);  
Z = peaks(V);  
Z = peaks(X,Y);
```

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

```
X,Y,Z] = peaks;  
[X,Y,Z] = peaks(n);  
[X,Y,Z] = peaks(V);
```

Description

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

`Z = peaks;` returns a 49-by-49 matrix.

`Z = peaks(n);` returns an n-by-n matrix.

`Z = peaks(V);` returns an n-by-n matrix, where `n = length(V)`.

`Z = peaks(X,Y);` evaluates `peaks` at the given `X` and `Y` (which must be the same size) and returns a matrix the same size.

`peaks(...)` (with no output argument) plots the peaks function with `surf`.

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

```
[X,Y] = meshgrid(V,V)
```

where `V` is a given vector, or `V` is a vector of length `n` with elements equally spaced from `-3` to `3`. If no input argument is given, the default `n` is `49`.

See Also

`meshgrid`, `surf`

perl

Purpose Call Perl script using appropriate operating system executable

Syntax

```
perl('perlfile')  
perl('perlfile',arg1,arg2,...)  
result = perl(...)
```

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

`perl('perlfile',arg1,arg2,...)` calls the Perl script `perlfile`, using the appropriate operating system Perl executable, and passes the arguments `arg1`, `arg2`, and so on, to `perlfile`.

`result = perl(...)` returns the results of attempted Perl call to `result`.

Examples Given the Perl script, `hello.pl`

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

run the following statement in MATLAB

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

MATLAB returns

```
ans =  
Hello World.
```

It is sometimes beneficial to use Perl scripts instead of MATLAB code. The `perl` function allows you to run those scripts from within MATLAB. Specific examples where you might choose to use a Perl script include

- Perl script already exists

- Perl script preprocesses data quickly, formatting it in a way more easily read by MATLAB
- Perl has features not supported by MATLAB

See Also

! (exclamation point), dos, regexp, system, unix

perms

Purpose All possible permutations

Syntax `P = perms(v)`

Description `P = perms(v)`, where `v` is a row vector of length `n`, creates a matrix whose rows consist of all possible permutations of the `n` elements of `v`. Matrix `P` contains `n!` rows and `n` columns.

Examples The command `perms(2:2:6)` returns *all* the permutations of the numbers 2, 4, and 6:

```
6     4     2
6     2     4
4     6     2
4     2     6
2     4     6
2     6     4
```

Limitations This function is only practical for situations where `n` is less than about 15.

See Also `nchoosek`, `permute`, `randperm`

Purpose Rearrange dimensions of N-D array

Syntax `B = permute(A,order)`

Description `B = permute(A,order)` rearranges the dimensions of `A` so that they are in the order specified by the vector `order`. `B` has the same values of `A` but the order of the subscripts needed to access any particular element is rearranged as specified by `order`. All the elements of `order` must be unique.

Remarks `permute` and `ipermute` are a generalization of transpose (`.'`) for multidimensional arrays.

Examples Given any matrix `A`, the statement

```
permute(A,[2 1])
```

is the same as `A'`.

For example:

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

The following code permutes a three-dimensional array:

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

See Also `ipermute`, `circshift`

persistent

Purpose Define persistent variable

Syntax persistent X Y Z

Description persistent X Y Z defines X, Y, and Z as variables that are local to the function in which they are declared; yet their values are retained in memory between calls to the function. Persistent variables are similar to global variables because MATLAB creates permanent storage for both. They differ from global variables in that persistent variables are known only to the function in which they are declared. This prevents persistent variables from being changed by other functions or from the MATLAB command line.

Persistent variables are cleared when the M-file is cleared from memory or when the M-file is changed. To keep an M-file in memory until MATLAB quits, use `mlock`.

If the persistent variable does not exist the first time you issue the persistent statement, it is initialized to the empty matrix.

It is an error to declare a variable persistent if a variable with the same name exists in the current workspace.

Remarks There is no function form of the persistent command (i.e., you cannot use parentheses and quote the variable names).

Example This function prompts a user to enter a directory name to use in locating one or more files. If the user has already entered this information, and it requires no modification, they do not need to enter it again. This is because the function stores it in a persistent variable (`lastDir`), and offers it as the default selection. Here is the function definition:

```
function find_file(file)
persistent lastDir

if isempty(lastDir)
    prompt = 'Enter directory: ';
else
```



```

        prompt = ['Enter directory[' lastDir ']: '];
    end
    response = input(prompt, 's');

    if ~isempty(response)
        dirName = response;
    else
        dirName = lastDir;
    end

    dir(strcat(dirName, file))
    lastDir = dirName;

```

Execute the function twice. The first time, it prompts you to enter the information and does not offer a default:

```

cd(matlabroot)

find_file('is*.m')
Enter directory:  toolbox/matlab/strfun/

iscellstr.m  ischar.m      isletter.m   isspace.m    isstr.m
isstrprop.m

```

The second time, it does offer a default taken from the persistent variable `dirName`:

```

find_file('is*.m')
Enter directory[toolbox/matlab/strfun/]:  toolbox/matlab/elmat/

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

```

See Also

`global`, `clear`, `mislocked`, `mlock`, `munlock`, `isempty`

pi

Purpose Ratio of circle's circumference to its diameter, π

Syntax `pi`

Description `pi` returns the floating-point number nearest the value of π . The expressions `4*atan(1)` and `imag(log(-1))` provide the same value.

Examples The expression `sin(pi)` is not exactly zero because `pi` is not exactly π .

```
sin(pi)
```

```
ans =
```


```
1.2246e-16
```

See Also `ans`, `eps`, `i`, `Inf`, `j`, `NaN`

Purpose

Pie chart

**GUI Alternatives**

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
pie(X)
pie(X,explode)
pie(...,labels)
pie(axes_handle,...)
h = pie(...)
```

Description

`pie(X)` draws a pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

`pie(X,explode)` offsets a slice from the pie. `explode` is a vector or matrix of zeros and nonzeros that correspond to `X`. A nonzero value offsets the corresponding slice from the center of the pie chart, so that `X(i,j)` is offset from the center if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

`pie(...,labels)` specifies text labels for the slices. The number of labels must equal the number of elements in `X`. For example,

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

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

`h = pie(...)` returns a vector of handles to patch and text graphics objects.

pie

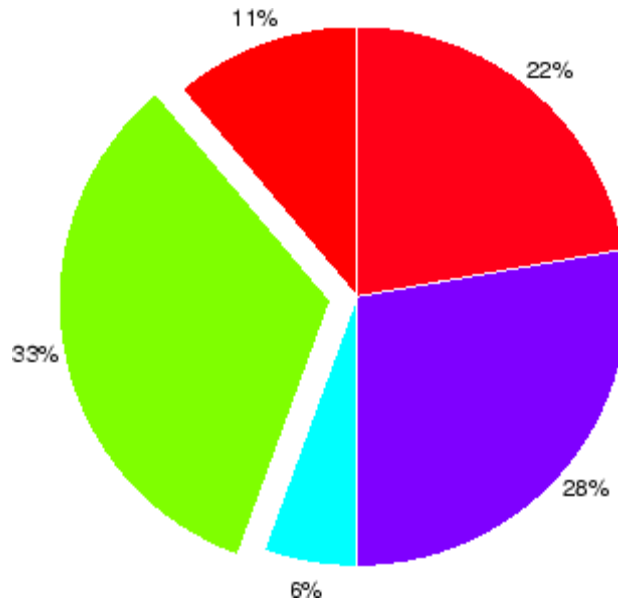
Remarks

The values in X are normalized via $X/\text{sum}(X)$ to determine the area of each slice of the pie. If $\text{sum}(X) \leq 1$, the values in X directly specify the area of the pie slices. MATLAB draws only a partial pie if $\text{sum}(X) < 1$.

Examples

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

```
x = [1 3 0.5 2.5 2];  
explode = [0 1 0 0 0];  
pie(x,explode)  
colormap jet
```




See Also

[pie3](#)

Purpose

3-D pie chart

**GUI Alternatives**

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
pie3(X)
pie3(X,explode)
pie3(...,labels)
pie3(axes_handle,...)
h = pie3(...)
```

Description

`pie3(X)` draws a three-dimensional pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

`pie3(X,explode)` specifies whether to offset a slice from the center of the pie chart. `X(i,j)` is offset from the center of the pie chart if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

`pie3(...,labels)` specifies text labels for the slices. The number of labels must equal the number of elements in `X`. For example,

```
pie3(1:3,{'Taxes','Expenses','Profit'})
```

`pie3(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = pie3(...)` returns a vector of handles to patch, surface, and text graphics objects.

pie3

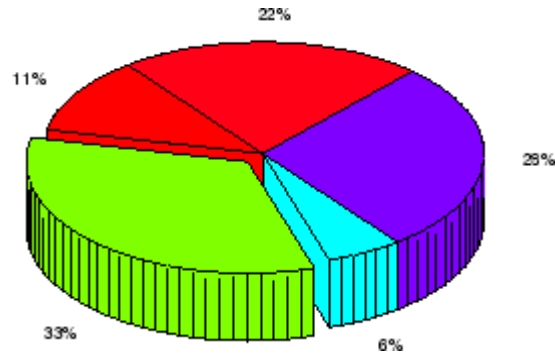
Remarks

The values in X are normalized via $X/\text{sum}(X)$ to determine the area of each slice of the pie. If $\text{sum}(X) \leq 1$, the values in X directly specify the area of the pie slices. MATLAB draws only a partial pie if $\text{sum}(X) < 1$.

Examples

Offset a slice in the pie chart by setting the corresponding `explode` element to 1:

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



See Also

`pie`

Purpose Moore-Penrose pseudoinverse of matrix

Syntax
 $B = \text{pinv}(A)$
 $B = \text{pinv}(A, \text{tol})$

Definition The Moore-Penrose pseudoinverse is a matrix B of the same dimensions as A' satisfying four conditions:

$$\begin{aligned} A*B*A &= A \\ B*A*B &= B \\ A*B &\text{ is Hermitian} \\ B*A &\text{ is Hermitian} \end{aligned}$$

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

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

$B = \text{pinv}(A, \text{tol})$ returns the Moore-Penrose pseudoinverse and overrides the default tolerance, $\max(\text{size}(A)) * \text{norm}(A) * \text{eps}$.

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

If A has more rows than columns and is not of full rank, then the overdetermined least squares problem

$$\text{minimize } \text{norm}(A*x - b)$$

does not have a unique solution. Two of the infinitely many solutions are

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

and

$$y = A \setminus b$$

These two are distinguished by the facts that $\text{norm}(x)$ is smaller than the norm of any other solution and that y has the fewest possible nonzero components.

For example, the matrix generated by

```
A = magic(8); A = A(:,1:6)
```

is an 8-by-6 matrix that happens to have $\text{rank}(A) = 3$.

```
A =
 64     2     3    61    60     6
   9    55    54    12    13    51
  17    47    46    20    21    43
  40    26    27    37    36    30
  32    34    35    29    28    38
  41    23    22    44    45    19
  49    15    14    52    53    11
   8    58    59     5     4    62
```

The right-hand side is $b = 260 \cdot \text{ones}(8, 1)$,

```
b =
 260
 260
 260
 260
 260
 260
 260
 260
 260
```

The scale factor 260 is the 8-by-8 magic sum. With all eight columns, one solution to $A \cdot x = b$ would be a vector of all 1's. With only six columns, the equations are still consistent, so a solution exists, but it is not all 1's. Since the matrix is rank deficient, there are infinitely many solutions. Two of them are

```
x = pinv(A)*b
```


which is

```
x =  
    1.1538  
    1.4615  
    1.3846  
    1.3846  
    1.4615  
    1.1538
```

and

```
y = A\b
```

which produces this result.

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

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

```
norm(x) = 3.2817
```

is smaller than the norm of any other solution, including

```
norm(y) = 6.4807
```

On the other hand, the solution y is special because it has only three nonzero components.

See Also

inv, qr, rank, svd

planerot

Purpose Givens plane rotation

Syntax `[G,y] = planerot(x)`

Description `[G,y] = planerot(x)` where x is a 2-component column vector, returns a 2-by-2 orthogonal matrix G so that $y = G*x$ has $y(2) = 0$.

Examples

```
x = [3 4];  
[G,y] = planerot(x')
```

```
G =  
    0.6000    0.8000  
   -0.8000    0.6000
```

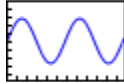
```
y =  
    5  
    0
```

See Also `qrdelete`, `qrinsert`


Purpose	Run M-file demo (deprecated; use echodemo instead)
Syntax	<code>playshow filename</code>
Description	<code>playshow filename</code> runs <code>filename</code> , which is a demo. Replace <code>playshow filename</code> with <code>echodemo filename</code> . Note that other arguments supported by <code>playshow</code> are not supported by <code>echodemo</code> .
See Also	<code>demo</code> , <code>echodemo</code> , <code>helpbrowser</code>

plot

Purpose 2-D line plot



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 *Using Plot Edit Mode*, and *The Figure Palette* in the MATLAB Graphics documentation, and also *Creating Graphics from the Workspace Browser* in the MATLAB Desktop documentation.

Syntax

```
plot(Y)
plot(X1,Y1,...)
plot(X1,Y1,LineStyle,...)
plot(...,'PropertyName',PropertyValue,...)
plot(axes_handle,...)
h = plot(...)
hlines = plot('v6',...)
```

Description

`plot(Y)` plots the columns of Y versus their index if Y is a real number. If Y is complex, `plot(Y)` is equivalent to `plot(real(Y), imag(Y))`. In all other uses of `plot`, the imaginary component is ignored.

`plot(X1,Y1,...)` plots all lines defined by X_n versus Y_n pairs. If only X_n or Y_n is a matrix, the vector is plotted versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix. If X_n is a scalar and Y_n is a vector, disconnected line objects are created and plotted as discrete points vertically at X_n .

`plot(X1,Y1,LineStyle,...)` plots all lines defined by the $X_n, Y_n, LineSpec$ triples, where `LineStyle` is a line specification that 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:
`plot(X1,Y1,X2,Y2,LineStyle,X3,Y3)`.

Note See `LineStyleSpec` for a list of line style, marker, and color specifiers.

`plot(..., 'PropertyName', PropertyValue, ...)` sets properties to the specified property values for all lineseries graphics objects created by `plot`. (See the Examples section for examples.)

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

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

Backward-Compatible Version

`hlines = plot('v6', ...)` returns the handles to line objects instead of lineseries objects.

Remarks

If you do not specify a color when plotting more than one line, `plot` automatically cycles through the colors in the order specified by the current axes `ColorOrder` property. After cycling through all the colors defined by `ColorOrder`, `plot` then cycles through the line styles defined in the axes `LineStyleOrder` property.

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

Cycling Through Line Colors and Styles

By default, MATLAB resets the `ColorOrder` and `LineStyleOrder` properties each time you call `plot`. If you want changes you make to these properties to persist, then you must define these changes as default values. For example,

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

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

Prevent Resetting of Color and Styles with `hold all`

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

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

Additional Information

- See [Creating Line Plots and Annotating Graphs](#) for more information on plotting.
- See [LineStyleOrder](#) for more information on specifying line styles and colors.

Examples

Specifying the Color and Size of Markers

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

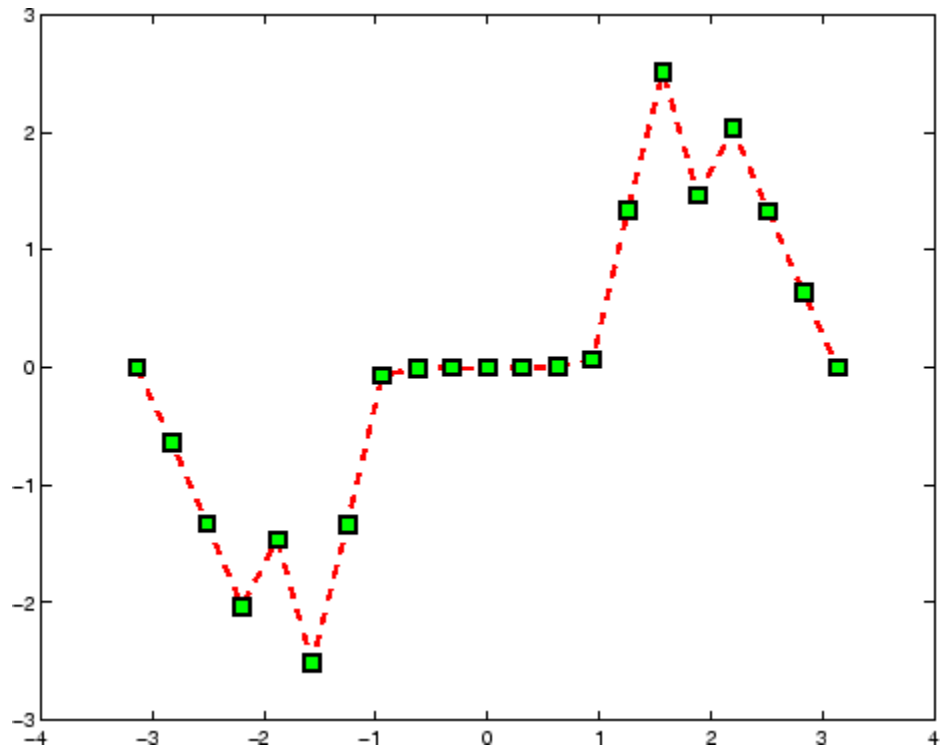
- `LineWidth` — Specifies the width (in points) of the line.
- `MarkerEdgeColor` — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- `MarkerFaceColor` — Specifies the color of the face of filled markers.
-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

For example, these statements,

```
x = -pi:pi/10:pi;
```

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

produce this graph.



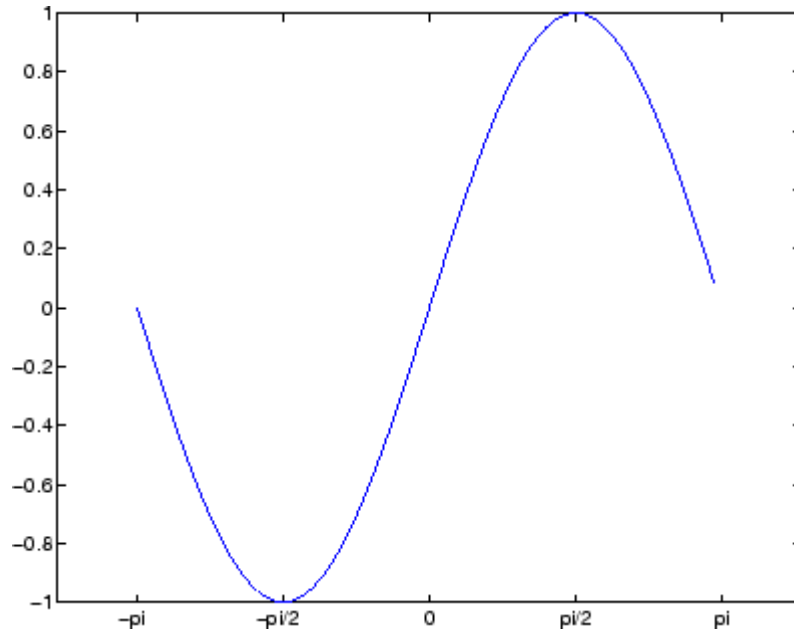
Specifying Tick-Mark Location and Labeling

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

plot

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

Now add axis labels and annotate the point $-\pi/4, \sin(-\pi/4)$.



Adding Titles, Axis Labels, and Annotations

MATLAB enables you to add axis labels and titles. For example, using the graph from the previous example, add an x - and y -axis label:

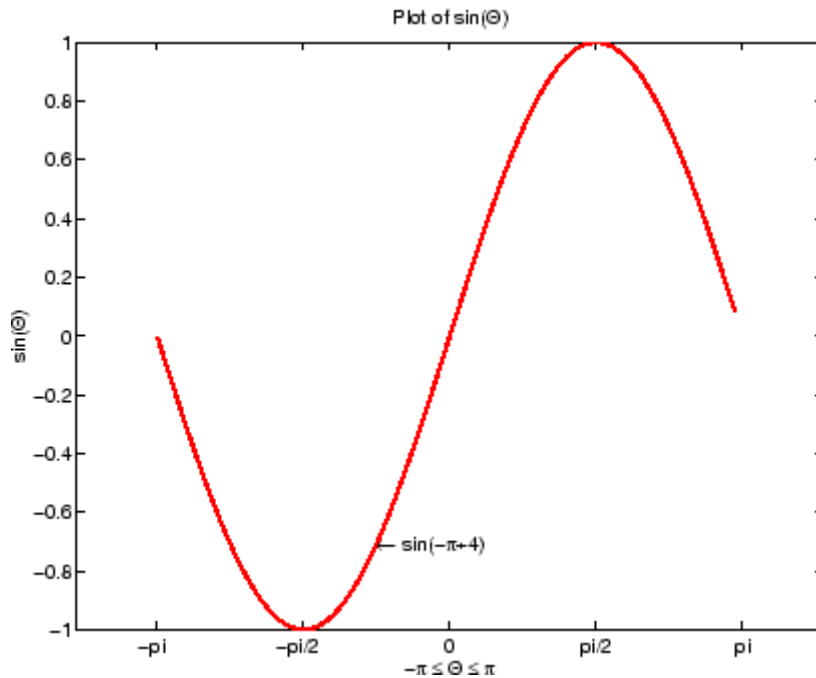
```
xlabel('-\pi \leq \Theta \leq \pi')
ylabel('sin(\Theta)')
title('Plot of sin(\Theta)')
text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi\div4),...')
```



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

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

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



See Also

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

plot

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

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

See Basic Plots and Graphs for related functions.


Purpose	Plot time series
Syntax	<code>plot(ts)</code> <code>plot(tsc.tsname)</code> <code>plot(function)</code>
Description	<p><code>plot(ts)</code> plots the time-series data against time and interpolates values between samples by using either zero-order-hold ('zoh') or linear interpolation.</p> <p><code>plot(tsc.tsname)</code> plots the timeseries object <code>tsname</code> that is part of the <code>tscollection tsc</code>.</p> <p><code>plot(function)</code> accepts the modifiers used by the MATLAB plotting utility for numerical arrays. These modifiers can be specified as auxiliary inputs for modifying the appearance of the plot. See Examples below.</p>
Remarks	Time-series events, when defined, are marked in the plot by a red circular marker.
Examples	<p><code>plot(ts, '-r*')</code> uses a regular line with the color red and marker '*' to render the plot.</p> <p><code>plot(ts, 'ko', 'MarkerSize', 3)</code> uses black circular markers of size 3 to render the plot.</p>

plot3

Purpose 3-D line plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
plot3(X1,Y1,Z1,...)
plot3(X1,Y1,Z1,LineStyle,...)
plot3(...,'PropertyName',PropertyValue,...)
h = plot3(...)
```

Description

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

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

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

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

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

Remarks

If one or more of $X1$, $Y1$, $Z1$ is a vector, the vectors are plotted versus the rows or columns of the matrix, depending whether the vectors' lengths equal the number of rows or the number of columns.

You can mix Xn, Yn, Zn triples with $Xn, Yn, Zn, LineSpec$ quads, for example,

```
plot3(X1,Y1,Z1,X2,Y2,Z2,LineSpec,X3,Y3,Z3)
```

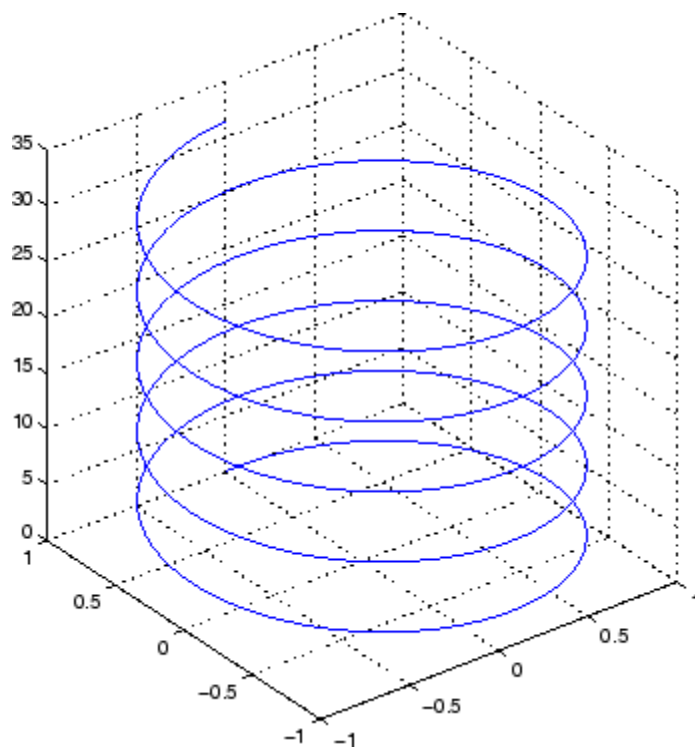
See `LineSpec` and `plot` for information on line types and markers.

Examples

Plot a three-dimensional helix.

```
t = 0:pi/50:10*pi;  
plot3(sin(t),cos(t),t)  
grid on  
axis square
```

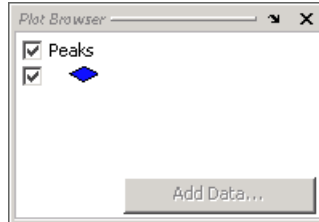
plot3





See Also

`axis`, `bar3`, `grid`, `line`, `LineStyle`, `loglog`, `plot`, `semilogx`, `semilogy`, `subplot`

Purpose Show or hide figure plot browser



GUI Alternatives

Click the larger Plotting Tools icon  on the figure toolbar to collectively enable plotting tools, and the smaller icon  to collectively disable them. Open or close the **Plot Browser** tool from the figure's **View** menu. For details, see “The Plot Browser” in the MATLAB Graphics documentation.

Syntax

```
plotbrowser('on')  
plotbrowser('off')  
plotbrowser('toggle')  
plotbrowser(figure_handle,...)
```

Description

`plotbrowser('on')` displays the Plot Browser on the current figure.

`plotbrowser('off')` hides the Plot Browser on the current figure.

`plotbrowser('toggle')` or `plotbrowser` toggles the visibility of the Plot Browser on the current figure.

`plotbrowser(figure_handle,...)` shows or hides the Plot Browser on the figure specified by *figure_handle*.

See Also

`plottools`, `figurepalette`, `propertyeditor`

plottedit

Purpose Interactively edit and annotate plots

Syntax

```
plottedit on
plottedit off
plottedit
plottedit(h)
plottedit('state')
plottedit(h, 'state')
```

Description `plottedit on` starts plot edit mode for the current figure, allowing you to use a graphical interface to annotate and edit plots easily. In plot edit mode, you can label axes, change line styles, and add text, line, and arrow annotations.

`plottedit off` ends plot mode for the current figure.

`plottedit` toggles the plot edit mode for the current figure.

`plottedit(h)` toggles the plot edit mode for the figure specified by figure handle `h`.

`plottedit('state')` specifies the `plottedit` state for the current figure. Values for `state` can be as shown.

Value for state	Description
<code>on</code>	Starts plot edit mode
<code>off</code>	Ends plot edit mode
<code>showtoolsmenu</code>	Displays the Tools menu in the menu bar
<code>hidetoolsmenu</code>	Removes the Tools menu from the menu bar

Note `hidetoolsmenu` is intended for GUI developers who do not want the **Tools** menu to appear in applications that use the figure window.

`plottedit(h, 'state')` specifies the plottedit state for figure handle `h`.

Remarks

Plot Editing Mode Graphical Interface Components

Use these toolbar buttons to add a legend, text, and arrows.

To start plot edit mode, click this button.

Use the Edit, Insert, and Tools menus to add objects or edit existing objects in a graph.

Double-click on an object to select it.

Position labels, legends, and other objects by clicking and dragging.

Access object-specific plot edit functions through context-sensitive pop-up menus.

Examples

Start plot edit mode for figure 2.

```
plottedit(2)
```

End plot edit mode for figure 2.

```
plottedit(2, 'off')
```

plottedit

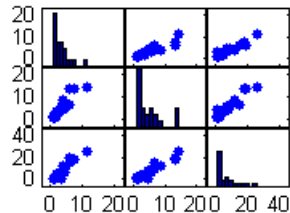
Hide the **Tools** menu for the current figure:

```
plottedit('hidetoolsmenu')
```

See Also

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

Purpose Scatter plot matrix



Syntax

```
plotmatrix(X,Y)
plotmatrix(...,'LineStyle')
[H,AX,BigAx,P] = plotmatrix(...)
```

Description `plotmatrix(X,Y)` scatter plots the columns of X against the columns of Y . If X is p -by- m and Y is p -by- n , `plotmatrix` produces an n -by- m matrix of axes. `plotmatrix(Y)` is the same as `plotmatrix(Y,Y)` except that the diagonal is replaced by `hist(Y(:,i))`.

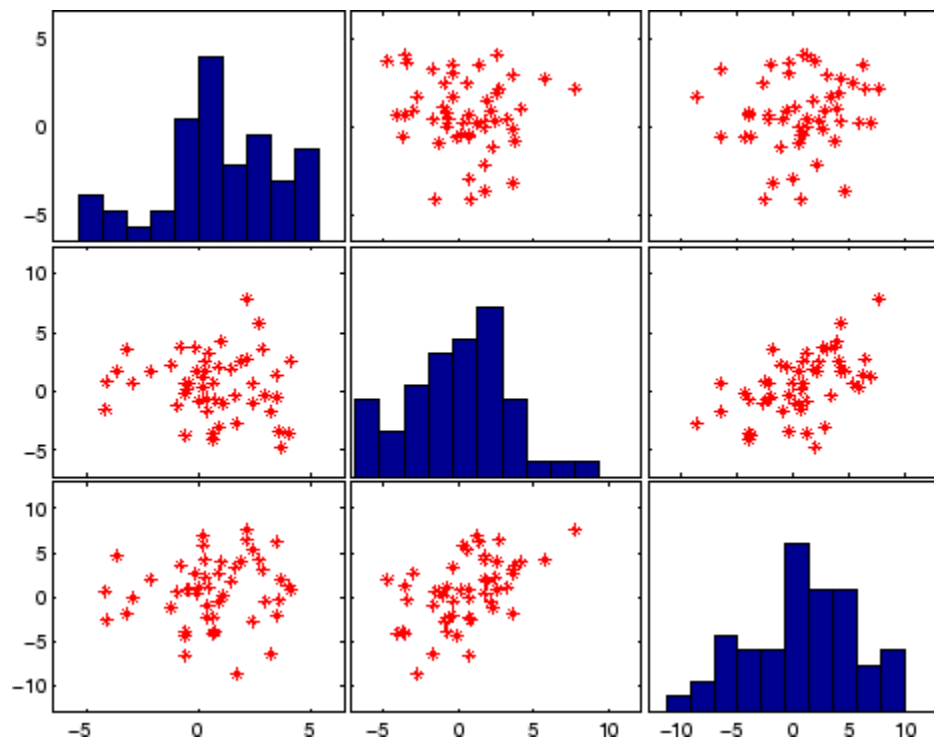
`plotmatrix(...,'LineStyle')` uses a `LineStyle` to create the scatter plot. The default is `'.'`.

`[H,AX,BigAx,P] = plotmatrix(...)` returns a matrix of handles to the objects created in H , a matrix of handles to the individual subaxes in AX , a handle to a big (invisible) axes that frames the subaxes in $BigAx$, and a matrix of handles for the histogram plots in P . $BigAx$ is left as the current axes so that a subsequent `title`, `xlabel`, or `ylabel` command is centered with respect to the matrix of axes.

Examples Generate plots of random data.

```
x = randn(50,3); y = x*[-1 2 1;2 0 1;1 -2 3]';
plotmatrix(y,'*r')
```

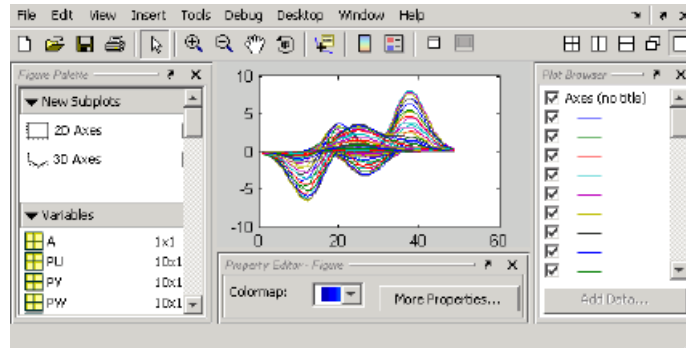
plotmatrix





See Also

`scatter`, `scatter3`

Purpose Show or hide plot tools



GUI Alternatives

Click the larger Plotting Tools icon  on the figure toolbar to collectively enable plotting tools, and the smaller icon  to collectively disable them. Individually select the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu. For details, see “Plotting Tools — Interactive Plotting” in the MATLAB Graphics documentation.

Syntax

```
plottools('on')
plottools('off')
plottools
plottools(figure_handle,...)
plottools(...,'tool')
```

Description

`plottools('on')` displays the Figure Palette, Plot Browser, and Property Editor on the current figure, configured as you last used them.

`plottools('off')` hides the Figure Palette, Plot Browser, and Property Editor on the current figure.

`plottools` with no arguments, is the same as `plottools('on')`

`plottools(figure_handle,...)` displays or hides the plot tools on the specified figure instead of the current figure.

plottools

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

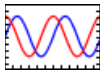
- `figurepalette`
- `plotbrowser`
- `propertyeditor`

Note The first time you open the plotting tools, all three of them appear, grouped around the current figure as shown above. If you close, move, or undock any of the tools, MATLAB remembers the configuration you left them in and will restore it when you invoke the tools for subsequent figures, both within and across MATLAB sessions.


See Also

`figurepalette`, `plotbrowser`, `propertyeditor`

Purpose 2-D line plots with y-axes on both left and right side



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see “Plotting Tools — Interactive Plotting” in the MATLAB Graphics documentation and “Creating Plots from the Workspace Browser” in the MATLAB Desktop Tools documentation.

Syntax

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

Description

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

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

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

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

For example,

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

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

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

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

Examples

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

```
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);
[AX,H1,H2] = plotyy(x,y1,x,y2,'plot');
```

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

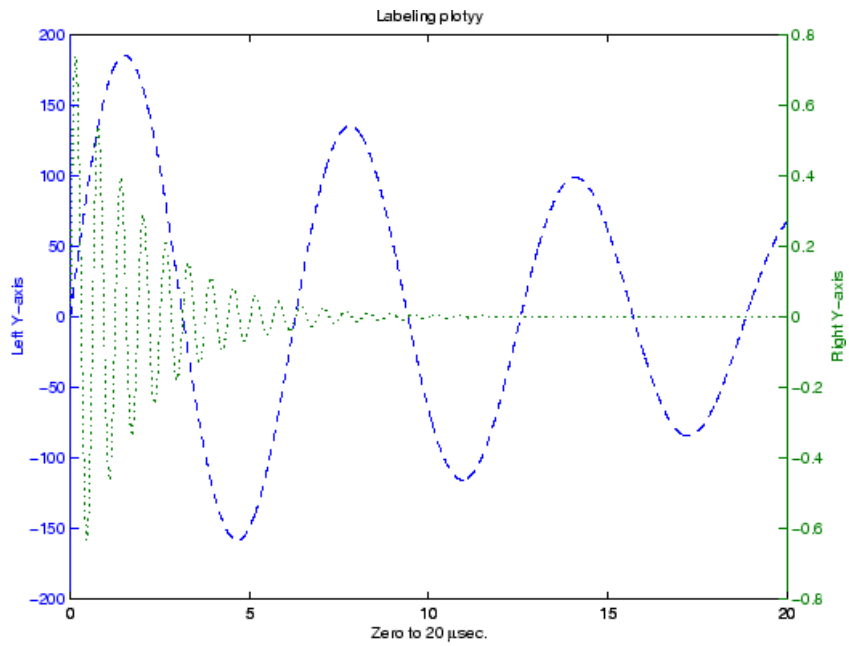
```
set(get(AX(1),'Ylabel'),'String','Slow Decay')
set(get(AX(2),'Ylabel'),'String','Fast Decay')
```

Use the `xlabel` and `title` commands to label the `x`-axis and add a title:

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

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

```
set(H1,'LineStyle','--')
set(H2,'LineStyle',':')
```

See Also

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

See “Using Multiple X- and Y-Axes” for more information.

pol2cart

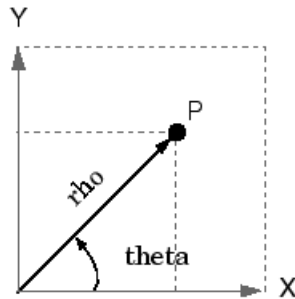
Purpose Transform polar or cylindrical coordinates to Cartesian

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

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

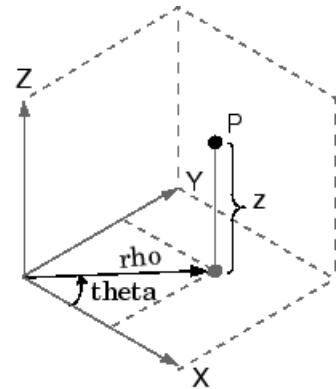
xyz , $[X,Y,Z] = \text{pol2cart}(\text{THETA},\text{RHO},Z)$ transforms the cylindrical coordinate data stored in corresponding elements of THETA, RHO, and Z to three-dimensional Cartesian, or coordinates. The arrays THETA, RHO, and Z must be the same size (or any can be scalar). The values in THETA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

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



Polar to Cartesian Mapping

$$\begin{aligned}\text{theta} &= \text{atan2}(y,x) \\ \text{rho} &= \text{sqrt}(x.^2 + y.^2)\end{aligned}$$



Cylindrical to Cartesian Mapping

$$\begin{aligned}\text{theta} &= \text{atan2}(y,x) \\ \text{rho} &= \text{sqrt}(x.^2 + y.^2) \\ Z &= Z\end{aligned}$$

See Also `cart2pol`, `cart2sph`, `sph2cart`


polar

Purpose

Polar coordinate plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
polar(theta,rho)
polar(theta,rho,LineStyle)
polar(axes_handle,...)
h = polar(...)
```

Description

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

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

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

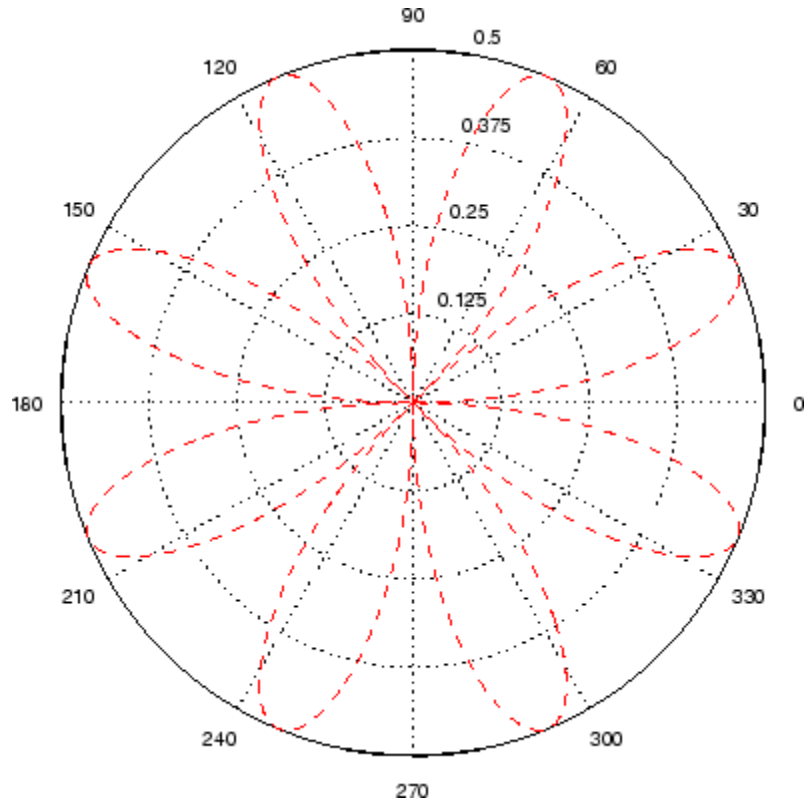
`polar(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

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

Examples

Create a simple polar plot using a dashed red line:

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

**See Also**

`cart2pol`, `compass`, `LineSpec`, `plot`, `pol2cart`, `rose`

poly

Purpose Polynomial with specified roots

Syntax
`p = poly(A)`
`p = poly(r)`

Description `p = poly(A)` where A is an n -by- n matrix returns an $n+1$ element row vector whose elements are the coefficients of the characteristic polynomial, $\det(sI - A)$. The coefficients are ordered in descending powers: if a vector c has $n+1$ components, the polynomial it represents is $c_1s^n + \dots + c_n s + c_{n+1}$

`p = poly(r)` where r is a vector returns a row vector whose elements are the coefficients of the polynomial whose roots are the elements of r .

Remarks Note the relationship of this command to

`r = roots(p)`

which returns a column vector whose elements are the roots of the polynomial specified by the coefficients row vector p . For vectors, `roots` and `poly` are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples MATLAB displays polynomials as row vectors containing the coefficients ordered by descending powers. The characteristic equation of the matrix

```
A =  
    1    2    3  
    4    5    6  
    7    8    0
```

is returned in a row vector by `poly`:

```
p = poly(A)
```

```
p =
```

```
1 -6 -72 -27
```

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

```
r = roots(p)
```

```
r =
```

```
12.1229
```

```
-5.7345
```

```
-0.3884
```

Algorithm

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

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

$$\det(\lambda I - A) = c_1 \lambda^n + \dots + c_n \lambda + c_{n+1}$$

The algorithm is

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

This recursion is easily derived by expanding the product.

$$(\lambda - \lambda_1)(\lambda - \lambda_2) \dots (\lambda - \lambda_n)$$

poly

It is possible to prove that `poly(A)` produces the coefficients in the characteristic polynomial of a matrix within roundoff error of A . This is true even if the eigenvalues of A are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial, which do not use the eigenvalues, do not have such satisfactory numerical properties.

See Also

`conv`, `polyval`, `residue`, `roots`

Purpose Area of polygon

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

Description `A = polyarea(X,Y)` returns the area of the polygon specified by the vertices in the vectors `X` and `Y`.

If `X` and `Y` are matrices of the same size, then `polyarea` returns the area of polygons defined by the columns `X` and `Y`.

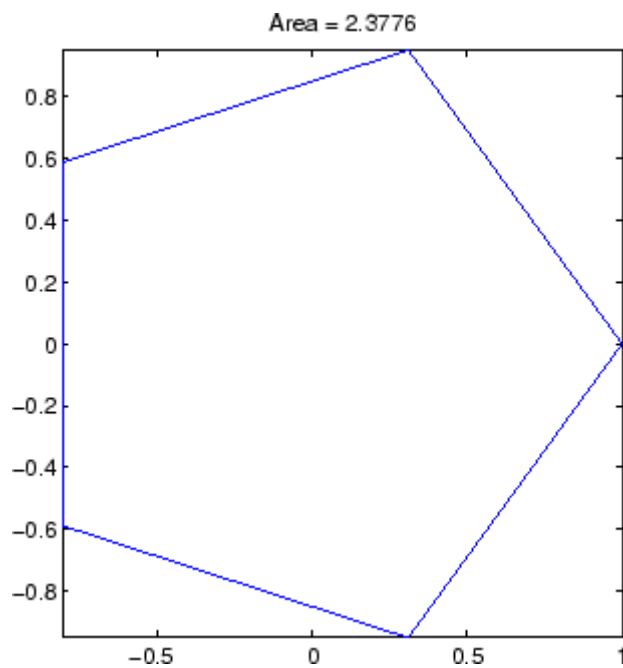
If `X` and `Y` are multidimensional arrays, `polyarea` returns the area of the polygons in the first nonsingleton dimension of `X` and `Y`.

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

Examples

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

polyarea



See Also

`convhull`, `inpolygon`, `rectint`

Purpose	Polynomial derivative
Syntax	<pre>k = polyder(p) k = polyder(a,b) [q,d] = polyder(b,a)</pre>
Description	<p>The <code>polyder</code> function calculates the derivative of polynomials, polynomial products, and polynomial quotients. The operands <code>a</code>, <code>b</code>, and <code>p</code> are vectors whose elements are the coefficients of a polynomial in descending powers.</p> <p><code>k = polyder(p)</code> returns the derivative of the polynomial <code>p</code>.</p> <p><code>k = polyder(a,b)</code> returns the derivative of the product of the polynomials <code>a</code> and <code>b</code>.</p> <p><code>[q,d] = polyder(b,a)</code> returns the numerator <code>q</code> and denominator <code>d</code> of the derivative of the polynomial quotient <code>b/a</code>.</p>
Examples	<p>The derivative of the product</p> $(3x^2 + 6x + 9)(x^2 + 2x)$ <p>is obtained with</p> <pre>a = [3 6 9]; b = [1 2 0]; k = polyder(a,b) k = 12 36 42 18</pre> <p>This result represents the polynomial</p> $12x^3 + 36x^2 + 42x + 18$
See Also	<code>conv</code> , <code>deconv</code>

polyeig

Purpose Polynomial eigenvalue problem

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

Description $[X, e] = \text{polyeig}(A_0, A_1, \dots, A_p)$ solves the polynomial eigenvalue problem of degree p

$$(A_0 + \lambda A_1 + \dots + \lambda^p A_p)x = 0$$

where polynomial degree p is a non-negative integer, and A_0, A_1, \dots, A_p are input matrices of order n . The output consists of a matrix X of size n -by- $n \times p$ whose columns are the eigenvectors, and a vector e of length $n \times p$ containing the eigenvalues.

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

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

$[X, e, s] = \text{polyeig}(A_0, A_1, \dots, A_p)$ also returns a vector s of length $p \times n$ containing condition numbers for the eigenvalues. At least one of A_0 and A_p must be nonsingular. Large condition numbers imply that the problem is close to a problem with multiple eigenvalues.

Remarks Based on the values of p and n , `polyeig` handles several special cases:

- $p = 0$, or `polyeig(A)` is the standard eigenvalue problem: `eig(A)`.
- $p = 1$, or `polyeig(A,B)` is the generalized eigenvalue problem: `eig(A,-B)`.
- $n = 1$, or `polyeig(a0,a1,...ap)` for scalars a_0, a_1, \dots, a_p is the standard polynomial problem: `roots([ap ... a1 a0])`.

If both A_0 and A_p are singular the problem is potentially ill-posed. Theoretically, the solutions might not exist or might not be unique. Computationally, the computed solutions might be inaccurate. If one, but not both, of A_0 and A_p is singular, the problem is well posed, but some of the eigenvalues might be zero or infinite.

Note that scaling A_0, A_1, \dots, A_p to have $\text{norm}(A_i)$ roughly equal 1 may increase the accuracy of `polyeig`. In general, however, this cannot be achieved. (See Tisseur [3] for more detail.)

Algorithm

The `polyeig` function uses the QZ factorization to find intermediate results in the computation of generalized eigenvalues. It uses these intermediate results to determine if the eigenvalues are well-determined. See the descriptions of `eig` and `qz` for more on this.

See Also

`condeig`, `eig`, `qz`

References

- [1] Dedieu, Jean-Pierre Dedieu and Françoise Tisseur, “Perturbation theory for homogeneous polynomial eigenvalue problems,” *Linear Algebra Appl.*, Vol. 358, pp. 71-94, 2003.
- [2] Tisseur, Françoise and Karl Meerbergen, “The quadratic eigenvalue problem,” *SIAM Rev.*, Vol. 43, Number 2, pp. 235-286, 2001.
- [3] Françoise Tisseur, “Backward error and condition of polynomial eigenvalue problems” *Linear Algebra Appl.*, Vol. 309, pp. 339-361, 2000.

polyfit

Purpose Polynomial curve fitting

Syntax
`p = polyfit(x,y,n)`
`[p,S] = polyfit(x,y,n)`
`[p,S,mu] = polyfit(x,y,n)`

Description `p = polyfit(x,y,n)` finds the coefficients of a polynomial $p(x)$ of degree n that fits the data, $p(x(i))$ to $y(i)$, in a least squares sense. The result `p` is a row vector of length $n+1$ containing the polynomial coefficients in descending powers

$$p(x) = p_1x^n + p_2x^{n-1} + \dots + p_nx + p_{n+1}$$

`[p,S] = polyfit(x,y,n)` returns the polynomial coefficients `p` and a structure `S` for use with `polyval` to obtain error estimates or predictions. Structure `S` contains fields `R`, `df`, and `normr`, for the triangular factor from a QR decomposition of the Vandermonde matrix of `X`, the degrees of freedom, and the norm of the residuals, respectively. If the data `Y` are random, an estimate of the covariance matrix of `P` is $(Rinv * Rinv') * normr^2 / df$, where `Rinv` is the inverse of `R`. If the errors in the data `y` are independent normal with constant variance, `polyval` produces error bounds that contain at least 50% of the predictions.

`[p,S,mu] = polyfit(x,y,n)` finds the coefficients of a polynomial in

$$\hat{x} = \frac{x - \mu_1}{\mu_2}$$

where $\mu_1 = \text{mean}(x)$ and $\mu_2 = \text{std}(x)$. `mu` is the two-element vector $[\mu_1, \mu_2]$. This centering and scaling transformation improves the numerical properties of both the polynomial and the fitting algorithm.

Examples

This example involves fitting the error function, $\text{erf}(x)$, by a polynomial in x . This is a risky project because $\text{erf}(x)$ is a bounded function, while polynomials are unbounded, so the fit might not be very good.

First generate a vector of x points, equally spaced in the interval **[0, 2.5]**; then evaluate $\text{erf}(x)$ at those points.

```
x = (0: 0.1: 2.5)';
y = erf(x);
```

The coefficients in the approximating polynomial of degree 6 are

```
p = polyfit(x,y,6)
```

```
p =
```

```
0.0084 -0.0983 0.4217 -0.7435 0.1471 1.1064 0.0004
```

There are seven coefficients and the polynomial is

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

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

```
f = polyval(p,x);
```

A table showing the data, fit, and error is

```
table = [x y f y-f]
```

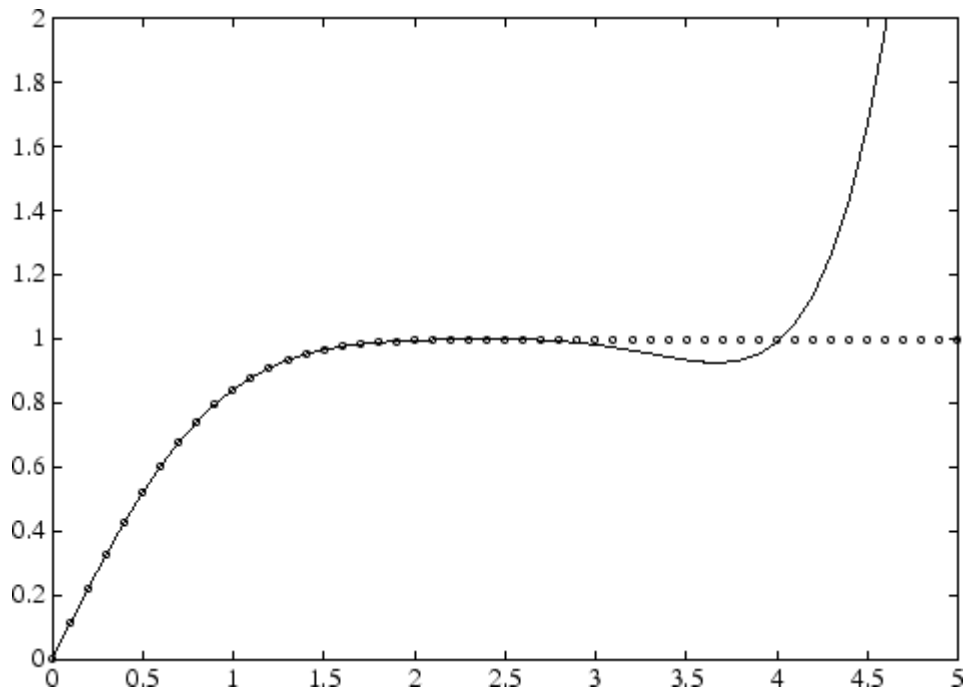
```
table =
```

0	0	0.0004	-0.0004
0.1000	0.1125	0.1119	0.0006
0.2000	0.2227	0.2223	0.0004
0.3000	0.3286	0.3287	-0.0001
0.4000	0.4284	0.4288	-0.0004
...			
2.1000	0.9970	0.9969	0.0001
2.2000	0.9981	0.9982	-0.0001
2.3000	0.9989	0.9991	-0.0003
2.4000	0.9993	0.9995	-0.0002

2.5000 0.9996 0.9994 0.0002

So, on this interval, the fit is good to between three and four digits. Beyond this interval the graph shows that the polynomial behavior takes over and the approximation quickly deteriorates.

```
x = (0: 0.1: 5)';  
y = erf(x);  
f = polyval(p,x);  
plot(x,y,'o',x,f,'-')  
axis([0 5 0 2])
```



Algorithm

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

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

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

$$Vp \cong y$$

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

See Also

`poly`, `polyval`, `roots`

polyint

Purpose	Integrate polynomial analytically
Syntax	<code>polyint(p,k)</code> <code>polyint(p)</code>
Description	<code>polyint(p,k)</code> returns a polynomial representing the integral of polynomial <code>p</code> , using a scalar constant of integration <code>k</code> . <code>polyint(p)</code> assumes a constant of integration <code>k=0</code> .
See Also	<code>polyder</code> , <code>polyval</code> , <code>polyvalm</code> , <code>polyfit</code>

Purpose	Polynomial evaluation
Syntax	<pre> y = polyval(p,x) y = polyval(p,x,[],mu) [y,delta] = polyval(p,x,S) [y,delta] = polyval(p,x,S,mu) </pre>
Description	<p><code>y = polyval(p,x)</code> returns the value of a polynomial of degree n evaluated at x. The input argument p is a vector of length $n+1$ whose elements are the coefficients in descending powers of the polynomial to be evaluated.</p> $y = p_1x^n + p_2x^{n-1} + \dots + p_nx + p_{n+1}$ <p>x can be a matrix or a vector. In either case, <code>polyval</code> evaluates p at each element of x.</p> <p><code>y = polyval(p,x,[],mu)</code> uses $\hat{x} = (x - \mu_1)/\mu_2$ in place of x. In this equation, $\mu_1 = \text{mean}(x)$ and $\mu_2 = \text{std}(x)$. The centering and scaling parameters $\text{mu} = [\mu_1, \mu_2]$ are optional output computed by <code>polyfit</code>.</p> <p><code>[y,delta] = polyval(p,x,S)</code> and <code>[y,delta] = polyval(p,x,S,mu)</code> use the optional output structure S generated by <code>polyfit</code> to generate error estimates, $y \pm \text{delta}$. If the errors in the data input to <code>polyfit</code> are independent normal with constant variance, $y \pm \text{delta}$ contains at least 50% of the predictions.</p>
Remarks	The <code>polyvalm(p,x)</code> function, with x a matrix, evaluates the polynomial in a matrix sense. See <code>polyvalm</code> for more information.
Examples	<p>The polynomial $p(x) = 3x^2 + 2x + 1$ is evaluated at $x = 5, 7,$ and 9 with</p> <pre> p = [3 2 1]; polyval(p,[5 7 9]) </pre> <p>which results in</p>

polyval

```
ans =
```

```
86 162 262
```

For another example, see `polyfit`.

See Also

`polyfit`, `polyvalm`

Purpose Matrix polynomial evaluation

Syntax `Y = polyvalm(p,X)`

Description `Y = polyvalm(p,X)` evaluates a polynomial in a matrix sense. This is the same as substituting matrix `X` in the polynomial `p`.

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

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

```
X = pascal(4)
X =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

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

```
p = poly(X)
p =
     1    -29     72    -29     1
```

This represents the polynomial $x^4 - 29x^3 + 72x^2 - 29x + 1$.

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

Evaluating this polynomial at each element is not very interesting.

```
polyval(p,X)
ans =
     16     16     16     16
     16     15    -140    -563
     16    -140   -2549  -12089
```

polyvalm

```
16    -563   -12089  -43779
```

But evaluating it in a matrix sense is interesting.

```
polyvalm(p,X)
ans =
    0     0     0     0
    0     0     0     0
    0     0     0     0
    0     0     0     0
```

The result is the zero matrix. This is an instance of the Cayley-Hamilton theorem: a matrix satisfies its own characteristic equation.

See Also

polyfit, polyval

Purpose Base 2 power and scale floating-point numbers

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

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

`X = pow2(F,E)` computes $x = f * 2^e$ for corresponding elements of `F` and `E`. The result is computed quickly by simply adding `E` to the floating-point exponent of `F`. Arguments `F` and `E` are real and integer arrays, respectively.

Remarks This function corresponds to the ANSI C function `ldexp()` and the IEEE floating-point standard function `scalbn()`.

Examples For IEEE arithmetic, the statement `X = pow2(F,E)` yields the values:

F	E	X
1/2	1	1
pi/4	2	pi
-3/4	2	-3
1/2	-51	eps
1-eps/2	1024	realmax
1/2	-1021	realmin

See Also `log2`, `exp`, `hex2num`, `realmax`, `realmin`
 The arithmetic operators `^` and `.^`

power

Purpose Array power

Syntax $Z = X.^Y$

Description $Z = X.^Y$ denotes element-by-element powers. X and Y must have the same dimensions unless one is a scalar. A scalar is expanded to an array of the same size as the other input.

$C = \text{power}(A,B)$ is called for the syntax ' $A.^B$ ' when A or B is an object.

See Also `nthroot`, `realpow`

Purpose Evaluate piecewise polynomial

Syntax `v = ppval(pp,xx)`

Description `v = ppval(pp,xx)` returns the value of the piecewise polynomial f , contained in `pp`, at the entries of `xx`. You can construct `pp` using the functions `interp1`, `pchip`, `spline`, or the spline utility `mkpp`.

`v` is obtained by replacing each entry of `xx` by the value of f there. If f is scalar-valued, `v` is of the same size as `xx`. `xx` may be N -dimensional.

If `pp` was constructed by `pchip`, `spline`, or `mkpp` using the orientation of non-scalar function values specified for those functions, then:

If f is $[D1, \dots, Dr]$ -valued, and `xx` is a vector of length N , then `V` has size $[D1, \dots, Dr, N]$, with `V(:, ..., :, J)` the value of f at `xx(J)`.

If f is $[D1, \dots, Dr]$ -valued, and `xx` has size $[N1, \dots, Ns]$, then `V` has size $[D1, \dots, Dr, N1, \dots, Ns]$, with `V(:, ..., :, J1, ..., Js)` the value of f at `xx(J1, ..., Js)`.

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

If f is $[D1, \dots, Dr]$ -valued, and `xx` is a vector of length N , then `V` has size $[N, D1, \dots, Dr]$, with `V(J, :, ..., :)` the value of f at `xx(J)`.

If f is $[D1, \dots, Dr]$ -valued, and `xx` has size $[N1, \dots, Ns]$, then `V` has size $[N1, \dots, Ns, D1, \dots, Dr]$, with `V(J1, ..., Js, :, ..., :)` the value of f at `xx(J1, ..., Js)`.

Examples Compare the results of integrating the function `cos`

```
a = 0; b = 10;
int1 = quad(@cos,a,b)
```

```
int1 =
    -0.5440
```

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

```
x = a:b;
y = cos(x);
pp = spline(x,y);
int2 = quad(@(x)ppval(pp,x),a,b)

int2 =
    -0.5485
```

`int1` provides the integral of the cosine function over the interval `[a,b]`, while `int2` provides the integral over the same interval of the piecewise polynomial `pp`.

See Also

`mkpp`, `spline`, `unmkpp`

Purpose

Directory containing preferences, history, and layout files

Syntax

```
prefdir
d = prefdir
d = prefdir(1)
```

Description

prefdir returns the directory that contains

- Preferences for MATLAB and related products (matlab.prf)
- Command history file (history.m)
- MATLAB shortcuts (shortcuts.xml)
- MATLAB desktop layout files (MATLABDesktop.xml and Your_Saved_LayoutMATLABLayout.xml)
- Other related files

The directory might be in a hidden folder, for example, myname/.matlab/R2006b. How to access hidden folders depends on your platform:

- On Windows, in any folder window, select **Tools > Folder Options**. Click the **View** tab, and under **Advanced** settings, select **Show hidden files and folders**. Then you should be able to see the folder returned by prefdir.
- On Macintosh platforms, in the Finder, select **Go -> Go to Folder**. In the resulting dialog box, type the path returned by prefdir and press **Enter**.

d = prefdir assigns to d the name of the directory containing preferences and related files.

d = prefdir(1) creates a directory for preferences and related files if one does not exist. If the directory does exist, the name is assigned to d.

Remarks

The preferences directory MATLAB uses depends on the release. The practice used from R13 through R14SP2 was changed starting in R14SP3 to address backwards compatibility problems. The differences are relevant primarily if you run multiple versions of MATLAB, and especially if one version is prior to R14SP3:

- For R2006b, R2006a, and R14SP3, MATLAB uses the R2006b, R2006a, and R14SP3 preferences directories, respectively. When you install R2006b, MATLAB migrates the files in the R2006a preferences directory to the R2006b preferences directory. While running R2006b, R2006a or R14SP3, any changes made to files in those preferences directories (R2006b, R2006a or R14SP3) are used only in their respective versions. As an example, commands you run in R2006b will *not* appear in the Command History when you run R2006a, or R14SP3, and the converse is also true.
- The R14 through R14SP2 releases all share the R14 preferences directory. While running R14SP1, for example, any changes made to files in the preferences directory, R14, are used when you run R14SP2 and R14. As another example, commands you run in R14 appear in the Command History when you run R14SP2, and the converse is also true. The preferences are not used when you run R14SP3, R2006a, or R2006b because those versions each use their own preferences directories.
- All R13 releases use the R13 preferences directory. While running R13SP1, for example, any changes made to files in the preferences directory, R13, are used when you run R13. As an example, commands you run in R13 will appear in the Command History when you run R13SP1, and the converse is true. The preferences are not used when you run any R14 or later releases because R14 and later releases use different preferences directories, and the converse is true.
- Upon startup, MATLAB 7.3 (R2006b) looks for and if found, uses the R2006b preferences directory. If not found, MATLAB creates an R2006b preferences directory. This happens when the R2006b preferences directory is deleted. MATLAB then looks for the R2006a preferences directory, and if found, migrates the R2006a preferences

to the R2006b preferences. If it does not find the R2006a preferences directory, it uses the default preferences for R2006b. Similarly, upon starting MATLAB 7.2 (R2006a), it looks for the R2006a preferences directory, and if not found creates it and migrates preferences from R14SP3 if it finds that preferences directory. The process also applies when MATLAB 7.1 (R14SP3) starts.

- If you want to use default preferences for R2006b, and do not want MATLAB to migrate preferences from R2006a, the R2006b preferences directory must exist but be empty when you start MATLAB. If you want to maintain some of your R2006b preferences, but restore the defaults for others, in the R2006b preferences directory, delete the files for which you want the defaults to be restored. One file you might want to maintain is `history.m`—for more information about the file, see “Viewing Statements in the Command History Window” in the MATLAB Desktop Tools and Development Environment documentation.

Examples

Run

```
prefdir
```

MATLAB returns

```
ans =
```

```
C:\WINNT\Profiles\tbear.MATHWORKS  
\Application Data\MathWorks\MATLAB\R2006b
```

Running `dir` for the directory shows these files

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

and possibly other files for MATLAB and other MathWorks products.

prefdir

In MATLAB, run `cd(prefdir)` to change to that directory.

On Windows platforms, go directly to the preferences directory in Explorer by running `winopen(prefdir)`.

See Also

`preferences`, `winopen`

Fonts, Colors, and Other Preferences in the MATLAB Desktop Tools and Development Environment documentation

Purpose	Open Preferences dialog box for MATLAB and related products
GUI Alternatives	As an alternative to the preferences function, select File > Preferences in the MATLAB desktop or any desktop tool.
Syntax	<code>preferences</code> <code>preferences -nodesktop</code>
Description	<p><code>preferences</code> displays the Preferences dialog box, from which you can make changes to options for MATLAB and related products.</p> <p><code>preferences -nodesktop</code> displays a special preferences dialog box you use to change the Command Window font and color after having started MATLAB with the <code>-nodesktop</code> option. Use <code>preferences</code> (with no option) to set other available Command Window preferences. Any font and color changes you make using <code>preferences</code> (with no option) do not apply to the Command Window.</p>
See Also	<code>prefdir</code> Fonts, Colors, and Other Preferences in the MATLAB Desktop Tools and Development Environment documentation

primes

Purpose Generate list of prime numbers

Syntax `p = primes(n)`

Description `p = primes(n)` returns a row vector of the prime numbers less than or equal to `n`. A prime number is one that has no factors other than 1 and itself.

Examples `p = primes(37)`

```
p = 2 3 5 7 11 13 17 19 23 29 31 37
```

See Also `factor`

Purpose	Print figure or save to file and configure printer defaults
GUI Alternative	Use File → Print on the figure window menu to access the Print dialog and File → Print Preview to access the Print Preview GUI. For details, see How to Print or Export in the MATLAB Graphics documentation.
Syntax	<pre>print print filename print -ddriver print -dformat print -dformat filename print -smodelname print -options print(...) [pcmd,dev] = printopt</pre>
Description	<p><code>print</code> and <code>printopt</code> produce hardcopy output. All arguments to the <code>print</code> command are optional. You can use them in any combination or order.</p> <p><code>print</code> sends the contents of the current figure, including bitmap representations of any user interface controls, to the printer using the device and system printing command defined by <code>printopt</code>.</p> <p><code>print filename</code> directs the output to the PostScript file designated by <code>filename</code>. If <code>filename</code> does not include an extension, <code>print</code> appends an appropriate extension.</p> <p><code>print -ddriver</code> prints the figure using the specified printer <i>driver</i>, (such as color PostScript). If you omit <code>-ddriver</code>, <code>print</code> uses the default value stored in <code>printopt.m</code>. The Printer Driver table lists all supported device types.</p> <p><code>print -dformat</code> copies the figure to the system clipboard (Windows only). A valid <i>format</i> for this operation is either <code>-dmeta</code> (Windows Enhanced Metafile) or <code>-dbitmap</code> (Windows Bitmap).</p>

print, printopt

`print -dformat filename` exports the figure to the specified file using the specified graphics *format*, (such as TIFF). The Graphics Format table lists all supported graphics file formats.

`print -smodelname` prints the current Simulink model *modelname*.

`print -options` specifies print options that modify the action of the `print` command. (For example, the `noui` option suppresses printing of user interface controls.) The Options section lists available options.

`print(...)` is the function form of `print`. It enables you to pass variables for any input arguments. This form is useful for passing filenames and handles. See Batch Processing for an example.

`[pcmd,dev] = printopt` returns strings containing the current system-dependent printing command and output device. `printopt` is an M-file used by `print` to produce the hardcopy output. You can edit the M-file `printopt.m` to set your default printer type and destination.

`pcmd` and `dev` are platform-dependent strings. `pcmd` contains the command that `print` uses to send a file to the printer. `dev` contains the printer driver or graphics format option for the `print` command. Their defaults are platform dependent.

Platform	System Printing Command	Driver or Format
UNIX	<code>lpr -r</code>	<code>dps2</code>
Windows	<code>COPY /B %s LPT1:</code>	<code>dwin</code>

Drivers

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

```
print -d
```

at the MATLAB prompt.

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

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

Printer Driver	PRINT Command Option String	Ghostscript
Canon BubbleJet BJ10e	-dbj10e	Yes
Canon BubbleJet BJ200 color	-dbj200	Yes
Canon Color BubbleJet BJC-70/BJC-600/BJC-4000	-dbjc600	Yes
Canon Color BubbleJet BJC-800	-dbjc800	Yes
DEC LN03	-dln03	Yes
Epson and compatible 9- or 24-pin dot matrix print drivers	-depson	Yes
Epson and compatible 9-pin with interleaved lines (triple resolution)	-deps9high	Yes
Epson LQ-2550 and compatible; color (not supported on HP-700)	-depsonc	Yes
Fujitsu 3400/2400/1200	-depsonc	Yes
HP DesignJet 650C color (not supported on Windows)	-ddnj650c	Yes
HP DeskJet 500	-ddjet500	Yes

print, printopt

Printer Driver	PRINT Command Option String	Ghostscript
HP DeskJet 500C (creates black and white output)	-dcdjmono	Yes
HP DeskJet 500C (with 24 bit/pixel color and high-quality Floyd-Steinberg color dithering) (not supported on Windows)	-dcdjcolor	Yes
HP DeskJet 500C/540C color (not supported on Windows)	-dcdj500	Yes
HP Deskjet 550C color (not supported on Windows)	-dcdj550	Yes
HP DeskJet and DeskJet Plus	-ddeskjet	Yes
HP LaserJet	-dlaserjet	Yes
HP LaserJet+	-dljetplus	Yes
HP LaserJet IIP	-dljet2p	Yes
HP LaserJet III	-dljet3	Yes
HP LaserJet 4.5L and 5P	-dljet4	Yes
HP LaserJet 5 and 6	-dpxlmono	Yes
HP PaintJet color	-dpaintjet	Yes
HP PaintJet XL color	-dpjxl	Yes
HP PaintJet XL color	-dpjetxl	Yes

Printer Driver	PRINT Command Option String	Ghostscript
HP PaintJet XL300 color (not supported on Windows)	-dpjx1300	Yes
HPGL for HP 7475A and other compatible plotters. (Renderer cannot be set to Z-buffer.)	-dhpgl	No
IBM 9-pin Proprinter	-dibmpro	Yes
PostScript black and white	-dps	No
PostScript color	-dpsc	No
PostScript Level 2 black and white	-dps2	No
PostScript Level 2 color	-dpsc2	No
Windows color (Windows only)	-dwinc	No
Windows monochrome (Windows only)	-dwin	No

Note Generally, Level 2 PostScript files are smaller and are rendered more quickly when printing than Level 1 PostScript files. However, not all PostScript printers support Level 2, so determine the capabilities of your printer before using those drivers. Level 2 PostScript is the default for UNIX. You can change this default by editing the `printopt.m` file. Likewise, if you want color PostScript to be the default instead of black-and-white PostScript, edit the line in the `printopt.m` file that reads `dev = '-dps2';` to be `dev = '-dpsc2';`

Graphics Format Files

To save your figure as a graphics-format file, specify a format switch and filename. To set the resolution of the output file for a built-in MATLAB format, use the `-r` switch. (For example, `-r300` sets the output resolution to 300 dots per inch.) The `-r` switch is also supported for Windows Enhanced Metafiles, JPEG, and PNG files, but is not supported for Ghostscript formats.

The table below shows the supported output formats for exporting from MATLAB and the switch settings to use. In some cases, a format is available both as a MATLAB output filter and as a Ghostscript output filter. All formats except for EMF are supported on both the PC and UNIX platforms.

Graphics Format	Bitmap or Vector	PRINT Command Option String	MATLAB or Ghostscript
BMP monochrome BMP	Bitmap	-dbmpmono	Ghostscript
BMP 24-bit BMP	Bitmap	-dbmp16m	Ghostscript
BMP 8-bit (256-color) BMP (this format uses a fixed colormap)	Bitmap	-dbmp256	Ghostscript
BMP 24-bit	Bitmap	-dbmp	MATLAB
EMF	Vector	-dmeta	MATLAB
EPS black and white	Vector	-deps	MATLAB
EPS color	Vector	-depsc	MATLAB
EPS Level 2 black and white	Vector	-deps2	MATLAB
EPS Level 2 color	Vector	-depsc2	MATLAB
HDF 24-bit	Bitmap	-dhdf	MATLAB

Graphics Format	Bitmap or Vector	PRINT Command Option String	MATLAB or Ghostscript
ILL (Adobe Illustrator)	Vector	-dill	MATLAB
JPEG 24-bit	Bitmap	-djpeg	MATLAB
PBM (plain format) 1-bit	Bitmap	-dpbm	Ghostscript
PBM (raw format) 1-bit	Bitmap	-dpbmraw	Ghostscript
PCX 1-bit	Bitmap	-dpcxmono	Ghostscript
PCX 24-bit color PCX file format, three 8-bit planes	Bitmap	-dpcx24b	Ghostscript
PCX 8-bit newer color PCX file format (256-color)	Bitmap	-dpcx256	Ghostscript
PCX Older color PCX file format (EGA/VGA, 16-color)	Bitmap	-dpcx16	Ghostscript
PCX 8-bit	Bitmap	-dpcx	MATLAB
PDF Color PDF file format	Vector	-dpdf	Ghostscript
PGM Portable Graymap (plain format)	Bitmap	-dpgm	Ghostscript
PGM Portable Graymap (raw format)	Bitmap	-dpgmraw	Ghostscript

print, printopt

Graphics Format	Bitmap or Vector	PRINT Command Option String	MATLAB or Ghostscript
PNG 24-bit	Bitmap	-dpng	MATLAB
PPM Portable Pixmap (plain format)	Bitmap	-dppm	Ghostscript
PPM Portable Pixmap (raw format)	Bitmap	-dppmraw	Ghostscript
TIFF 24-bit	Bitmap	-dtiff or -dtiffn	MATLAB
TIFF preview for EPS files	Bitmap	-tiff	

The TIFF image format is supported on all platforms by almost all word processors for importing images. JPEG is a lossy, highly compressed format that is supported on all platforms for image processing and for inclusion into HTML documents on the World Wide Web. To create these formats, MATLAB renders the figure using the Z-buffer rendering method and the resulting bitmap is then saved to the specified file.

Options

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

Option	Description
-adobecset	PostScript only. Use PostScript default character set encoding. See Early PostScript 1 Printers.
-append	PostScript only. Append figure to existing PostScript file. See Settings That Are Driver Specific.

Option	Description
-cmyk	PostScript only. Print with CMYK colors instead of RGB. See Setting CMYK Color.
-ddriver	Printing only. Printer driver to use. See Drivers table.
-dformat	Exporting only. Graphics format to use. See Graphics Format Files table.
-dsetup	Display the Print Setup dialog.
-fhandle	Handle of figure to print. Note that you cannot specify both this option and the <i>-windowtitle</i> option. See Which Figure Is Printed.
-loose	PostScript and Ghostscript only. Use loose bounding box for PostScript. See Producing Uncropped Figures.
-noui	Suppress printing of user interface controls. See Excluding User Interface Controls.
-opengl	Render using the OpenGL algorithm. Note that you cannot specify this method in conjunction with <i>-zbuffer</i> or <i>-painters</i> . See Selecting a Renderer.
-painters	Render using the Painter's algorithm. Note that you cannot specify this method in conjunction with <i>-zbuffer</i> or <i>-opengl</i> . See Selecting a Renderer.
-Pprinter	Specify name of printer to use. See Selecting Printer.

print, printopt

Option	Description
-rnumber	PostScript, JPEG, PNG, and Ghostscript only. Specify resolution in dots per inch. Defaults to 90 for Simulink, 150 for figures in image formats and when printing in Z-buffer or OpenGL mode, screen resolution for metafiles, and 864 otherwise. Use -r0 to specify screen resolution. See Setting the Resolution.
-swindowtitle	Specify name of Simulink system window to print. Note that you cannot specify both this option and the -fhandle option. See Which Figure Is Printed.
-v	Windows only. Display the Windows Print dialog box. The v stands for verbose mode.
-zbuffer	Render using the Z-buffer algorithm. Note that you cannot specify this method in conjunction with -opengl or -painters. See Selecting a Renderer.

Paper Sizes

MATLAB supports a number of standard paper sizes. You can select from the following list by setting the PaperType property of the figure or selecting a supported paper size from the Print dialog box.

Property Value	Size (Width by Height)
usletter	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

Property Value	Size (Width by 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

Printing Tips

This section includes information about specific printing issues.

Figures with Resize Functions

The print command produces a warning when you print a figure having a callback routine defined for the figure `ResizeFcn`. To avoid the warning, set the figure `PaperPositionMode` property to `auto` or select **Match Figure Screen Size** in the **File->Page Setup** dialog box.

Troubleshooting MS Windows Printing

If you encounter problems such as segmentation violations, general protection faults, or application errors, or the output does not appear as you expect when using MS-Windows printer drivers, try the following:

- If your printer is PostScript compatible, print with one of the MATLAB built-in PostScript drivers. There are various PostScript device options that you can use with the print command: they all start with `-dps`.
- The behavior you are experiencing might occur only with certain versions of the print driver. Contact the print driver vendor for information on how to obtain and install a different driver.
- Try printing with one of the MATLAB built-in Ghostscript devices. These devices use Ghostscript to convert PostScript files into other formats, such as HP LaserJet, PCX, Canon BubbleJet, and so on.
- Copy the figure as a Windows Enhanced Metafile using the **Edit**→**Copy Figure** menu item on the figure window menu or the `print -dmeta` option at the command line. You can then import the file into another application for printing.

You can set copy options in the figure's **File**→**Preferences**→**Copying Options** dialog box. The Windows Enhanced Metafile clipboard format produces a better quality image than Windows Bitmap.

Printing MATLAB GUIs

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

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

To set `PaperPositionMode` on the current figure, use the command

```
set(gcf, 'PaperPositionMode', 'auto')
```

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

To set `InvertHardcopy` on the current figure, use the command

```
set(gcf, 'InvertHardcopy', 'off')
```

- Use a color device if you want lines and text that are in color on the screen to be written to the output file as colored objects. Black and white devices convert colored lines and text to black or white to provide the best contrast with the background and to avoid dithering.
- Use the print command's `-loose` option to prevent MATLAB from using a bounding box that is tightly wrapped around objects contained in the figure. This is important if you have intentionally used space between uicontrols or axes and the edge of the figure and you want to maintain this appearance in the printed output.

Notes on Printing Interpolated Shading with PostScript Drivers

MATLAB can print surface objects (such as graphs created with `surf` or `mesh`) using interpolated colors. However, only patch objects that are composed of triangular faces can be printed using interpolated shading.

Printed output is always interpolated in RGB space, not in the colormap colors. This means that if you are using indexed color and interpolated face coloring, the printed output can look different from what is displayed on screen.

PostScript files generated for interpolated shading contain the color information of the graphics object's vertices and require the printer to perform the interpolation calculations. This can take an excessive amount of time and in some cases, printers might time out before

finishing the print job. One solution to this problem is to interpolate the data and generate a greater number of faces, which can then be flat shaded.

To ensure that the printed output matches what you see on the screen, print using the `-zbuffer` option. To obtain higher resolution (for example, to make text look better), use the `-r` option to increase the resolution. There is, however, a tradeoff between the resolution and the size of the created PostScript file, which can be quite large at higher resolutions. The default resolution of 150 dpi generally produces good results. You can reduce the size of the output file by making the figure smaller before printing it and setting the figure `PaperPositionMode` to `auto`, or by just setting the `PaperPosition` property to a smaller size.

Examples

Specifying the Figure to Print

You can print a noncurrent figure by specifying the figure's handle. If a figure has the title Figure 2, its handle is 2. The syntax is

```
print -fhandle
```

This example prints the figure whose handle is 2, regardless of which figure is the current figure.

```
print -f2
```

Note You must use the `-f` option if the figure's handle is hidden (i.e., its `HandleVisibility` property is set to `off`).

This example saves the figure with the handle `-f2` to a PostScript file named `Figure2`, which can be printed later.

```
print -f2 -dps 'Figure2.ps'
```

If the figure uses noninteger handles, use the `figure` command to get its value, and then pass it in as the first argument.

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

You can also pass a figure handle as a variable to the function form of `print`. For example,

```
h = figure; plot(1:4,5:8)
print(h)
```

This example uses the function form of `print` to enable a filename to be passed in as a variable.

```
filename = 'mydata';
print('-f3', '-dp3c', filename);
```

(Because a filename is specified, the figure will be printed to a file.)

Specifying the Model to Print

To print a noncurrent Simulink model, use the `-s` option with the title of the window. For example, this command prints the Simulink window titled `f14`.

```
print -sf14
```

If the window title includes any spaces, you must call the function form rather than the command form of `print`. For example, this command saves Simulink window title `Thruster Control`.

```
print('-sThruster Control')
```

To print the current system, use

```
print -s
```

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

Printing Figures at Screen Size

This example prints a surface plot with interpolated shading. Setting the current figure's (gcf) PaperPositionMode to auto enables you to resize the figure window and print it at the size you see on the screen. See Options and the previous section for information on the -zbuffer and -r200 options.

```
surf(peaks)
shading interp
set(gcf,'PaperPositionMode','auto')
print -dpsc2 -zbuffer -r200
```

For additional details, see Printing Images in the MATLAB Graphics documentation.

Batch Processing

You can use the function form of print to pass variables containing file names. For example, this for loop uses filenames stored in a cell array to create a series of graphs and prints each one with a different file name.

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

Tiff Preview

The command

```
print -depsec -tiff -r300 picture1
```

saves the current figure at 300 dpi, in a color Encapsulated PostScript file named picture1.eps. The -tiff option creates a 72 dpi TIFF preview, which many word processor applications can display on screen after you import the EPS file. This enables you to view the picture

on screen within your word processor and print the document to a PostScript printer using a resolution of 300 dpi.

See Also

orient, figure

printdlg

Purpose Print dialog box

Syntax

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

Description printdlg prints the current figure.

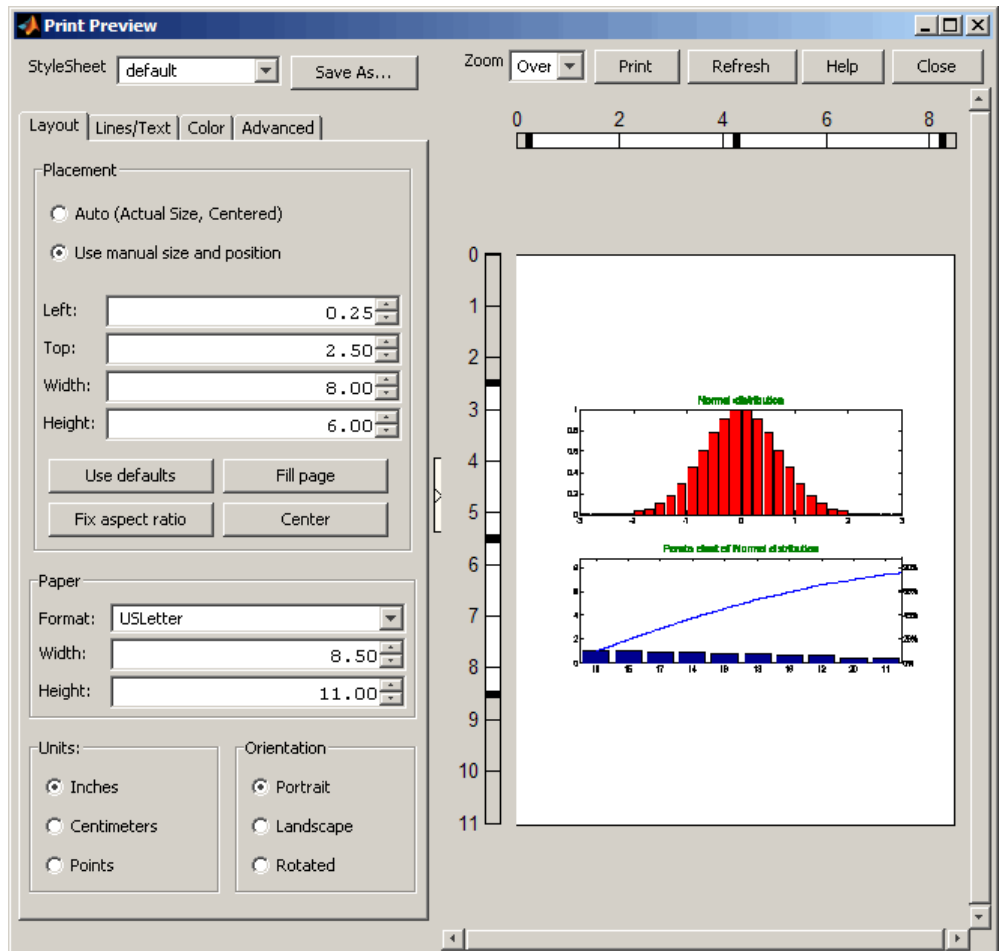
printdlg(fig) creates a dialog box from which you can print the figure window identified by the handle fig. Note that uimenu do not print.

printdlg('-crossplatform',fig) displays the standard cross-platform MATLAB printing dialog rather than the built-in printing dialog box for Microsoft Windows computers. Insert this option before the fig argument.

printdlg('-setup',fig) forces the printing dialog to appear in a setup mode. Here one can set the default printing options without actually printing.

See Also pagesetupdlg, printpreview

Purpose	Preview figure to print
GUI Alternative	Use File > Print Preview on the figure window menu to access the Print Preview dialog box, described below. For details, see “Using Print Preview” in the MATLAB Graphics documentation.
Syntax	<code>printpreview</code> <code>printpreview(f)</code>
Description	<p><code>printpreview</code> displays a dialog box showing the figure in the currently active figure window as it will print. A scaled version of the figure displays in the right-hand pane of the GUI.</p> <p><code>printpreview(f)</code> displays a dialog box showing the figure having the handle <code>f</code> as it will print.</p> <p>Use the Print Preview dialog box, shown below, to control the layout and appearance of figures before sending them to a printer or print file. Controls are grouped into four tabbed panes: Layout, Lines/Text, Color, and Advanced.</p>



Right Pane Controls

You can position and scale plots on the printed page using the rulers in the right-hand pane of the Print Preview dialog. Use the outer ruler handlebars to change margins. Moving them changes plot proportions. Use the center ruler handlebars to change the position of the plot on the page. Plot proportions do not change, but you can move portions of

the plot off the paper. The buttons on that pane let you refresh the plot, close the dialog (preserving all current settings), print the page immediately, or obtain context-sensitive help. Use the Zoom box and scroll bars to view and position page elements more precisely.

The Layout Tab

Use the **Layout** tab, shown above, to control the paper format and placement of the plot on printed pages. The following table summarizes the **Layout** options:

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

Group	Option	Description
Orientation	Portrait	Upright paper orientation
	Landscape	Sideways paper orientation

The Lines/Text Tab

Use the **Lines/Text** tab, shown below, to control the line weights, font characteristics, and headers for printed pages. The following table summarizes the **Lines/Text** options:

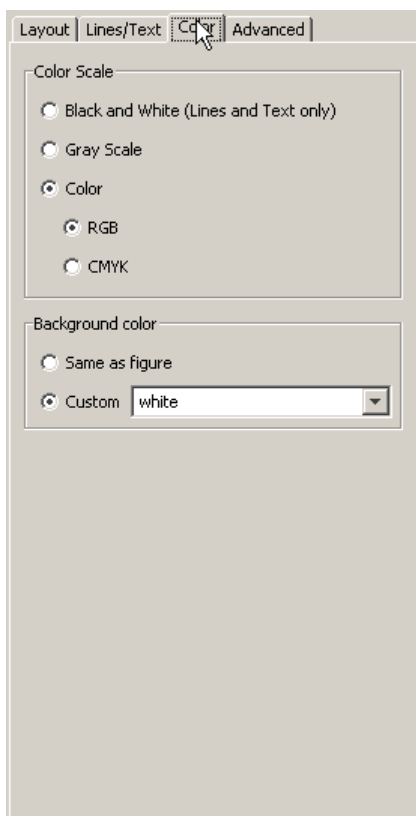
The screenshot shows a dialog box with four tabs: "Layout", "Lines/Text", "Color", and "Advanced". The "Lines/Text" tab is selected. It is divided into three sections: "Lines", "Text", and "Header".

- Lines Section:**
 - Line Width:** Radio buttons for "Default", "Scale By" (with a text box containing "0" and a "%" symbol), and "Custom" (with a text box containing "0.5" and a "points" label).
 - Min Width:** Radio buttons for "Default" and "Custom" (with an empty text box).
- Text Section:**
 - Font Name:** Radio buttons for "Default" and "Custom" (with a dropdown menu showing "Helvetica").
 - Font Size:** Radio buttons for "Default", "Scale By" (with a text box containing "0" and a "%" symbol), and "Custom" (with a dropdown menu showing "10" and a "points" label).
 - Font Weight:** A dropdown menu showing "Default".
 - Font Angle:** A dropdown menu showing "Default".
- Header Section:**
 - Header Text:** A text input field.
 - Font...:** A button next to the header text field.
 - Date Style:** A dropdown menu showing "none".

Group	Option	Description
Lines	Line Width	Scale all lines by a percentage from 0 upwards (100 being no change), print lines at a specified point size, or default line widths used on the plot
	Min Width	Smallest line width (in points) to use when printing; defaults to 0.5 point
Text	Font Name	Select a system font for all text on plot, or default to fonts currently used on the plot
	Font Size	Scale all text by a percentage from 0 upwards (100 being no change), print text at a specified point size, or default to this
	Font Weight	Select Normal ... Bold font styling for all text from drop-down menu or default to the font weights used on the plot
	Font Angle	Select Normal, Italic or Oblique font styling for all text from drop-down menu or default to the font angles used on the plot
Header	Header Text	Type the text to appear on the header at the upper left of printed pages, or leave blank for no header
	Date Style	Select a date format to have today's date appear at the upper left of printed pages, or none for no date

The Color Tab

Use the **Color** tab, shown below, to control how colors are printed for lines and backgrounds. The following table summarizes the **Color** options:

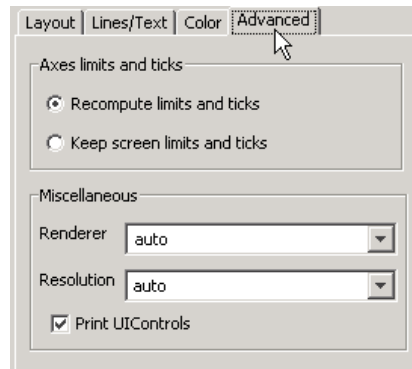


Group	Option	Description
Color Scale	Black and White	Select to print lines and text in black and white, but use color for patches and other objects
	Gray Scale	Convert colors to shades of gray on printed pages

Group	Option	Description
	Color	Print everything in color, matching colors on plot; select RGB (default) or CMYK color model for printing
Background Color	Same as figure	Print the figure's background color as it is
	Custom	Select a color name, or type a colorspec for the background; white (default) implies no background color, even on colored paper.

The Advanced Tab

Use the **Advanced** tab, shown below, to control finer details of printing, such as limits and ticks, renderer, resolution, and the printing of UIControls. The following table summarizes the **Advanced** options:



Group	Option	Description
Axes limits and ticks	Recompute limits and ticks	Redraw x - and y -axes ticks and limits based on printed plot size (default)

printpreview

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

See Also

printdlg, pagesetupdlg

For more information, see [How to Print or Export](#) in the MATLAB Graphics documentation.

Purpose

Product of array elements

Syntax

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

Description

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

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

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

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

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

Examples

The magic square of order 3 is

```
M = magic(3)
```

```
M =
     8     1     6
     3     5     7
     4     9     2
```

The product of the elements in each column is

```
prod(M) =
     96     45     84
```

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

```
prod(M,2) =
     48
```

prod

105
72

See Also

cumprod, diff, sum

Purpose	Profile execution time for function
GUI Alternatives	As an alternative to the <code>profile</code> function, select Desktop > Profiler to open the Profiler.
Syntax	<pre>profile on profile on -detail level profile on -history profile on -nohistory profile on -timer clock profile on -detail level -history -timer clock profile off profile resume profile clear profile viewer S = profile('status') stats = profile('info')</pre>
Description	<p>The <code>profile</code> function helps you debug and optimize M-files by tracking their execution time. For each function in the M-file, <code>profile</code> records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. Some people use <code>profile</code> simply to see the child functions; see also <code>depfun</code> for that purpose. To open the Profiler graphical user interface, use the <code>profile viewer</code> syntax. Profile time is CPU time. The total time reported by the Profiler is not the same as the time reported using the <code>tic</code> and <code>toc</code> functions or the time you would observe using a stopwatch. To change options, stop profiling and then start or resume profiling with new options.</p> <p><code>profile on</code> starts the Profiler, clearing previously recorded profile statistics.</p> <p><code>profile on -detail level</code> starts the Profiler, clearing previously recorded profile statistics, and specifies the set of functions you want to profile. The level applies to subsequent uses of <code>profile</code> or the Profiler, until you change it. Allowable values for <i>level</i> are</p>

profile

- `'builtin'`—Gathers information about M-functions, M-subfunctions, and MEX-functions, plus built-in functions, such as `eig`.
- `'mmex'`—Gathers information about M-functions, M-subfunctions, and MEX-functions. This is the default value.

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

`profile on -nohistory` starts the Profiler, clearing previously recorded profile statistics, and disables further recording of the history (exact sequence of function calls). Use the `-nohistory` option after having previously set the `-history` option. All other profiling statistics continue to accumulate.

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

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

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

`profile on -detail level -history -timer clock` starts the Profiler using all of these specified options. Any order is acceptable, as is a subset.

`profile off` stops the Profiler.

`profile resume` restarts the Profiler without clearing previously recorded statistics.

`profile clear` clears the statistics recorded by `profile`.

`profile viewer` stops the Profiler and displays the results in the Profiler window. For more information, see *Profiling for Improving Performance in the Desktop Tools and Development Environment* documentation.

`S = profile('status')` returns a structure containing information about the current status of the Profiler. The table lists the fields in the order they appear in the structure.

Field	Values
ProfilerStatus	'on' or 'off'
DetailLevel	'mmex' or 'builtin'
Timer	'cpu' or 'real'
HistoryTracking	'on' or 'off'

`stats = profile('info')` stops the Profiler and displays a structure containing the results. Use this function to access the data generated by `profile`. The table lists the fields in the order they appear in the structure.

Field	Description
FunctionTable	Structure array containing statistics about each function called
FunctionHistory	Array containing function call history
ClockPrecision	Precision of <code>profile</code> 's time measurement
ClockSpeed	Estimated clock speed of the CPU
Name	Name of the profiler

The `FunctionTable` field is an array of structures, where each structure contains information about one of the functions or subfunctions called during execution. The following table lists these fields in the order they appear in the structure.

Field	Description
CompleteName	Full path to FunctionName, including subfunctions
FunctionName	Function name; includes subfunctions
FileName	Full path to FunctionName, with file extension, excluding subfunctions
Type	M-functions, MEX-functions, and many other types of functions including M-subfunctions, nested functions, and anonymous functions
NumCalls	Number of times the function was called
TotalTime	Total time spent in the function and its child functions
TotalRecursiveTime	No longer used.
Children	FunctionTable indices to child functions
Parents	FunctionTable indices to parent functions
ExecutedLines	<p>Array containing line-by-line details for the function being profiled.</p> <p>Column 1: Number of the line that executed. If a line was not executed, it does not appear in this matrix.</p> <p>Column 2: Number of times the line was executed</p> <p>Column 3: Total time spent on that line. Note: The sum of Column 3 entries does not necessarily add up to the function's TotalTime.</p>

Field	Description
IsRecursive	BOOLEAN value: Logical 1 (true) if recursive, otherwise logical 0 (false)
PartialData	BOOLEAN value: Logical 1 (true) if function was modified during profiling, for example by being edited or cleared. In that event, data was collected only up until the point the function was modified.

Examples

Profile and Display Results

This example profiles the MATLAB magic command and then displays the results in the Profiler window. The example then retrieves the profile data on which the HTML display is based and uses the `profsave` command to save the profile data in HTML form.

```
profile on
plot(magic(35))
profile viewer
p = profile('info');
profsave(p,'profile_results')
```

Profile and Save Results

Another way to save profile data is to store it in a MAT-file. This example stores the profile data in a MAT-file, clears the profile data from memory, and then loads the profile data from the MAT-file. This example also shows a way to bring the reloaded profile data into the Profiler graphical interface as live profile data, not as a static HTML page.

```
p = profile('info');
save myprofiledata p
clear p
load myprofiledata
profview(0,p)
```

Profile and Show Results Including History

This example illustrates an effective way to view the results of profiling when the history option is enabled. The history data describes the sequence of functions entered and exited during execution. The `profile` command returns history data in the `FunctionHistory` field of the structure it returns. The history data is a 2-by-n array. The first row contains Boolean values, where 1 means entrance into a function and 0 means exit from a function. The second row identifies the function being entered or exited by its index in the `FunctionTable` field. This example reads the history data and displays it in the MATLAB Command Window.

```
profile on -history
plot(magic(4));
p = profile('info');

for n = 1:size(p.FunctionHistory,2)
    if p.FunctionHistory(1,n)==0
        str = 'entering function: ';
    else
        str = 'exiting function: ';
    end
    disp([str p.FunctionTable(p.FunctionHistory(2,n)).FunctionName])
end
```

See Also

`depsdir`, `depfun`, `mlint`, `profsave`

Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation

Purpose Save profile report in HTML format

Syntax

```
profsave
profsave(profinfo)
profsave(profinfo,dirname)
```

Description profsave executes the `profile('info')` function and saves the results in HTML format. profsave creates a separate HTML file for each function listed in the `FunctionTable` field of the structure returned by `profile`. By default, profsave stores the HTML files in a subdirectory of the current directory named `profile_results`.

`profsave(profinfo)` saves the profiling results, `profinfo`, in HTML format. `profinfo` is a structure of profiling information returned by the `profile('info')` function.

`profsave(profinfo,dirname)` saves the profiling results, `profinfo`, in HTML format. profsave creates a separate HTML file for each function listed in the `FunctionTable` field of `profinfo` and stores them in the directory specified by `dirname`.

Examples Run profile and save the results.

```
profile on
plot(magic(5))
profile off
profsave(profile('info'),'myprofile_results')
```

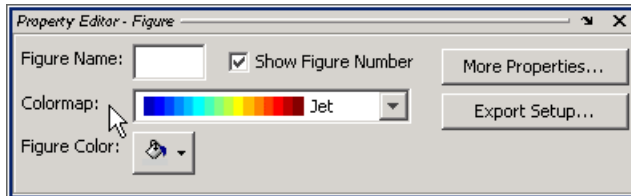
See Also `profile`

Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation

propedit

Purpose

Open Property Editor



Syntax

```
propedit  
propedit(handle_list)
```

Description

`propedit` starts the Property Editor, a graphical user interface to the properties of graphics objects. If no current figure exists, `propedit` will create one.

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

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

See Also

`inspect`, `plottedit`, `propertyeditor`

Purpose Open built-in property page for control

Syntax `h.propedit`
`propedit(h)`

Description `h.propedit` requests the control to display its built-in property page. Note that some controls do not have a built-in property page. For those controls, this command fails.

`propedit(h)` is an alternate syntax for the same operation.

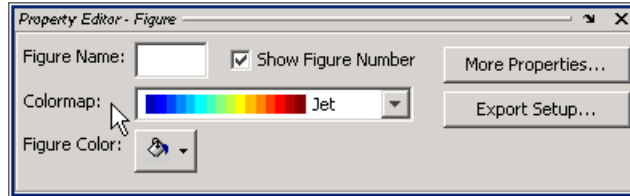
Examples Create a Microsoft Calendar control and display its property page:

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



See Also `inspect`, `get`

propertyeditor

Purpose Show or hide property editor



GUI Alternatives

Click the larger Plotting Tools icon  on the figure toolbar to collectively enable plotting tools, and the smaller icon  to collectively disable them. Open or close the **Property Editor** tool from the figure's **View** menu. For details, see “The Property Editor” in the MATLAB Graphics documentation.

Syntax

```
propertyeditor('on')
propertyeditor('off')
propertyeditor('toggle')
propertyeditor(figure_handle,...)
```

Description

`propertyeditor('on')` displays the property editor on the current figure.

`propertyeditor('off')` hides the property editor on the current figure.

`propertyeditor('toggle')` or `propertyeditor` toggles the visibility of the property editor on the current figure.

`propertyeditor(figure_handle,...)` displays or hides the property editor on the figure specified by `figure_handle`.

See Also

`plottools`, `plotbrowser`, `figurepalette`, `inspect`

Purpose

Psi (polygamma) function

Syntax

```
Y = psi(X)
Y = psi(k,X)
Y = psi(k0:k1,X)
```

Description

$Y = \text{psi}(X)$ evaluates the Ψ function for each element of array X . X must be real and nonnegative. The Ψ function, also known as the digamma function, is the logarithmic derivative of the gamma function

$$\begin{aligned}\psi(x) &= \text{digamma}(x) \\ &= \frac{d(\log(\Gamma(x)))}{dx} \\ &= \frac{d(\Gamma(x))/dx}{\Gamma(x)}\end{aligned}$$

$Y = \text{psi}(k,X)$ evaluates the k th derivative of Ψ at the elements of X . $\text{psi}(0,X)$ is the digamma function, $\text{psi}(1,X)$ is the trigamma function, $\text{psi}(2,X)$ is the tetragamma function, etc.

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

Examples**Example 1**

Use the psi function to calculate Euler's constant, γ .

```
format long
-psi(1)
ans =
    0.57721566490153

-psi(0,1)
ans =
    0.57721566490153
```

Example 2

The trigamma function of 2, $\text{psi}(1,2)$, is the same as $(\pi^2/6) - 1$.

```
format long
psi(1,2)
ans =
    0.64493406684823
```

```
pi^2/6 - 1
ans =
    0.64493406684823
```

Example 3

This code produces the first page of Table 6.1 in Abramowitz and Stegun [1].

```
x = (1:.005:1.250)';
[x gamma(x) gammaln(x) psi(0:1,x)' x-1]
```

Example 4

This code produces a portion of Table 6.2 in [1].

```
psi(2:3,1:.01:2)'
```

See Also

gamma, gammainc, gammaln

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Sections 6.3 and 6.4.

Purpose Run M-file script containing cells, saving results to file of specified type

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

Syntax

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

Description `publish('script')` runs the file named `script` in the base workspace one cell at a time, and saves the code, comments, and results to an HTML output file. The output file is named `script.html` and is stored, along with other supporting output files, in an `html` subdirectory in `script`'s directory.

`publish('script','format')` runs the file named `script`, one cell at a time in the base workspace, and publishes the code, comments, and results to an output file using the specified *format*. Allowable values for *format* are `html` (the default), `xml`, `latex` for LaTeX, `doc` for Microsoft Word documents, and `ppt` for Microsoft PowerPoint documents. The output file is named `script.format` and is stored, along with other supporting output files, in an `html` subdirectory in `script`'s directory. The `doc` format requires Microsoft Word, and the `ppt` format requires PowerPoint. When publishing to HTML, the M-file code is included at the end of published HTML file as comments, even when the `showCode` option is set to `false`. Use the `grabcode` function to extract the code from the HTML file.

`publish('script',options)` publishes using the structure *options*, which can contain any of the following fields and corresponding value for each field. Create and save structures for the options you use regularly. For details about the values, see the description of the corresponding Editor/Debugger preferences for Publishing and Publishing Images preferences in the online documentation for MATLAB.

Field	Allowable Values
format	'doc', 'html' (default), 'latex', 'ppt', 'xml'
stylesheet	' ' (default), XSL filename (used only when format is html, latex, or xml)
outputDir	' ' (default, a subfolder named html), full pathname
imageFormat	'png' (default unless format is latex), 'eps2' (default when format is latex), any format supported by print when figureSnapMethod is print, any format supported by imwrite functions when figureSnapMethod is getframe.
figureSnapMethod	'print' (default), 'getframe'
useNewFigure	true (default), false
maxHeight	[] (default), positive integer specifying number of pixels
maxWidth	[] (default), positive integer specifying number of pixels
showCode	true (default), false
evalCode	true (default), false
catchError	true (default, continues publishing and includes the error in the published file), false (displays the error and publishing ends)
stopOnError	true (default), false
createThumbnail	true (default), false

Examples

Publish to HTML Format

To publish the file `d:/mymfiles/sine_wave.m` to HTML, run

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

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

Publish with Options

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

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

See Also

`grabcode`, `notebook`

MATLAB Desktop Tools and Development Environment documentation, specifically

- Publishing to HTML, XML, LaTeX, Word, and PowerPoint Using Cells
- Defining Cells

PutCharArray

Purpose Store character array in server

Syntax **MATLAB Client**
h.PutCharArray('varname', 'workspace', 'string')
PutCharArray(h, 'varname', 'workspace', 'string')
invoke(h, 'PutCharArray', 'varname', 'workspace', 'string')

Method Signature
PutCharArray([in] BSTR varname, [in] BSTR workspace,
[in] BSTR string)

Visual Basic Client
PutCharArray(varname As String, workspace As String,
string As String)

Description PutCharArray stores the character array in string in the specified workspace of the server attached to handle h, assigning to it the variable varname. The workspace argument can be either base or global.

Remarks The character array specified in the string argument can have any dimensions. However, PutCharArray changes the dimensions to a 1-by-n column-wise representation, where n is the number of characters in the array. Executing the following commands in MATLAB illustrates this behavior:

```
h = actxserver('matlab.application');  
chArr = ['abc'; 'def'; 'ghk']  
chArr =  
abc  
def  
ghk  
  
h.PutCharArray('Foo', 'base', chArr)  
tstArr = h.GetCharArray('Foo', 'base')  
tstArr =  
adgbehcfk
```

Server function names, like `PutCharArray`, are case sensitive when using the dot notation syntax shown in the [Syntax](#) section.

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

Examples

Store string `str` in the base workspace of the server using `PutCharArray`. Retrieve the string with `GetCharArray`.

MATLAB Client

```
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

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

See Also

[GetCharArray](#), [PutWorkspaceData](#), [GetWorkspaceData](#), [Execute](#)

PutFullMatrix

Purpose Store matrix in server

Syntax

MATLAB Client

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

Method Signature

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

Visual Basic Client

```
PutFullMatrix([in] varname As String, [in] workspace As String,
[in] xreal As Double, [in] ximag As Double)
```

Description

PutFullMatrix stores a matrix in the specified workspace of the server attached to handle h, assigning to it the variable varname. Enter the real and imaginary parts of the matrix in the xreal and ximag input arguments. The workspace argument can be either base or global.

Remarks

The matrix specified in the xreal and ximag arguments cannot be scalar, an empty array, or have more than two dimensions.

Server function names, like PutFullMatrix, are case sensitive when using the first syntax shown.

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

For VBScript clients, use the GetWorkspaceData and PutWorkspaceData functions to pass numeric data to and from the MATLAB workspace. These functions use the variant data type instead of safearray which is not supported by VBScript.

Examples

Example 1 – Writing to the Base Workspace

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

MATLAB Client

```
h = actxserver('matlab.application');
h.PutFullMatrix('M', 'base', rand(5), zeros(5))
% One output returns real, use two for real and imag
xreal = h.GetFullMatrix('M', 'base', zeros(5), zeros(5))
xreal =
    0.9501    0.7621    0.6154    0.4057    0.0579
    0.2311    0.4565    0.7919    0.9355    0.3529
    0.6068    0.0185    0.9218    0.9169    0.8132
    0.4860    0.8214    0.7382    0.4103    0.0099
    0.8913    0.4447    0.1763    0.8936    0.1389
```

Visual Basic.net Client

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

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

Matlab = CreateObject(matlab.application)
MatLab.PutFullMatrix(M, base, XReal, XImag)
MatLab.GetFullMatrix(M, base, ZReal, ZImag)
```

PutFullMatrix

Example 2 – Writing to the Global Workspace

Write a matrix to the global workspace of the server and then examine the server's global workspace from the client.

MATLAB Client

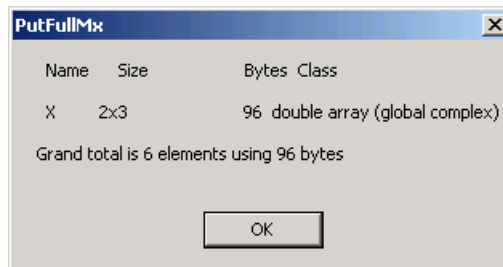
```
h = actxserver('matlab.application');
h.PutFullMatrix('X', 'global', [1 3 5; 2 4 6], [1 1 1; 1 1 1])
h.invoke('Execute', 'whos global')
ans =
    Name      Size      Bytes Class
    X          2x3         96 double array (global complex)
Grand total is 6 elements using 96 bytes
```

Visual Basic.net Client

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

For i = 0 To 1
    For j = 0 To 2
        XReal(i, j) = (j * 2 + 1) + i
        XImag(i, j) = 1
    Next j
Next i

Matlab = CreateObject(matlab.application)
MatLab.PutFullMatrix(M, global, XReal, XImag)
result = Matlab.Execute(whos global)
MsgBox(result)
```

See Also

GetFullMatrix, PutWorkspaceData, , GetWorkspaceDataExecute

PutWorkspaceData

Purpose Store data in server workspace

Syntax **MATLAB Client**
h.PutWorkspaceData('varname', 'workspace', data)
PutWorkspaceData(h, 'varname', 'workspace', data)
invoke(h, 'PutWorkspaceData', 'varname', 'workspace', data)

Method Signature
PutWorkspaceData([in] BSTR varname, [in] BSTR workspace,
[in] VARIANT data)

Visual Basic Client
PutWorkspaceData(varname As String, workspace As String,
data As Object)

Description PutWorkspaceData stores data in the specified workspace of the server attached to handle h, assigning to it the variable varname. The workspace argument can be either base or global.

Note PutWorkspaceData works on all MATLAB data types except sparse arrays, structure arrays, and function handles. Use the Execute method for these data types.

Passing Character Arrays

MATLAB enables you to define 2-D character arrays such as the following:

```
chArr = ['abc'; 'def'; 'ghk']  
chArr =  
abc  
def  
ghk  
  
size(chArr)  
ans =
```

3 3

However, PutWorkspaceData does not preserve the dimensions of character arrays when passing them to a COM server. 2-D arrays are converted to 1-by-n arrays of characters, where n equals the number of characters in the original array plus one newline character for each row in the original array. This means that chArr above is converted to a 1-by-12 array, but the newline characters make it display with three rows in the MATLAB command window. For example,

```
h = actxserver('matlab.application');
h.PutWorkspaceData('Foo','base',chArr);
tstArr = h.GetWorkspaceData('Foo','base')
tstArr =
abc
def
ghk

size(tstArr)
ans =
     1     12
```

Remarks

You can use PutWorkspaceData in place of PutFullMatrix and PutCharArray to pass numeric and character array data respectively to the server.

Server function names, like PutWorkspaceData, are case sensitive when using the first syntax shown.

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

The GetWorkspaceData and PutWorkspaceData functions pass numeric data as a variant data type. These functions are especially useful for VBScript clients as VBScript does not support the safearray data type used by GetFullMatrix and PutFullMatrix.

Examples

Create an array in the client and assign it to variable A in the base workspace of the server:

PutWorkspaceData

MATLAB Client

```
h = actxserver('matlab.application');
for i = 0:6
    data(i+1) = i * 15;
end
h.PutWorkspaceData('A', 'base', data)
```

Visual Basic.net Client

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

See Also


GetWorkspaceData, PutFullMatrix, , GetFullMatrix, PutCharArray, GetCharArrayExecute

See “Executing Commands in the MATLAB Server” for more examples.

Purpose

Identify current directory

Graphical Interface

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

Syntax

```
pwd  
s = pwd
```

Description

`pwd` displays the current working directory.

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

On Windows platforms, go directly to the current working directory using

```
winopen(pwd)
```

See Also

`cd`, `dir`, `fileparts`, `mfilename`, `path`, `what`, `winopen`

Purpose

Quasi-minimal residual method

Syntax

```
x = qmr(A,b)
qmr(A,b,tol)
qmr(A,b,tol,maxit)
qmr(A,b,tol,maxit,M)
qmr(A,b,tol,maxit,M1,M2)
qmr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = qmr(A,b,...)
[x,flag,relres] = qmr(A,b,...)
[x,flag,relres,iter] = qmr(A,b,...)
[x,flag,relres,iter,resvec] = qmr(A,b,...)
```

Description

`x = qmr(A,b)` attempts to solve the system of linear equations $A^*x=b$ for x . The n -by- n coefficient matrix A must be square and should be large and sparse. The column vector b must have length n . A can be a function handle `afun` such that `afun(x, 'notransp')` returns A^*x and `afun(x, 'transp')` returns A'^*x . See “Function Handles” in the MATLAB Programming documentation for more information.

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

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

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

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

`qmr(A,b,tol,maxit,M)` and `qmr(A,b,tol,maxit,M1,M2)` use preconditioners M or $M = M1*M2$ and effectively solve the system

$\text{inv}(M)*A*x = \text{inv}(M)*b$ for x . If M is $[]$ then `qmr` applies no preconditioner. M can be a function handle `mfun` such that `mfun(x, 'notransp')` returns $M \setminus x$ and `mfun(x, 'transp')` returns $M' \setminus x$.

`qmr(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If x_0 is $[]$, then `qmr` uses the default, an all zero vector.

`[x,flag] = qmr(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	qmr converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.
1	qmr iterated <code>maxit</code> times but did not converge.
2	Preconditioner M was ill-conditioned.
3	The method stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during <code>qmr</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] = qmr(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] = qmr(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] = qmr(A,b,...)` also returns a vector of the residual norms at each iteration, including $\text{norm}(b-A*x_0)$.

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 = qmr(A,b,tol,maxit,M1,M2);
```

displays the message

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

Example 2

This example replaces the matrix A in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in an M-file `run_qmr` that

- Calls `qmr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_qmr` are available to `afun`.

The following shows the code for `run_qmr`:

```
function x1 = run_qmr
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = qmr(@afun,b,tol,maxit,M1,M2);

function y = afun(x,transp_flag)
    if strcmp(transp_flag,'transp')
        % y = A'*x
```



```

        y = 4 * x;
        y(1:n-1) = y(1:n-1) - 2 * x(2:n);
        y(2:n) = y(2:n) - x(1:n-1);
    elseif strcmp(transp_flag,'notransp') % y = A*x
        y = 4 * x;
        y(2:n) = y(2:n) - 2 * x(1:n-1);
        y(1:n-1) = y(1:n-1) - x(2:n);
    end
end
end
end

```

When you enter

```
x1=run_qmr;
```

MATLAB displays the message

```

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

```

Example 3

```

load west0479;
A = west0479;
b = sum(A,2);
[x,flag] = qmr(A,b)

```

flag is 1 because qmr does not converge to the default tolerance $1e-6$ within the default 20 iterations.

```

[L1,U1] = luinc(A,1e-5);
[x1,flag1] = qmr(A,b,1e-6,20,L1,U1)

```

flag1 is 2 because the upper triangular U1 has a zero on its diagonal, and qmr fails in the first iteration when it tries to solve a system such as $U1*y = r$ for y using backslash.

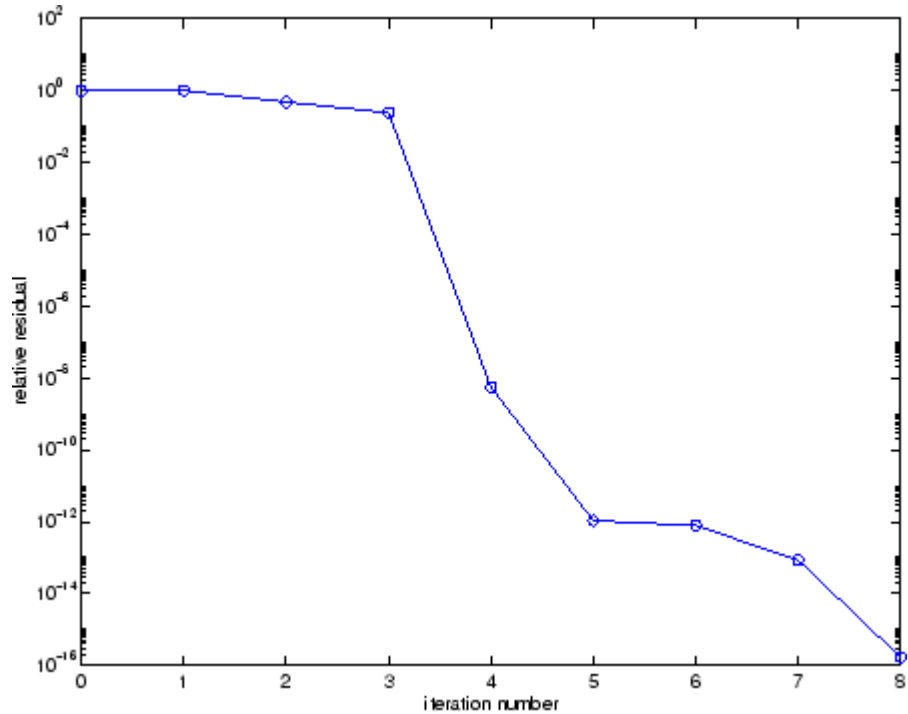
```

[L2,U2] = luinc(A,1e-6);
[x2,flag2,relres2,iter2,resvec2] = qmr(A,b,1e-15,10,L2,U2)

```

flag2 is 0 because qmr converges to the tolerance of $1.6571e-016$ (the value of relres2) at the eighth iteration (the value of iter2) when preconditioned by the incomplete LU factorization with a drop tolerance of $1e-6$. $\text{resvec2}(1) = \text{norm}(b)$ and $\text{resvec2}(9) = \text{norm}(b-A*x2)$. You can follow the progress of qmr by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:iter2,resvec2/norm(b),'-o')  
xlabel('iteration number')  
ylabel('relative residual')
```



See Also

bicg, bicgstab, cgs, gmres, lsqr, luinc, minres, pcg, symmlq,
function_handle(@), mldivide(\)

References

- [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] Freund, Roland W. and Noël M. Nachtigal, "QMR: A quasi-minimal residual method for non-Hermitian linear systems," *SIAM Journal: Numer. Math.* 60, 1991, pp. 315-339.

Purpose Orthogonal-triangular decomposition

Syntax

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

Description orthonormal or

The `qr` function performs the orthogonal-triangular decomposition of a matrix. This factorization is useful for both square and rectangular matrices. It expresses the matrix as the product of a real complex unitary matrix and an upper triangular matrix.

$[Q,R] = \text{qr}(A)$ produces an upper triangular matrix R of the same dimension as A and a unitary matrix Q so that $A = Q^*R$. For sparse matrices, Q is often nearly full. If $[m \ n] = \text{size}(A)$, then Q is m -by- m and R is m -by- n .

$[Q,R] = \text{qr}(A,0)$ produces an “economy-size” decomposition. If $[m \ n] = \text{size}(A)$, and $m > n$, then `qr` computes only the first n columns of Q and R is n -by- n . If $m \leq n$, it is the same as $[Q,R] = \text{qr}(A)$.

$[Q,R,E] = \text{qr}(A)$ for full matrix A , produces a permutation matrix E , an upper triangular matrix R with decreasing diagonal elements, and a unitary matrix Q so that $A^*E = Q^*R$. The column permutation E is chosen so that $\text{abs}(\text{diag}(R))$ is decreasing.

$[Q,R,E] = \text{qr}(A,0)$ for full matrix A , produces an “economy-size” decomposition in which E is a permutation vector, so that $A(:,E) = Q^*R$. The column permutation E is chosen so that $\text{abs}(\text{diag}(R))$ is decreasing.

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

$R = \text{qr}(A)$ for sparse matrix A , produces only an upper triangular matrix, R . The matrix R provides a Cholesky factorization for the matrix associated with the normal equations,

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

This approach avoids the loss of numerical information inherent in the computation of $A' * A$. It may be preferred to $[Q, R] = \text{qr}(A)$ since Q is always nearly full.

$[C, R] = \text{qr}(A, B)$ for sparse matrix A , applies the orthogonal transformations to B , producing $C = Q' * B$ without computing Q . B and A must have the same number of rows.

$R = \text{qr}(A, 0)$ and $[C, R] = \text{qr}(A, B, 0)$ for sparse matrix A , produce “economy-size” results.

For sparse matrices, the Q-less QR factorization allows the solution of sparse least squares problems

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

with two steps

$$\begin{aligned} [C, R] &= \text{qr}(A, b) \\ x &= R \backslash c \end{aligned}$$

If A is sparse but not square, MATLAB uses the two steps above for the linear equation solving backslash operator, i.e., $x = A \backslash b$.

Examples

Example 1

Start with

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

This is a rank-deficient matrix; the middle column is the average of the other two columns. The rank deficiency is revealed by the factorization:

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

Q =

-0.0776	-0.8331	0.5444	0.0605
-0.3105	-0.4512	-0.7709	0.3251
-0.5433	-0.0694	-0.0913	-0.8317
-0.7762	0.3124	0.3178	0.4461

R =

-12.8841	-14.5916	-16.2992
0	-1.0413	-2.0826
0	0	0.0000
0	0	0

The triangular structure of R gives it zeros below the diagonal; the zero on the diagonal in R(3,3) implies that R, and consequently A, does not have full rank.

Example 2

This example uses matrix A from the first example. The QR factorization is used to solve linear systems with more equations than unknowns. For example, let

$$b = [1;3;5;7]$$

The linear system $Ax = b$ represents four equations in only three unknowns. The best solution in a least squares sense is computed by

$$x = A \backslash b$$

which produces

Warning: Rank deficient, rank = 2, tol = 1.4594E-014

```
x =
    0.5000
         0
    0.1667
```

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

```
tol = max(size(A))*eps*abs(R(1,1))
```

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

```
y = Q'*b;
x = R\y
```

The computed solution can be checked by forming Ax . This equals b to within roundoff error, which indicates that even though the simultaneous equations $Ax = b$ are overdetermined and rank deficient, they happen to be consistent. There are infinitely many solution vectors x ; the QR factorization has found just one of them.

Algorithm

Inputs of Type Double

For inputs of type double, `qr` uses the LAPACK routines listed in the following table to compute the QR decomposition.

Syntax	Real	Complex
<code>X = qr(A)</code> <code>X = qr(A,0)</code>	DGEQRF	ZGEQRF
<code>[Q,R] = qr(A)</code> <code>[Q,R] = qr(A,0)</code>	DGEQRF, DORGQR	ZGEQRF, ZUNGQR
<code>[Q,R,e] = qr(A)</code> <code>[Q,R,e] = qr(A,0)</code>	DGEQP3, DORGQR	ZGEQP3, ZUNGQR

Inputs of Type Single

For inputs of type `single`, `qr` uses the LAPACK routines listed in the following table to compute the QR decomposition.

Syntax	Real	Complex
$R = qr(A)$ $R = qr(A, 0)$	SGEQRF	CGEQRF
$[Q, R] = qr(A)$ $[Q, R] = qr(A, 0)$	SGEQRF, SORGQR	CGEQRF, CUNGQR
$[Q, R, e] = qr(A)$ $[Q, R, e] = qr(A, 0)$	SGEQP3, SORGQR	CGEQP3, CUNGQR

See Also

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

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), Third Edition, SIAM, Philadelphia, 1999.

Purpose Remove column or row from QR factorization

Syntax

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

Description

`[Q1,R1] = qrdelete(Q,R,j)` returns the QR factorization of the matrix `A1`, where `A1` is `A` with the column `A(:,j)` removed and `[Q,R] = qr(A)` is the QR factorization of `A`.

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

`[Q1,R1] = qrdelete(Q,R,j,'row')` returns the QR factorization of the matrix `A1`, where `A1` is `A` with the row `A(j,:)` removed and `[Q,R] = qr(A)` is the QR factorization of `A`.

Examples

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

```
Q1 =
    0.5274   -0.5197   -0.6697   -0.0578
    0.7135    0.6911    0.0158    0.1142
    0.3102   -0.1982    0.4675   -0.8037
    0.3413   -0.4616    0.5768    0.5811
```

```
R1 =
   32.2335   26.0908   19.9482   21.4063   23.3297
         0  -19.7045  -10.9891    0.4318   -1.4873
         0         0   22.7444    5.8357   -3.1977
         0         0         0  -14.5784    3.7796
```

returns a valid QR factorization, although possibly different from

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

qrdelete

```
Q2 =
-0.5274    0.5197    0.6697   -0.0578
-0.7135   -0.6911   -0.0158    0.1142
-0.3102    0.1982   -0.4675   -0.8037
-0.3413    0.4616   -0.5768    0.5811
```

```
R2 =
-32.2335  -26.0908  -19.9482  -21.4063  -23.3297
         0   19.7045   10.9891   -0.4318    1.4873
         0         0  -22.7444   -5.8357    3.1977
         0         0         0  -14.5784    3.7796
```

Algorithm

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

See Also

`planerot`, `qr`, `qrinsert`

Purpose Insert column or row into QR factorization

Syntax

```
[Q1,R1] = qrinsert(Q,R,j,x)
[Q1,R1] = qrinsert(Q,R,j,x,'col')
[Q1,R1] = qrinsert(Q,R,j,x,'row')
```

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

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

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

Examples

```
A = magic(5);
[Q,R] = qr(A);
j = 3;
x = 1:5;
[Q1,R1] = qrinsert(Q,R,j,x,'row')
```

```
Q1 =
    0.5231    0.5039   -0.6750    0.1205    0.0411    0.0225
    0.7078   -0.6966    0.0190   -0.0788    0.0833   -0.0150
    0.0308    0.0592    0.0656    0.1169    0.1527   -0.9769
    0.1231    0.1363    0.3542    0.6222    0.6398    0.2104
    0.3077    0.1902    0.4100    0.4161   -0.7264   -0.0150
    0.3385    0.4500    0.4961   -0.6366    0.1761    0.0225
```

```
R1 =
   32.4962   26.6801   21.4795   23.8182   26.0031
         0   19.9292   12.4403    2.1340    4.3271
         0         0   24.4514   11.8132    3.9931
         0         0         0   20.2382   10.3392
```

qrinsert

```
0      0      0      0  16.1948
0      0      0      0      0
```

returns a valid QR factorization, although possibly different from

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

```
Q2 =
-0.5231    0.5039    0.6750   -0.1205    0.0411    0.0225
-0.7078   -0.6966   -0.0190    0.0788    0.0833   -0.0150
-0.0308    0.0592   -0.0656   -0.1169    0.1527   -0.9769
-0.1231    0.1363   -0.3542   -0.6222    0.6398    0.2104
-0.3077    0.1902   -0.4100   -0.4161   -0.7264   -0.0150
-0.3385    0.4500   -0.4961    0.6366    0.1761    0.0225
```

```
R2 =
-32.4962  -26.6801  -21.4795  -23.8182  -26.0031
         0   19.9292   12.4403    2.1340    4.3271
         0         0  -24.4514  -11.8132   -3.9931
         0         0         0  -20.2382  -10.3392
         0         0         0         0   16.1948
         0         0         0         0         0
```

Algorithm

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

See Also

`planerot`, `qr`, `qrdelete`

Description Rank 1 update to QR factorization

Syntax `[Q1,R1] = qrupdate(Q,R,u,v)`

Description `[Q1,R1] = qrupdate(Q,R,u,v)` when `[Q,R] = qr(A)` is the original QR factorization of A , returns the QR factorization of $A + u \cdot v'$, where u and v are column vectors of appropriate lengths.

Remarks `qrupdate` works only for full matrices.

Examples The matrix

```
mu = sqrt(eps)
```

```
mu =
```

```
1.4901e-08
```

```
A = [ones(1,4); mu*eye(4)];
```

is a well-known example in least squares that indicates the dangers of forming $A' \cdot A$. Instead, we work with the QR factorization – orthonormal Q and upper triangular R .

```
[Q,R] = qr(A);
```

As we expect, R is upper triangular.

```
R =
```

```
-1.0000    -1.0000    -1.0000    -1.0000
         0     0.0000     0.0000     0.0000
         0         0     0.0000     0.0000
         0         0         0     0.0000
         0         0         0         0
```

qrupdate

In this case, the upper triangular entries of R, excluding the first row, are on the order of $\sqrt{\text{eps}}$.

Consider the update vectors

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

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

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

QT =

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \end{bmatrix}$$

RT =

1.0e-007 *

$$\begin{bmatrix} -0.1490 & 0 & 0 & 0 & 0 \\ 0 & -0.1490 & 0 & 0 & 0 \\ 0 & 0 & -0.1490 & 0 & 0 \\ 0 & 0 & 0 & -0.1490 & 0 \\ 0 & 0 & 0 & 0 & -0.1490 \end{bmatrix}$$

we may use qrupdate.

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

Q1 =

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

```

0.0000    1.0000   -0.0000   -0.0000    0.0000
0.0000    0.0000    1.0000   -0.0000    0.0000
-0.0000   -0.0000   -0.0000    1.0000    0.0000

```

R1 =

```

1.0e-007 *
0.1490    0.0000    0.0000    0.0000
0         0.1490    0.0000    0.0000
0         0         0.1490    0.0000
0         0         0         0.1490
0         0         0         0

```

Note that both factorizations are correct, even though they are different.

Algorithm

qrupdate uses the algorithm in section 12.5.1 of the third edition of *Matrix Computations* by Golub and van Loan. qrupdate is useful since, if we take $N = \max(m, n)$, then computing the new QR factorization from scratch is roughly an $O(N^3)$ algorithm, while simply updating the existing factors in this way is an $O(N^2)$ algorithm.

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

See Also

cholupdate, qr

quad

Purpose Numerically evaluate integral, adaptive Simpson quadrature

Syntax

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

Description *Quadrature* is a numerical method used to find the area under the graph of a function, that is, to compute a definite integral.

$$q = \int_a^b f(x) dx$$

`q = quad(fun,a,b)` tries to approximate the integral of function `fun` from `a` to `b` to within an error of $1e-6$ using recursive adaptive Simpson quadrature. `fun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. Limits `a` and `b` must be finite. The function `y = fun(x)` should accept a vector argument `x` and return a vector result `y`, the integrand evaluated at each element of `x`.

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

`q = quad(fun,a,b,tol)` uses an absolute error tolerance `tol` instead of the default which is $1.0e-6$. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results. In MATLAB version 5.3 and earlier, the `quad` function used a less reliable algorithm and a default relative tolerance of $1.0e-3$.

`q = quad(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q]` during the recursion.

`[q,fcnt] = quad(...)` returns the number of function evaluations.

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

Example

To compute the integral

$$\int_0^2 \frac{1}{x^3 - 2x - 5} dx$$

write an M-file function myfun that computes the integrand:

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

Then pass @myfun, a function handle to myfun, to quad, along with the limits of integration, 0 to 2:

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

```
Q =
```

```
-0.4605
```

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

```
F = @(x)1./(x.^3-2*x-5);
Q = quad(F,0,2);
```

Algorithm

quad implements a low order method using an adaptive recursive Simpson's rule.

Diagnostics

quad may issue one of the following warnings:

'Minimum step size reached' indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

'Maximum function count exceeded' indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

quad

'Infinite or Not-a-Number function value encountered' indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

See Also

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

References

[1] Gander, W. and W. Gautschi, "Adaptive Quadrature – Revisited," BIT, Vol. 40, 2000, pp. 84-101. This document is also available at <http://www.inf.ethz.ch/personal/gander>.

Purpose

Numerically evaluate integral, adaptive Lobatto quadrature

Syntax

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

Description

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

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

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

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

`[q,fcnt] = quadl(...)` returns the number of function evaluations.

Use array operators `.*`, `./` and `.^` in the definition of `fun` so that it can be evaluated with a vector argument.

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

Examples

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

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

where the M-file `myfun.m` is

quadl

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

Pass anonymous function handle F to quadl:

```
F = @(x) 1./(x.^3-2*x-5);
Q = quadl(F,0,2);
```

Algorithm

quadl implements a high order method using an adaptive Gauss/Lobatto quadrature rule.

Diagnostics

quadl may issue one of the following warnings:

'Minimum step size reached' indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

'Maximum function count exceeded' indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

'Infinite or Not-a-Number function value encountered' indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

See Also

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

References

[1] Gander, W. and W. Gautschi, “Adaptive Quadrature – Revisited,” BIT, Vol. 40, 2000, pp. 84-101. This document is also available at <http://www.inf.ethz.ch/personal/gander>.

Purpose

Vectorized quadrature

Syntax

```
Q = quadv(fun,a,b)
Q = quadv(fun,a,b,tol)
Q = quadv(fun,a,b,tol,trace)
[Q,fcnt] = quadv(...)
```

Description

`Q = quadv(fun,a,b)` approximates the integral of the complex array-valued function `fun` from `a` to `b` to within an error of $1.e-6$ using recursive adaptive Simpson quadrature. `fun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. The function $Y = \text{fun}(x)$ should accept a scalar argument `x` and return an array result `Y`, whose components are the integrands evaluated at `x`. Limits `a` and `b` must be finite.

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

`Q = quadv(fun,a,b,tol)` uses the absolute error tolerance `tol` for all the integrals instead of the default, which is $1.e-6$.

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

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

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

Example

For the parameterized array-valued function `myarrayfun`, defined by

```
function Y = myarrayfun(x,n)
Y = 1./((1:n)+x);
```

quadv

the following command integrates `myarrayfun`, for the parameter value `n = 10` between `a = 0` and `b = 1`:

```
Qv = quadv(@(x)myarrayfun(x,10),0,1);
```

The resulting array `Qv` has 10 elements estimating $Q(k) = \log((k+1)./(k))$, for `k = 1:10`.

The entries in `Qv` are slightly different than if you compute the integrals using `quad` in a loop:

```
for k = 1:10
    Qs(k) = quad(@(x)myscalarfun(x,k),0,1);
end
```

where `myscalarfun` is:

```
function y = myscalarfun(x,k)
    y = 1./(k+x);
```

See Also

`quad`, `quadl`, `dblquad`, `triplequad`, `function_handle` (@)

Purpose

Create and open question dialog box

Syntax

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

Description

`button = questdlg('qstring')` displays a modal dialog presenting the question 'qstring'. The dialog has three default buttons, **Yes**, **No**, and **Cancel**. If the user presses one of these three buttons, `button` is set to the name of the button pressed. If the user presses the close button on the dialog, `button` is set to the empty string. If the user presses the **Return** key, `button` is set to 'Yes'. 'qstring' is a cell array or a string that automatically wraps to fit within the dialog box.

`button = questdlg('qstring','title')` displays a question dialog with 'title' displayed in the dialog's title bar.

`button = questdlg('qstring','title','default')` specifies which push button is the default in the event that the **Return** key is pressed. 'default' must be 'Yes', 'No', or 'Cancel'.

`button = questdlg('qstring','title','str1','str2','default')` creates a question dialog box with two push buttons labeled 'str1' and 'str2'. 'default' specifies the default button selection and must be 'str1' or 'str2'.

`button = questdlg('qstring','title','str1','str2','str3','default')` creates a question dialog box with three push buttons labeled 'str1', 'str2', and 'str3'. 'default' specifies the default button selection and must be 'str1', 'str2', or 'str3'.

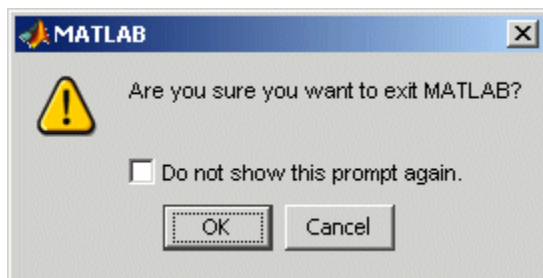
In all cases where 'default' is specified, if 'default' is not set to one of the button names, pressing the **Enter** key displays a warning and the dialog remains open.

questdlg

See Also

dialog, errordlg, helpdlg, inputdlg, listdlg, msgbox, warndlg
figure, textwrap, uiwait, uiresume
“Predefined Dialog Boxes” on page 1-100 for related functions

Purpose	Terminate MATLAB
GUI Alternatives	As an alternative to the quit function, use the Close box or select File > Exit MATLAB in the MATLAB desktop.
Syntax	<pre>quit quit cancel quit force</pre>
Description	<p>quit displays a confirmation dialog box if the confirm upon quitting preference is selected, and if confirmed or if the confirmation preference is not selected, terminates MATLAB after running <code>finish.m</code>, if <code>finish.m</code> exists. The workspace is not automatically saved by quit. To save the workspace or perform other actions when quitting, create a <code>finish.m</code> file to perform those actions. For example, you can display a custom dialog box to confirm quitting using a <code>finish.m</code> file—see the following examples for details. If an error occurs while <code>finish.m</code> is running, quit is canceled so that you can correct your <code>finish.m</code> file without losing your workspace.</p> <p>quit cancel is for use in <code>finish.m</code> and cancels quitting. It has no effect anywhere else.</p> <p>quit force bypasses <code>finish.m</code> and terminates MATLAB. Use this to override <code>finish.m</code>, for example, if an errant <code>finish.m</code> will not let you quit.</p>
Remarks	<p>When using Handle Graphics in <code>finish.m</code>, use <code>uiwait</code>, <code>waitfor</code>, or <code>drawnow</code> so that figures are visible. See the reference pages for these functions for more information.</p> <p>If you want MATLAB to display the following confirmation dialog box after running quit, select File > Preferences > General > Confirmation Dialogs. Then select the check box for Confirm before exiting MATLAB, and click OK.</p>



Examples

Two sample `finish.m` files are included with MATLAB. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` to use it.

- `finishesav.m`—Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m`—Displays a dialog allowing you to cancel quitting; it uses `quit cancel` and contains the following code:

```
button = questdlg('Ready to quit?', ...  
                 'Exit Dialog','Yes','No','No');  
switch button  
    case 'Yes',  
        disp('Exiting MATLAB');  
        %Save variables to matlab.mat  
        save  
    case 'No',  
        quit cancel;  
end
```

See Also

`exit`, `finish`, `save`, `startup`

Purpose Terminate MATLAB server

Syntax

MATLAB Client
h.Quit
Quit(h)
invoke(h, 'Quit')

Method Signature
void Quit(void)

Visual Basic Client
Quit

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

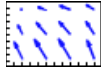
Remarks Server function names, like Quit, are case sensitive when using the first syntax shown.

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


quiver

Purpose

Quiver or velocity plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
quiver(x,y,u,v)
quiver(u,v)
quiver(...,scale)
quiver(...,LineStyle)
quiver(...,LineStyle,'filled')
quiver(axes_handles,...)
h = quiver(...)
hlines = quiver('v6',...)
```

Description

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

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

`quiver(x,y,u,v)` plots vectors as arrows at the coordinates specified in each corresponding pair of elements in x and y . The matrices x , y , u , and v must all be the same size and contain corresponding position and velocity components. However, x and y can also be vectors, as explained in the next section. By default, the arrows are scaled to just not overlap, but you can scale them to be longer or shorter if you want.

Expanding x- and y-Coordinates

MATLAB expands x and y if they are not matrices. This expansion is equivalent to calling `meshgrid` to generate matrices from vectors:

```
[x,y] = meshgrid(x,y);
quiver(x,y,u,v)
```


In this case, the following must be true:

`length(x) = n` and `length(y) = m`, where `[m,n] = size(u) = size(v)`.

The vector `x` corresponds to the columns of `u` and `v`, and vector `y` corresponds to the rows of `u` and `v`.

`quiver(u,v)` draws vectors specified by `u` and `v` at equally spaced points in the x - y plane.

`quiver(...,scale)` automatically scales the arrows to fit within the grid and then stretches them by the factor `scale`. `scale = 2` doubles their relative length and `scale = 0.5` halves the length. Use `scale = 0` to plot the velocity vectors without automatic scaling. You can also tune the length of arrows after they have been drawn by choosing the plot

edit  tool, selecting the `quivergroup` object, opening the Property Editor, and adjusting the **Length** slider.

`quiver(...,LineStyle)` specifies line style, marker symbol, and color using any valid `LineStyle`. `quiver` draws the markers at the origin of the vectors.

`quiver(...,LineStyle,'filled')` fills markers specified by `LineStyle`.

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

`h = quiver(...)` returns the handle to the `quivergroup` object.

Backward-Compatible Version

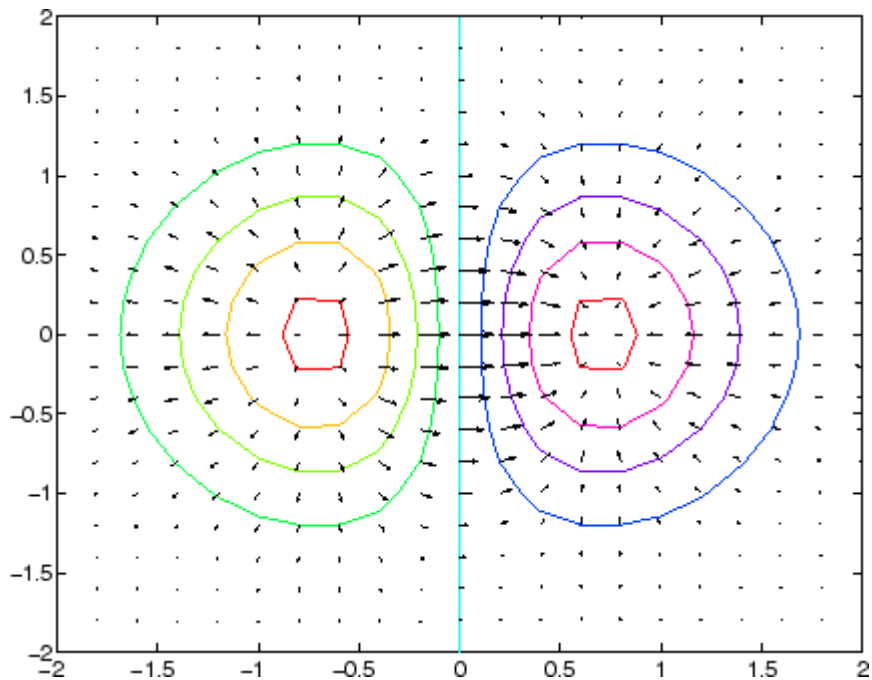
`hlines = quiver('v6',...)` returns the handles of line objects instead of `quivergroup` objects for compatibility with MATLAB 6.5 and earlier.

Examples

Showing the Gradient with Quiver Plots

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

```
[X,Y] = meshgrid(-2:.2:2);  
Z = X.*exp(-X.^2 - Y.^2);  
[DX,DY] = gradient(Z,.2,.2);  
contour(X,Y,Z)  
hold on  
quiver(X,Y,DX,DY)  
colormap hsv  
hold off
```



See Also

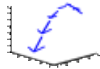
contour, LineSpec, plot, quiver3

“Direction and Velocity Plots” on page 1-85 for related functions


Two-Dimensional Quiver Plots for more examples

See Quivergroup Properties for property descriptions

Purpose 3-D quiver or velocity plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
quiver3(x,y,z,u,v,w)
quiver3(z,u,v,w)
quiver3(...,scale)
quiver3(...,LineStyle)
quiver3(...,LineStyle,'filled')
quiver3(axes_handles,...)
h = quiver3(...)
```

Description

A three-dimensional quiver plot displays vectors with components (u,v,w) at the points (x,y,z) .

`quiver3(x,y,z,u,v,w)` plots vectors with components (u,v,w) at the points (x,y,z) . The matrices x,y,z,u,v,w must all be the same size and contain the corresponding position and vector components.

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

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

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

quiver3

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

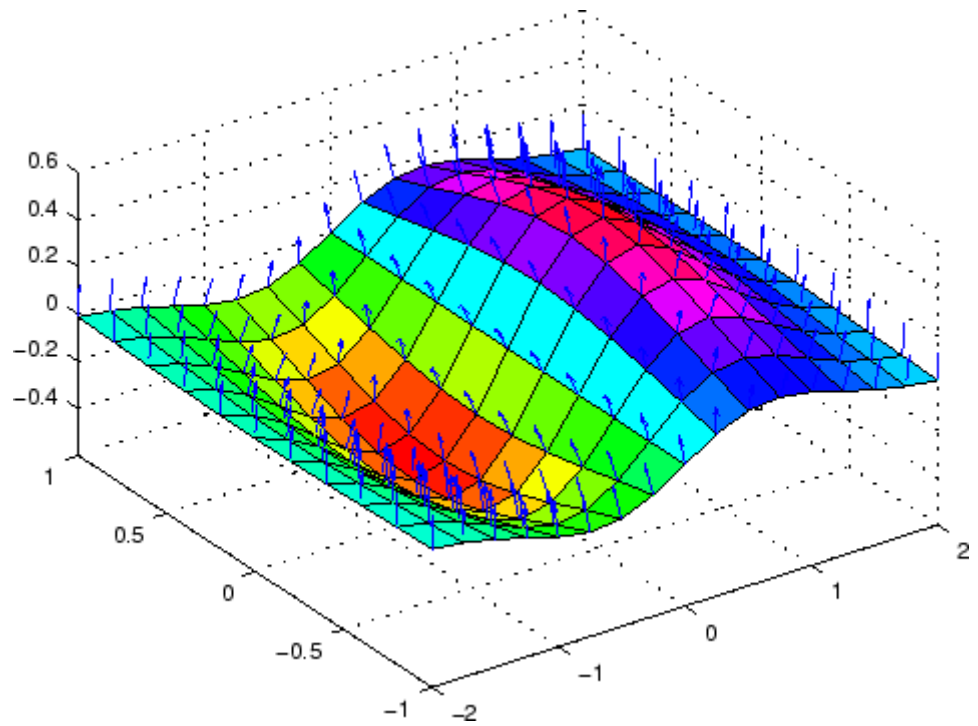
`quiver3(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

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

Examples

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

```
[X,Y] = meshgrid(-2:0.25:2,-1:0.2:1);
Z = X.* exp(-X.^2 - Y.^2);
[U,V,W] = surfnorm(X,Y,Z);
quiver3(X,Y,Z,U,V,W,0.5);
hold on
surf(X,Y,Z);
colormap hsv
view(-35,45)
axis ([-2 2 -1 1 -.6 .6])
hold off
```


**See Also**

axis, contour, LineSpec, plot, plot3, quiver, surfnorm, view
“Direction and Velocity Plots” on page 1-85 for related functions
Three-Dimensional Quiver Plots for more examples

Quivergroup Properties

Purpose Defines the quivergroup properties

Modifying Properties You can set and query graphics object properties using the set and get commands or the Property Editor (propertyeditor).

Note that you cannot define default properties for areaseries objects.

See Plot Objects for more information on quivergroup objects.

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

AutoScale
{on} | off

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

AutoScaleFactor
scalar (default = 0.9)

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

BeingDeleted
on | {off} Read Only

This object is being deleted. The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the BeingDeleted property to on when the object's delete function callback is called (see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

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

`BusyAction`
cancel | {queue}

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is off, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over the quivergroup object.

This property can be

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

Quivergroup Properties

- A function handle

The expression executes in the MATLAB workspace.

See [Function Handle Callbacks](#) for information on how to use function handles to define the callbacks.

Children

array of graphics object handles

Children of the quivergroup object. An array containing the handles of all line objects parented to the quivergroup object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in the quiver `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

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

Clipping

{on} | off

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

Color

ColorSpec

Color of arrows. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color. See the [ColorSpec](#) reference page for more information on specifying color. For example, the following statement shows the arrow color set to blue.

```
h = quiver(u,v, 'Color', 'b');
```

CreateFcn

string or function handle

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

```
quiver(u,v,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other quivergroup properties. Setting this property on an existing quivergroup object has no effect.

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

See [Function Handle Callbacks](#) for information on how to use function handles to define the callback function.

DeleteFcn

string or function handle

Callback executed during object deletion. A callback that executes when the quivergroup object is deleted (e.g., this might happen when you issue a `delete` command on the quivergroup object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so that the callback routine can query these values.

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

Quivergroup Properties

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

See the BeingDeleted property for related information.

DisplayName
string

Label used by plot legends. The legend and the plot browser use this text for labels for any quivergroup objects appearing in these legends.

EraseMode
{normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase quiver child objects (the lines used to construct the arrows). Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster but do not perform a complete redraw and are therefore less accurate.
- **none** — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with EraseMode none, you cannot print these objects because MATLAB stores no information about their former locations.
- **xor** — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is

over the axes background color (or the figure background color if the axes Color property is set to none). That is, it isn't erased correctly if there are objects behind it.

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

Printing with Nonnormal Erase Modes

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

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

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

`HandleVisibility`
{on} | callback | off

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

- on — Handles are always visible when `HandleVisibility` is on.

Quivergroup Properties

- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

HitTest
{on} | off

Selectable by mouse click. HitTest determines whether the quivergroup object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the line objects that compose the quiver plot. If HitTest is off, clicking the quivergroup object selects the object below it (which is usually the axes containing it).

HitTestArea
on | {off}

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

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

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

Interruptible
{on} | off

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

Only callbacks defined for the `ButtonDownFcn` property are affected by the Interruptible property. MATLAB checks for

Quivergroup 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 a quiver property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

LineStyle

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

Line style. This property specifies the line style used for the quiver arrows. Available line styles are shown in the following table.

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

Width of the quiver arrows. Specify this value in points (1 point = $\frac{1}{72}$ inch). The default `LineWidth` is 0.5 points.

Marker

character (see table)

Marker symbol. The Marker property specifies the type of markers that are displayed at the x - and y -coordinates. You can set values for the Marker property independently from the LineStyle property. Supported markers include those shown in the following table.

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 quiver Color property.

Quivergroup Properties

MarkerFaceColor
ColorSpec | {none} | auto

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the axes color, or to the figure color, if the axes Color property is set to none (which is the factory default for axes).

MarkerSize
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the '.' symbol) at one-third the specified size.

MaxHeadSize
scalar (default = 0.2)

Maximum size of arrowhead. A value determining the maximum size of the arrowhead relative to the length of the arrow.

Parent
axes handle

Parent of quivergroup object. This property contains the handle of the quivergroup object's parent object. The parent of a quivergroup object is the axes, 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 at the corners and midpoints if the SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this property to on, thereby indicating that the quivergroup object is selected.

SelectionHighlight
{on} | off

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

ShowArrowHead
{on} | off

Display arrowheads on vectors. When this property is on, MATLAB draws arrowheads on the vectors displayed by quiver. When you set this property to off, quiver draws the vectors as lines without arrowheads.

Tag
string

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

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

```
t = quiver(u,v,'Tag','quiver1')
```

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

Quivergroup Properties

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

Type

string (read only)

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

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

UIContextMenu

handle of a uicontextmenu object

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

UserData

array

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

Visible

{on} | off

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

UData

matrix

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

UDataSource

string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

VData

matrix

Quivergroup Properties

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

VDataSource
string (MATLAB variable)

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

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

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

See the refreshdata reference page for more information.

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

WData
matrix

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

WDataSource

string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

XData

vector or matrix

X-axis coordinates of arrows. The quiver function draws an individual arrow at each *x*-axis location in the XData array. XData can be either a matrix equal in size to all other data properties or for 2-D, a vector equal in length to the number of columns in UData or VData. That is, `length(XData) == size(UData,2)`.

If you do not specify XData (i.e., the input argument X), the quiver function uses the indices of UData to create the quiver graph. See the XDataMode property for related information.

Quivergroup Properties

XDataMode

{auto} | manual

Use automatic or user-specified x-axis values. If you specify XData (by setting the XData property or specifying the input argument X), the quiver function sets this property to manual.

If you set XDataMode to auto after having specified XData, the quiver function resets the x tick-mark labels to the indices of the U, V, and W data, overwriting any previous values.

XDataSource

string (MATLAB variable)

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

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

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

See the refreshdata reference page for more information.

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

YData

vector or matrix

Y-axis coordinates of arrows. The quiver function draws an individual arrow at each y -axis location in the YData array. YData can be either a matrix equal in size to all other data properties or for 2-D, a vector equal in length to the number of rows in UData or VData. That is, `length(YData) == size(UData,1)`.

If you do not specify YData (i.e., the input argument Y), the quiver function uses the indices of VData to create the quiver graph. See the YDataMode property for related information.

The input argument y in the quiver function calling syntax assigns values to YData.

YDataMode

{auto} | manual

Use automatic or user-specified y-axis values. If you specify YData (by setting the YData property or specifying the input argument Y), MATLAB sets this property to manual.

If you set YDataMode to auto after having specified YData, MATLAB resets the y tick-mark labels to the indices of the U, V, and W data, overwriting any previous values.

YDataSource

string (MATLAB variable)

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

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

Quivergroup Properties

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

See the `refreshdata` reference page for more information.

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

ZData

vector or matrix

Z-axis coordinates of arrows. The `quiver` function draws an individual arrow at each *z*-axis location in the `ZData` array. `ZData` must be a matrix equal in size to `XData` and `YData`.

The input argument `z` in the `quiver3` function calling syntax assigns values to `ZData`.

Purpose

QZ factorization for generalized eigenvalues

Syntax

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

Description

The qz function gives access to intermediate results in the computation of generalized eigenvalues.

$[AA, BB, Q, Z] = \text{qz}(A, B)$ for square matrices A and B , produces upper quasitriangular matrices AA and BB , and unitary matrices Q and Z such that $Q^*A^*Z = AA$, and $Q^*B^*Z = BB$. For complex matrices, AA and BB are triangular.

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

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

'complex'	Produces a possibly complex decomposition with a triangular AA . For compatibility with earlier versions, 'complex' is the default.
'real'	Produces a real decomposition with a quasitriangular AA , containing 1-by-1 and 2-by-2 blocks on its diagonal.

If AA is triangular, the diagonal elements of AA and BB , $\alpha = \text{diag}(AA)$ and $\beta = \text{diag}(BB)$, are the generalized eigenvalues that satisfy

$$A^*V*\beta = B^*V*\alpha$$

$$\beta^*W'^*A = \alpha^*W'^*B$$

The eigenvalues produced by

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

are the ratios of the α s and β s.

$$\lambda = \alpha ./ \beta$$

If AA is triangular, the diagonal elements of AA and BB,

```
alpha = diag(AA)
beta = diag(BB)
```

are the generalized eigenvalues that satisfy

```
A*V*diag(beta) = B*V*diag(alpha)
diag(beta)*W'*A = diag(alpha)*W'*B
```

The eigenvalues produced by

```
lambda = eig(A,B)
```

are the element-wise ratios of alpha and beta.

```
lambda = alpha ./ beta
```

If AA is not triangular, it is necessary to further reduce the 2-by-2 blocks to obtain the eigenvalues of the full system.

Algorithm

For full matrices A and B, qz uses the LAPACK routines listed in the following table.

	A and B Real	A or B Complex
A and B double	DGGES, DTGEVC (if you request the fifth output V)	ZGGES, ZTGEVC (if you request the fifth output V)
A or B single	SGGES, STGEVC (if you request the fifth output V)	CGGES, CTGEVC (if you request the fifth output V)

See Also

eig

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), Third Edition, SIAM, Philadelphia, 1999.

rand

Purpose Uniformly distributed pseudorandom numbers

Syntax

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

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

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

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

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

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

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

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

The value of s is dependent upon which `method` is selected. If `method` is set to 'state' or 'twister', then s must be either a scalar integer value from 0 to $2^{32}-1$ or the output of `rand(method)`. If `method` is set to 'seed', then s must be either a scalar integer value from 0 to $2^{31}-2$

or the output of `rand(method)`. To set the generator to its default initial state, set `s` equal to zero.

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

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

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

For a full description of the Mersenne twister algorithm, see

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

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

Remarks

The sequence of numbers produced by `rand` is determined by the internal state of the generator. Setting the generator to the same fixed state enables you to repeat computations. Setting the generator to

different states leads to unique computations. It does not, however, improve statistical properties.

Because MATLAB resets the rand state at startup, rand generates the same sequence of numbers in each session unless you change the value of the state input.

Examples

Example 1

Make a random choice between two equally probable alternatives:

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

Example 2

Generate a 3-by-4 pseudorandom matrix:

```
R = rand(3,4)
R =
    0.2190    0.6793    0.5194    0.0535
    0.0470    0.9347    0.8310    0.5297
    0.6789    0.3835    0.0346    0.6711
```

Example 3

Set rand to its default initial state:

```
rand('state', 0);
```

Initialize rand to a different state each time:

```
rand('state', sum(100*clock));
```

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

```
s = rand('state');
```

```
u1 = rand(100);
rand('state',s);
u2 = rand(100);      % Contains exactly the same values as u1.
```

Example 4

Generate uniform integers on the set 1:n:

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

f(1:10)
ans =
    72
    34
    25
     1
    13
    63
    49
    25
     2
    27
```

Example 5

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

```
a = 10; b = 50;
x = a + (b-a) * rand(5)
x =

    18.1106    10.6110    26.7460    43.5247    30.1125
    17.9489    39.8714    43.8489    10.7856    38.3789
    34.1517    27.8039    31.0061    37.2511    27.1557
    20.8875    47.2726    18.1059    25.1792    22.1847
```

rand

17.9526 28.6398 36.8855 43.2718 17.5861

Example 6

Use the Mersenne twister generator, with the default initial state used by Nishimura and Matsumoto:

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

References

- [1] Moler, C.B., “Numerical Computing with MATLAB,” SIAM, (2004), 336 pp. Available online at <http://www.mathworks.com/moler>.
- [2] G. Marsaglia and A. Zaman A New Class of Random Number Generators, *Annals of Applied Probability*, (1991), 3:462-480.
- [3] Matsumoto, M. and Nishimura, T. “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator,” *ACM Transactions on Modeling and Computer Simulation*, (1998), 8(1):3-30.
- [4] Park, S.K. and Miller, K.W. “Random Number Generators: Good Ones Are Hard to Find,” *Communications of the ACM*, (1988), 31(10):1192-1201

See Also

randn, randperm, sprand, sprandn

Purpose

Normally distributed random numbers

Syntax

```
Y = randn
Y = randn(n)
Y = randn(m,n)
Y = randn([m n])
Y = randn(m,n,p,...)
Y = randn([m n p...])
Y = randn(size(A))
randn(method,s)
s = randn(method)
```

Description

`Y = randn` returns a pseudorandom, scalar value drawn from a normal distribution with mean 0 and standard deviation 1.

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

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

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

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

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

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

The value of `s` is dependent upon which `method` is selected. If `method` is set to 'state', then `s` must be either a scalar integer value from 0 to $2^{32}-1$ or the output of `rand(method)`. If `method` is set to 'seed', then `s` must be either a scalar integer value from 0 to $2^{31}-2$ or the

randn

output of `rand(method)`. To set the generator to its default initial state, set `s` equal to zero.

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

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

method	Description
'state'	Use Marsaglia's ziggurat algorithm (the default in MATLAB versions 5 and later). The period is approximately 2^{64} .
'seed'	Use the polar algorithm (the default in MATLAB version 4). The period is approximately $(2^{31}-1) * (\pi/8)$.

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

Examples

Example 1

`R = randn(3,4)` might produce

```
R =  
    1.1650    0.3516    0.0591    0.8717  
    0.6268   -0.6965    1.7971   -1.4462  
    0.0751    1.6961    0.2641   -0.7012
```

For a histogram of the `randn` distribution, see `hist`.

Example 2

Set `randn` to its default initial state:

```
randn('state', 0);
```

Initialize `randn` to a different state each time:

```
randn('state', sum(100*clock));
```

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

```
s = randn('state');
u1 = randn(100);
randn('state',s);
u2 = randn(100);           % Contains exactly the same values as u1.
```

Example 3

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

```
x = .6 + sqrt(0.1) * randn(5)
x =
```

```
    0.8713    0.4735    0.8114    0.0927    0.7672
    0.9966    0.8182    0.9766    0.6814    0.6694
    0.0960    0.8579    0.2197    0.2659    0.3085
    0.1443    0.8251    0.5937    1.0475   -0.0864
    0.7806    1.0080    0.5504    0.3454    0.5813
```

References

- [1] Moler, C.B., "Numerical Computing with MATLAB," SIAM, (2004), 336 pp. Available online at <http://www.mathworks.com/moler>.
- [2] Marsaglia, G. and Tsang, W.W., "The Ziggurat Method for Generating Random Variables," *Journal of Statistical Software*, (2000), 5(8). Available online at <http://www.jstatsoft.org/v05/i08/>.
- [3] Marsaglia, G. and Tsang, W.W., "A Fast, Easily Implemented Method for Sampling from Decreasing or Symmetric Unimodal Density Functions," *SIAM Journal of Scientific and Statistical Computing*, (1984), 5(2):349-359.

randn

[4] Knuth, D.E., "Seminumerical Algorithms," Volume 2 of *The Art of Computer Programming*, 3rd edition Addison-Wesley (1998).

See Also

rand, randperm, sprand, sprandn

Purpose Random permutation

Syntax `p = randperm(n)`

Description `p = randperm(n)` returns a random permutation of the integers `1:n`.

Remarks The `randperm` function calls `rand` and therefore, changes `rand`'s state.

Examples `randperm(6)` might be the vector

```
[3 2 6 4 1 5]
```

or it might be some other permutation of `1:6`.

See Also `permute`

rank

Purpose Rank of matrix

Syntax `k = rank(A)`
`k = rank(A,tol)`

Description The rank function provides an estimate of the number of linearly independent rows or columns of a full matrix.

`k = rank(A)` returns the number of singular values of A that are larger than the default tolerance, `max(size(A))*eps(norm(A))`.

`k = rank(A,tol)` returns the number of singular values of A that are larger than `tol`.

Remark Use `sprank` to determine the structural rank of a sparse matrix.

Algorithm There are a number of ways to compute the rank of a matrix. MATLAB uses the method based on the singular value decomposition, or SVD. The SVD algorithm is the most time consuming, but also the most reliable.

The rank algorithm is

```
s = svd(A);  
tol = max(size(A))*eps(max(s));  
r = sum(s > tol);
```

See Also `sprank`

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), Third Edition, SIAM, Philadelphia, 1999.

Purpose Rational fraction approximation

Syntax

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

Description Even though all floating-point numbers are rational numbers, it is sometimes desirable to approximate them by simple rational numbers, which are fractions whose numerator and denominator are small integers. The `rat` function attempts to do this. Rational approximations are generated by truncating continued fraction expansions. The `rats` function calls `rat`, and returns strings.

`[N,D] = rat(X)` returns arrays `N` and `D` so that `N./D` approximates `X` to within the default tolerance, `1.e-6*norm(X(:),1)`.

`[N,D] = rat(X,tol)` returns `N./D` approximating `X` to within `tol`.

`rat(X)`, with no output arguments, simply displays the continued fraction.

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

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

Examples Ordinarily, the statement

```
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7
```

produces

```
s =
```

rat, rats

0.7595

However, with

```
format rat
```

or with

```
rats(s)
```

the printed result is

```
s =  
319/420
```

This is a simple rational number. Its denominator is 420, the least common multiple of the denominators of the terms involved in the original expression. Even though the quantity s is stored internally as a binary floating-point number, the desired rational form can be reconstructed.

To see how the rational approximation is generated, the statement `rat(s)`

produces

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

And the statement

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

produces

```
n = 319, d = 420
```

The mathematical quantity π is certainly not a rational number, but the MATLAB quantity `pi` that approximates it is a rational number. `pi` is the ratio of a large integer and 2^{52} :

```
14148475504056880/4503599627370496
```

However, this is not a simple rational number. The value printed for pi with format rat, or with rats(pi), is

355/113

This approximation was known in Euclid's time. Its decimal representation is

3.14159292035398

and so it agrees with pi to seven significant figures. The statement

rat(pi)

produces

3 + 1/(7 + 1/(16))

This shows how the 355/113 was obtained. The less accurate, but more familiar approximation 22/7 is obtained from the first two terms of this continued fraction.

Algorithm

The rat(X) function approximates each element of X by a continued fraction of the form

$$\frac{n}{d} = d_1 + \frac{1}{d_2 + \frac{1}{\left(d_3 + \dots + \frac{1}{d_k}\right)}}$$

The d s are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when $X = \text{sqrt}(2)$. For $x = \text{sqrt}(2)$, the error with k terms is about $2.68 * (.173)^k$, so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.

rat, rats

See Also

format

Purpose	Create rubberband box for area selection
Syntax	<pre>rbbox rbbox(initialRect) rbbox(initialRect, fixedPoint) rbbox(initialRect, fixedPoint, stepSize) finalRect = rbbox(...)</pre>
Description	<p><code>rbbox</code> initializes and tracks a rubberband box in the current figure. It sets the initial rectangular size of the box to 0, anchors the box at the figure's <code>CurrentPoint</code>, and begins tracking from this point.</p> <p><code>rbbox(initialRect)</code> specifies the initial location and size of the rubberband box as <code>[x y width height]</code>, where <code>x</code> and <code>y</code> define the lower left corner, and <code>width</code> and <code>height</code> define the size. <code>initialRect</code> is in the units specified by the current figure's <code>Units</code> property, and measured from the lower left corner of the figure window. The corner of the box closest to the pointer position follows the pointer until <code>rbbox</code> receives a button-up event.</p> <p><code>rbbox(initialRect, fixedPoint)</code> specifies the corner of the box that remains fixed. All arguments are in the units specified by the current figure's <code>Units</code> property, and measured from the lower left corner of the figure window. <code>fixedPoint</code> is a two-element vector, <code>[x y]</code>. The tracking point is the corner diametrically opposite the anchored corner defined by <code>fixedPoint</code>.</p> <p><code>rbbox(initialRect, fixedPoint, stepSize)</code> specifies how frequently the rubberband box is updated. When the tracking point exceeds <code>stepSize</code> figure units, <code>rbbox</code> redraws the rubberband box. The default <code>stepSize</code> is 1.</p> <p><code>finalRect = rbbox(...)</code> returns a four-element vector, <code>[x y width height]</code>, where <code>x</code> and <code>y</code> are the x and y components of the lower left corner of the box, and <code>width</code> and <code>height</code> are the dimensions of the box.</p>
Remarks	<code>rbbox</code> is useful for defining and resizing a rectangular region:

- For box definition, `initialRect` is `[x y 0 0]`, where `(x,y)` is the figure's `CurrentPoint`.
- For box resizing, `initialRect` defines the rectangular region that you resize (e.g., a legend). `fixedPoint` is the corner diametrically opposite the tracking point.

`rbbox` returns immediately if a button is not currently pressed. Therefore, you use `rbbox` with `waitforbuttonpress` so that the mouse button is down when `rbbox` is called. `rbbox` returns when you release the mouse button.

Examples

Assuming the current view is `view(2)`, use the current axes' `CurrentPoint` property to determine the extent of the rectangle in dataspace units:

```
k = waitforbuttonpress;
point1 = get(gca, 'CurrentPoint');    % button down detected
finalRect = rbbox;                   % return figure units
point2 = get(gca, 'CurrentPoint');    % button up detected
point1 = point1(1,1:2);               % extract x and y
point2 = point2(1,1:2);
p1 = min(point1,point2);              % calculate locations
offset = abs(point1-point2);          % and dimensions
x = [p1(1) p1(1)+offset(1) p1(1)+offset(1) p1(1) p1(1)];
y = [p1(2) p1(2) p1(2)+offset(2) p1(2)+offset(2) p1(2)];
hold on
axis manual
plot(x,y)                             % redraw in dataspace units
```

See Also

`axis`, `dragrect`, `waitforbuttonpress`
“View Control” on page 1-96 for related functions

Purpose Matrix reciprocal condition number estimate

Syntax `c = rcond(A)`

Description `c = rcond(A)` returns an estimate for the reciprocal of the condition of A in 1-norm using the LAPACK condition estimator. If A is well conditioned, `rcond(A)` is near 1.0. If A is badly conditioned, `rcond(A)` is near 0.0. Compared to `cond`, `rcond` is a more efficient, but less reliable, method of estimating the condition of a matrix.

Algorithm For full matrices A , `rcond` uses the LAPACK routines listed in the following table to compute the estimate of the reciprocal condition number.

	Real	Complex
A double	DLANGE, DGETRF, DGECON	ZLANGE, ZGETRF, ZGECON
A single	SLANGE, SGETRF, SGECON	CLANGE, CGETRF, CGECON

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

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), Third Edition, SIAM, Philadelphia, 1999.

readasync

Purpose Read data asynchronously from device

Syntax `readasync(obj)`
`readasync(obj,size)`

Arguments

<code>obj</code>	A serial port object.
<code>size</code>	The number of bytes to read from the device.

Description `readasync(obj)` initiates an asynchronous read operation.
`readasync(obj,size)` asynchronously reads, at most, the number of bytes given by `size`. If `size` is greater than the difference between the `InputBufferSize` property value and the `BytesAvailable` property value, an error is returned.

Remarks

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

You should use `readasync` only when you configure the `ReadAsyncMode` property to `manual`. `readasync` is ignored if used when `ReadAsyncMode` is `continuous`.

The `TransferStatus` property indicates if an asynchronous read or write operation is in progress. You can write data while an asynchronous read is in progress because serial ports have separate read and write pins. You can stop asynchronous read and write operations with the `stopasync` function.

You can monitor the amount of data stored in the input buffer with the `BytesAvailable` property. Additionally, you can use the `BytesAvailableFcn` property to execute an M-file callback function when the terminator or the specified amount of data is read.

Rules for Completing an Asynchronous Read Operation

An asynchronous read operation with `readasync` completes when one of these conditions is met:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The specified number of bytes is read.
- The input buffer is filled (if size is not specified).

Because `readasync` checks for the terminator, this function can be slow. To increase speed, you might want to configure `ReadAsyncMode` to `continuous` and continuously return data to the input buffer as soon as it is available from the device.

Example

This example creates the serial port object `s`, connects `s` to a Tektronix TDS 210 oscilloscope, configures `s` to read data asynchronously only if `readasync` is issued, and configures the instrument to return the peak-to-peak value of the signal on channel 1.

```
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s, 'Measurement:Meas1:Source CH1')
fprintf(s, 'Measurement:Meas1:Type Pk2Pk')
fprintf(s, 'Measurement:Meas1:Value?')
```

Begin reading data asynchronously from the instrument using `readasync`. When the read operation is complete, return the data to the MATLAB workspace using `fscanf`.

```
readasync(s)
s.BytesAvailable
ans =
    15
out = fscanf(s)
```

readasync

```
out =  
2.0399999619E0  
fclose(s)
```

See Also

Functions

fopen, stopasync

Properties

BytesAvailable, BytesAvailableFcn, ReadAsyncMode, Status, TransferStatus

Purpose	Real part of complex number
Syntax	$X = \text{real}(Z)$
Description	$X = \text{real}(Z)$ returns the real part of the elements of the complex array Z .
Examples	<code>real(2+3*i)</code> is 2.
See Also	<code>abs</code> , <code>angle</code> , <code>conj</code> , <code>i</code> , <code>j</code> , <code>imag</code>

reallog

Purpose Natural logarithm for nonnegative real arrays

Syntax `Y = reallog(X)`

Description `Y = reallog(X)` returns the natural logarithm of each element in array `X`. Array `X` must contain only nonnegative real numbers. The size of `Y` is the same as the size of `X`.

Examples `M = magic(4)`

```
M =  
    16     2     3    13  
     5    11    10     8  
     9     7     6    12  
     4    14    15     1
```

```
reallog(M)
```

```
ans =  
    2.7726    0.6931    1.0986    2.5649  
    1.6094    2.3979    2.3026    2.0794  
    2.1972    1.9459    1.7918    2.4849  
    1.3863    2.6391    2.7081         0
```

See Also `log`, `realpow`, `realsqrt`

Purpose	Largest positive floating-point number
Syntax	<code>n = realmax</code>
Description	<p><code>n = realmax</code> returns the largest floating-point number representable on your computer. Anything larger overflows.</p> <p><code>realmax('double')</code> is the same as <code>realmax</code> with no arguments.</p> <p><code>realmax('single')</code> is the largest single precision floating point number representable on your computer. Anything larger overflows to <code>single(Inf)</code>.</p>
Examples	<code>realmax</code> is one bit less than 2^{1024} or about <code>1.7977e+308</code> .
Algorithm	<p>The <code>realmax</code> function is equivalent to <code>pow2(2-eps,maxexp)</code>, where <code>maxexp</code> is the largest possible floating-point exponent.</p> <p>Execute type <code>realmax</code> to see <code>maxexp</code> for various computers.</p>
See Also	<code>eps</code> , <code>realmin</code> , <code>intmax</code>

realmin

Purpose	Smallest positive floating-point number
Syntax	<code>n = realmin</code>
Description	<p><code>n = realmin</code> returns the smallest positive normalized floating-point number on your computer. Anything smaller underflows or is an IEEE “denormal.”</p> <p><code>REALMIN('double')</code> is the same as <code>REALMIN</code> with no arguments.</p> <p><code>REALMIN('single')</code> is the smallest positive normalized single precision floating point number on your computer.</p>
Examples	<code>realmin</code> is $2^{(-1022)}$ or about $2.2251e-308$.
Algorithm	<p>The <code>realmin</code> function is equivalent to <code>pow2(1,minexp)</code> where <code>minexp</code> is the smallest possible floating-point exponent.</p> <p>Execute type <code>realmin</code> to see <code>minexp</code> for various computers.</p>
See Also	<code>eps</code> , <code>realmax</code> , <code>intmin</code>

Purpose Array power for real-only output

Syntax `Z = realpow(X,Y)`

Description `Z = realpow(X,Y)` raises each element of array `X` to the power of its corresponding element in array `Y`. Arrays `X` and `Y` must be the same size. The range of `realpow` is the set of all real numbers, i.e., all elements of the output array `Z` must be real.

Examples

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

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

```
Y = pascal(3)
```

```
ans =  
     1     1     1  
     1     2     3  
     1     3     6
```

```
realpow(X,Y)
```

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

See Also `reallog`, `realsqrt`, `.`[^] (array power operator)

realsqrt

Purpose Square root for nonnegative real arrays

Syntax `Y = realsqrt(X)`

Description `Y = realsqrt(X)` returns the square root of each element of array `X`. Array `X` must contain only nonnegative real numbers. The size of `Y` is the same as the size of `X`.

Examples

```
M = magic(4)
```

```
M =  
    16     2     3    13  
     5    11    10     8  
     9     7     6    12  
     4    14    15     1
```

```
realsqrt(M)
```

```
ans =  
    4.0000    1.4142    1.7321    3.6056  
    2.2361    3.3166    3.1623    2.8284  
    3.0000    2.6458    2.4495    3.4641  
    2.0000    3.7417    3.8730    1.0000
```

See Also `reallog`, `realpow`, `sqrt`, `sqrtm`

Purpose	Record data and event information to file
Syntax	<pre>record(obj) record(obj, 'switch')</pre>
Arguments	<p><code>obj</code> A serial port object.</p> <p><code>'switch'</code> Switch recording capabilities on or off.</p>
Description	<p><code>record(obj)</code> toggles the recording state for <code>obj</code>.</p> <p><code>record(obj, 'switch')</code> initiates or terminates recording for <code>obj</code>. <code>switch</code> can be on or off. If <code>switch</code> is on, recording is initiated. If <code>switch</code> is off, recording is terminated.</p>
Remarks	<p>Before you can record information to disk, <code>obj</code> must be connected to the device with the <code>fopen</code> function. A connected serial port object has a <code>Status</code> property value of <code>open</code>. An error is returned if you attempt to record information while <code>obj</code> is not connected to the device. Each serial port object must record information to a separate file. Recording is automatically terminated when <code>obj</code> is disconnected from the device with <code>fclose</code>.</p> <p>The <code>RecordName</code> and <code>RecordMode</code> properties are read-only while <code>obj</code> is recording, and must be configured before using <code>record</code>.</p> <p>For a detailed description of the record file format and the properties associated with recording data and event information to a file, refer to Debugging: Recording Information to Disk.</p>
Example	<p>This example creates the serial port object <code>s</code>, connects <code>s</code> to the device, configures <code>s</code> to record information to a file, writes and reads text data, and then disconnects <code>s</code> from the device.</p> <pre>s = serial('COM1'); fopen(s) s.RecordDetail = 'verbose';</pre>

record

```
s.RecordName = 'MySerialFile.txt';  
record(s, 'on')  
fprintf(s, '*IDN?')  
out = fscanf(s);  
record(s, 'off')  
fclose(s)
```

See Also

Functions

fclose, fopen

Properties

RecordDetail, RecordMode, RecordName, RecordStatus, Status

Purpose

Create a 2-D rectangle object

Syntax

Description

`rectangle` draws a rectangle with Position `[0,0,1,1]` and Curvature `[0,0]` (i.e., no curvature).

`rectangle('Position', [x,y,w,h])` draws the rectangle from the point `x,y` and having a width of `w` and a height of `h`. Specify values in axes data units.

Note that, to display a rectangle in the specified proportions, you need to set the axes data aspect ratio so that one unit is of equal length along both the `x` and `y` axes. You can do this with the command `axis equal` or `daspect([1,1,1])`.

`rectangle(..., 'Curvature', [x,y])` specifies the curvature of the rectangle sides, enabling it to vary from a rectangle to an ellipse. The horizontal curvature `x` is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature `y` is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of `x` and `y` can range from 0 (no curvature) to 1 (maximum curvature). A value of `[0,0]` creates a rectangle with square sides. A value of `[1,1]` creates an ellipse. If you specify only one value for Curvature, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

`h = rectangle(...)` returns the handle of the rectangle object created.

Remarks

Rectangle objects are 2-D and can be drawn in an axes only if the view is `[0 90]` (i.e., `view(2)`). Rectangles are children of axes and are defined in coordinates of the axes data.

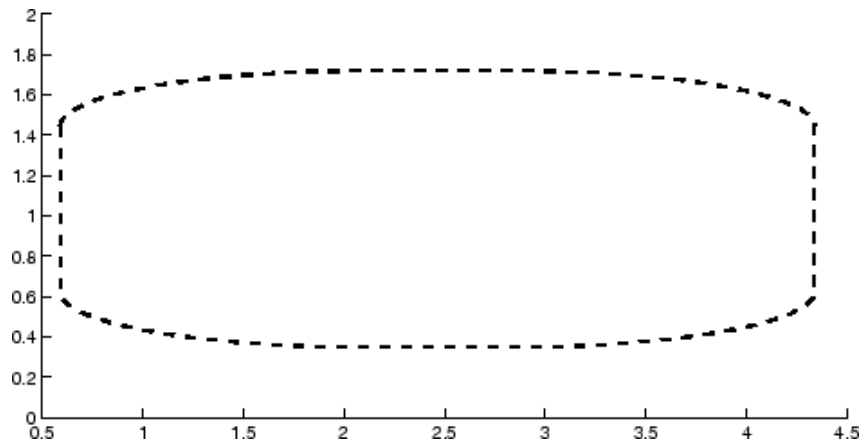
Examples

This example sets the data aspect ratio to `[1,1,1]` so that the rectangle is displayed in the specified proportions (`daspect`). Note that the

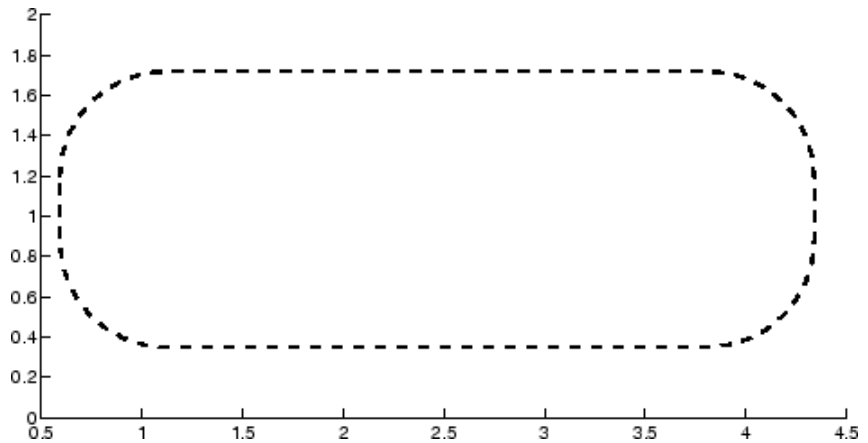
rectangle

horizontal and vertical curvature can be different. Also, note the effects of using a single value for Curvature.

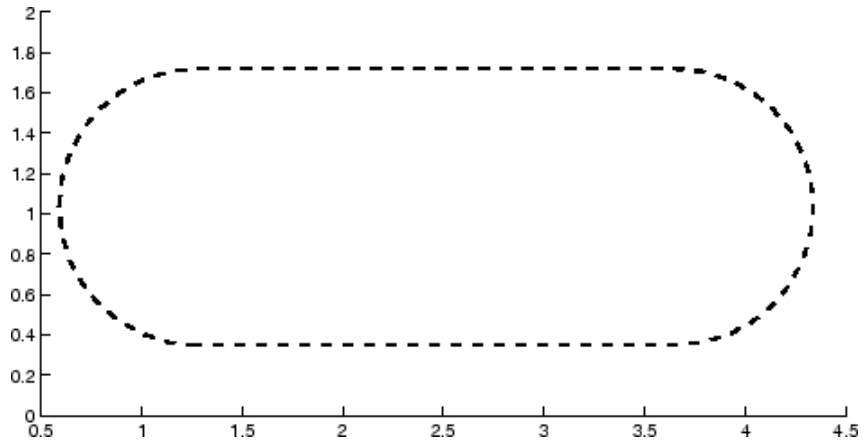
```
rectangle('Position',[0.59,0.35,3.75,1.37],...  
         'Curvature',[0.8,0.4],...  
         'LineWidth',2,'LineStyle','--')  
daspect([1,1,1])
```



Specifying a single value of [0.4] for Curvature produces



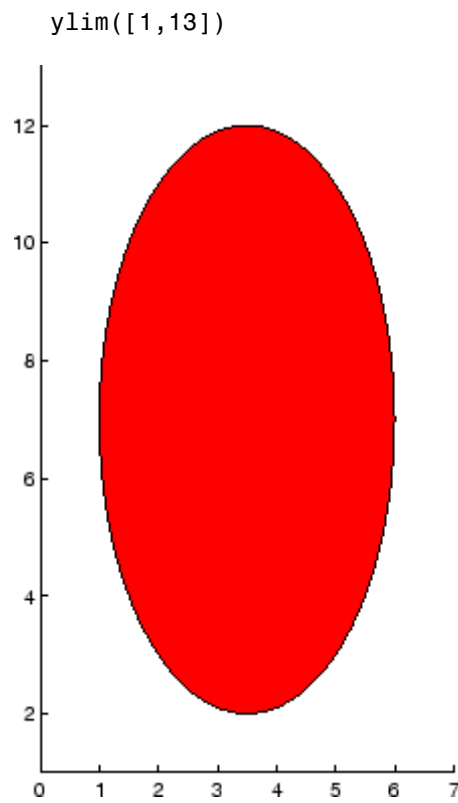
A Curvature of [1] produces a rectangle with the shortest side completely round:



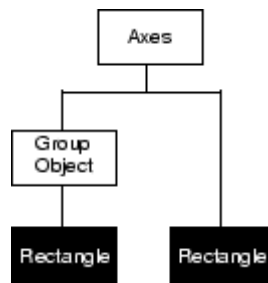
This example creates an ellipse and colors the face red.

```
rectangle('Position',[1,2,5,10],'Curvature',[1,1],...  
         'FaceColor','r')  
daspect([1,1,1])  
xlim([0,7])
```

rectangle



Object Hierarchy



Setting Default Properties

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

```
set(0, 'DefaultRectangleProperty', PropertyValue...)  
set(gcf, 'DefaultRectangleProperty', PropertyValue...)  
set(gca, 'DefaultRectangleProperty', PropertyValue...)
```

where *Property* is the name of the rectangle property whose default value you want to set and *PropertyValue* is the value you are specifying. Use `set` and `get` to access the surface properties.

See Also

`line`, `patch`, `rectangle` properties

“Object Creation Functions” on page 1-90 for related functions

See the annotation function for information about the rectangle annotation object.

Rectangle Properties for property descriptions

Rectangle Properties

Purpose

Defines the rectangle properties.

Modifying Properties

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

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

To change the default values of properties, see “Setting Default Property Values”.

See “Core Graphics Objects” for general information about this type of object.

Rectangle Property Descriptions

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

BeingDeleted
on | {off} read only

This object is being deleted. The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the BeingDeleted property to on when the object’s delete function callback is called (see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

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

BusyAction
cancel | {queue}

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is off, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

string or function handle

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the rectangle object.

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

Set this property to a function handle that references the callback. You can also use a string that is a valid MATLAB expression or the name of an M-file. The expressions execute in the MATLAB workspace.

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

`Children`

vector of handles

The empty matrix; rectangle objects have no children.

Rectangle Properties

Clipping
{on} | off

Clipping mode. MATLAB clips rectangles to the axes plot box by default. If you set `Clipping` to off, rectangles are displayed outside the axes plot box. This can occur if you create a rectangle, set `hold` to on, freeze axis scaling (axis set to manual), and then create a larger rectangle.

CreateFcn
string or function handle

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a rectangle object. You must define this property as a default value for rectangles or in a call to the `rectangle` function to create a new rectangle object. For example, the statement

```
set(0, 'DefaultRectangleCreateFcn', ...  
    'set(gca, 'DataAspectRatio', [1,1,1])')
```

defines a default value on the root level that sets the axes `DataAspectRatio` whenever you create a rectangle object. MATLAB executes this routine after setting all rectangle properties. Setting this property on an existing rectangle object has no effect.

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

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

Curvature
one- or two-element vector [x,y]

Amount of horizontal and vertical curvature. This property specifies the curvature of the rectangle sides, which enables the shape of the rectangle to vary from rectangular to ellipsoidal. The horizontal curvature x is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature y is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of x and y can range from 0 (no curvature) to 1 (maximum curvature). A value of $[0, 0]$ creates a rectangle with square sides. A value of $[1, 1]$ creates an ellipse. If you specify only one value for Curvature, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

DeleteFcn

string or function handle

Delete rectangle callback routine. A callback routine that executes when you delete the rectangle object (e.g., when you issue a delete command or clear the axes or figure). MATLAB executes the routine before deleting the object's properties so these values are available to the callback routine.

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

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

EdgeColor

{ColorSpec} | none

Color of the rectangle edges. This property specifies the color of the rectangle edges as a color or specifies that no edges be drawn.

Rectangle Properties

EraseMode

{normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase rectangle objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** (the default) — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- **none** — Do not erase the rectangle when it is moved or destroyed. While the object is still visible on the screen after erasing with EraseMode none, you cannot print it because MATLAB stores no information about its former location.
- **xor** — Draw and erase the rectangle by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the rectangle. However, the rectangle's color depends on the color of whatever is beneath it on the display.
- **background** — Erase the rectangle by drawing it in the axes background Color, or the figure background Color if the axes Color is set to none. This damages objects that are behind the erased rectangle, but rectangles are always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the EraseMode of all objects is normal. This means graphics objects created with EraseMode set to none, xor, or background can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional

sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

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

FaceColor
ColorSpec | {none}

Color of rectangle face. This property specifies the color of the rectangle face, which is not colored by default.

HandleVisibility
{on} | callback | off

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is on.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

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

Rectangle Properties

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

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

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

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

`HitTest`
{on} | off

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

`Interruptible`
{on} | off

Callback routine interruption mode. The `Interruptible` property controls whether a rectangle callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback

routine only when it encounters a drawnow, figure, getframe, or pause command in the routine.

LineStyle

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

Line style of rectangle edge. This property specifies the line style of the edges. The available line styles are

Symbol	Line Style
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line
none	No line

LineWidth

scalar

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

Parent

handle of axes, hggroup, or hgtransform

Parent of rectangle object. This property contains the handle of the rectangle object's parent. The parent of a rectangle object is the axes, hggroup, or hgtransform object that contains it.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

Position

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

Rectangle Properties

Location and size of rectangle. This property specifies the location and size of the rectangle in the data units of the axes. The point defined by x , y specifies one corner of the rectangle, and width and height define the size in units along the x - and y -axes respectively.

Selected
on | off

Is object selected? When this property is on MATLAB displays selection handles if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

SelectionHighlight
{on} | off

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

Tag
string

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

Type
string (read only)

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

UIContextMenu
handle of a uicontextmenu object

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

`UserData`
matrix

User-specified data. Any data you want to associate with the rectangle object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
{on} | off

Rectangle visibility. By default, all rectangles are visible. When set to `off`, the rectangle is not visible, but still exists, and you can get and set its properties.

rectint

Purpose Rectangle intersection area

Syntax `area = rectint(A,B)`

Description `area = rectint(A,B)` returns the area of intersection of the rectangles specified by position vectors A and B.

If A and B each specify one rectangle, the output area is a scalar.

A and B can also be matrices, where each row is a position vector. area is then a matrix giving the intersection of all rectangles specified by A with all the rectangles specified by B. That is, if A is n-by-4 and B is m-by-4, then area is an n-by-m matrix where `area(i, j)` is the intersection area of the rectangles specified by the ith row of A and the jth row of B.

Note A position vector is a four-element vector `[x,y,width,height]`, where the point defined by x and y specifies one corner of the rectangle, and width and height define the size in units along the x and y axes respectively.

See Also `polyarea`

Purpose	Set option to move deleted files to recycle folder
Syntax	<code>S = recycle</code> <code>S = recycle state</code> <code>S = recycle('state')</code>
Description	<p><code>S = recycle</code> returns a character array <code>S</code> that shows the current state of the MATLAB file recycling option. This state can be either on or off. When file recycling is on, MATLAB moves all files that you delete with the <code>delete</code> function to either the recycle bin on the PC or Macintosh, or a temporary directory on UNIX. (To locate this directory on UNIX, see the Remarks section below.) When file recycling is off, any files you delete are actually removed from the system.</p> <p>The default recycle state is off. You can turn recycling on for all of your MATLAB sessions using the Preferences dialog box (Select File > Preferences > General). Under the heading Default behavior of the delete function select Move files to the Recycle Bin.</p> <p><code>S = recycle state</code> sets the MATLAB recycle option to the given state, either on or off. Return value <code>S</code> shows the previous recycle state.</p> <p><code>S = recycle('state')</code> is the function format for this command.</p>
Remarks	<p>On UNIX systems, you can locate the system temporary directory by entering the MATLAB function <code>tempdir</code>. The recycle directory is a subdirectory of this temporary directory, and is named according to the format</p> <pre>MATLAB_Files_<day>-<mo>-<yr>_<hr>_<min>_<sec></pre> <p>For example, files recycled on a UNIX system at 2:09:28 in the afternoon of November 9, 2004 would be copied to a directory named</p> <pre>/tmp/MATLAB_Files_09-Nov-2004_14_09_28</pre> <p>To set the recycle state for all MATLAB sessions, use the Preferences dialog box. Open the Preferences dialog and select General. To</p>

recycle

enable or disable recycling, click **Move files to the recycle bin** or **Delete files permanently**. See “General Preferences for MATLAB” in the Desktop Tools and Development Environment documentation for more information.

You can recycle files that are stored on your local computer system, but not files that you access over a network connection. On Windows systems, when you use the `delete` function on files accessed over a network, MATLAB removes the file entirely.

Examples

Start from a state where file recycling has been turned off. Check the current recycle state:

```
recycle
ans =
    off
```

Turn file recycling on. Delete a file and verify that it has been transferred to the recycle bin or temporary folder:

```
recycle on;
delete myfile.txt
```

See Also

`delete`, `dir`, `ls`, `fileparts`, `mkdir`, `rmdir`

Purpose Reduce number of patch faces

Syntax

```
nfv = reducepatch(p,r)
nfv = reducepatch(fv,r)
nfv = reducepatch(p) or nfv = reducepatch(fv)
reducepatch(...,'fast')
reducepatch(...,'verbose')
nfv = reducepatch(f,v,r)
[nf,nv] = reducepatch(...)
```

Description `reducepatch(p,r)` reduces the number of faces of the patch identified by handle `p`, while attempting to preserve the overall shape of the original object. MATLAB interprets the reduction factor `r` in one of two ways depending on its value:

- If `r` is less than 1, `r` is interpreted as a fraction of the original number of faces. For example, if you specify `r` as 0.2, then the number of faces is reduced to 20% of the number in the original patch.
- If `r` is greater than or equal to 1, then `r` is the target number of faces. For example, if you specify `r` as 400, then the number of faces is reduced until there are 400 faces remaining.

`nfv = reducepatch(p,r)` returns the reduced set of faces and vertices but does not set the `Faces` and `Vertices` properties of patch `p`. The struct `nfv` contains the faces and vertices after reduction.

`nfv = reducepatch(fv,r)` performs the reduction on the faces and vertices in the struct `fv`.

`nfv = reducepatch(p)` or `nfv = reducepatch(fv)` uses a reduction value of 0.5.

`reducepatch(...,'fast')` assumes the vertices are unique and does not compute shared vertices.

`reducepatch(...,'verbose')` prints progress messages to the command window as the computation progresses.

reducepatch

`nfv = reducepatch(f,v,r)` performs the reduction on the faces in `f` and the vertices in `v`.

`[nf,nv] = reducepatch(...)` returns the faces and vertices in the arrays `nf` and `nv`.

Remarks

If the patch contains nonshared vertices, MATLAB computes shared vertices before reducing the number of faces. If the faces of the patch are not triangles, MATLAB triangulates the faces before reduction. The faces returned are always defined as triangles.

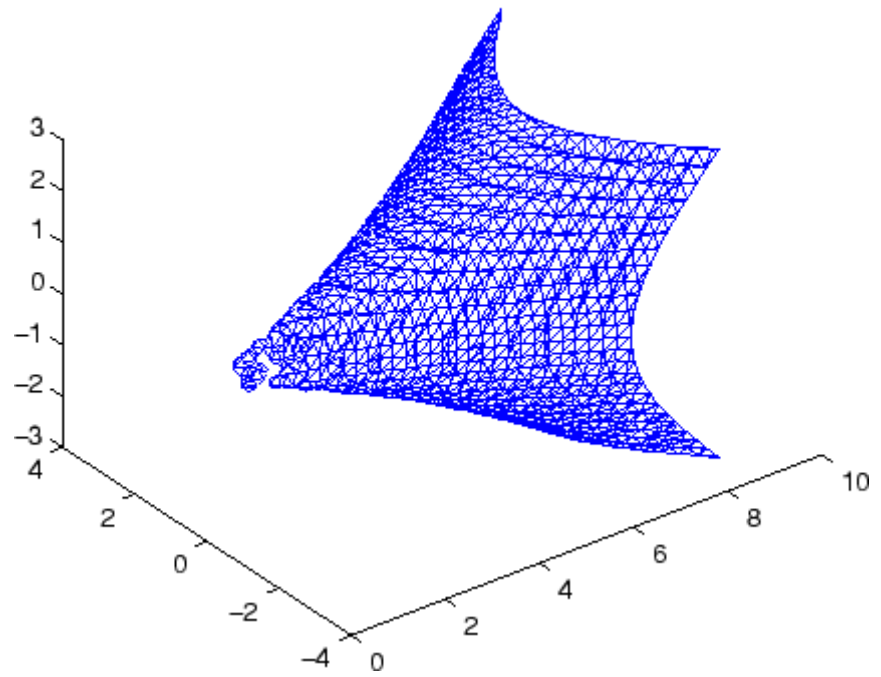
The number of output triangles may not be exactly the number specified with the reduction factor argument (`r`), particularly if the faces of the original patch are not triangles.

Examples

This example illustrates the effect of reducing the number of faces to only 15% of the original value.

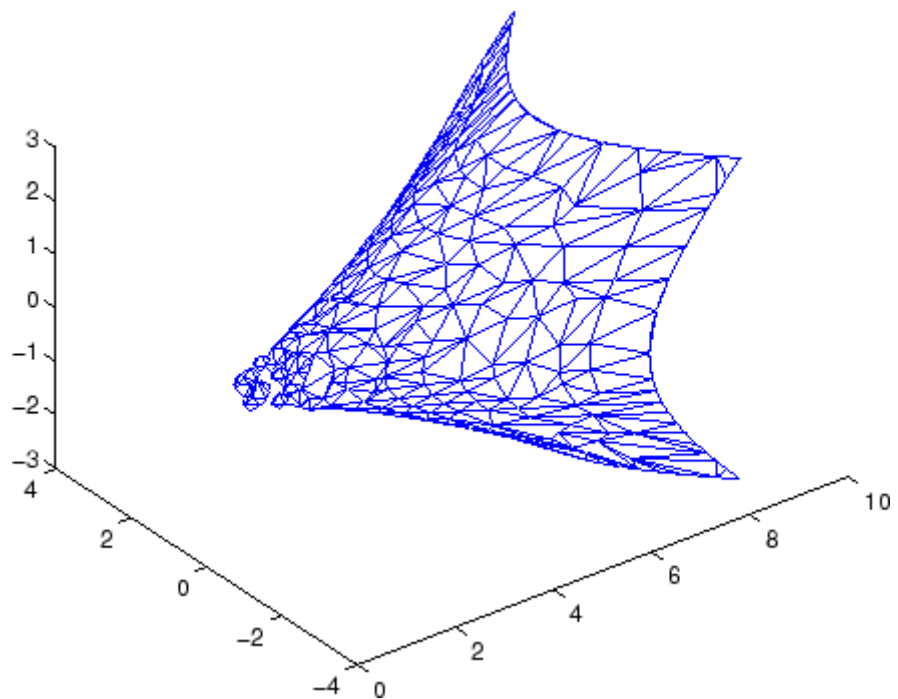
```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
set(p,'facecolor','w','EdgeColor','b');
daspect([1,1,1])
view(3)
figure;
h = axes;
p2 = copyobj(p,h);
reducepatch(p2,0.15)
daspect([1,1,1])
view(3)
```


Before Reduction



reducepatch

After Reduction to 15% of Original Number of Faces



See Also

`isosurface`, `isocaps`, `isonormals`, `smooth3`, `subvolume`, `reducevolume`
“Volume Visualization” on page 1-98 for related functions
Vector Field Displayed with Cone Plots for another example

Purpose

Reduce number of elements in volume data set

Syntax

```
[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])
[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])
nv = reducevolume(...)
```

Description

`[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])` reduces the number of elements in the volume by retaining every R_x^{th} element in the x direction, every R_y^{th} element in the y direction, and every R_z^{th} element in the z direction. If a scalar R is used to indicate the amount of reduction instead of a three-element vector, MATLAB assumes the reduction to be `[R R R]`.

The arrays X , Y , and Z define the coordinates for the volume V . The reduced volume is returned in nv , and the coordinates of the reduced volume are returned in nx , ny , and nz .

`[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])` assumes the arrays X , Y , and Z are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p] = size(V)`.

`nv = reducevolume(...)` returns only the reduced volume.

Examples

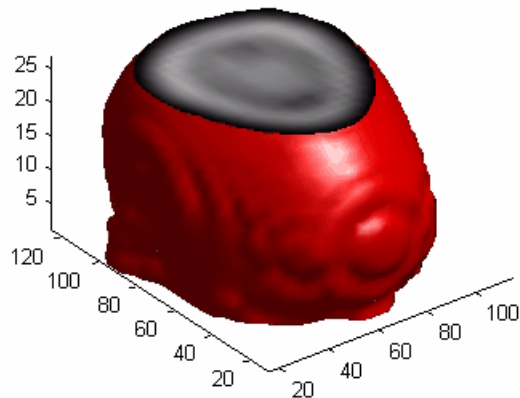
This example uses a data set that is a collection of MRI slices of a human skull. This data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then reduced (`reducevolume`) so that what remains is every fourth element in the x and y directions and every element in the z direction.
- The reduced data is smoothed (`smooth3`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with an interpolated face color draws the end caps (`FaceColor`, `isocaps`).
- The view of the object is set (`view`, `axis`, `daspect`).

reducevolume

- A 100-element grayscale colormap provides coloring for the end caps (colormap).
- Adding a light to the right of the camera illuminates the object (camlight, lighting).

```
load mri
D = squeeze(D);
[x,y,z,D] = reducevolume(D,[4,4,1]);
D = smooth3(D);
p1 = patch(isosurface(x,y,z,D, 5,'verbose'),...
    'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
    'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight; lighting gouraud
```



See Also

isosurface, isocaps, isonormals, smooth3, subvolume, reducepatch

“Volume Visualization” on page 1-98 for related functions

refresh

Purpose	Redraw current figure
Syntax	<code>refresh</code> <code>refresh(h)</code>
Description	<code>refresh</code> erases and redraws the current figure. <code>refresh(h)</code> redraws the figure identified by <code>h</code> .
See Also	“Figure Windows” on page 1-92 for related functions

Purpose

Refresh data in graph when data source is specified

Syntax

```
refreshdata
refreshdata(figure_handle)
refreshdata(object_handles)
refreshdata(object_handles, 'workspace')
```

Description

`refreshdata` evaluates any data source properties (XDataSource, YDataSource, or ZDataSource) on all objects in graphs in the current figure. If the specified data source has changed, MATLAB updates the graph to reflect this change.

Note that the variable assigned to the data source property must be in the base workspace.

`refreshdata(figure_handle)` refreshes the data of the objects in the specified figure.

`refreshdata(object_handles)` refreshes the data of the objects specified in `object_handles` or the children of those objects. Therefore, `object_handles` can contain figure, axes, or plot object handles.

`refreshdata(object_handles, 'workspace')` enables you to specify whether the data source properties are evaluated in the base workspace or the workspace of the function in which `refreshdata` was called. `workspace` is a string that can be

- `base` — Evaluate the data source properties in the base workspace.
- `caller` — Evaluate the data source properties in the workspace of the function that called `refreshdata`.

Examples

This example creates a contour plot and changes its data source. The call to `refreshdata` causes the graph to update.

```
z = peaks(5);
[c h] = contour(z, 'ZDataSource', 'z');
drawnow
pause(3) % Wait 3 seconds and the graph will update
```

refreshdata

```
z = peaks(20);  
refreshdata(h)
```

See Also

The [X,Y,Z]DataSource properties of plot objects.

Purpose

Match regular expression

Syntax

```
regexp('str', 'expr')  
[start end extents match tokens names] = regexp('str',  
    'expr')  
[v1 v2 ...] = regexp('str', 'expr', q1, q2, ...)  
[v1 v2 ...] = regexp('str', 'expr', ..., options)
```

Each of these syntaxes apply to both `regexp` and `regexpi`. The `regexp` function is case sensitive in matching regular expressions to a string, and `regexpi` is case insensitive:

Description

The following descriptions apply to both `regexp` and `regexpi`:

`regexp('str', 'expr')` returns a row vector containing the starting index of each substring of `str` that matches the regular expression string `expr`. If no matches are found, `regexp` returns an empty array. The `str` and `expr` arguments can also be cell arrays of strings.

To specify more than one string to parse or more than one expression to match, see the guidelines listed below under “Multiple Strings or Expressions” on page 2-2533.

`[start end extents match tokens names] = regexp('str', 'expr')` returns up to six values, one for each output variable you specify, and in the default order (as shown in the table below).

Note The `str` and `expr` inputs are required and must be entered as the first and second arguments, respectively. Any other input arguments (all are described below) are optional and can be entered following the two required inputs in any order.

`[v1 v2 ...] = regexp('str', 'expr', q1, q2, ...)` returns up to six values, one for each output variable you specify, and ordered according to the order of the qualifier arguments, `q1`, `q2`, etc.

regexp, regexpi

Return Values for Regular Expressions

Default Order	Description	Qualifier
1	Row vector containing the starting index of each substring of <code>str</code> that matches <code>expr</code>	start
2	Row vector containing the ending index of each substring of <code>str</code> that matches <code>expr</code>	end
3	Cell array containing the starting and ending indices of each substring of <code>str</code> that matches a token in <code>expr</code> . (This is a double array when used with 'once'.)	tokenExtents
4	Cell array containing the text of each substring of <code>str</code> that matches <code>expr</code> . (This is a string when used with 'once'.)	match
5	Cell array of cell arrays of strings containing the text of each token captured by <code>regexp</code> . (This is a cell array of strings when used with 'once'.)	tokens
6	Structure array containing the name and text of each <i>named</i> token captured by <code>regexp</code> . If there are no named tokens in <code>expr</code> , <code>regexp</code> returns a structure array with no fields. Field names of the returned structure are set to the token names, and field values are the text of those tokens. Named tokens are generated by the expression (?<tokenname>).	names

[v1 v2 ...] = `regexp('str', 'expr', ..., options)` calls `regexp` with one or more of the nondefault options listed in the following table. These options must follow `str` and `expr` in the input argument list.

Option	Description
mode	See the section on “Modes” on page 2-2531 below.

Option	Description
'once'	Return only the first match found.
'warnings'	Display any hidden warning messages issued by MATLAB during the execution of the command. This option only enables warnings for the one command being executed. See Example 10.

Modes

You can specify one or more of the following modes with the `regexp`, `regexpi`, and `regexprep` functions. You can enable or disable any of these modes using the mode specifier keyword (e.g., 'lineanchors') or the mode flag (e.g., (?m)). Both are shown in the tables that follow. Use the keyword to enable or disable the mode for the entire string being parsed. Use the flag to both enable and disable the mode for selected pieces of the string.

Case-Sensitivity Mode

Use the Case-Sensitivity mode to control whether or not MATLAB considers letter case when matching an expression to a string. Example 6 illustrates the this mode.

Keyword	Flag	Description
'matchcase'	(?-i)	Letter case must match when matching patterns to a string. (The default for <code>regexp</code>).
'ignorecase'	(?i)	Do not consider letter case when matching patterns to a string. (The default for <code>regexpi</code>).

Dot Matching Mode

Use the Dot Matching mode to control whether or not MATLAB includes the newline (\n) character when matching the dot (.) metacharacter in a regular expression. Example 7 illustrates the Dot Matching mode.

regexp, regexpi

Mode Keyword	Flag	Description
'dotall'	(?s)	Match dot ('.') in the pattern string with any character. (This is the default).
'dotexceptnewline'	(?-s)	Match dot in the pattern with any character that is not a newline.

Anchor Type Mode

Use the Anchor Type mode to control whether MATLAB considers the ^ and \$ metacharacters to represent the beginning and end of a string or the beginning and end of a line. Example 8 illustrates the Anchor mode.

Mode Keyword	Flag	Description
'stringanchors'	(?m)	Match the ^ and \$ metacharacters at the beginning and end of a string. (This is the default).
'lineanchors'	(?m)	Match the ^ and \$ metacharacters at the beginning and end of a line.

Spacing Mode

Use the Spacing mode to control how MATLAB interprets space characters and comments within the string being parsed. Example 9 illustrates the Spacing mode.

Mode Keyword	Flag	Description
' literalspacing '	(?-x)	Parse space characters and comments (the # character and any text to the right of it) in the same way as any other characters in the string. (This is the default).
' freespacing '	(?x)	Ignore spaces and comments when parsing the string. (You must use '\ ' and '\#' to match space and # characters.)

Remarks

See “Regular Expressions” in the MATLAB Programming documentation for a listing of all regular expression elements supported by MATLAB.

Multiple Strings or Expressions

Either the str or expr argument, or both, can be a cell array of strings, according to the following guidelines:

- If str is a cell array of strings, then each of the regexp outputs is a cell array having the same dimensions as str.
- If str is a single string but expr is a cell array of strings, then each of the regexp outputs is a cell array having the same dimensions as expr.
- If both str and expr are cell arrays of strings, these two cell arrays must contain the same number of elements.

Examples

Example 1 – Matching a Simple Pattern

Return a row vector of indices that match words that start with c, end with t, and contain one or more vowels between them. Make the matches insensitive to letter case (by using regexpi):

```
str = 'bat cat can car COAT court cut ct CAT-scan';
```

regexp, regexpi

```
regexpi(str, 'c[aeiou]+t')
ans =
     5     17     28     35
```

Example 2 – Parsing Multiple Input Strings

Return a cell array of row vectors of indices that match capital letters and white spaces in the cell array of strings `str`:

```
str = {'Madrid, Spain' 'Romeo and Juliet' 'MATLAB is great'};
s1 = regexp(str, '[A-Z]');
s2 = regexp(str, '\s');
```

Capital letters, '[A-Z]', were found at these `str` indices:

```
s1{:}
ans =
     1     9
ans =
     1    11
ans =
     1     2     3     4     5     6
```

Space characters, '\s', were found at these `str` indices:

```
s2{:}
ans =
     8
ans =
     6    10
ans =
     7    10
```

Example 3 – Selecting Return Values

Return the text and the starting and ending indices of words containing the letter `x`:

```
str = 'regexp helps you relax';
[m s e] = regexp(str, '\w*x\w*', 'match', 'start', 'end')
```

```
m =  
    'regexp'    'relax'  
s =  
    1    18  
e =  
    6    22
```

Example 4 – Using Tokens

Search a string for opening and closing HTML tags. Use the expression `<(\w+)` to find the opening tag (e.g., `<tagname>`) and to create a token for it. Use the expression `</\1>` to find another occurrence of the same token, but formatted as a closing tag (e.g., `</tagname>`):

```
str = 'if <code>A</code> == x<sup>2</sup>, <em>disp(x)</em>';  
expr = '<(\w+).*?>.*?</\1>';  
  
[tok mat] = regexp(str, expr, 'tokens', 'match');  
  
tok{:}  
ans =  
    'code'  
ans =  
    'sup'  
ans =  
    'em'  
  
mat{:}  
ans =  
    <code>A</code>  
ans =  
    <sup>2</sup>  
ans =  
    <em>disp(x)</em>
```

See “Tokens” in the MATLAB Programming documentation for information on using tokens.

Example 5 – Using Named Capture

Enter a string containing two names, the first and last names being in a different order:

```
str = sprintf('John Davis\nRogers, James')
str =
    John Davis
    Rogers, James
```

Create an expression that generates first and last name tokens, assigning the names `first` and `last` to the tokens. Call `regexp` to get the text and names of each token found:

```
expr = ...
    '(?<first>\w+)\s+(?<last>\w+)|(?<last>\w+),\s+(?<first>\w+)';

[tokens names] = regexp(str, expr, 'tokens', 'names');
```

Examine the `tokens` cell array that was returned. The first and last name tokens appear in the order in which they were generated: first name–last name, then last name–first name:

```
tokens{:}
ans =
    'John'    'Davis'
ans =
    'Rogers'  'James'
```

Now examine the `names` structure that was returned. First and last names appear in a more usable order:

```
names(:,1)
ans =
    first: 'John'
    last:  'Davis'

names(:,2)
ans =
```



```
first: 'James'  
last: 'Rogers'
```

Example 6 – Using the Case-Sensitive Mode

Given a string that has both uppercase and lowercase letters,

```
str = 'A string with UPPERCASE and lowercase text.';
```

Use the regexp default mode (case-sensitive) to locate only the lowercase instance of the word case:

```
regexp(str, 'case', 'match')  
ans =  
    'case'
```

Now disable case-sensitive matching to find both instances of case:

```
regexp(str, 'case', 'ignorecase', 'match')  
ans =  
    'CASE'    'case'
```

Match 5 letters that are followed by 'CASE'. Use the (?-i) flag to turn on case-sensitivity for the first match and (?i) to turn it off for the second:

```
M = regexp(str, {'(?-i)\w{5}(?=CASE)', ...  
                '(?i)\w{5}(?=CASE)'} , 'match');  
  
M{:}  
ans =  
    'UPPER'  
ans =  
    'UPPER'    'lower'
```

Example 7 – Using the Dot Matching Mode

Parse the following string that contains a newline (\n) character:

```
str = sprintf('abc\ndef')  
str =
```

regexp, regexpi

```
abc
def
```

When you use the default mode, `dotall`, MATLAB includes the newline in the characters matched:

```
regexp(str, '.', 'match')
ans =
    'a'    'b'    'c'    [1x1 char]    'd'    'e'    'f'
```

When you use the `dotexceptnewline` mode, MATLAB skips the newline character:

```
regexp(str, '.', 'match', 'dotexceptnewline')
ans =
    'a'    'b'    'c'    'd'    'e'    'f'
```

Example 8 – Using the Anchor Type Mode

Given the following two-line string,

```
str = sprintf('%s\n%s', 'Here is the first line', ...
              'followed by the second line')
str =
    Here is the first line
    followed by the second line
```

In `stringanchors` mode, MATLAB interprets the `$` metacharacter as an end-of-string specifier, and thus finds the last two words of the entire *string*:

```
regexp(str, '\w+\W\w+$', 'match', 'stringanchors')
ans =
    'second line'
```

While in `lineanchors` mode, MATLAB interprets `$` as an end-of-line specifier, and finds the last two words of each *line*:

```
regexp(str, '\w+\W\w+$', 'match', 'lineanchors')
```

```
ans =
    'first line'    'second line'
```

Example 9 – Using the Freespacing Mode

Create a file called `regexp_str.txt` containing the following text. Because the first line enables freespacing mode, MATLAB ignores all spaces and comments that appear in the file:

```
(?x)  # turn on freespacing.

# This pattern matches a string with a repeated letter.

\w*   # First, match any number of preceding word characters.

(     # Mark a token.
  \w # Match a word character.
)     # Finish capturing said token.
\1    # Backreference to match what token #1 matched.

\w*   # Finally, match the remainder of the word.
```

Here is the string to parse:

```
str = ['Looking for words with letters that ' ...
      'appear twice in succession.'];
```

Use the pattern expression read from the file to find those words that have consecutive matching letters:

```
patt = fileread('regexp_str.txt');
regexp(str, patt, 'match')
ans =
    'Looking'    'letters'    'appear'    'succession'
```

Example 10 – Displaying Parsing Warnings

To help debug problems in parsing a string with `regexp`, `regexpi`, or `regexprep`, use the `'warnings'` option to view all warning messages:

regexp, regexp_i

```
regexp('$.', '[a-]', 'warnings')
Warning: Unbound range.
[a-]
|
```

See Also

regxprep, strfind, findstr, strmatch, strcmp, strcmpi, strncmp, strncmpi

Purpose Replace string using regular expression

Syntax
`s = regexprep('str', 'expr', 'repstr')`
`s = regexprep('str', 'expr', 'repstr' options)`

Description `s = regexprep('str', 'expr', 'repstr')` replaces all occurrences of the regular expression `expr` in string `str` with the string `repstr`. The new string is returned in `s`. If no matches are found, return string `s` is the same as input string `str`. You can use character representations (e.g., `'\t'` for tab, or `'\n'` for newline) in replacement string `repstr`.

If `str` is a cell array of strings, then the `regexprep` return value `s` is always a cell array of strings having the same dimensions as `str`.

To specify more than one expression to match or more than one replacement string, see the guidelines listed below under “Multiple Expressions or Replacement Strings” on page 2-2542.

You can capture parts of the input string as tokens and then reuse them in the replacement string. Specify the parts of the string to capture using the `(...)` operator. Specify the tokens to use in the replacement string using the operators `$1`, `$2`, `$N` to reference the first, second, and `N`th tokens captured. (See “Tokens” and the example “Using Tokens in a Replacement String” in the MATLAB Programming documentation for information on using tokens.)

`s = regexprep('str', 'expr', 'repstr' options)` By default, `regexprep` replaces all matches and is case sensitive. You can use one or more of the following options with `regexprep`.

Option	Description
<code>mode</code>	See mode descriptions on the <code>regexp</code> reference page.
<code>N</code>	Replace only the <code>N</code> th occurrence of <code>expr</code> in <code>str</code> .
<code>'once'</code>	Replace only the first occurrence of <code>expr</code> in <code>str</code> .
<code>'ignorecase'</code>	Ignore case when matching and when replacing.

Option	Description
'preservecase'	Ignore case when matching (as with 'ignorecase'), but override the case of replace characters with the case of corresponding characters in str when replacing.
'warnings'	Display any hidden warning messages issued by MATLAB during the execution of the command. This option only enables warnings for the one command being executed.

Remarks

See “Regular Expressions” in the MATLAB Programming documentation for a listing of all regular expression metacharacters supported by MATLAB.

Multiple Expressions or Replacement Strings

In the case of multiple expressions and/or replacement strings, `regexprep` attempts to make all matches and replacements. The first match is against the initial input string. Successive matches are against the string resulting from the previous replacement.

The `expr` and `repstr` inputs follow these rules:

- If `expr` is a cell array of strings and `repstr` is a single string, `regexprep` uses the same replacement string on each expression in `expr`.
- If `expr` is a single string and `repstr` is a cell array of `N` strings, `regexprep` attempts to make `N` matches and replacements.
- If both `expr` and `repstr` are cell arrays of strings, then `expr` and `repstr` must contain the same number of elements, and `regexprep` pairs each `repstr` element with its matching element in `expr`.

Examples

Example 1 – Making a Case-Sensitive Replacement

Perform a case-sensitive replacement on words starting with `m` and ending with `y`:

```
str = 'My flowers may bloom in May';
pat = 'm(\w*)y';
regexprep(str, pat, 'April')
ans =
    My flowers April bloom in May
```

Replace all words starting with `m` and ending with `y`, regardless of case, but maintain the original case in the replacement strings:

```
regexprep(str, pat, 'April', 'preserveCase')
ans =
    April flowers april bloom in April
```

Example 2 – Using Tokens In the Replacement String

Replace all variations of the words 'walk up' using the letters following walk as a token. In the replacement string

```
str = 'I walk up, they walked up, we are walking up.';
pat = 'walk(\w*) up';
regexprep(str, pat, 'ascend$1')
ans =
    I ascend, they ascended, we are ascending.
```

Example 3 – Operating on Multiple Strings

This example operates on a cell array of strings. It searches for consecutive matching letters (e.g., 'oo') and uses a common replacement value ('--') for all matches. The function returns a cell array of strings having the same dimensions as the input cell array:

```
str = {
    'Whose woods these are I think I know.' ; ...
    'His house is in the village though;' ; ...
    'He will not see me stopping here' ; ...
    'To watch his woods fill up with snow.'};

a = regexprep(str, '(.)\1', '--', 'ignoreCase')
a =
```

regexprep

```
'Whose w--ds these are I think I know.'  
'His house is in the vi--age though;'  
'He wi-- not s-- me sto--ing here'  
'To watch his w--ds fi-- up with snow.'
```

See Also

regexp, regexp_i, strfind, findstr, strmatch, strcmp, strcmpi, strncmp, strncmpi

Purpose Translate string into regular expression

Syntax `s2 = regexptranslate(type, s1)`

Description `s2 = regexptranslate(type, s1)` translates string `s1` into a regular expression string `s2` that you can then use as input into one of the MATLAB regular expression functions such as `regexp`. The `type` input can be either one of the following strings that define the type of translation to be performed.

Type	Description
'escape'	Translate all special characters (e.g., '\$', '.', '?', '[') in string <code>s1</code> so that they are treated as literal characters when used in the <code>regexp</code> and <code>regexprep</code> functions. The translation inserts an escape character ('\') before each special character in <code>s1</code> . Return the new string in <code>s2</code> .
'wildcard'	Translate all wildcard and '.' characters in string <code>s1</code> so that they are treated as literal wildcards and periods when used in the <code>regexp</code> and <code>regexprep</code> functions. The translation replaces all instances of '*' with '.', all instances of '?' with '.', and all instances of '.' with '\.'. Return the new string in <code>s2</code> .

Examples

Example 1 – Using the 'escape' Option

Because `regexp` interprets the sequence '\n' as a newline character, it cannot locate the two consecutive characters '\n' and 'n' in this string:

```
str = 'The sequence \n generates a new line';
pat = '\n';

regexp(str, pat)
ans =
     []
```

regexptranslate

To have `regexp` interpret the expression `expr` as the characters `'\'` and `'n'`, first translate the expression using `regexptranslate`:

```
pat2 = regexptranslate('escape', pat)
pat2 =
    \n

regexp(str, pat2)
ans =
    14
```

Example 2 – Using 'escape' In a Replacement String

Replace the word 'walk' with 'ascend' in this string, treating the characters '\$1' as a token designator:

```
str = 'I walk up, they walked up, we are walking up.';
pat = 'walk(\w*) up';

regexprep(str, pat, 'ascend$1')
ans =
    I ascend, they ascended, we are ascending.
```

Make another replacement on the same string, this time treating the '\$1' as literal characters:

```
regexprep(str, pat, regexptranslate('escapereplace', 'ascend$1'))
ans =
    I ascend$1, they ascend$1, we are ascend$1.
```

Example 3 – Using the 'wildcard' Option

Given the following string of filenames, pick out just the MAT-files. Use `regexptranslate` to interpret the '*' wildcard as '\w+' instead of as a regular expression quantifier:

```
files = ['test1.mat, myfile.mat, newfile.txt, ' ...
        'jan30.mat, table3.xls'];
regexp(str, regexptranslate('wildcard', '*.mat'), 'match')
```

```
ans =  
    'test1.mat' 'myfile.mat' 'jan30.mat'
```

To see the translation, you can type

```
regexptranslate('wildcard', '*.mat')  
ans =  
    \w+\.mat
```

See Also

regexp, regexpi, regexprep

registerevent

Purpose Register event handler with control's event

Syntax `h.registerevent(event_handler)`
`registerevent(h, event_handler)`

Description `h.registerevent(event_handler)` registers certain event handler routines with their corresponding events. Once an event is registered, the control responds to the occurrence of that event by invoking its event handler routine. The `event_handler` argument can be either a string that specifies the name of the event handler function, or a function handle that maps to that function.

`registerevent(h, event_handler)` is an alternate syntax for the same operation.

You can either register events at the time you create the control (using `actxcontrol`), or register them dynamically at any time after the control has been created (using `registerevent`). Both events and event handlers are specified in the `event_handler` argument (see [Specifying Event Handlers in the External Interfaces documentation](#)).

Examples

Example 1

Create an `mwsamp` control and list all events associated with the control:

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

h.events
ans =
    Click = void Click()
    DblClick = void DblClick()
    MouseDown = void MouseDown(int16 Button, int16 Shift,
        Variant x, Variant y)
```

Register all events with the same event handler routine, `sampev`. Use the `eventlisteners` function to see the event handler used by each event:

```
h.registerevent('sampev');
h.eventlisteners
ans =
    'click'          'sampev'
    'dblclick'      'sampev'
    'mousedown'     'sampev'

h.unregisterallevents;
```

Register the Click and DblClick events with event handlers myclick and my2click, respectively:

```
h.registerevent({'click' 'myclick'; 'dblclick' 'my2click'});
h.eventlisteners
ans =
    'click'          'myclick'
    'dblclick'      'my2click'
```

Example 2

Register all events with the same event handler routine, sampev, but use a function handle (@sampev) instead of the function name:

```
h = actxcontrol('mwsamp.mwsampctrl1.2', [0 0 200 200]);
registerevent(h, @sampev);
```

See Also

events, eventlisteners, unregisterevent, unregisterallevents, isevent

rehash

Purpose Refresh function and file system path caches

Syntax

```
rehash  
rehash path  
rehash toolbox  
rehash pathreset  
rehash toolboxreset  
rehash toolboxcache
```

Description `rehash` with no arguments updates the MATLAB list of known files and classes for directories on the search path that are not in `matlabroot/toolbox`. It compares the timestamps for loaded shadowed functions (functions that have been called but not cleared in the current session) against their timestamps on disk. It clears loaded functions if the files on disk are newer. All of this normally happens each time MATLAB displays the Command Window prompt. Therefore, use `rehash` with no arguments only when you run an M-file that updates another M-file, and the calling file needs to reuse the updated version before it has finished running.

`rehash path` performs the same updates as `rehash`, but uses a different technique for detecting the files and directories that require updates. If you receive a warning during MATLAB startup notifying you that MATLAB could not tell if a directory has changed and you encounter problems with MATLAB using the most current versions of your M-files, run `rehash path`.

`rehash toolbox` updates all directories in `matlabroot/toolbox`. Run this when you add or remove files in `matlabroot/toolbox` during a session by some means other than MATLAB tools.

`rehash pathreset` performs the same updates as `rehash path`, and also ensures the known files and classes list follows precedence rules for shadowed functions.

`rehash toolboxreset` performs the same updates as `rehash toolbox`, and also ensures the known files and classes list follows precedence rules for shadowed functions.

rehash **toolboxcache** performs the same updates as rehash **toolbox**, and also updates the cache file. This is the equivalent of clicking the **Update Toolbox Path Cache** button in **Preferences > General**.

See Also

addpath, clear, path, rmpath

“Toolbox Path Caching in MATLAB” in the MATLAB Desktop Tools and Development Environment documentation

release

Purpose Release interface

Syntax `h.release`
`release(h)`

Description `h.release` releases the interface and all resources used by the interface. Each interface handle must be released when you are finished manipulating its properties and invoking its methods. Once an interface has been released, it is no longer valid. Subsequent operations on the MATLAB object that represents that interface will result in errors.

`release(h)` is an alternate syntax for the same operation.

Note Releasing the interface does not delete the control itself (see `delete`), since other interfaces on that object may still be active. See *Releasing Interfaces in the External Interfaces documentation* for more information.

Examples Create a Microsoft Calender application. Then create a `TitleFont` interface and use it to change the appearance of the font of the calendar's title:

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

TFont = cal.TitleFont
TFont =
    Interface.Standard_OLE_Types.Font

TFont.Name = 'Viva BoldExtraExtended';
TFont.Bold = 0;
```

When you're finished working with the title font, release the `TitleFont` interface:


```
TFont.release;
```

Now create a GridFont interface and use it to modify the size of the calendar's date numerals:

```
GFont = cal.GridFont
GFont =
    Interface.Standard_OLE_Types.Font

GFont.Size = 16;
```

When you're done, delete the cal object and the figure window:

```
cal.delete;
delete(f);
clear f;
```

See Also

delete, save, load, actxcontrol, actxserver

rem

Purpose Remainder after division

Syntax $R = \text{rem}(X, Y)$

Description $R = \text{rem}(X, Y)$ if $Y \neq 0$, returns $X - n \cdot Y$ where $n = \text{fix}(X./Y)$. If Y is not an integer and the quotient $X./Y$ is within roundoff error of an integer, then n is that integer. The inputs X and Y must be real arrays of the same size, or real scalars.

The following are true by convention:

- $\text{rem}(X, 0)$ is NaN
- $\text{rem}(X, X)$ for $X \neq 0$ is 0
- $\text{rem}(X, Y)$ for $X \neq Y$ and $Y \neq 0$ has the same sign as X .

Remarks $\text{mod}(X, Y)$ for $X \neq Y$ and $Y \neq 0$ has the same sign as Y .

$\text{rem}(X, Y)$ and $\text{mod}(X, Y)$ are equal if X and Y have the same sign, but differ by Y if X and Y have different signs.

The `rem` function returns a result that is between 0 and $\text{sign}(X) \cdot \text{abs}(Y)$. If Y is zero, `rem` returns NaN.

See Also `mod`

Purpose Remove timeseries objects from tscollection object

Syntax `tsc = removets(tsc,Name)`

Description `tsc = removets(tsc,Name)` removes one or more timeseries objects with the name specified in `Name` from the `tscollection` object `tsc`. `Name` can either be a string or a cell array of strings.

Examples The following example shows how to remove a time series from a `tscollection`.

1 Create two timeseries objects, `ts1` and `ts2`.

```
ts1=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'name','acceleration');  
ts2=timeseries([3.2 4.2 6.2 8.5 1.1],1:5,'name','speed');
```

2 Create a `tscollection` object `tsc`, which includes `ts1` and `ts2`.

```
tsc=tscollection({ts1 ts2});
```

3 To view the members of `tsc`, type the following at the MATLAB prompt:

```
tsc
```

MATLAB responds with

```
Time Series Collection Object: unnamed
```

```
Time vector characteristics
```

```
Start time          1 seconds  
End time            5 seconds
```

```
Member Time Series Objects:
```

removets

```
acceleration
speed
```

The members of `tsc` are listed by name at the bottom: `acceleration` and `speed`. These are the `Name` properties of `ts1` and `ts2`, respectively.

- 4 Remove `ts2` from `tsc`.

```
tsc=removets(tsc,'speed');
```

- 5 To view the current members of `tsc`, type the following at the MATLAB prompt:

```
tsc
```

MATLAB responds with

```
Time Series Collection Object: unnamed
```

```
Time vector characteristics
```

```
Start time          1 seconds
End time            5 seconds
```

```
Member Time Series Objects:
acceleration
```

The remaining member of `tsc` is `acceleration`. The timeseries `speed` has been removed.

See Also

`addts`, `tscollection`

Purpose Rename file on FTP server

Syntax `rename(f, 'oldname', 'newname')`

Description `rename(f, 'oldname', 'newname')` changes the name of the file `oldname` to `newname` in the current directory of the FTP server `f`, where `f` was created using `ftp`.

Examples Connect to server `testsite`, view the contents, and change the name of `testfile.m` to `showresults.m`.

```
test=ftp('ftp.testsite.com');
dir(test)
.          ..          testfile.m
rename(test, 'testfile.m', 'showresults.m')
dir(test)
.          ..          showresults.m
```

See Also `dir (ftp)`, `delete (ftp)`, `ftp`, `mget`, `mput`

repmat

Purpose Replicate and tile array

Syntax
B = repmat(A,m,n)
B = repmat(A,[m n])
B = repmat(A,[m n p...])

Description B = repmat(A,m,n) creates a large matrix B consisting of an m-by-n tiling of copies of A. The size of B is [size(A,1)*m, (size(A,2)*n)]. The statement repmat(A,n) creates an n-by-n tiling.
B = repmat(A,[m n]) accomplishes the same result as repmat(A,m,n).
B = repmat(A,[m n p...]) produces a multidimensional array B composed of copies of A. The size of B is [size(A,1)*m, size(A,2)*n, size(A,3)*p, ...].

Remarks repmat(A,m,n), when A is a scalar, produces an m-by-n matrix filled with A's value and having A's class. For certain values, you can achieve the same results using other functions, as shown by the following examples:

- repmat(NaN,m,n) returns the same result as NaN(m,n).
- repmat(single(inf),m,n) is the same as inf(m,n,'single').
- repmat(int8(0),m,n) is the same as zeros(m,n,'int8').
- repmat(uint32(1),m,n) is the same as ones(m,n,'uint32').
- repmat(eps,m,n) is the same as eps(ones(m,n)).

Examples In this example, repmat replicates 12 copies of the second-order identity matrix, resulting in a “checkerboard” pattern.

```
B = repmat(eye(2),3,4)
```

```
B =  
    1     0     1     0     1     0     1     0  
    0     1     0     1     0     1     0     1  
    1     0     1     0     1     0     1     0
```

```
0    1    0    1    0    1    0    1
1    0    1    0    1    0    1    0
0    1    0    1    0    1    0    1
```

The statement `N = repmat(NaN,[2 3])` creates a 2-by-3 matrix of NaNs.

See Also

NaN, Inf, ones, zeros

resample (timeseries)

Purpose Select or interpolate timeseries data using new time vector

Syntax

```
ts = resample(ts,Time)
ts = resample(ts,Time,interp_method)
ts = resample(ts,Time,interp_method,code)
```

Description

`ts = resample(ts,Time)` resamples the timeseries object `ts` using the new `Time` vector. When `ts` uses date strings and `Time` is numeric, `Time` is treated as specified relative to the `ts.TimeInfo.StartDate` property and in the same units that `ts` uses. The resample operation uses the default interpolation method, which you can view by using the `getinterpmethod(ts)` syntax.

`ts = resample(ts,Time,interp_method)` resamples the timeseries object `ts` using the interpolation method given by the string `interp_method`. Valid interpolation methods include 'linear' and 'zoh' (zero-order hold).

`ts = resample(ts,Time,interp_method,code)` resamples the timeseries object `ts` using the interpolation method given by the string `interp_method`. The integer `code` is a user-defined Quality code for resampling, applied to all samples.

Examples The following example shows how to resample a timeseries object.

1 Create a timeseries object.

```
ts=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'Name','speed');
```

2 Transpose `ts` to make the data columnwise.

```
ts=transpose(ts)
```

MATLAB displays

```
Time Series Object: speed
```

```
Time vector characteristics
```



```
Length          5
Start time      1 seconds
End time        5 seconds
```

Data characteristics

```
Interpolation method linear
Size              [5 1]
Data type         double
```

Time	Data	Quality
1	1.1	
2	2.9	
3	3.7	
4	4	
5	3	

Note that the interpolation method is set to linear, by default.

3 Resample ts using its default interpolation method.

```
res_ts=resample(ts,[1 1.5 3.5 4.5 4.9])
```

MATLAB displays the resampled time series as follows:

```
Time Series Object: speed
```

Time vector characteristics

```
Length          5
Start time      1 seconds
End time        4.900000e+000 seconds
```

resample (timeseries)

Data characteristics

Interpolation method	linear
Size	[5 1]
Data type	double

Time	Data	Quality
1	1.1	
1.5	2	
3.5	3.85	
4.5	3.5	
4.9	3.1	

See Also

getinterpmethod, setinterpmethod, synchronize, timeseries

Purpose Select or interpolate data in `tscollection` using new time vector

Syntax

```
tsc = resample(tsc,Time)
tsc = resample(tsc,Time,interp_method)
tsc = resample(tsc,Time,interp_method,code)
```

Description

`tsc = resample(tsc,Time)` resamples the `tscollection` object `tsc` on the new `Time` vector. When `tsc` uses date strings and `Time` is numeric, `Time` is treated as numerical specified relative to the `tsc.TimeInfo.StartDate` property and in the same units that `tsc` uses. The `resample` method uses the default interpolation method for each time series member.

`tsc = resample(tsc,Time,interp_method)` resamples the `tscollection` object `tsc` using the interpolation method given by the string `interp_method`. Valid interpolation methods include 'linear' and 'zoh' (zero-order hold).

`tsc = resample(tsc,Time,interp_method,code)` resamples the `tscollection` object `tsc` using the interpolation method given by the string `interp_method`. The integer `code` is a user-defined quality code for resampling, applied to all samples.

Examples The following example shows how to resample a `tscollection` that consists of two `timeseries` members.

1 Create two `timeseries` objects.

```
ts1=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'name','acceleration');
ts2=timeseries([3.2 4.2 6.2 8.5 1.1],1:5,'name','speed');
```

2 Create a `tscollection` `tsc`.

```
tsc=tscollection({ts1 ts2});
```

The time vector of the collection `tsc` is `[1:5]`, which is the same as for `ts1` and `ts2` (individually).

resample (tscollection)

- 3** Get the interpolation method for acceleration by typing

```
tsc.acceleration
```

MATLAB responds with

```
Time Series Object: acceleration
```

```
Time vector characteristics
```

```
Length          5
Start time      1 seconds
End time        5 seconds
```

```
Data characteristics
```

```
Interpolation method linear
Size                [1 1 5]
Data type           double
```

- 4** Set the interpolation method for speed to zero-order hold by typing

```
setinterpmethod(tsc.speed, 'zoh')
```

MATLAB responds with

```
Time Series Object: acceleration
```

```
Time vector characteristics
```

```
Length          5
Start time      1 seconds
End time        5 seconds
```

Data characteristics

Interpolation method	zoh
Size	[1 1 5]
Data type	double

- 5 Resample the time-series collection `tsc` by individually resampling each time-series member of the collection and using its interpolation method.

```
res_tsc=resample(tsc,[1 1.5 3.5 4.5 4.9])
```

See Also

`getinterpmethod`, `setinterpmethod`, `tscollection`

reset

Purpose Reset graphics object properties to their defaults

Syntax `reset(h)`

Description `reset(h)` resets all properties having factory defaults on the object identified by `h`. To see the list of factory defaults, use the statement

```
get(0, 'factory')
```

If `h` is a figure, MATLAB does not reset `Position`, `Units`, `Windowstyle`, or `PaperUnits`. If `h` is an axes, MATLAB does not reset `Position` and `Units`.

Examples `reset(gca)` resets the properties of the current axes.
`reset(gcf)` resets the properties of the current figure.

See Also `cla`, `clf`, `gca`, `gcf`, `hold`
“Object Manipulation” on page 1-97 for related functions

Purpose

Reshape array

Syntax

```
B = reshape(A,m,n)
B = reshape(A,m,n,p,...)
B = reshape(A,[m n p ...])
B = reshape(A,...,[],...)
B = reshape(A,siz)
```

Description

`B = reshape(A,m,n)` returns the m -by- n matrix B whose elements are taken column-wise from A . An error results if A does not have $m*n$ elements.

`B = reshape(A,m,n,p,...)` or `B = reshape(A,[m n p ...])` returns an n -dimensional array with the same elements as A but reshaped to have the size m -by- n -by- p -by-... The product of the specified dimensions, $m*n*p*...$, must be the same as `prod(size(A))`.

`B = reshape(A,...,[],...)` calculates the length of the dimension represented by the placeholder `[]`, such that the product of the dimensions equals `prod(size(A))`. The value of `prod(size(A))` must be evenly divisible by the product of the specified dimensions. You can use only one occurrence of `[]`.

`B = reshape(A,siz)` returns an n -dimensional array with the same elements as A , but reshaped to `siz`, a vector representing the dimensions of the reshaped array. The quantity `prod(siz)` must be the same as `prod(size(A))`.

Examples

Reshape a 3-by-4 matrix into a 2-by-6 matrix.

```
A =
     1     4     7    10
     2     5     8    11
     3     6     9    12
```

```
B = reshape(A,2,6)
```

```
B =
```

reshape

```
      1   3   5   7   9  11
      2   4   6   8  10  12
B = reshape(A,2,[])
```

```
B =
      1   3   5   7   9  11
      2   4   6   8  10  12
```

See Also

`shiftdim`, `squeeze`

The colon operator :

Purpose Convert between partial fraction expansion and polynomial coefficients

Syntax
 $[r,p,k] = \text{residue}(b,a)$
 $[b,a] = \text{residue}(r,p,k)$

Description The residue function converts a quotient of polynomials to pole-residue representation, and back again.

$[r,p,k] = \text{residue}(b,a)$ finds the residues, poles, and direct term of a partial fraction expansion of the ratio of two polynomials, $b(s)$ and $a(s)$, of the form

$$\frac{b(s)}{a(s)} = \frac{b_1 s^m + b_2 s^{m-1} + b_3 s^{m-2} + \dots + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + a_3 s^{n-2} + \dots + a_{n+1}}$$

where b_j and a_j are the j th elements of the input vectors b and a .

$[b,a] = \text{residue}(r,p,k)$ converts the partial fraction expansion back to the polynomials with coefficients in b and a .

Definition If there are no multiple roots, then

$$\frac{b(s)}{a(s)} = \frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} + \dots + \frac{r_n}{s-p_n} + k(s)$$

The number of poles n is

$$n = \text{length}(a) - 1 = \text{length}(r) = \text{length}(p)$$

The direct term coefficient vector is empty if $\text{length}(b) < \text{length}(a)$; otherwise

$$\text{length}(k) = \text{length}(b) - \text{length}(a) + 1$$

If $p(j) = \dots = p(j+m-1)$ is a pole of multiplicity m , then the expansion includes terms of the form

residue

$$\frac{r_j}{s-p_j} + \frac{r_{j+1}}{(s-p_j)^2} + \dots + \frac{r_{j+m-1}}{(s-p_j)^m}$$

Arguments

- `b, a` Vectors that specify the coefficients of the polynomials in descending powers of `s`
- `r` Column vector of residues
- `p` Column vector of poles
- `k` Row vector of direct terms

Algorithm

It first obtains the poles with roots. Next, if the fraction is nonproper, the direct term `k` is found using `deconv`, which performs polynomial long division. Finally, the residues are determined by evaluating the polynomial with individual roots removed. For repeated roots, `resid2` computes the residues at the repeated root locations.

Limitations

Numerically, the partial fraction expansion of a ratio of polynomials represents an ill-posed problem. If the denominator polynomial, $a(s)$, is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can make arbitrarily large changes in the resulting poles and residues. Problem formulations making use of state-space or zero-pole representations are preferable.

Examples

If the ratio of two polynomials is expressed as

$$\frac{b(s)}{a(s)} = \frac{5s^3 + 3s^2 - 2s + 7}{-4s^3 + 8s + 3}$$

then

$$\begin{aligned} b &= [5 \ 3 \ -2 \ 7] \\ a &= [-4 \ 0 \ 8 \ 3] \end{aligned}$$

and you can calculate the partial fraction expansion as

$$[r, p, k] = \text{residue}(b,a)$$

$$r = \begin{array}{l} -1.4167 \\ -0.6653 \\ 1.3320 \end{array}$$

$$p = \begin{array}{l} 1.5737 \\ -1.1644 \\ -0.4093 \end{array}$$

$$k = \begin{array}{l} -1.2500 \end{array}$$

Now, convert the partial fraction expansion back to polynomial coefficients.

$$[b,a] = \text{residue}(r,p,k)$$

$$b = \begin{array}{cccc} -1.2500 & -0.7500 & 0.5000 & -1.7500 \end{array}$$

$$a = \begin{array}{cccc} 1.0000 & -0.0000 & -2.0000 & -0.7500 \end{array}$$

The result can be expressed as

$$\frac{b(s)}{a(s)} = \frac{-1.25s^3 - 0.75s^2 + 0.50s - 1.75}{s^3 - 2.00s - 0.75}$$

Note that the result is normalized for the leading coefficient in the denominator.

See Also

deconv, poly, roots

References

- [1] Oppenheim, A.V. and R.W. Schaffer, *Digital Signal Processing*, Prentice-Hall, 1975, p. 56.

Purpose Restore default MATLAB search path

Syntax `restoredefaultpath`
`restoredefaultpath; matlabrc`

Description `restoredefaultpath` sets the search path to include only installed products from The MathWorks. Run `restoredefaultpath` if you are having problems with the search path. If `restoredefaultpath` seems to correct the problem, run `savepath`. Start MATLAB again to be sure the problem does not reappear.

`restoredefaultpath; matlabrc` sets the search path to include only installed products from The MathWorks and corrects path problems encountered during startup. Run `restoredefaultpath; matlabrc` if you are having problems with the search path and `restoredefaultpath` by itself does not correct the problem. After the problem seems to be resolved, run `savepath`. Start MATLAB again to be sure the problem does not reappear.

See Also `addpath`, `path`, `pathdef`, `rmpath`, `savepath`

Search Path in the MATLAB Desktop Tools and Development Environment documentation

rethrow

Purpose Reissue error

Syntax rethrow(err)

Description rethrow(err) reissues the error specified by err. The currently running M-file terminates and control returns to the keyboard (or to any enclosing catch block). The err argument must be a MATLAB structure containing at least one of the following fields.

Fieldname	Description
message	Text of the error message
identifier	Message identifier of the error message
stack	Information about the error from the program stack

See Message Identifiers in the MATLAB documentation for more information on the syntax and usage of message identifiers.

A convenient way to get a valid err structure for the last error issued is by using the lasterror function.

Remarks The err input can contain the field stack, identical in format to the output of the dbstack command. If the stack field is present, the stack of the rethrown error will be set to that value. Otherwise, the stack will be set to the line at which the rethrow occurs.

Examples rethrow is usually used in conjunction with try-catch statements to reissue an error from a catch block after performing catch-related operations. For example,

```
try
    do_something
catch
    do_cleanup
    rethrow(lasterror)
end
```

See Also error, lasterror, try, catch, dbstop

return

Purpose Return to invoking function

Syntax return

Description return causes a normal return to the invoking function or to the keyboard. It also terminates keyboard mode.

Examples If the determinant function were an M-file, it might use a return statement in handling the special case of an empty matrix, as follows:

```
function d = det(A)
%DET det(A) is the determinant of A.
if isempty(A)
    d = 1;
    return
else
    ...
end
```

See Also break, continue, disp, end, error, for, if, keyboard, switch, while

Purpose Convert RGB colormap to HSV colormap

Syntax

```
cmap = rgb2hsv(M)  
hsv_image = rgb2hsv(rgb_image)
```

Description `cmap = rgb2hsv(M)` converts an RGB colormap `M` to an HSV colormap `cmap`. Both colormaps are m -by-3 matrices. The elements of both colormaps are in the range 0 to 1.

The columns of the input matrix `M` represent intensities of red, green, and blue, respectively. The columns of the output matrix `cmap` represent hue, saturation, and value, respectively.

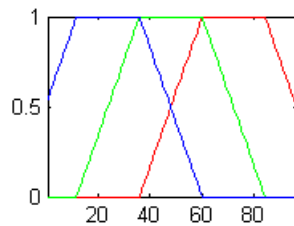
`hsv_image = rgb2hsv(rgb_image)` converts the RGB image to the equivalent HSV image. RGB is an m -by- n -by-3 image array whose three planes contain the red, green, and blue components for the image. HSV is returned as an m -by- n -by-3 image array whose three planes contain the hue, saturation, and value components for the image.

See Also `brighten`, `colormap`, `hsv2rgb`, `rgbplot`
“Color Operations” on page 1-95 for related functions

rgbplot

Purpose

Plot colormap



Syntax

```
rgbplot(cmap)
```

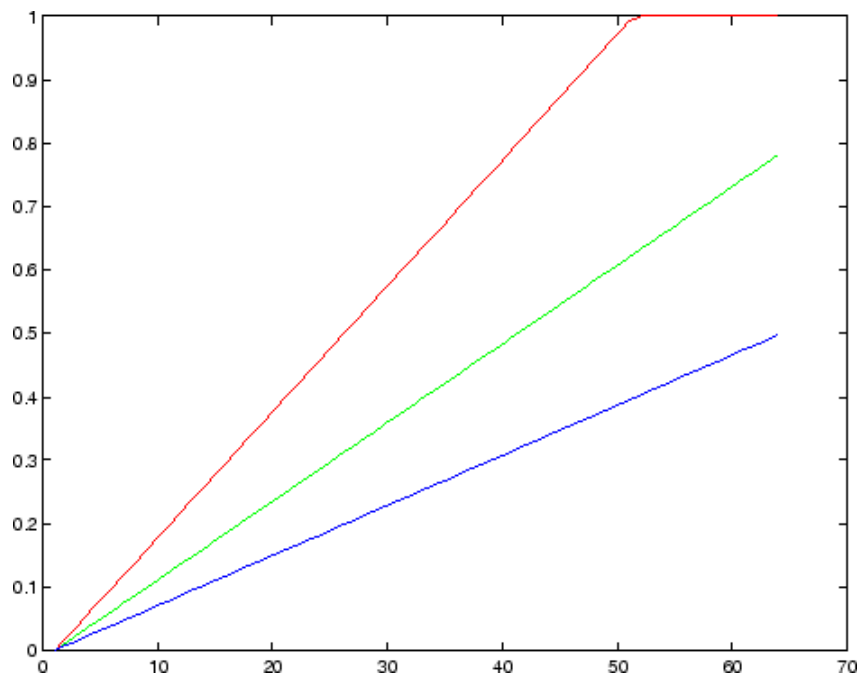
Description

`rgbplot(cmap)` plots the three columns of `cmap`, where `cmap` is an m -by-3 colormap matrix. `rgbplot` draws the first column in red, the second in green, and the third in blue.

Examples

Plot the RGB values of the copper colormap.

```
rgbplot(copper)
```



See Also

`colormap`

“Color Operations” on page 1-95 for related functions


ribbon

Purpose

Ribbon plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
ribbon(Y)
ribbon(X,Y)
ribbon(X,Y,width)
ribbon(axes_handle,...)
h = ribbon(...)
```

Description

`ribbon(Y)` plots the columns of `Y` as separate three-dimensional ribbons using `X = 1:size(Y,1)`.

`ribbon(X,Y)` plots `X` versus the columns of `Y` as three-dimensional strips. `X` and `Y` are vectors of the same size or matrices of the same size. Additionally, `X` can be a row or a column vector, and `Y` a matrix with `length(X)` rows.

`ribbon(X,Y,width)` specifies the width of the ribbons. The default is 0.75.

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

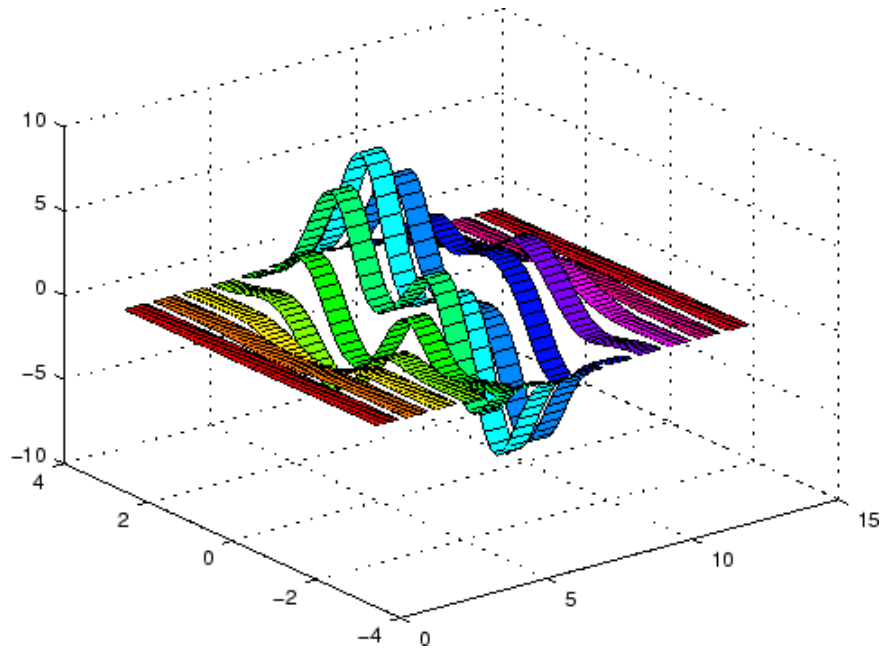
`h = ribbon(...)` returns a vector of handles to surface graphics objects. `ribbon` returns one handle per strip.

Examples

Create a ribbon plot of the peaks function.

```
[x,y] = meshgrid(-3:.5:3,-3:.1:3);
```

```
z = peaks(x,y);  
ribbon(y,z)  
colormap hsv
```

**See Also**

plot, plot3, surface, waterfall

“Polygons and Surfaces” on page 1-86 for related functions

rmappdata

Purpose Remove application-defined data

Syntax `rmappdata(h, name)`

Description `rmappdata(h, name)` removes the application-defined data name from the object specified by handle `h`.

See Also `getappdata`, `isappdata`, `setappdata`

Purpose	Remove directory
Graphical Interface	As an alternative to the <code>rmdir</code> function, use the delete feature in the “Current Directory Browser”.
Syntax	<pre>rmdir('dirname') rmdir('dirname','s') [status, message, messageid] = rmdir('dirname','s')</pre>
Description	<p><code>rmdir('dirname')</code> removes the directory <code>dirname</code> from the current directory. If the directory is not empty, you must use the <code>s</code> argument. If <code>dirname</code> is not in the current directory, specify the relative path to the current directory or the full path for <code>dirname</code>.</p> <p><code>rmdir('dirname','s')</code> removes the directory <code>dirname</code> and its contents from the current directory. This removes all subdirectories and files in the current directory regardless of their write permissions.</p> <p><code>[status, message, messageid] = rmdir('dirname','s')</code> removes the directory <code>dirname</code> and its contents from the current directory, returning the status, a message, and the MATLAB error message ID (see <code>error</code> and <code>lasterror</code>). Here, <code>status</code> is 1 for success and is 0 for error, and <code>message</code>, <code>messageid</code>, and the <code>s</code> input argument are optional.</p>
Examples	<p>Remove Empty Directory</p> <p>To remove <code>myfiles</code> from the current directory, where <code>myfiles</code> is empty, type</p> <pre>rmdir('myfiles')</pre> <p>If the current directory is <code>matlabr13/work</code>, and <code>myfiles</code> is in <code>d:/matlabr13/work/project/</code>, use the relative path to <code>myfiles</code></p> <pre>rmdir('project/myfiles')</pre> <p>or the full path to <code>myfiles</code></p> <pre>rmdir('d:/matlabr13/work/project/myfiles')</pre>

Remove Directory and All Contents

To remove myfiles, its subdirectories, and all files in the directories, assuming myfiles is in the current directory, type

```
rmdir('myfiles','s')
```

Remove Directory and Return Results

To remove myfiles from the current directory, type

```
[stat, mess, id]=rmdir('myfiles')
```

MATLAB returns

```
stat =  
      0
```

```
mess =
```

```
The directory is not empty.
```

```
id =
```

```
MATLAB:RMDIR:OSError
```

indicating the directory myfiles is not empty.

To remove myfiles and its contents, run

```
[stat, mess]=rmdir('myfiles','s')
```

and MATLAB returns

```
stat =  
      1
```

```
mess =
```

```
''
```


indicating myfiles and its contents were removed.

See Also

cd, copyfile, delete, dir, error, fileattrib, filebrowser, lasterror, mkdir, movefile

rmdir (ftp)

Purpose Remove directory on FTP server

Syntax `rmdir(f, 'dirname')`

Description `rmdir(f, 'dirname')` removes the directory `dirname` from the current directory of the FTP server `f`, where `f` was created using `ftp`.

Examples Connect to server `testsite`, view the contents of `testdir`, and remove the directory `newdir` from the directory `testdir`.

```
test=ftp('ftp.testsite.com');
cd(test, 'testdir');
dir(test)
.          ..          newdir
dir(test, 'newdir')
.          ..
rmdir(test, 'newdir');
dir(test, 'testdir')
.          ..
```

See Also `cd (ftp)`, `delete (ftp)`, `dir (ftp)`, `ftp`, `mkdir (ftp)`

Purpose Remove fields from structure

Syntax `s = rmfield(s, 'fieldname')`
`s = rmfield(s, fields)`

Description `s = rmfield(s, 'fieldname')` removes the specified field from the structure array `s`.

`s = rmfield(s, fields)` removes more than one field at a time. `fields` is a character array of field names or cell array of strings.

See Also `fieldnames`, `setfield`, `getfield`, `isfield`, `orderfields`, “Using Dynamic Field Names”

rmpath

Purpose	Remove directories from MATLAB search path
GUI Alternatives	As an alternative to the <code>rmpath</code> function, use the Set Path dialog box. To open it, select File > Set Path in the MATLAB desktop.
Syntax	<pre>rmpath('directory') rmpath directory</pre>
Description	<p><code>rmpath('directory')</code> removes the specified directory from the current MATLAB search path. Use the full pathname for <code>directory</code>.</p> <p><code>rmpath directory</code> is the command form of the syntax.</p>
Examples	Remove <code>/usr/local/matlab/mytools</code> from the search path. <pre>rmpath /usr/local/matlab/mytools</pre>
See Also	<code>addpath</code> , <code>cd</code> , <code>dir</code> , <code>genpath</code> , <code>matlabroot</code> , <code>partialpath</code> , <code>path</code> , <code>pathdef</code> , <code>pathsep</code> , <code>pathtool</code> , <code>rehash</code> , <code>restoredefaultpath</code> , <code>savepath</code> , <code>what</code> Search Path in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Remove preference

Syntax

```
rmpref('group','pref')  
rmpref('group',{'pref1','pref2',... 'prefn'})  
rmpref('group')
```

Description

`rmpref('group','pref')` removes the preference specified by `group` and `pref`. It is an error to remove a preference that does not exist.

`rmpref('group',{'pref1','pref2',... 'prefn'})` removes each preference specified in the cell array of preference names. It is an error if any of the preferences do not exist.

`rmpref('group')` removes all the preferences for the specified group. It is an error to remove a group that does not exist.

Examples

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

See Also

`addpref`, `getpref`, `ispref`, `setpref`, `uigetpref`, `uisetpref`

root object

Purpose

Root object properties

Description

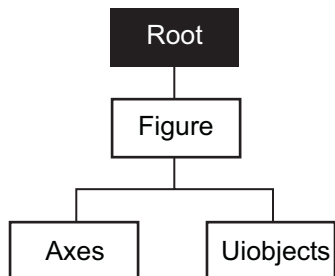
The root is a graphics object that corresponds to the computer screen. There is only one root object and it has no parent. The children of the root object are figures.

The root object exists when you start MATLAB; you never have to create it and you cannot destroy it. Use `set` and `get` to access the root properties.

See Also

`diary`, `echo`, `figure`, `format`, `gcf`, `get`, `set`

Object Hierarchy



Purpose

Defines root properties.

Modifying Properties

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

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

To change the default values of properties, see “Setting Default Property Values”.

Root Properties

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

BusyAction
cancel | {queue}

Not used by the root object.

ButtonDownFcn
string

Not used by the root object.

CallbackObject
handle (read only)

Handle of current callback’s object. This property contains the handle of the object whose callback routine is currently executing. If no callback routines are executing, this property contains the empty matrix []. See also the gco command.

CaptureMatrix
(obsolete)

This property has been superseded by the getframe command.

Root Properties

CaptureRect
(obsolete)

This property has been superseded by the `getframe` command.

Children
vector of handles

Handles of child objects. A vector containing the handles of all nonhidden figure objects (see `HandleVisibility` for more information). You can change the order of the handles and thereby change the stacking order of the figures on the display.

Clipping
{on} | off

Clipping has no effect on the root object.

CommandWindowSize
[columns rows]

Current size of command window. This property contains the size of the MATLAB command window in a two-element vector. The first element is the number of columns wide and the second element is the number of rows tall.

CreateFcn
The root does not use this property.

CurrentFigure
figure handle

Handle of the current figure window, which is the one most recently created, clicked in, or made current with the statement

```
figure(h)
```

which restacks the figure to the top of the screen, or

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


which does not restack the figures. In these statements, `h` is the handle of an existing figure. If there are no figure objects,

```
get(0, 'CurrentFigure')
```

returns the empty matrix. Note, however, that `gcf` always returns a figure handle, and creates one if there are no figure objects.

DeleteFcn
string

This property is not used, because you cannot delete the root object.

Diary
on | {off}

Diary file mode. When this property is on, MATLAB maintains a file (whose name is specified by the `DiaryFile` property) that saves a copy of all keyboard input and most of the resulting output. See also the `diary` command.

DiaryFile
string

Diary filename. The name of the diary file. The default name is `diary`.

Echo
on | {off}

Script echoing mode. When `Echo` is on, MATLAB displays each line of a script file as it executes. See also the `echo` command.

ErrorMessage
string

Text of last error message. This property contains the last error message issued by MATLAB.

Root Properties

FixedWidthFontName
font name

Fixed-width font to use for axes, text, and uicontrols whose FontName is set to FixedWidth. MATLAB uses the font name specified for this property as the value for axes, text, and uicontrol FontName properties when their FontName property is set to FixedWidth. Specifying the font name with this property eliminates the need to hardcode font names in MATLAB applications and thereby enables these applications to run without modification in locales where non-ASCII character sets are required. In these cases, MATLAB attempts to set the value of FixedWidthFontName to the correct value for a given locale.

MATLAB application developers should not change this property, but should create axes, text, and uicontrols with FontName properties set to FixedWidth when they want to use a fixed-width font for these objects.

MATLAB end users can set this property if they do not want to use the preselected value. In locales where Latin-based characters are used, Courier is the default.

Format

short | {shortE} | long | longE | bank |
hex | + | rat

Output format mode. This property sets the format used to display numbers. See also the format command.

- short — Fixed-point format with 5 digits
- shortE — Floating-point format with 5 digits
- shortG — Fixed- or floating-point format displaying as many significant figures as possible with 5 digits
- long — Scaled fixed-point format with 15 digits
- longE — Floating-point format with 15 digits

- `longG` — Fixed- or floating-point format displaying as many significant figures as possible with 15 digits
- `bank` — Fixed-format of dollars and cents
- `hex` — Hexadecimal format
- `+` — Displays + and - symbols
- `rat` — Approximation by ratio of small integers

`FormatSpacing`

`compact` | `{loose}`

Output format spacing (see also `format` command).

- `compact` — Suppress extra line feeds for more compact display.
- `loose` — Display extra line feeds for a more readable display.

`HandleVisibility`

`{on}` | `callback` | `off`

This property is not useful on the root object.

`HitTest`

`{on}` | `off`

This property is not useful on the root object.

`Interruptible`

`{on}` | `off`

This property is not useful on the root object.

`Language`

`string`

System environment setting.

`MonitorPosition`

`[x y width height;x y width height]`

Root Properties

Width and height of primary and secondary monitors, in pixels.
This property contains the width and height of each monitor connected to your computer. The x and y values for the primary monitor are 0, 0 and the width and height of the monitor are specified in pixels.

The secondary monitor position is specified as

```
x = primary monitor width + 1  
y = primary monitor height + 1
```

Querying the value of the figure `MonitorPosition` on a multiheaded system returns the position for each monitor on a separate line.

```
v = get(0,'MonitorPosition')  
v =  
x y width height % Primary monitor  
x y width height % Secondary monitor
```

Note that MATLAB sets the value of the `ScreenSize` property to the combined size of the monitors.

Parent
handle

Handle of parent object. This property always contains the empty matrix, because the root object has no parent.

PointerLocation
[x,y]

Current location of pointer. A vector containing the x - and y -coordinates of the pointer position, measured from the lower left corner of the screen. You can move the pointer by changing the values of this property. The `Units` property determines the units of this measurement.

This property always contains the current pointer location, even if the pointer is not in a MATLAB window. A callback routine querying the `PointerLocation` can get a value different from the location of the pointer when the callback was triggered. This difference results from delays in callback execution caused by competition for system resources.

On Macintosh platforms, you cannot change the pointer location using the `set` command.

`PointerWindow`
handle (read only)

Handle of window containing the pointer. MATLAB sets this property to the handle of the figure window containing the pointer. If the pointer is not in a MATLAB window, the value of this property is 0. A callback routine querying the `PointerWindow` can get the wrong window handle if you move the pointer to another window before the callback executes. This error results from delays in callback execution caused by competition for system resources.

`RecursionLimit`
integer

Number of nested M-file calls. This property sets a limit to the number of nested calls to M-files MATLAB will make before stopping (or potentially running out of memory). By default the value is set to a large value. Setting this property to a smaller value (something like 150, for example) should prevent MATLAB from running out of memory and will instead cause MATLAB to issue an error when the limit is reached.

`ScreenDepth`
bits per pixel

Root Properties

Screen depth. The depth of the display bitmap (i.e., the number of bits per pixel). The maximum number of simultaneously displayed colors on the current graphics device is 2 raised to this power.

ScreenDepth supersedes the BlackAndWhite property. To override automatic hardware checking, set this property to 1. This value causes MATLAB to assume the display is monochrome. This is useful if MATLAB is running on color hardware but is being displayed on a monochrome terminal. Such a situation can cause MATLAB to determine erroneously that the display is color.

ScreenPixelsPerInch
Display resolution

DPI setting for your display. This property contains the setting of your display resolution specified in your system preferences.

ScreenSize
four-element rectangle vector (read only)

Screen size. A four-element vector,
[left,bottom,width,height]

that defines the display size. left and bottom are 0 for all Units except pixels, in which case left and bottom are 1. width and height are the screen dimensions in units specified by the Units property.

Determining Screen Size

Note that the screen size in absolute units (e.g., inches) is determined by dividing the number of pixels in width and height by the screen DPI (see the ScreenPixelPerInch property). This value is approximate and might not represent the actual size of the screen.

Note that the `ScreenSize` property is static. Its values are read only at MATLAB startup and not updated if system display settings change. Also, the values returned might not represent the usable screen size for application developers due to the presence of other GUIs, such as the Windows task bar.

`Selected`
on | off

This property has no effect on the root level.

`SelectionHighlight`
{on} | off

This property has no effect on the root level.

`ShowHiddenHandles`
on | {off}

Show or hide handles marked as hidden. When set to on, this property disables handle hiding and exposes all object handles regardless of the setting of an object's `HandleVisibility` property. When set to off, all objects so marked remain hidden within the graphics hierarchy.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. While it is not necessary to identify the root object with a tag (since its handle is always 0), you can use this property to store any string value that you can later retrieve using `set`.

`Type`
string (read only)

Class of graphics object. For the root object, `Type` is always 'root'.

Root Properties

UIContextMenu
handle

This property has no effect on the root level.

Units

{pixels} | normalized | inches | centimeters
| points | characters

Unit of measurement. This property specifies the units MATLAB uses to interpret size and location data. All units are measured from the lower left corner of the screen. Normalized units map the lower left corner of the screen to (0,0) and the upper right corner to (1.0,1.0). inches, centimeters, and points are absolute units (one point equals 1/72 of an inch). Characters are units defined by characters from the default system font; the width of one unit is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

This property affects the `PointerLocation` and `ScreenSize` properties. If you change the value of `Units`, it is good practice to return it to its default value after completing your operation, so as not to affect other functions that assume `Units` is set to the default value.

UserData
matrix

User-specified data. This property can be any data you want to associate with the root object. MATLAB does not use this property, but you can access it using the `set` and `get` functions.

Visible
{on} | off

Object visibility. This property has no effect on the root object.

Purpose	Polynomial roots
Syntax	<code>r = roots(c)</code>
Description	<p><code>r = roots(c)</code> returns a column vector whose elements are the roots of the polynomial <code>c</code>.</p> <p>Row vector <code>c</code> contains the coefficients of a polynomial, ordered in descending powers. If <code>c</code> has $n+1$ components, the polynomial it represents is $c_1s^n + \dots + c_n s + c_{n+1}$.</p>
Remarks	<p>Note the relationship of this function to <code>p = poly(r)</code>, which returns a row vector whose elements are the coefficients of the polynomial. For vectors, <code>roots</code> and <code>poly</code> are inverse functions of each other, up to ordering, scaling, and roundoff error.</p>
Examples	<p>The polynomial $s^3 - 6s^2 - 72s - 27$ is represented in MATLAB as</p> <pre>p = [1 -6 -72 -27]</pre> <p>The roots of this polynomial are returned in a column vector by</p> <pre>r = roots(p)</pre> <pre>r = 12.1229 -5.7345 -0.3884</pre>
Algorithm	<p>The algorithm simply involves computing the eigenvalues of the companion matrix:</p> <pre>A = diag(ones(n-1,1),-1); A(1,:) = -c(2:n+1)./c(1); eig(A)</pre>

roots

It is possible to prove that the results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix A , but this does not mean that they are the exact roots of a polynomial with coefficients within roundoff error of those in c .


See Also

fzero, poly, residue

Purpose

Angle histogram plot

**GUI Alternatives**

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
rose  
rose(theta)  
rose(theta,x)  
rose(theta,nbins)  
rose(axes_handle,...)  
h = rose(...)  
[tout,rout] = rose(...)
```

Description

`rose` creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range. Each group is shown as one bin.

`rose(theta)` plots an angle histogram showing the distribution of `theta` in 20 angle bins or less. The vector `theta`, expressed in radians, determines the angle of each bin from the origin. The length of each bin reflects the number of elements in `theta` that fall within a group, which ranges from 0 to the greatest number of elements deposited in any one bin.

`rose(theta,x)` uses the vector `x` to specify the number and the locations of bins. `length(x)` is the number of bins and the values of `x` specify the center angle of each bin. For example, if `x` is a five-element vector, `rose` distributes the elements of `theta` in five bins centered at the specified `x` values.

rose

`rose(theta,nbins)` plots `nbins` equally spaced bins in the range `[0, 2*pi]`. The default is 20.

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

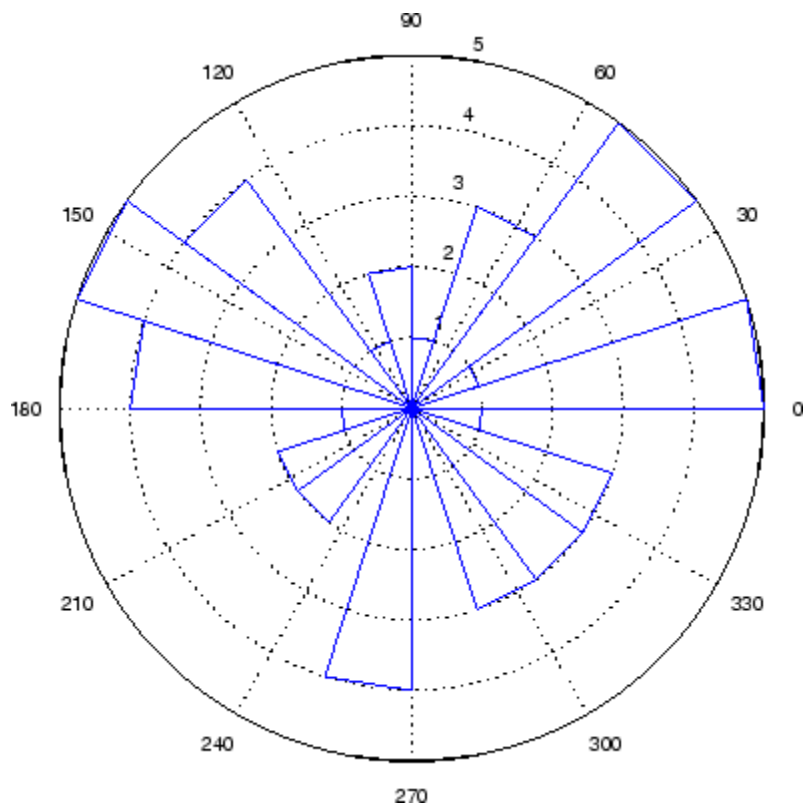
`h = rose(...)` returns the handles of the line objects used to create the graph.

`[tout,rout] = rose(...)` returns the vectors `tout` and `rout` so `polar(tout,rout)` generates the histogram for the data. This syntax does not generate a plot.

Example

Create a rose plot showing the distribution of 50 random numbers.

```
theta = 2*pi*rand(1,50);  
rose(theta)
```

**See Also**

compass, feather, hist, line, polar

“Histograms” on page 1-86 for related functions

Histograms in Polar Coordinates for another example

Purpose Classic symmetric eigenvalue test problem

Syntax A = rosser

Description A = rosser returns the Rosser matrix. This matrix was a challenge for many matrix eigenvalue algorithms. But LAPACK's DSYEV routine used in MATLAB has no trouble with it. The matrix is 8-by-8 with integer elements. It has:

- A double eigenvalue
- Three nearly equal eigenvalues
- Dominant eigenvalues of opposite sign
- A zero eigenvalue
- A small, nonzero eigenvalue

Examples

```
rosser
```

```
ans =
```

```
    611    196   -192    407     -8    -52   -49    29
    196    899    113   -192    -71   -43    -8   -44
   -192    113    899    196     61    49     8    52
    407   -192    196    611     8    44    59   -23
     -8    -71     61     8    411  -599   208   208
   -52   -43    49    44  -599   411   208   208
   -49    -8     8    59   208   208    99  -911
    29   -44    52   -23   208   208  -911    99
```

Purpose Rotate matrix 90 degrees

Syntax $B = \text{rot90}(A)$
 $B = \text{rot90}(A, k)$

Description $B = \text{rot90}(A)$ rotates matrix A counterclockwise by 90 degrees.
 $B = \text{rot90}(A, k)$ rotates matrix A counterclockwise by $k \cdot 90$ degrees, where k is an integer.

Examples The matrix

```
X =  
    1    2    3  
    4    5    6  
    7    8    9
```

rotated by 90 degrees is

```
Y = rot90(X)  
Y =  
    3    6    9  
    2    5    8  
    1    4    7
```

See Also `flipdim`, `fliplr`, `flipud`

rotate

Purpose Rotate object in specified direction

Syntax `rotate(h,direction,alpha)`
`rotate(...,origin)`

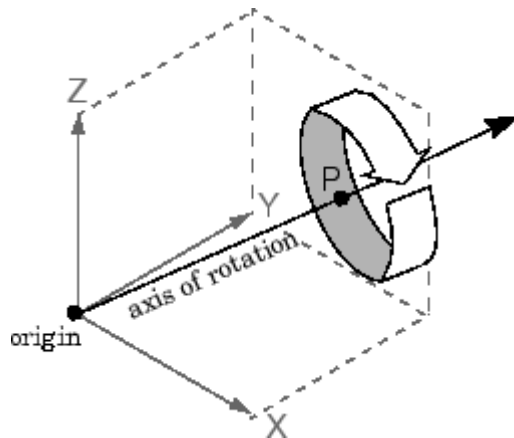
Description The rotate function rotates a graphics object in three-dimensional space, according to the right-hand rule.

`rotate(h,direction,alpha)` rotates the graphics object `h` by `alpha` degrees. `direction` is a two- or three-element vector that describes the axis of rotation in conjunction with the origin.

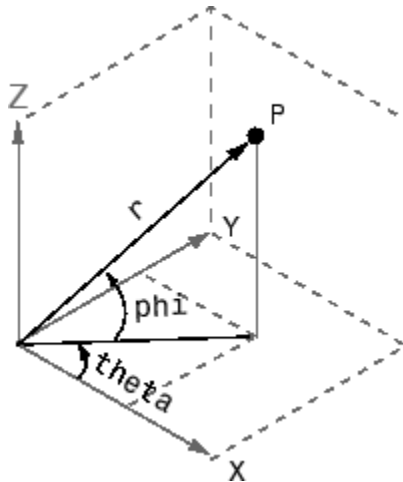
`rotate(...,origin)` specifies the origin of the axis of rotation as a three-element vector. The default origin is the center of the plot box.

Remarks The graphics object you want rotated must be a child of the same axes. The object's data is modified by the rotation transformation. This is in contrast to `view` and `rotate3d`, which only modify the viewpoint.

The axis of rotation is defined by an origin and a point P relative to the origin. P is expressed as the spherical coordinates `[theta phi]` or as Cartesian coordinates.



The two-element form for direction specifies the axis direction using the spherical coordinates $[\text{theta } \text{phi}]$. theta is the angle in the x - y plane counterclockwise from the positive x -axis. phi is the elevation of the direction vector from the x - y plane.



The three-element form for direction specifies the axis direction using Cartesian coordinates. The direction vector is the vector from the origin to (X,Y,Z) .

Examples

Rotate a graphics object 180° about the x -axis.

```
h = surf(peaks(20));
rotate(h,[1 0 0],180)
```

Rotate a surface graphics object 45° about its center in the z direction.

```
h = surf(peaks(20));
zdir = [0 0 1];
center = [10 10 0];
rotate(h,zdir,45,center)
```

rotate

Remarks

rotate changes the Xdata, Ydata, and Zdata properties of the appropriate graphics object.

See Also

rotate3d, sph2cart, view


The axes CameraPosition, CameraTarget, CameraUpVector, CameraViewAngle

“Object Manipulation” on page 1-97 for related functions

Purpose

Rotate 3-D view using mouse

GUI Alternatives

Use the Rotate3D tool  on the figure toolbar to enable and disable rotate3D mode on a plot, or select **Rotate 3D** from the figure's **Tools** menu. For details, see “Rotate 3D — Interactive Rotation of 3-D Views” in the MATLAB Graphics documentation.

Syntax

```
rotate3d
rotate3d
rotate3d
rotate3d(figure_handle,...)
rotate3d(axes_handle,...)
h = rotate3d(figure_handle)
```

Description

`rotate3d on` enables mouse-base rotation on all axes within the current figure.

`rotate3d off` disables interactive axes rotation in the current figure.

`rotate3d toggle` toggles interactive axes rotation in the current figure.

`rotate3d(figure_handle,...)` enables rotation within the specified figure instead of the current figure.

`rotate3d(axes_handle,...)` enables rotation only in the specified axes.

`h = rotate3d(figure_handle)` returns a `rotate3d mode object` for figure `figure_handle` for you to customize the mode's behavior.

Using Rotate Mode Objects

You access the following properties of pan mode objects via `get` and modify some of them using `set`:

`FigureHandle` <*handle*>

The associated figure handle. This read-only property cannot be set.

Enable 'on' | 'off'

Specifies whether this figure mode is currently enabled on the figure.

```
RotateMotion 'orbit'|'box'
```

Sets the method of rotation. 'orbit' rotates the entire axes; 'box' rotates a plot-box outline of the axes.

```
ButtonDownFilter <function_handle>
```

The application can inhibit the rotate operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

```
function [res] = myfunction(obj,event_obj)
% OBJ      handle to the object that has been clicked on.
% EVENT_OBJ handle to event object (empty in this release).
% RES      a logical flag to determine whether the rotate
           operation should take place or the
           'ButtonDownFcn' property of the object should
           take precedence.
```

```
ActionPreCallback <function_handle>
```

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

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

The event object has the following read-only property:

Axes The handle of the axes that is being rotated.

```
ActionPostCallback <function_handle>
```

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

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

```
flags = isAllowAxesRotate(h,axes)
```

Calling the function `isAllowAxesRotate` on the `rotate3d` object, `h`, with a vector of axes handles, `axes`, as input will return a logical array of the same dimension as the axes handle vector which indicate whether a rotate operation is permitted on the axes objects.

```
setAllowAxesRotate(h,axes,flag)
```

Calling the function `setAllowAxesRotate` on the `rotate3d` object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, will either allow or disallow a rotate operation on the axes objects.

Examples

Example 1

Simple 3-D rotation

```
surf(peaks);
rotate3d on
% rotate the plot using the mouse pointer.
```

Example 2

Rotate the plot using the Plot Box rotate style:

```
surf(peaks);
h = rotate3d;
set(h,'RotateMotion','box','Enable','on');
% Rotate the plot.
```

Example 3

Create two axes as subplots and then prevent one from rotating:

```
ax1 = subplot(1,2,1);
surf(peaks);
h = rotate3d;
ax2 = subplot(1,2,2);
surf(membrane);
setAllowAxesRotate(h,ax2,false);
% rotate the plots.
```

Example 4

Create a `ButtonDown` callback for rotate mode objects to trigger. Copy the following code to a new M-file, execute it, and observe rotation behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
h = rotate3d;
set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse-click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
    flag = true;
else
    flag = false;
end
```

Example 5

Create callbacks for pre- and post-buttonDown events for rotate3D mode objects to trigger. Copy the following code to a new M-file, execute it, and observe rotation behavior:

```
function demo
% Listen to rotate events
surf(peaks);
h = rotate3d;
set(h,'ActionPreCallback',@myprecallback);
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');
%
function myprecallback(obj,evd)
disp('A rotation is about to occur.');
```

```
%
function mypostcallback(obj,evd)
newView = round(get(evd.Axes,'View'));
msgbox(sprintf('The new view is [%d %d].',newView));
```

Remarks

When enabled, rotate3d provides continuous rotation of axes and the objects it contains through mouse movement. A numeric readout appears in the lower left corner of the figure during rotation, showing the current azimuth and elevation of the axes. Releasing the mouse button removes the animated box and the readout.

You can also enable 3-D rotation from the figure **Tools** menu or the figure toolbar.

You can create a rotate3D mode object once and use it to customize the behavior of different axes, as example 3 illustrates. You can also change its callback functions on the fly.

When you assign different 3-D rotation behaviors to different subplot axes via a mode object and then link them using the linkaxes function, the behavior of the axes you manipulate with the mouse will carry over to the linked axes, regardless of the behavior you previously set for the other axes.

rotate3d

See Also

camorbit, pan, rotate, view, zoom

Object Manipulation for related functions

Purpose Round to nearest integer

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

Description $Y = \text{round}(X)$ rounds the elements of X to the nearest integers. For complex X , the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]
```

```
a =  
Columns 1 through 4  
-1.9000    -0.2000    3.4000    5.6000  
Columns 5 through 6  
7.0000    2.4000 + 3.6000i
```

```
round(a)
```

```
ans =  
Columns 1 through 4  
-2.0000    0    3.0000    6.0000  
Columns 5 through 6  
7.0000    2.0000 + 4.0000i
```

See Also `ceil`, `fix`, `floor`

rref

Purpose Reduced row echelon form

Syntax
`R = rref(A)`
`[R, jb] = rref(A)`
`[R, jb] = rref(A, tol)`

Description `R = rref(A)` produces the reduced row echelon form of `A` using Gauss Jordan elimination with partial pivoting. A default tolerance of `(max(size(A))*eps *norm(A, inf))` tests for negligible column elements.

`[R, jb] = rref(A)` also returns a vector `jb` such that:

- `r = length(jb)` is this algorithm's idea of the rank of `A`.
- `x(jb)` are the pivot variables in a linear system `Ax = b`.
- `A(:, jb)` is a basis for the range of `A`.
- `R(1:r, jb)` is the `r`-by-`r` identity matrix.

`[R, jb] = rref(A, tol)` uses the given tolerance in the rank tests.

Roundoff errors may cause this algorithm to compute a different value for the rank than `rank`, `orth` and `null`.

Examples Use `rref` on a rank-deficient magic square:

```
A = magic(4), R = rref(A)
```

```
A =  
 16   2   3  13  
  5  11  10   8  
  9   7   6  12  
  4  14  15   1
```

```
R =  
  1   0   0   1  
  0   1   0   3
```

$$\begin{array}{cccc} 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 0 \end{array}$$
See Also

inv, lu, rank

Purpose Convert real Schur form to complex Schur form

Syntax `[U,T] = rsf2csf(U,T)`

Description The *complex Schur form* of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The *real Schur form* has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

`[U,T] = rsf2csf(U,T)` converts the real Schur form to the complex form.

Arguments `U` and `T` represent the unitary and Schur forms of a matrix `A`, respectively, that satisfy the relationships: $A = U * T * U'$ and $U' * U = \text{eye}(\text{size}(A))$. See `schur` for details.

Examples

Given matrix `A`,

```
1    1    1    3
1    2    1    1
1    1    3    1
-2   1    1    4
```

with the eigenvalues

```
4.8121    1.9202 + 1.4742i    1.9202 + 1.4742i    1.3474
```

Generating the Schur form of `A` and converting to the complex Schur form

```
[u,t] = schur(A);
[U,T] = rsf2csf(u,t)
```

yields a triangular matrix `T` whose diagonal (underlined here for readability) consists of the eigenvalues of `A`.

```
U =
```

-0.4916	-0.2756 - 0.4411i	0.2133 + 0.5699i	-0.3428
-0.4980	-0.1012 + 0.2163i	-0.1046 + 0.2093i	0.8001
-0.6751	0.1842 + 0.3860i	-0.1867 - 0.3808i	-0.4260
-0.2337	0.2635 - 0.6481i	0.3134 - 0.5448i	0.2466

T =

4.8121	-0.9697 + 1.0778i	-0.5212 + 2.0051i	-1.0067
0	1.9202 + 1.4742i	2.3355	0.1117 + 1.6547i
0	0	1.9202 - 1.4742i	0.8002 + 0.2310i
0	0	0	1.3474

See Also

schur

run

Purpose Run script that is not on current path

Syntax `run scriptname`

Description `run scriptname` runs the MATLAB script specified by `scriptname`. If `scriptname` contains the full pathname to the script file, then `run` changes the current directory to be the one in which the script file resides, executes the script, and sets the current directory back to what it was. The script is run within the caller's workspace.

`run` is a convenience function that runs scripts that are not currently on the path. Typically, you just type the name of a script at the MATLAB prompt to execute it. This works when the script is on your path. Use the `cd` or `addpath` function to make a script executable by entering the script name alone.

See Also `cd`, `addpath`

Purpose

Save workspace variables to disk

Graphical Interface

As an alternative to the save function, select **Save Workspace As** from the **File** menu in the MATLAB desktop, or use the Workspace browser.

Syntax

```
save
save filename
save filename var1 var2 ...
save -struct s
save filename -struct, s
save -struct s f1 f2 ...
save filename -struct s f1 f2 ...
save -regexp expr1 expr2 ...
save filename -regexp expr1 expr2 ...
save ... format
save ('filename', 'var1', 'var2', ...)
```

Description

save by itself stores all workspace variables in a binary format in the current directory in a file named matlab.mat. Retrieve the data with load. MAT-files are double-precision, binary, MATLAB format files. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the different formats allow. They can also be manipulated by other programs external to MATLAB.

save filename stores all workspace variables in the current directory to file filename. If you do not specify an extension to the filename, MATLAB uses .mat. The file must be writable. To save to another directory, use the full pathname for the filename. If filename is the special string `stdio`, the save command sends the data as standard output.

save filename var1 var2 ... saves only the specified workspace variables in filename.mat. Use the * wildcard to save only those variables that match the specified pattern. For example, save('A*') saves all variables that start with A.

save

`save -struct s` saves all fields of the scalar structure `s` as individual variables within the default save file `matlab.mat`.

`save filename -struct, s` saves these variables in the file specified by `filename`.

`save -struct s f1 f2 ...` saves as individual variables only those structure fields specified (`s.f1`, `s.f2`, ...). The variables are saved in file `matlab.mat`.

`save filename -struct s f1 f2 ...` saves these variables in the file specified by `filename`.

`save -regexp expr1 expr2 ...` saves those variables that match any of the “Regular Expressions” `expr1`, `expr2`, etc. The variables are saved in file `matlab.mat`.

`save filename -regexp expr1 expr2 ...` saves these variables in the file specified by `filename`.

`save ... format` enables you to make use of other data formats available with the `save` function. Enter each format specifier separately. For example, `save('myfile', '-ascii', '-double', '-tabs')`. The following table lists the valid format specifiers.

Format	How Data Is Stored
<code>-append</code>	The specified existing MAT-file, appended to the end. See Remarks, below.
<code>-ascii</code>	8-digit ASCII format
<code>-ascii -double</code>	16-digit ASCII format
<code>-ascii -tabs</code>	Delimits with tabs
<code>-ascii -double -tabs</code>	16-digit ASCII format, tab delimited
<code>-mat</code>	Binary MAT-file form (default)
<code>-v4</code>	A format that MATLAB Version 4 can open

Format	How Data Is Stored
-v6	A format that MATLAB Version 6 and earlier can open
-v7	A format that MATLAB Version 7 can open
-v7.3	An HDF5-based format that supports data items \geq 2 GB

`save ('filename', 'var1', 'var2', ...)` is the function form of the syntax.

Remarks

By default, MATLAB compresses the data it saves to MAT-files. MATLAB also uses Unicode character encoding when saving character data.

MAT-files saved with compression and Unicode encoding cannot be loaded into versions of MATLAB prior to MATLAB 7.0. If you save data to a MAT-file that you intend to load using MATLAB Version 6 or earlier, then you must specify the `-v6` option when saving. This disables compression and Unicode encoding for that particular save operation.

To override the compression and Unicode setting for all of your MATLAB sessions, use the **Preferences** dialog box. Open the **Preferences** dialog and select **General** and then **MAT-Files**. To disable data compression and Unicode encoding, click **Ensure backward compatibility (-v6)**. To turn these features back on, click **Use default features (Unicode and compression)**. See “General Preferences for MATLAB” in the Desktop Tools and Development Environment documentation for more information.

If you want to save to a file that you can then load into a Version 4 MATLAB session, you must use the `-V4` option when saving. When you use this option, variables that are incompatible with MATLAB 4 are not saved to the MAT-file. For example, ND arrays, structs, cells, etc. cannot be saved to a MATLAB 4 MAT-file. Also, variables with names that are longer than 19 characters cannot be saved to a MATLAB 4 MAT-file.

When working on 64-bit platforms, you can have data items in your workspace that occupy more than 2 GB. To save data of this size, you must use the HDF5-based version of the MATLAB MAT-file. Use the `-v7.3` switch to do this:

```
save -v7.3 myfile v1 v2
```

For information on any of the following topics related to saving to MAT-files, see “Exporting Data to MAT-Files” in the MATLAB Programming documentation:

- Appending variables to an existing MAT-file
- Compressing data in the MAT-file
- Saving in ASCII format
- Saving in MATLAB Version 4 format
- Saving with Unicode character encoding
- Data storage requirements
- Saving from external programs

For information on saving figures, see the documentation for `hgsave` and `saveas`. For information on exporting figures to other graphics formats, see the documentation for `print`.

Examples

Example 1

Save all variables from the workspace in binary MAT-file `test.mat`:

```
save test.mat
```

Example 2

Save variables `p` and `q` in binary MAT-file `test.mat`.

In this example, the file name is stored in a variable, `savefile`. You must call `save` using the function syntax of the command if you intend to reference the file name through a variable.

```

savefile = 'test.mat';
p = rand(1, 10);
q = ones(10);
save(savefile, 'p', 'q')

```

Example 3

Save the variables `vol` and `temp` in ASCII format to a file named `june10`:

```

save('d:\myfiles\june10', 'vol', 'temp', '-ASCII')

```

Example 4

Save the fields of structure `s1` as individual variables rather than as an entire structure.

```

s1.a = 12.7; s1.b = {'abc', [4 5; 6 7]}; s1.c = 'Hello!';
save newstruct.mat -struct s1;
clear

```

Check what was saved to `newstruct.mat`:

```

whos -file newstruct.mat

```

Name	Size	Bytes	Class
a	1x1	8	double array
b	1x2	158	cell array
c	1x6	12	char array

Grand total is 16 elements using 178 bytes

Read only the `b` field into the MATLAB workspace.

```

str = load('newstruct.mat', 'b')
str =
    b: {'abc' [2x2 double]}

```

Example 5

Using regular expressions, save in MAT-file `mydata.mat` those variables with names that begin with Mon, Tue, or Wed:

```
save('mydata', '-regexp', '^Mon|^Tue|^Wed');
```

Here is another way of doing the same thing. In this case, there are three separate expression arguments:

```
save('mydata', '-regexp', '^Mon', '^Tue', '^Wed');
```

Example 6

Save a 3000-by-3000 matrix uncompressed to file `c1.mat`, and compressed to file `c2.mat`. The compressed file uses about one quarter the disk space required to store the uncompressed data:

```
x = ones(3000);
y = uint32(rand(3000) * 100);

save c1 x y
save c2 x y -compress

d1 = dir('c1.mat');
d2 = dir('c2.mat');

d1.bytes
ans =
    45000240           % Size of the uncompressed data
d2.bytes
ans =
    11985634           % Size of the compressed data

d2.bytes/d1.bytes
ans =
    0.2663             % Ratio of compressed to uncompressed
```

See Also

load, clear, diary, fprintf, fwrite, genvarname, who, whos, workspace, regexp

save (COM)

Purpose Serialize control object to file

Syntax `h.save('filename')`
`save(h, 'filename')`

Description `h.save('filename')` saves the COM control object, `h`, to the file specified in the string, `filename`.
`save(h, 'filename')` is an alternate syntax for the same operation.

Note The COM save function is only supported for controls at this time.

Examples Create an `mwsamp` control and save its original state to the file `mwsample`:

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

Now, alter the figure by changing its label and the radius of the circle:

```
h.Label = 'Circle';  
h.Radius = 50;  
h.Redraw;
```

Using the load function, you can restore the control to its original state:

```
h.load('mwsample');  
h.get  
ans =  
    Label: 'Label'  
    Radius: 20
```

See Also `load`, `actxcontrol`, `actxserver`, `release`, `delete`

Purpose Save serial port objects and variables to MAT-file

Syntax

```
save filename  
save filename obj1 obj2...
```

Arguments

filename	The MAT-file name.
obj1	Serial port objects or arrays of serial port objects.
obj2...	

Description `save filename` saves all MATLAB variables to the MAT-file `filename`. If an extension is not specified for `filename`, then the `.mat` extension is used.

`save filename obj1 obj2...` saves the serial port objects `obj1 obj2...` to the MAT-file `filename`.

Remarks You can use `save` in the functional form as well as the command form shown above. When using the functional form, you must specify the filename and serial port objects as strings. For example, to save the serial port object `s` to the file `MySerial.mat`

```
s = serial('COM1');  
save('MySerial','s')
```

Any data that is associated with the serial port object is not automatically stored in the MAT-file. For example, suppose there is data in the input buffer for `obj`. To save that data to a MAT-file, you must bring it into the MATLAB workspace using one of the synchronous read functions, and then save to the MAT-file using a separate variable name. You can also save data to a text file with the `record` function.

You return objects and variables to the MATLAB workspace with the `load` command. Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

save (serial)

Example

This example illustrates how to use the command and functional form of save.

```
s = serial('COM1');
set(s, 'BaudRate', 2400, 'StopBits', 1)
save MySerial1 s
set(s, 'BytesAvailableFcn', @mycallback)
save('MySerial2', 's')
```

See Also

Functions

load, record

Properties

Status

Purpose

Save figure or Simulink block diagram using specified format

GUI Alternative

Use **File** → **Save As** on the figure window menu to access the Save As dialog, in which you can select a graphics format. For details, see “Exporting in a Specific Graphics Format” in the MATLAB Graphics documentation. Note that sizes of files written to image formats by this GUI and by saveas can differ, due to disparate resolution settings.

Syntax

```
saveas(h, 'filename.ext')
saveas(h, 'filename', 'format')
```

Description

saveas(h, 'filename.ext') saves the figure or Simulink block diagram with the handle h to the file filename.ext. The format of the file is determined by the extension, ext. Allowable values for ext are listed in this table.

You can pass the handle of any Handle Graphics object to saveas, which then saves the parent figure to the object you specified should h not be a figure handle. This means that saveas cannot save a subplot without also saving all subplots in its parent figure.

ext Value	Format
ai	Adobe Illustrator '88
bmp	Windows bitmap
emf	Enhanced metafile
eps	EPS Level 1
fig	MATLAB figure (invalid for Simulink block diagrams)
jpg	JPEG image (invalid for Simulink block diagrams)
m	MATLAB M-file (invalid for Simulink block diagrams)
pbm	Portable bitmap

ext Value	Format
pcx	Paintbrush 24-bit
pgm	Portable Graymap
png	Portable Network Graphics
ppm	Portable Pixmap
tif	TIFF image, compressed

`saveas(h, 'filename', 'format')` saves the figure or Simulink block diagram with the handle `h` to the file called `filename` using the specified format. The filename can have an extension, but the extension is not used to define the file format. If no extension is specified, the standard extension corresponding to the specified format is automatically appended to the filename.

Allowable values for `format` are the extensions in the table above and the device drivers and graphic formats supported by `print`. The drivers and graphic formats supported by `print` include additional file formats not listed in the table above. When using a `print` device type to specify format for `saveas`, do not prefix it with `-d`.

Remarks

You can use `open` to open files saved using `saveas` with an `m` or `fig` extension. Other `saveas` and `print` formats are not supported by `open`. Both the **Save As** and **Export** dialog boxes that you access from a figure's **File** menu use `saveas` with the `format` argument, and support all device and file types listed above.

If you want to control the size or resolution of figures saved in image (bitmapped) formats (such as BMP or JPG), use the `print` command and specify dots-per-inch resolution with the `r` switch.

Examples

Example 1: Specify File Extension

Save the current figure that you annotated using the Plot Editor to a file named `pred_pre.y` using the MATLAB `fig` format. This allows you

to open the file `pred_prej.fig` at a later time and continue editing it with the Plot Editor.

```
saveas(gcf, 'pred_prej.fig')
```

Example 2: Specify File Format but No Extension

Save the current figure, using Adobe Illustrator format, to the file `logo`. Use the `ai` extension from the above table to specify the format. The file created is `logo.ai`.

```
saveas(gcf, 'logo', 'ai')
```

This is the same as using the Adobe Illustrator format from the print devices table, which is `-dill`; use `doc print` or `help print` to see the table for print device types. The file created is `logo.ai`. MATLAB automatically appends the `ai` extension for an Illustrator format file because no extension was specified.

```
saveas(gcf, 'logo', 'ill')
```

Example 3: Specify File Format and Extension

Save the current figure to the file `star.eps` using the Level 2 Color PostScript format. If you use `doc print` or `help print`, you can see from the table for print device types that the device type for this format is `-dpsc2`. The file created is `star.eps`.

```
saveas(gcf, 'star.eps', 'psc2')
```

In another example, save the current Simulink block diagram to the file `trans.tiff` using the TIFF format with no compression. From the table for print device types, you can see that the device type for this format is `-dtiffn`. The file created is `trans.tiff`.

```
saveas(gcf, 'trans.tiff', 'tiffn')
```

See Also

`hgsave`, `open`, `print`

saveas

“Printing” on page 1-88 for related functions

Simulink users, see also `save_system`

Purpose	User-defined extension of save function for user objects
Syntax	<code>B = saveobj(A)</code>
Description	<p><code>B = saveobj(A)</code> is called by the MATLAB save function when object A is saved to a MAT-file. This call executes the saveobj method for the object's class, if such a method exists. The return value B is subsequently used by save to populate the MAT-file.</p> <p>When you issue a save command on an object, MATLAB looks for a method called saveobj in the class directory. You can overload this method to modify the object before the save operation. For example, you could define a saveobj method that saves related data along with the object.</p>
Remarks	<p>saveobj can be overloaded only for user objects. save will not call saveobj for a built-in datatype, such as double, even if @double/saveobj exists.</p> <p>saveobj will be separately invoked for each object to be saved.</p> <p>A child object does not inherit the saveobj method of its parent class. To implement saveobj for any class, including a class that inherits from a parent, you must define a saveobj method within that class directory.</p>
Examples	<p>The following example shows a saveobj method written for the portfolio class. The method determines if a portfolio object has already been assigned an account number from a previous save operation. If not, saveobj calls getAccountNumber to obtain the number and assigns it to the account_number field. The contents of b is saved to the MAT-file.</p> <pre>function b = saveobj(a) if isempty(a.account_number) a.account_number = getAccountNumber(a); end b = a;</pre>

saveobj

See Also

save, load, loadobj

Purpose

Save current MATLAB search path to pathdef.m file

GUI Alternatives

As an alternative to the savepath function, use the **Set Path** dialog box. To open it, select **File > Set Path** in the MATLAB desktop.

Syntax

```
savepath  
savepath newfile
```

Description

savepath saves the current MATLAB search path to pathdef.m. It returns

0	If the file was saved successfully
1	If the save failed

savepath newfile saves the current MATLAB search path to newfile, where newfile is in the current directory or is a relative or absolute path.

Examples

The statement

```
savepath myfiles/pathdef.m
```

saves the current search path to the file pathdef.m, which is located in the myfiles directory in the MATLAB current directory.

Consider using savepath in your MATLAB finish.m file to save the path when you exit MATLAB.

See Also

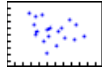
addpath, cd, dir, finish, genpath, matlabroot, partialpath, pathdef, pathsep, pathtool, rehash, restoredefaultpath, rmpath, savepath, startup, what

Search Path in the MATLAB Desktop Tools and Development Environment documentation


scatter

Purpose

Scatter plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
scatter(X,Y,S,C)
scatter(X,Y)
scatter(X,Y,S)
scatter(...,markertype)
scatter(...,'filled')
scatter(...,'PropertyName',propertyvalue)
scatter(axes_handles,...)
h = scatter(...)
hpatch = scatter('v6',...)
```

Description

`scatter(X,Y,S,C)` displays colored circles at the locations specified by the vectors `X` and `Y` (which must be the same size).

`S` determines the area of each marker (specified in points^2). `S` can be a vector the same length as `X` and `Y` or a scalar. If `S` is a scalar, MATLAB draws all the markers the same size. If `S` is empty, the default size is used.

`C` determines the color of each marker. When `C` is a vector the same length as `X` and `Y`, the values in `C` are linearly mapped to the colors in the current colormap. When `C` is a $\text{length}(X)$ -by-3 matrix, it specifies the colors of the markers as RGB values. `C` can also be a color string (see `ColorSpec` for a list of color string specifiers).

`scatter(X,Y)` draws the markers in the default size and color.

`scatter(X,Y,S)` draws the markers at the specified sizes (S) with a single color. This type of graph is also known as a bubble plot.

`scatter(...,markertype)` uses the marker type specified instead of 'o' (see `LineStyle` for a list of marker specifiers).

`scatter(...,'filled')` fills the markers.

`scatter(...,'PropertyName',propertyvalue)` creates the scatter graph, applying the specified property settings. See `scattergroup` properties for a description of properties.

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

`h = scatter(...)` returns the handle of the scattergroup object created.

Backward-Compatible Version

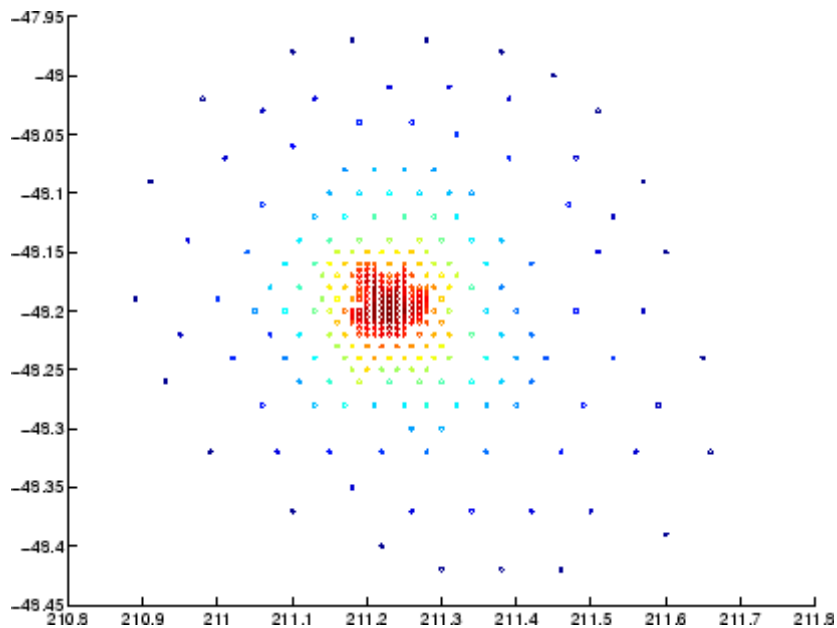
`hpatch = scatter('v6',...)` returns the handles to the patch objects created by `scatter` (see `Patch Properties` for a list of properties you can specify using the object handles and `set`).

See `Plot Objects and Backward Compatibility` for more information.

Example

```
load seamount
scatter(x,y,5,z)
```

scatter



See Also

`scatter3`, `plot3`

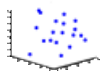
“Scatter/Bubble Plots” on page 1-87 for related functions


See [Triangulation and Interpolation of Scatter Data](#) for related information.

See [Scattergroup Properties](#) for property descriptions.

Purpose

3-D scatter plot

**GUI Alternatives**

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
scatter3(X,Y,Z,S,C)
scatter3(X,Y,Z)
scatter3(X,Y,Z,S)
scatter3(...,markertype)
scatter3(...,'filled')
h = scatter3(...)
hpatch = scatter3('v6',...)
```

Description

`scatter3(X,Y,Z,S,C)` displays colored circles at the locations specified by the vectors X , Y , and Z (which must all be the same size).

S determines the size of each marker (specified in points). S can be a vector the same length as X , Y , and Z or a scalar. If S is a scalar, MATLAB draws all the markers the same size.

C determines the colors of each marker. When C is a vector the same length as X , Y , and Z , the values in C are linearly mapped to the colors in the current colormap. When C is a $\text{length}(X)$ -by-3 matrix, it specifies the colors of the markers as RGB values. C can also be a color string (see `ColorSpec` for a list of color string specifiers).

`scatter3(X,Y,Z)` draws the markers in the default size and color.

`scatter3(X,Y,Z,S)` draws markers at the specified sizes (S) in a single color.

scatter3

`scatter3(...,markertype)` uses the marker type specified instead of 'o' (see `LineStyleSpec` for a list of marker specifiers).

`scatter3(..., 'filled')` fills the markers.

`h = scatter3(...)` returns handles to the `scattergroup` objects created by `scatter3`. See `Scattergroup Properties` for property descriptions.

Backward-Compatible Version

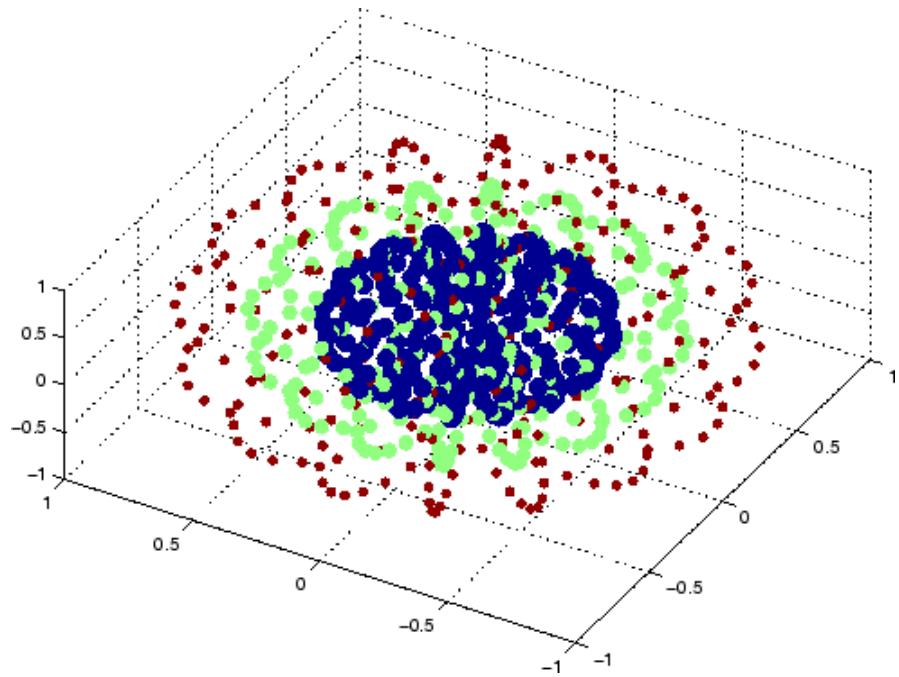
`hpatch = scatter3('v6',...)` returns the handles to the patch objects created by `scatter3` (see `Patch` for a list of properties you can specify using the object handles and `set`).

Remarks

Use `plot3` for single color, single marker size 3-D scatter plots.

Examples

```
[x,y,z] = sphere(16);
X = [x(:)*.5 x(:)*.75 x(:)];
Y = [y(:)*.5 y(:)*.75 y(:)];
Z = [z(:)*.5 z(:)*.75 z(:)];
S = repmat([1 .75 .5]*10,prod(size(x)),1);
C = repmat([1 2 3],prod(size(x)),1);
scatter3(X(:),Y(:),Z(:),S(:),C(:),'filled'), view(-60,60)
```

**See Also**

scatter, plot3

See Scattergroup Properties for property descriptions

“Scatter/Bubble Plots” on page 1-87 for related functions

Scattergroup Properties

Purpose

Defines the scattergroup properties

Modifying Properties

You can set and query graphics object properties using the set and get commands or the Property Editor (propertyeditor).

Note that you cannot define default property values for scattergroup objects.

See Plot Objects for information on scattergroup objects.

Scattergroup Property Descriptions

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

BeingDeleted
on | {off} Read Only

This object is being deleted. The BeingDeleted property provides a mechanism that you can use to determine whether objects are in the process of being deleted. MATLAB sets the BeingDeleted property to on when the object's delete function callback is called (see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

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

BusyAction
cancel | {queue}

Callback routine interruption. The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over the scattergroup object.

This property can be

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

The expression executes in the MATLAB workspace.

See [Function Handle Callbacks](#) for information on how to use function handles to define the callbacks.

`CData`

vector, m-by-3 matrix, `ColorSpec`

Color of markers. When `CData` is a vector the same length as `XData` and `YData`, the values in `CData` are linearly mapped to the colors in the current colormap. When `CData` is a `length(XData)-by-3` matrix, it specifies the colors of the markers as RGB values.

Scattergroup Properties

CData can also be a color string (see ColorSpec for a list of color string specifiers).

CDataSource
string (MATLAB variable)

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

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

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

See the refreshdata reference page for more information.

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

Children
array of graphics object handles

Children of the scattergroup object. An array containing the handle of a patch object parented to the scattergroup object (whether visible or not).

Note that if a child object's HandleVisibility property is set to callback or off, its handle does not show up in the stem

Children property unless you set the root ShowHiddenHandles property to on:

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

Clipping

{on} | off

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

CreateFcn

string or function handle

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

```
scatter(x,y, 'CreateFcn', @CallbackFcn)
```

where @CallbackFcn is a function handle that references the callback function.

MATLAB executes this routine after setting all other scattergroup properties. Setting this property on an existing scattergroup object has no effect.

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

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

DeleteFcn

string or function handle

Scattergroup Properties

Callback executed during object deletion. A callback that executes when the scattergroup object is deleted (e.g., this might happen when you issue a `delete` command on the scattergroup object, its parent axes object, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

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

See [Function Handle Callbacks](#) for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string

Label used by plot legends. The legend and the plot browser use this text for labels for any scattergroup objects appearing in these legends.

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

Erase mode. This property controls the technique MATLAB uses to draw and erase scatter child objects (the patch used to construct the scatter graph). Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster,

but do not perform a complete redraw and are therefore less accurate.

- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

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

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

Scattergroup Properties

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

`HandleVisibility`

{on} | callback | off

Control access to object's handle by command-line users and GUIs.

This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the scattergroup object.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property,

figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

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

Handle Validity

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

`HitTest`
`{on} | off`

Selectable by mouse click. `HitTest` determines whether the scattergroup object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the line objects that compose the stem plot. If `HitTest` is `off`, clicking the `stemseries` object selects the object below it (which is usually the axes object containing it).

`HitTestArea`
`on | {off}`

Select scattergroup object on markers or area of scatter graph. This property enables you to select scattergroup objects in two ways:

- Select by clicking on scatter markers (default).
- Select by clicking anywhere in the extent of the scatter graph.

Scattergroup Properties

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

Interruptible
{on} | off

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

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to on allows any graphics object's callback to interrupt callback routines originating from a stem property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

LineWidth
width in points (default: 0.5 points)

Width of line that draws the edge of markers. You can set the thickness of the lines used to draw the markers to a value in points.

Marker
character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed on the scatter graph. The following table shows supported markers.

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 uses the CData property to determine the MarkerEdgeColor.

MarkerFaceColor

ColorSpec | {none} | auto

Scattergroup Properties

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the axes color, or the figure color if the axes Color property is set to none (which is the factory default for axes objects).

Parent

handle of axes, hggroup, or hgtransform object

Parent of scattergroup object. This property contains the handle of the scattergroup object's parent. The parent of a scattergroup object is the axes object that contains it. You can reparent scattergroup objects to other axes object, hggroup, or hgtransform objects.

See [Objects That Can Contain Other Objects](#) for more information on parenting graphics objects.

Selected

on | {off}

Is object selected. When you set this property to on, MATLAB displays selection handles at the corners and midpoints if the SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this property to on, thereby indicating that the scattergroup object is selected.

SelectionHighlight

{on} | off

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

SizeData

square points

Size of markers in square points. This property specifies the area of the marker in the scatter graph in units of points. Since there are 72 points to one inch, to specify a marker that has an area of one square inch you would use a value of 72^2 .

Tag

string

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

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

```
t = scatter(x,y,'Tag','scatter1')
```

When you want to access the scattergroup object, you can use `findobj` to find the scattergroup object's handle. The following statement changes the `MarkerFaceColor` property of the object whose Tag is `scatter1`.

```
set(findobj('Tag','scatter1'),'MarkerFaceColor','red')
```

Type

string (read only)

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

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

UIContextMenu

handle of a `uicontextmenu` object

Scattergroup Properties

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

`UserData`
array

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

`Visible`
{on} | off

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

`XData`
array

X-coordinates of scatter markers. The scatter function draws individual markers at each *x*-axis location in the `XData` array. The input argument `x` in the scatter function calling syntax assigns values to `XData`.

`XDataSource`
string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

YData

scalar, vector, or matrix

Y-coordinates of scatter markers. The scatter function draws individual markers at each *y*-axis location in the YData array.

The input argument *y* in the scatter function calling syntax assigns values to YData.

YDataSource

string (MATLAB variable)

Links YData to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the YData.

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

Scattergroup Properties

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

See the `refreshdata` reference page for more information.

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

ZData

vector of coordinates

Z-coordinates. A vector of *z*-coordinates defining the scattergroup object. `XData` and `YData` must be the same length.

ZDataSource

string (MATLAB variable)

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

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

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

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.

schur

Purpose Schur decomposition

Syntax
`T = schur(A)`
`T = schur(A, flag)`
`[U, T] = schur(A, ...)`

Description The `schur` command computes the Schur form of a matrix.
`T = schur(A)` returns the Schur matrix `T`.
`T = schur(A, flag)` for real matrix `A`, returns a Schur matrix `T` in one of two forms depending on the value of `flag`:

'complex'	<code>T</code> is triangular and is complex if <code>A</code> has complex eigenvalues.
'real'	<code>T</code> has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal. 'real' is the default.

If `A` is complex, `schur` returns the complex Schur form in matrix `T`. The complex Schur form is upper triangular with the eigenvalues of `A` on the diagonal.

The function `rsf2csf` converts the real Schur form to the complex Schur form.

`[U, T] = schur(A, ...)` also returns a unitary matrix `U` so that `A = U*T*U'` and `U'*U = eye(size(A))`.

Examples `H` is a 3-by-3 eigenvalue test matrix:

```
H = [ -149   -50  -154
       537   180   546
       -27    -9   -25 ]
```

Its Schur form is

```
schur(H)
```

```

ans =
    1.0000   -7.1119  -815.8706
           0    2.0000  -55.0236
           0         0    3.0000

```

The eigenvalues, which in this case are 1, 2, and 3, are on the diagonal. The fact that the off-diagonal elements are so large indicates that this matrix has poorly conditioned eigenvalues; small changes in the matrix elements produce relatively large changes in its eigenvalues.

Algorithm

Input of Type Double

If A has type `double`, `schur` uses the LAPACK routines listed in the following table to compute the Schur form of a matrix:

Matrix A	Routine
Real symmetric	DSYTRD, DSTEQR DSYTRD, DORGTR, DSTEQR (with output U)
Real nonsymmetric	DGEHRD, DHSEQR DGEHRD, DORGHR, DHSEQR (with output U)
Complex Hermitian	ZHETRD, ZSTEQR ZHETRD, ZUNGTR, ZSTEQR (with output U)
Non-Hermitian	ZGEHRD, ZHSEQR ZGEHRD, ZUNGHR, ZHSEQR (with output U)

Input of Type Single

If A has type `single`, `schur` uses the LAPACK routines listed in the following table to compute the Schur form of a matrix:

Matrix A	Routine
Real symmetric	SSYTRD, SSTEQR SSYTRD, SORGTR, SSTEQR (with output U)
Real nonsymmetric	SGEHRD, SHSEQR SGEHRD, SORGHR, SHSEQR (with output U)
Complex Hermitian	CHETRD, CSTEQR CHETRD, CUNGTR, CSTEQR (with output U)
Non-Hermitian	CGEHRD, CHSEQR CGEHRD, CUNGHR, CHSEQR (with output U)

See Also

eig, hess, qz, rsf2csf

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), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Script M-file description

Description

A script file is an external file that contains a sequence of MATLAB statements. By typing the filename, you can obtain subsequent MATLAB input from the file. Script files have a filename extension of `.m` and are often called M-files.

Scripts are the simplest kind of M-file. They are useful for automating blocks of MATLAB commands, such as computations you have to perform repeatedly from the command line. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, so you can use them in further computations. In addition, scripts can produce graphical output using commands like `plot`.

Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters.

Like any M-file, scripts can contain comments. Any text following a percent sign (`%`) on a given line is comment text. Comments can appear on lines by themselves, or you can append them to the end of any executable line.

See Also

`echo`, `function`, `type`

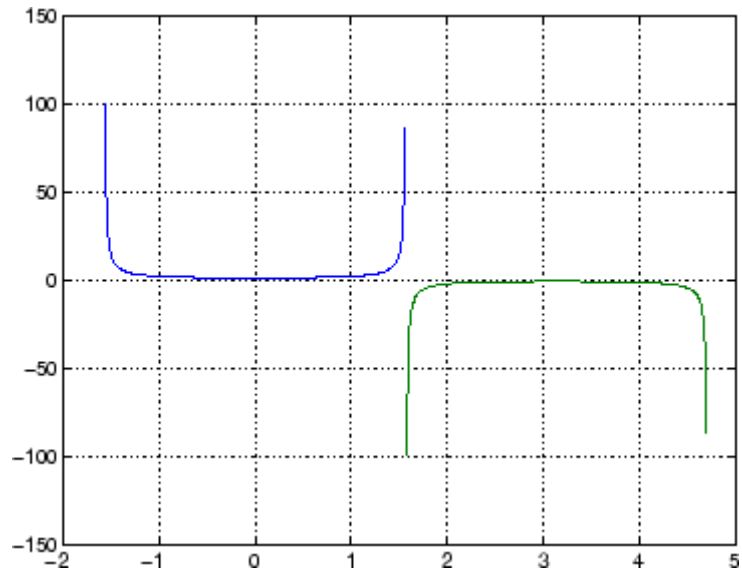
Purpose Secant of argument in radians

Syntax $Y = \sec(X)$

Description The sec function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians. $Y = \sec(X)$ returns an array the same size as X containing the secant of the elements of X .

Examples Graph the secant over the domains $-\pi/2 < x < \pi/2$ and $\pi/2 < x < 3\pi/2$.

```
x1 = -pi/2+0.01:0.01:pi/2-0.01;  
x2 = pi/2+0.01:0.01:(3*pi/2)-0.01;  
plot(x1,sec(x1),x2,sec(x2)), grid on
```



The expression $\text{sec}(\text{pi}/2)$ does not evaluate as infinite but as the reciprocal of the floating-point accuracy eps , because pi is a floating-point approximation to the exact value of π .

Definition

The secant can be defined as

$$\text{sec}(z) = \frac{1}{\cos(z)}$$

Algorithm

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

See Also

`secd`, `sech`, `asec`, `asecd`, `asech`

secd

Purpose Secant of argument in degrees

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

Description $Y = \text{secd}(X)$ is the secant of the elements of X , expressed in degrees. For odd integers n , $\text{secd}(n \cdot 90)$ is infinite, whereas $\text{sec}(n \cdot \pi/2)$ is large but finite, reflecting the accuracy of the floating point value of π .

See Also `sec`, `sech`, `asec`, `asecd`, `asech`

Purpose Hyperbolic secant

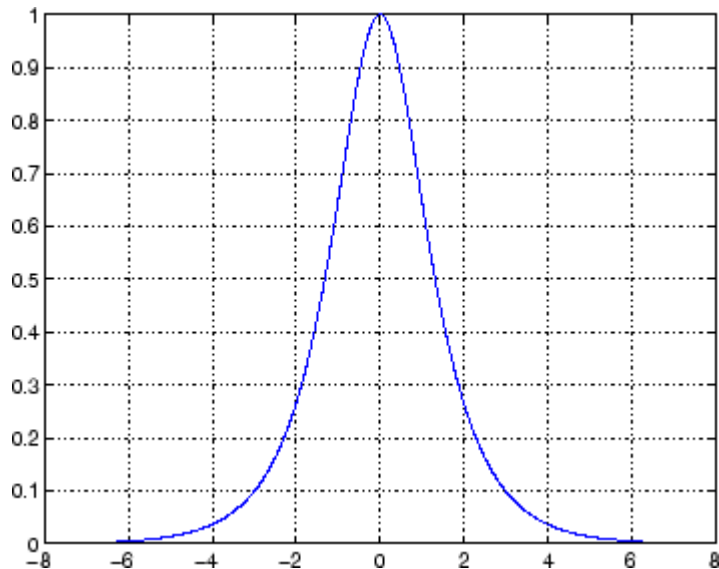
Syntax $Y = \text{sech}(X)$

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

$Y = \text{sech}(X)$ returns an array the same size as X containing the hyperbolic secant of the elements of X .

Examples Graph the hyperbolic secant over the domain $-2\pi \leq x \leq 2\pi$.

```
x = -2*pi:0.01:2*pi;  
plot(x,sech(x)), grid on
```



sech

Algorithm

sech uses this algorithm.

$$\mathbf{sech}(z) = \frac{1}{\mathbf{cosh}(z)}$$

Definition

The secant can be defined as

$$\mathbf{sech}(z) = \frac{1}{\mathbf{cosh}(z)}$$

Algorithm

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

See Also

asec, asech, sec

Purpose Select, move, resize, or copy axes and uicontrol graphics objects

Syntax `A = selectmoveresize`
`set(gca, 'ButtonDownFcn', 'selectmoveresize')`

Description `selectmoveresize` is useful as the callback routine for axes and uicontrol button down functions. When executed, it selects the object and allows you to move, resize, and copy it.

`A = selectmoveresize` returns a structure array containing

- `A.Type`: a string containing the action type, which can be `Select`, `Move`, `Resize`, or `Copy`
- `A.Handles`: a list of the selected handles, or, for a `Copy`, an `m`-by-2 matrix containing the original handles in the first column and the new handles in the second column

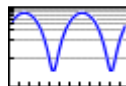
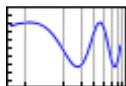
`set(gca, 'ButtonDownFcn', 'selectmoveresize')` sets the `ButtonDownFcn` property of the current axes to `selectmoveresize`:

See Also The `ButtonDownFcn` property of axes and uicontrol objects
“Object Manipulation” on page 1-97 for related functions


semilogx, semilogy

Purpose

Semilogarithmic plots



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
semilogx(Y)
semilogy(...)
semilogx(X1,Y1,...)
semilogx(X1,Y1,LineStyle,...)
semilogx(...,'PropertyName',PropertyValue,...)
h = semilogx(...)
hlines = semilogx('v6',...)
```

Description

`semilogx` and `semilogy` plot data as logarithmic scales for the x - and y -axis, respectively.

`semilogx(Y)` creates a plot using a base 10 logarithmic scale for the x -axis and a linear scale for the y -axis. It plots the columns of Y versus their index if Y contains real numbers. `semilogx(Y)` is equivalent to `semilogx(real(Y), imag(Y))` if Y contains complex numbers. `semilogx` ignores the imaginary component in all other uses of this function.

`semilogy(...)` creates a plot using a base 10 logarithmic scale for the y -axis and a linear scale for the x -axis.

`semilogx(X1,Y1,...)` plots all X_n versus Y_n pairs. If only X_n or Y_n is a matrix, `semilogx` plots the vector argument versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`semilogx(X1,Y1,LineStyle,...)` plots all lines defined by the `Xn,Yn,LineStyle` triples. `LineStyle` determines line style, marker symbol, and color of the plotted lines.

`semilogx(...,'PropertyName',PropertyValue,...)` sets property values for all `lineseries` graphics objects created by `semilogx`.

`h = semilogx(...)` and `h = semilogy(...)` return a vector of handles to `lineseries` graphics objects, one handle per line.

Backward-Compatible Version

`hlines = semilogx('v6',...)` and `hlines = semilogy('v6',...)` return the handles to line objects instead of `lineseries` objects.

Remarks

If you do not specify a color when plotting more than one line, `semilogx` and `semilogy` automatically cycle through the colors and line styles in the order specified by the current axes `ColorOrder` and `LineStyleOrder` properties.

You can mix `Xn,Yn` pairs with `Xn,Yn,LineStyle` triples; for example,

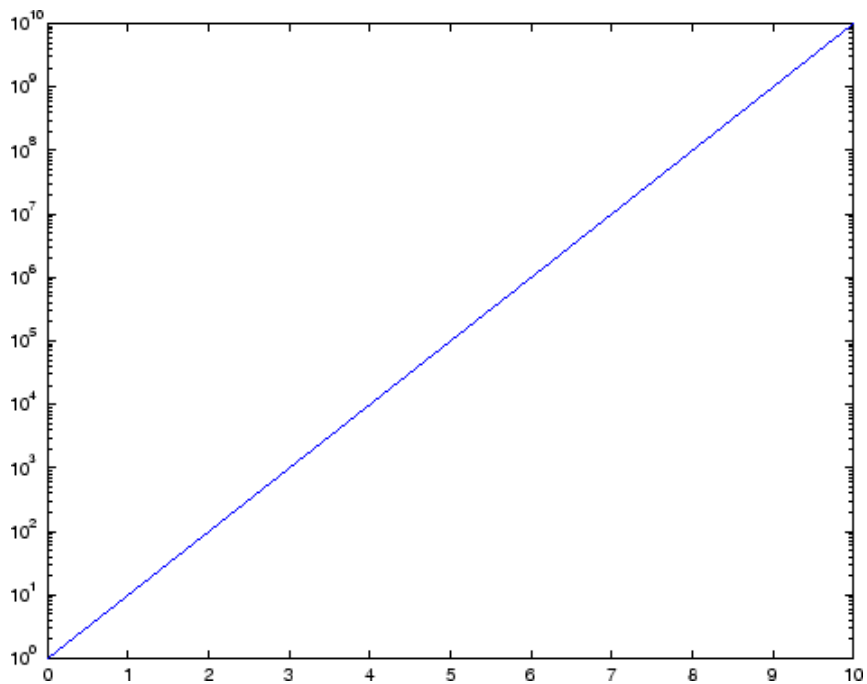
```
semilogx(X1,Y1,X2,Y2,LineStyle,X3,Y3)
```

Examples

Create a simple `semilogy` plot.

```
x = 0:.1:10;  
semilogy(x,10.^x)
```

semilogx, semilogy



See Also

line, LineSpec, loglog, plot

“Basic Plots and Graphs” on page 1-82 for related functions

Purpose

Return list of events control can trigger

Note Support for `send` will be removed in a future release of MATLAB. Use the `events` function instead of `send`.

sendmail

Purpose Send e-mail message to address list

Syntax
`sendmail('recipients','subject')`
`sendmail('recipients','subject','message','attachments')`

Description `sendmail('recipients','subject')` sends e-mail to recipients with the specified subject. For recipients, use a string for a single address, or a cell array of strings for multiple addresses.

`sendmail('recipients','subject','message','attachments')` sends message to recipients with the specified subject. For recipients, use a string for a single address, or a cell array of strings for multiple addresses. For message, use a string or cell array. When message is a string, the text automatically wraps at 75 characters. When message is a cell array, it does not wrap but rather each cell is a new line. To force text to start on a new line in strings or cells, use 10, as shown in the “Example of sendmail with New Lines Specified” on page 2-2677. Specify attachments as a cell array of files to send along with message.

To use `sendmail`, you must set the preferences for your e-mail server (Internet SMTP server) and your e-mail address must be set. MATLAB tries to read the SMTP mail server from your system registry, but if it cannot, it results in an error. In this event, identify the outgoing mail server for your electronic mail application, which is usually listed in the application’s preferences, or, consult your e-mail system administrator. Then provide the information to MATLAB using

```
setpref('Internet','SMTP_Server','myserver.myhost.com');
```

If you cannot easily determine your e-mail server, try using `mail`, as in

```
setpref('Internet','SMTP_Server','mail');
```

which might work because `mail` is often a default for mail systems.

Similarly, if MATLAB cannot determine your e-mail address and produces an error, specify your e-mail address using

```
setpref('Internet','E_mail','myaddress@example.com');
```

Note The sendmail function does not support e-mail servers that require authentication.

Examples

Example of sendmail with Two Attachments

```
sendmail('user@otherdomain.com','Test subject','Test message',  
{'directory/attach1.html','attach2.doc'});
```

Example of sendmail with New Lines Specified

This mail message forces the message to start new lines after each 10.

```
sendmail('user@otherdomain.com','New subject', ...  
['Line1 of message' 10 'Line2 of message' 10 ...  
'Line3 of message' 10 'Line4 of message']);
```

The resulting message is

```
Line1 of message  
Line2 of message  
Line3 of message  
Line4 of message
```

See Also

getpref, setpref

serial

Purpose Create serial port object

Syntax
`obj = serial('port')`
`obj = serial('port', 'PropertyName', PropertyValue, ...)`

Arguments

'port'	The serial port name.
'PropertyName'	A serial port property name.
PropertyValue	A property value supported by <i>PropertyName</i> .
obj	The serial port object.

Description

`obj = serial('port')` creates a serial port object associated with the serial port specified by `port`. If `port` does not exist, or if it is in use, you will not be able to connect the serial port object to the device.

`obj = serial('port', 'PropertyName', PropertyValue, ...)` creates a serial port object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned and the serial port object is not created.

Remarks When you create a serial port object, these property values are automatically configured:

- The `Type` property is given by `serial`.
- The `Name` property is given by concatenating `Serial` with the port specified in the `serial` function.
- The `Port` property is given by the port specified in the `serial` function.

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use

of property name completion. For example, the following commands are all valid.

```
s = serial('COM1','BaudRate',4800);
s = serial('COM1','baudrate',4800);
s = serial('COM1','BAUD',4800);
```

Refer to [Configuring Property Values](#) for a list of serial port object properties that you can use with `serial`.

Before you can communicate with the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while the object is not connected to the device. You can connect only one serial port object to a given serial port.

Example

This example creates the serial port object `s1` associated with the serial port `COM1`.

```
s1 = serial('COM1');
```

The `Type`, `Name`, and `Port` properties are automatically configured.

```
get(s1,{'Type','Name','Port'})
ans =
    'serial'    'Serial-COM1'    'COM1'
```

To specify properties during object creation

```
s2 = serial('COM2','BaudRate',1200,'DataBits',7);
```

See Also

Functions

`fclose`, `fopen`

Properties

`Name`, `Port`, `Status`, `Type`

serialbreak

Purpose Send break to device connected to serial port

Syntax `serialbreak(obj)`
`serialbreak(obj,time)`

Arguments

<code>obj</code>	A serial port object.
<code>time</code>	The duration of the break, in milliseconds.

Description `serialbreak(obj)` sends a break of 10 milliseconds to the device connected to `obj`.

`serialbreak(obj,time)` sends a break to the device with a duration, in milliseconds, specified by `time`. Note that the duration of the break might be inaccurate under some operating systems.

Remarks For some devices, the break signal provides a way to clear the hardware buffer.

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

`serialbreak` is a synchronous function, and blocks the command line until execution is complete.

If you issue `serialbreak` while data is being asynchronously written, an error is returned. In this case, you must call the `stopasync` function or wait for the write operation to complete.

See Also

Functions

`fopen`, `stopasync`

Properties

`Status`

Purpose

Set object properties

Syntax

```
set(H, 'PropertyName', PropertyValue, ...)  
set(H, a)  
set(H, pn, pv, ...)  
set(H, pn, <m-by-n cell array>)  
a = set(h)  
a = set(h, 'Default')  
a = set(h, 'DefaultObjectTypePropertyName')  
pv = set(h, 'PropertyName')
```

Description

`set(H, 'PropertyName', PropertyValue, ...)` sets the named properties to the specified values on the object(s) identified by H. H can be a vector of handles, in which case `set` sets the properties' values for all the objects.

`set(H, a)` sets the named properties to the specified values on the object(s) identified by H. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties.

`set(H, pn, pv, ...)` sets the named properties specified in the cell array `pn` to the corresponding value in the cell array `pv` for all objects identified in H.

`set(H, pn, <m-by-n cell array>)` sets `n` property values on each of `m` graphics objects, where `m = length(H)` and `n` is equal to the number of property names contained in the cell array `pn`. This allows you to set a given group of properties to different values on each object.

`a = set(h)` returns the user-settable properties and possible values for the object identified by `h`. `a` is a structure array whose field names are the object's property names and whose field values are the possible values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen. `h` must be scalar.

`a = set(h, 'Default')` returns the names of properties having default values set on the object identified by `h`. `set` also returns the possible values if they are strings. `h` must be scalar.

`a = set(h, 'DefaultObjectTypePropertyName')` returns the possible values of the named property for the specified object type, if the values are strings. The argument `DefaultObjectTypePropertyName` is the word `Default` concatenated with the object type (e.g., `axes`) and the property name (e.g., `CameraPosition`). For example, `DefaultAxesCameraPosition`. `h` must be scalar.

`pv = set(h, 'PropertyName')` returns the possible values for the named property. If the possible values are strings, `set` returns each in a cell of the cell array `pv`. For other properties, `set` returns an empty cell array. If you do not specify an output argument, MATLAB displays the information on the screen. `h` must be scalar.

Remarks

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to `set`.

Setting Property Units

Note that if you are setting both the `FontSize` and the `FontUnits` properties in one function call, you must set the `FontUnits` property first so that MATLAB can correctly interpret the specified `FontSize`. The same applies to figure and axes units — always set the `Units` property before setting properties whose values you want to be interpreted in those units. For example,

```
f = figure('Units','characters',...
          'Position',[30 30 120 35]);
```

Examples

Set the `Color` property of the current axes to blue.

```
set(gca, 'Color', 'b')
```

Change all the lines in a plot to black.

```
plot(peaks)
set(findobj('Type','line'),'Color','k')
```

You can define a group of properties in a structure to better organize your code. For example, these statements define a structure called `active`, which contains a set of property definitions used for the `uicontrol` objects in a particular figure. When this figure becomes the current figure, MATLAB changes colors and enables the controls.

```
active.BackgroundColor = [.7 .7 .7];
active.Enable = 'on';
active.ForegroundColor = [0 0 0];

if gcf == control_fig_handle
    set(findobj(control_fig_handle,'Type','uicontrol'),active)
end
```

You can use cell arrays to set properties to different values on each object. For example, these statements define a cell array to set three properties,

```
PropName(1) = {'BackgroundColor'};
PropName(2) = {'Enable'};
PropName(3) = {'ForegroundColor'};
```

These statements define a cell array containing three values for each of three objects (i.e., a 3-by-3 cell array).

```
PropVal(1,1) = {[.5 .5 .5]};
PropVal(1,2) = {'off'};
PropVal(1,3) = {[.9 .9 .9]};
PropVal(2,1) = {[1 0 0]};
PropVal(2,2) = {'on'};
PropVal(2,3) = {[1 1 1]};
PropVal(3,1) = {[.7 .7 .7]};
PropVal(3,2) = {'on'};
PropVal(3,3) = {[0 0 0]};
```

Now pass the arguments to `set`,

```
set(H,PropName,PropVal)
```

where `length(H) = 3` and each element is the handle to a uicontrol.

Setting Different Values for the Same Property on Multiple Objects

Suppose you want to set the value of the `Tag` property on five line objects, each to a different value. Note how the value cell array needs to be transposed to have the proper shape.

```
h = plot(rand(5));  
set(h,{'Tag'},{'line1','line2','line3','line4','line5'})
```

See Also

`findobj`, `gca`, `gcf`, `gco`, `gcbo`, `get`

“Finding and Identifying Graphics Objects” on page 1-89 for related functions

Purpose	Set object or interface property to specified value
Syntax	<pre>h.set('pname', value) h.set('pname1', value1, 'pname2', value2, ...) set(h, ...)</pre>
Description	<p><code>h.set('pname', value)</code> sets the property specified in the string <code>pname</code> to the given value.</p> <p><code>h.set('pname1', value1, 'pname2', value2, ...)</code> sets each property specified in the <code>pname</code> strings to the given value.</p> <p><code>set(h, ...)</code> is an alternate syntax for the same operation.</p> <p>See “Converting Data” in the External Interfaces documentation for information on how MATLAB converts workspace matrices to COM data types.</p>
Examples	<p>Create an <code>mwsamp</code> control and use <code>set</code> to change the <code>Label</code> and <code>Radius</code> properties:</p> <pre>f = figure ('position', [100 200 200 200]); h = actxcontrol ('mwsamp.mwsampctrl.1', [0 0 200 200], f); h.set('Label', 'Click to fire event', 'Radius', 40); h.invoke('Redraw');</pre> <p>Here is another way to do the same thing, only without <code>set</code> and <code>invoke</code>:</p> <pre>h.Label = 'Click to fire event'; h.Radius = 40; h.Redraw;</pre>
See Also	<code>get</code> , <code>inspect</code> , <code>isprop</code> , <code>addproperty</code> , <code>deleteproperty</code>

set (serial)

Purpose Configure or display serial port object properties

Syntax

```
set(obj)
props = set(obj)
set(obj, 'PropertyName')
props = set(obj, 'PropertyName')
set(obj, 'PropertyName', PropertyValue, ...)
set(obj, PN, PV)
set(obj, S)
```

Arguments

obj	A serial port object or an array of serial port objects.
'PropertyName'	A property name for obj.
PropertyValue	A property value supported by <i>PropertyName</i> .
PN	A cell array of property names.
PV	A cell array of property values.
S	A structure with property names and property values.
props	A structure array whose field names are the property names for obj, or cell array of possible values.

Description `set(obj)` displays all configurable properties values for obj. If a property has a finite list of possible string values, then these values are also displayed.

`props = set(obj)` returns all configurable properties and their possible values for obj to props. props is a structure whose field names are the property names of obj, and whose values are cell arrays of possible property values. If the property does not have a finite set of possible values, then the cell array is empty.

`set(obj, 'PropertyName')` displays the valid values for *PropertyName* if it possesses a finite list of string values.

`props = set(obj, 'PropertyName')` returns the valid values for *PropertyName* to `props`. `props` is a cell array of possible string values or an empty cell array if *PropertyName* does not have a finite list of possible values.

`set(obj, 'PropertyName', PropertyValue, ...)` configures multiple property values with a single command.

`set(obj, PN, PV)` configures the properties specified in the cell array of strings `PN` to the corresponding values in the cell array `PV`. `PN` must be a vector. `PV` can be `m-by-n` where `m` is equal to the number of serial port objects in `obj` and `n` is equal to the length of `PN`.

`set(obj, S)` configures the named properties to the specified values for `obj`. `S` is a structure whose field names are serial port object properties, and whose field values are the values of the corresponding properties.

Remarks

Refer to [Configuring Property Values](#) for a list of serial port object properties that you can configure with `set`.

You can use any combination of property name/property value pairs, structures, and cell arrays in one call to `set`. Additionally, you can specify a property name without regard to case, and you can make use of property name completion. For example, if `s` is a serial port object, then the following commands are all valid.

```
set(s, 'BaudRate')
set(s, 'baudrate')
set(s, 'BAUD')
```

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

```
help serial/set
```

set (serial)

Examples

This example illustrates some of the ways you can use `set` to configure or return property values for the serial port object `s`.

```
s = serial('COM1');
set(s, 'BaudRate', 9600, 'Parity', 'even')
set(s, {'StopBits', 'RecordName'}, {2, 'sydney.txt'})
set(s, 'Parity')
[ {none} | odd | even | mark | space ]
```

See Also

Functions

`get`

Purpose Configure or display timer object properties

Syntax

```
set(obj)
prop_struct = set(obj)
set(obj, 'PropertyName')
prop_cell=set(obj, 'PropertyName')
set(obj, 'PropertyName',PropertyValue,...)
set(obj,S)
set(obj,PN,PV)
```

Description `set(obj)` displays property names and their possible values for all configurable properties of timer object `obj`. `obj` must be a single timer object.

`prop_struct = set(obj)` returns the property names and their possible values for all configurable properties of timer object `obj`. `obj` must be a single timer object. The return value, `prop_struct`, is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values or empty cell arrays if the property does not have a finite set of possible string values.

`set(obj, 'PropertyName')` displays the possible values for the specified property, `PropertyName`, of timer object `obj`. `obj` must be a single timer object.

`prop_cell=set(obj, 'PropertyName')` returns the possible values for the specified property, `PropertyName`, of timer object `obj`. `obj` must be a single timer object. The returned array, `prop_cell`, is a cell array of possible value strings or an empty cell array if the property does not have a finite set of possible string values.

`set(obj, 'PropertyName',PropertyValue,...)` configures the property, `PropertyName`, to the specified value, `PropertyValue`, for timer object `obj`. You can specify multiple property name/property value pairs in a single statement. `obj` can be a single timer object or a vector of timer objects, in which case `set` configures the property values for all the timer objects specified.

set (timer)

`set(obj,S)` configures the properties of `obj`, with the values specified in `S`, where `S` is a structure whose field names are object property names.

`set(obj,PN,PV)` configures the properties specified in the cell array of strings, `PN`, to the corresponding values in the cell array `PV`, for the timer object `obj`. `PN` must be a vector. If `obj` is an array of timer objects, `PV` can be an `M`-by-`N` cell array, where `M` is equal to the length of timer object array and `N` is equal to the length of `PN`. In this case, each timer object is updated with a different set of values for the list of property names contained in `PN`.

Note When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to `set`.

Examples

Create a timer object.

```
t = timer;
```

Display all configurable properties and their possible values.

```
set(t)
  BusyMode: [ {drop} | queue | error ]
  ErrorFcn: string -or- function handle -or- cell array
  ExecutionMode: [ {singleShot} | fixedSpacing | fixedDelay | fixedRate ]
  Name
  ObjectVisibility: [ {on} | off ]
  Period
  StartDelay
  StartFcn: string -or- function handle -or- cell array
  StopFcn: string -or- function handle -or- cell array
  Tag
  TasksToExecute
  TimerFcn: string -or- function handle -or- cell array
  UserData
```

View the possible values of the `ExecutionMode` property.

```
set(t, 'ExecutionMode')  
[ {singleShot} | fixedSpacing | fixedDelay | fixedRate ]
```

Set the value of a specific timer object property.

```
set(t, 'ExecutionMode', 'FixedRate')
```

Set the values of several properties of the timer object.

```
set(t, 'TimerFcn', 'callback', 'Period', 10)
```

Use a cell array to specify the names of the properties you want to set and another cell array to specify the values of these properties.

```
set(t, {'StartDelay', 'Period'}, {30, 30})
```

See Also

timer, get

set (timeseries)

Purpose Set properties of timeseries object

Syntax
`set(ts, 'Property', Value)`
`set(ts, 'Property1', Value1, 'Property2', Value2, ...)`
`set(ts, 'Property')`
`set(ts)`

Description `set(ts, 'Property', Value)` sets the property 'Property' of the timeseries object `ts` to the value `Value`. The following syntax is equivalent:

```
ts.Property = Value
```

`set(ts, 'Property1', Value1, 'Property2', Value2, ...)` sets multiple property values for `ts` with a single statement.

`set(ts, 'Property')` displays values for the specified property of the timeseries object `ts`.

`set(ts)` displays all properties and values of the timeseries object `ts`.

See Also `get (timeseries)`

Purpose

Set properties of tscollection object

Syntax

```
set(tsc, 'Property', Value)
set(tsc, 'Property1', Value1, 'Property2', Value2, ...)
set(tsc, 'Property')
```

Description

set(tsc, 'Property', Value) sets the property 'Property' of the tscollection tsc to the value Value. The following syntax is equivalent:

```
tsc.Property = Value
```

set(tsc, 'Property1', Value1, 'Property2', Value2, ...) sets multiple property values for tsc with a single statement.

set(tsc, 'Property') displays values for the specified property in the time-series collection tsc.

set(tsc) displays all properties and values of the tscollection object tsc.

See Also

get (tscollection)

setabstime (timeseries)

Purpose Set times of timeseries object as date strings

Syntax

```
ts = setabstime(ts,Times)
ts = setabstime(ts,Times,Format)
```

Description `ts = setabstime(ts,Times)` sets the times in `ts` to the date strings specified in `Times`. `Times` must either be a cell array of strings, or a char array containing valid date or time values in the same date format.

`ts = setabstime(ts,Times,Format)` explicitly specifies the date-string format used in `Times`.

Examples **1** Create a time-series object.

```
ts = timeseries(rand(3,1))
```

2 Set the absolute time vector.

```
ts = setabstime(ts,{'12-DEC-2005 12:34:56',...
'12-DEC-2005 13:34:56','12-DEC-2005 14:34:56'})
```

See Also `datestr`, `getabstime (timeseries)`, `timeseries`

Purpose Set times of tscollection object as date strings

Syntax

```
tsc = setabstime(tsc,Times)
tsc = setabstime(tsc,Times,format)
```

Description

`tsc = setabstime(tsc,Times)` sets the times in `tsc` using the date strings `Times`. `Times` must be either a cell array of strings, or a char array containing valid date or time values in the same date format.

`tsc = setabstime(tsc,Times,format)` specifies the date-string format used in `Times` explicitly.

Examples

- 1 Create a tscollection object.

```
tsc = tscollection(timeseries(rand(3,1)))
```

- 2 Set the absolute time vector.

```
tsc = setabstime(tsc,{'12-DEC-2005 12:34:56',...
'12-DEC-2005 13:34:56','12-DEC-2005 14:34:56'})
```

See Also `datestr`, `getabstime (tscollection)`, `tscollection`

setappdata

Purpose Specify application-defined data

Syntax `setappdata(h, 'name', value)`

Description `setappdata(h, 'name', value)` sets application-defined data for the object with handle `h`. The application-defined data, which is created if it does not already exist, is assigned the specified name and value. The value can be any type of data.

See Also `getappdata`, `isappdata`, `rmappdata`

Purpose

Find set difference of two vectors

Syntax

```
c = setdiff(A, B)
c = setdiff(A, B, 'rows')
[c,i] = setdiff(...)
```

Description

`c = setdiff(A, B)` returns the values in `A` that are not in `B`. In set theory terms, $c = A - B$. Inputs `A` and `B` can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted in ascending order.

`c = setdiff(A, B, 'rows')`, when `A` and `B` are matrices with the same number of columns, returns the rows from `A` that are not in `B`.

`[c,i] = setdiff(...)` also returns an index vector `index` such that `c = a(i)` or `c = a(i,:)`.

Remarks

Because NaN is considered to be not equal to itself, it is always in the result `c` if it is in `A`.

Examples

```
A = magic(5);
B = magic(4);
[c, i] = setdiff(A(:), B(:));
c' =    17    18    19    20    21    22    23    24    25
i' =     1    10    14    18    19    23     2     6    15
```

See Also

`intersect`, `ismember`, `issorted`, `setxor`, `union`, `unique`

setenv

Purpose Set environment variable

Syntax `setenv(name, value)`
`setenv(name)`

Description `setenv(name, value)` sets the value of an environment variable belonging to the underlying operating system. Inputs `name` and `value` are both strings. If `name` already exists as an environment variable, then `setenv` replaces its current value with the string given in `value`. If `name` does not exist, `setenv` creates a new environment variable called `name` and assigns `value` to it.

`setenv(name)` is equivalent to `setenv(name, '')` and assigns a null value to the variable `name`. Under the Windows operating system, this is equivalent to undefining the variable. On most UNIX-like platforms, it is possible to have an environment variable defined as empty.

The maximum number of characters in `name` is $2^{15} - 2$ (or 32766). If `name` contains the character `=`, `setenv` throws an error. The behavior of environment variables with `=` in the name is not well-defined.

On all platforms, `setenv` passes the `name` and `value` strings to the operating system unchanged. Special characters such as `;`, `/`, `:`, `$`, `%`, etc. are left unexpanded and intact in the variable value.

Values assigned to variables using `setenv` are picked up by any process that is spawned using the MATLAB system, `unix`, `dos` or `!` functions. You can retrieve any value set with `setenv` by using `getenv(name)`.

Examples

```
% Set and retrieve a new value for the environment variable TEMP:
```

```
setenv('TEMP', 'C:\TEMP');  
getenv('TEMP')
```

```
% Append the Perl\bin directory to your system PATH variable:
```

```
setenv('PATH', [getenv('PATH') 'D:\Perl\bin']);
```

See Also

`getenv`, `system`, `unix`, `dos`, `!`

Purpose

Set value of structure array field

Syntax

```
s = setfield(s, 'field', v)
s = setfield(s, {i,j}, 'field', {k}, v)
```

Description

`s = setfield(s, 'field', v)`, where `s` is a 1-by-1 structure, sets the contents of the specified field to the value `v`. If `field` is not an existing field in structure `s`, MATLAB creates that field and assigns the value `v` to it. This is equivalent to the syntax `s.field = v`.

`s = setfield(s, {i,j}, 'field', {k}, v)` sets the contents of the specified field to the value `v`. If `field` is not an existing field in structure `s`, MATLAB creates that field and assigns the value `v` to it. This is equivalent to the syntax `s(i,j).field(k) = v`. All subscripts must be passed as cell arrays — that is, they must be enclosed in curly braces (similar to `{i,j}` and `{k}` above). Pass field references as strings.

See “Naming conventions for Structure Field Names” for guidelines to creating valid field names.

Remarks

In many cases, you can use dynamic field names in place of the `getfield` and `setfield` functions. Dynamic field names express structure fields as variable expressions that MATLAB evaluates at run-time. See Solution 1-19QWG for information about using dynamic field names versus the `getfield` and `setfield` functions.

Examples

Given the structure

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

You can change the name field of `mystr(2,1)` using

```
mystr = setfield(mystr, {2,1}, 'name', 'ted');
mystr(2,1).name
```

setfield

```
ans =
```

```
ted
```

The following example sets fields of a structure using `setfield` with variable and quoted field names and additional subscripting arguments.

```
class = 5; student = 'John_Doe';  
grades_Doe = [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];  
grades = [];  
  
grades = setfield(grades, {class}, student, 'Math', ...  
    {10, 21:30}, grades_Doe);
```

You can check the outcome using the standard structure syntax.

```
grades(class).John_Doe.Math(10, 21:30)
```

```
ans =
```

```
85    89    76    93    85    91    68    84    95    73
```

See Also

`getfield`, `fieldnames`, `isfield`, `orderfields`, `rmfield`, “Using Dynamic Field Names”

Purpose Set default interpolation method for timeseries object

Syntax

```
ts = setinterpmethod(ts,Method)
ts = setinterpmethod(ts,FHandle)
ts = setinterpmethod(ts,InterpObj),
```

Description `ts = setinterpmethod(ts,Method)` sets the default interpolation method for timeseries object `ts`, where `Method` is a string. `Method` in `ts`. `Method` is either 'linear' or 'zoh' (zero-order hold). For example:

```
ts = timeseries(rand(100,1),1:100);
ts = setinterpmethod(ts,'zoh');
```

`ts = setinterpmethod(ts,FHandle)` sets the default interpolation method for timeseries object `ts`, where `FHandle` is a function handle to the interpolation method defined by the function handle `FHandle`. For example:

```
ts = timeseries(rand(100,1),1:100);
myFuncHandle =
@(new_Time,Time,Data)interp1(Time,Data,new_Time,'linear',
'extrap');
ts = setinterpmethod(ts, myFuncHandle);
ts = resample(ts,[-5:0.1:10]);
plot(ts);
```

Note For `FHandle`, you must use three input arguments. The order of input arguments must be `new_Time`, `Time`, and `Data`. The single output argument must be the interpolated data only.

`ts = setinterpmethod(ts,InterpObj)`, where `InterpObj` is a `tsdata.interpolation` object that directly replaces the interpolation object stored in `ts`. For example:

```
ts = timeseries(rand(100,1),1:100);
```

setinterpmethod

```
myFuncHandle = @(new_Time,Time,Data)
interp1(Time,Data,new_Time,'linear','extrap');
myInterpObj = tsdata.interpolation(myFuncHandle);
ts = setinterpmethod(ts,myInterpObj);
```

This method is case sensitive.

See Also

getinterpmethod, timeseries, tsprops

Purpose

Set preference

Syntax

```
setpref('group','pref',val)
setpref('group',{'pref1','pref2',...,'prefn'},{val1,val2,...,
    valn})
```

Description

setpref('group','pref',val) sets the preference specified by group and pref to the value val. Setting a preference that does not yet exist causes it to be created.

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

setpref('group',{'pref1','pref2',...,'prefn'},{val1,val2,...,valn}) sets each preference specified in the cell array of names to the corresponding value.

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

Examples

```
addpref('mytoolbox','version','0.0')
setpref('mytoolbox','version','1.0')
getpref('mytoolbox','version')
```

```
ans =
    1.0
```

See Also

addpref, getpref, ispref, rmpref, uigetpref, uisetpref

setstr

Purpose Set string flag

Description This MATLAB 4 function has been renamed char in MATLAB 5.

Purpose	Change name of timeseries object in tscollection
Syntax	<code>tsc = settimeseriesnames(tsc,old,new)</code>
Description	<code>tsc = settimeseriesnames(tsc,old,new)</code> replaces the old name of timeseries object with the new name in tsc.
See Also	<code>tscollection</code>

setxor

Purpose Find set exclusive OR of two vectors

Syntax

```
c = setxor(A, B)
c = setxor(A, B, 'rows')
[c, ia, ib] = setxor(...)
```

Description `c = setxor(A, B)` returns the values that are not in the intersection of A and B. Inputs A and B can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted.

`c = setxor(A, B, 'rows')`, when A and B are matrices with the same number of columns, returns the rows that are not in the intersection of A and B.

`[c, ia, ib] = setxor(...)` also returns index vectors `ia` and `ib` such that `c` is a sorted combination of the elements `c = a(ia)` and `c = b(ib)` or, for row combinations, `c = a(ia,:)` and `c = b(ib,:)`.

Examples

```
a = [-1 0 1 Inf -Inf NaN];
b = [-2 pi 0 Inf];
c = setxor(a, b)
```

```
c =
    -Inf    -2.0000    -1.0000     1.0000     3.1416     NaN
```

See Also

`intersect`, `ismember`, `issorted`, `setdiff`, `union`, `unique`

Purpose	Set color shading properties
Syntax	<pre>shading flat shading faceted shading interp shading(axes_handle,...)</pre>
Description	<p>The shading function controls the color shading of surface and patch graphics objects.</p> <p><code>shading flat</code> each mesh line segment and face has a constant color determined by the color value at the endpoint of the segment or the corner of the face that has the smallest index or indices.</p> <p><code>shading faceted</code> flat shading with superimposed black mesh lines. This is the default shading mode.</p> <p><code>shading interp</code> varies the color in each line segment and face by interpolating the colormap index or true color value across the line or face.</p> <p><code>shading(axes_handle,...)</code> applies the shading type to the objects in the axes specified by <code>axes_handle</code>, instead of the current axes.</p>

Examples Compare a flat, faceted, and interpolated-shaded sphere.

```
subplot(3,1,1)
sphere(16)
axis square
shading flat
title('Flat Shading')
```

```
subplot(3,1,2)
sphere(16)
axis square
shading faceted
title('Faceted Shading')
```

```
subplot(3,1,3)
```

shading

```
sphere(16)
axis square
shading interp
title('Interpolated Shading')
```

Algorithm

shading sets the EdgeColor and FaceColor properties of all surface and patch graphics objects in the current axes. shading sets the appropriate values, depending on whether the surface or patch objects represent meshes or solid surfaces.

See Also

fill, fill3, hidden, mesh, patch, pcolor, surf

The EdgeColor and FaceColor properties for patch and surface graphics objects.

“Color Operations” on page 1-95 for related functions

Purpose Shift dimensions

Syntax `B = shiftdim(X,n)`
`[B,nshifts] = shiftdim(X)`

Description `B = shiftdim(X,n)` shifts the dimensions of `X` by `n`. When `n` is positive, `shiftdim` shifts the dimensions to the left and wraps the `n` leading dimensions to the end. When `n` is negative, `shiftdim` shifts the dimensions to the right and pads with singletons.

`[B,nshifts] = shiftdim(X)` returns the array `B` with the same number of elements as `X` but with any leading singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`. `nshifts` is the number of dimensions that are removed.

If `X` is a scalar, `shiftdim` has no effect.

Examples The `shiftdim` command is handy for creating functions that, like `sum` or `diff`, work along the first nonsingleton dimension.

```
a = rand(1,1,3,1,2);
[b,n] = shiftdim(a); % b is 3-by-1-by-2 and n is 2.
c = shiftdim(b,-n); % c == a.
d = shiftdim(a,3); % d is 1-by-2-by-1-by-1-by-3.
```

See Also `circshift`, `reshape`, `squeeze`

showplottool

Purpose Show or hide one of the figure plot tools

Syntax

```
showplottool('tool')  
showplottool('on','tool')  
showplottool('off','tool')  
showplottool('toggle','tool')  
showplottool(figure_handle,...)
```

Description `showplottool('tool')` shows the specified plot tool on the current figure. `tool` can be one of the following strings:

- `figurepalette`
- `plotbrowser`
- `propertyeditor`

`showplottool('on','tool')` shows the specified plot tool on the current figure.

`showplottool('off','tool')` hides the specified plot tool on the current figure.

`showplottool('toggle','tool')` toggles the visibility of the specified plot tool on the current figure.

`showplottool(figure_handle,...)` operates on the specified figure instead of the current figure.

See Also `getplottool`, `plottools`

Purpose Reduce the size of patch faces

Syntax

Description

`shrinkfaces(p,sf)` shrinks the area of the faces in patch `p` to shrink factor `sf`. A shrink factor of 0.6 shrinks each face to 60% of its original area. If the patch contains shared vertices, MATLAB creates nonshared vertices before performing the face-area reduction.

`nfv = shrinkfaces(p,sf)` returns the face and vertex data in the struct `nfv`, but does not set the `Faces` and `Vertices` properties of patch `p`.

`nfv = shrinkfaces(fv,sf)` uses the face and vertex data from the struct `fv`.

`shrinkfaces(p)` and `shrinkfaces(fv)` (without specifying a shrink factor) assume a shrink factor of 0.3.

`nfv = shrinkfaces(f,v,sf)` uses the face and vertex data from the arrays `f` and `v`.

`[nf,nv] = shrinkfaces(...)` returns the face and vertex data in two separate arrays instead of a struct.

Examples

This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type `help flow` for more information). Two isosurfaces provide a before and after view of the effects of shrinking the face size.

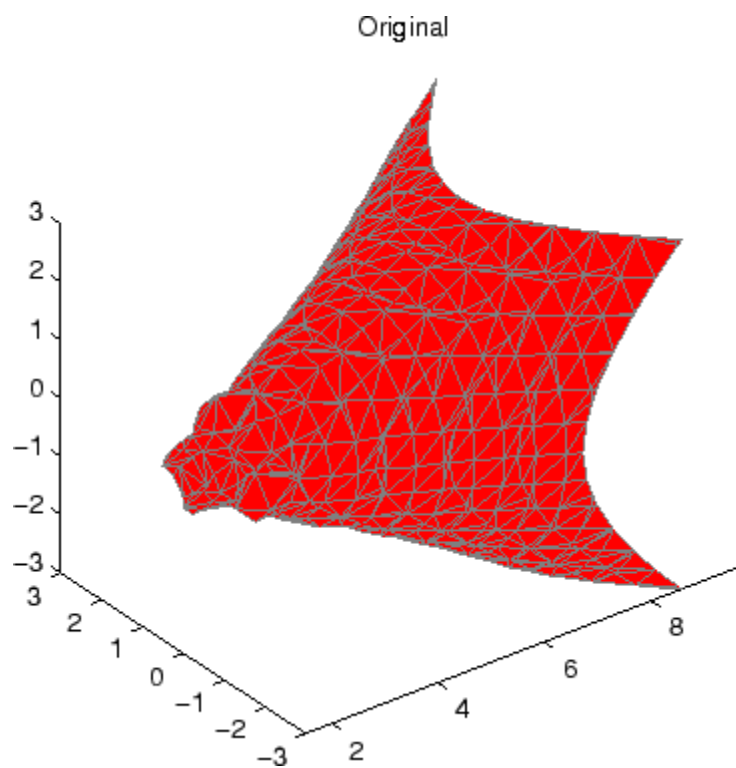
- First `reducevolume` samples the flow data at every other point and then `isosurface` generates the faces and vertices data.
- The `patch` command accepts the face/vertex struct and draws the first (`p1`) isosurface.
- Use the `daspect`, `view`, and `axis` commands to set up the view and then add a title.

shrinkfaces

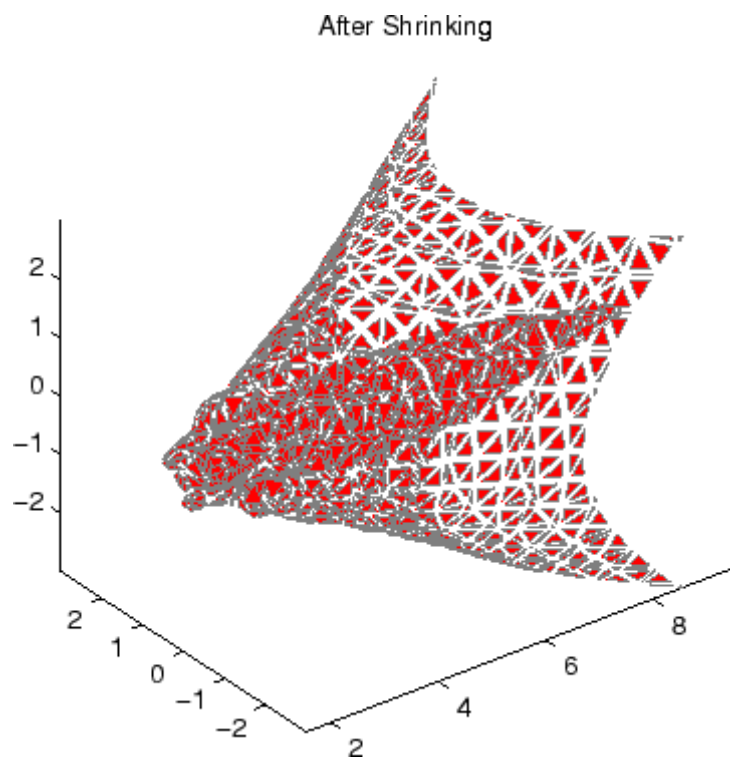
- The `shrinkfaces` command modifies the face/vertex data and passes it directly to `patch`.

```
[x,y,z,v] = flow;
[x,y,z,v] = reducevolume(x,y,z,v,2);
fv = isosurface(x,y,z,v,-3);
p1 = patch(fv);
set(p1,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
title('Original')

figure
p2 = patch(shrinkfaces(fv,.3));
set(p2,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
title('After Shrinking')
```

shrinkfaces



See Also

`isosurface`, `patch`, `reducevolume`, `daspect`, `view`, `axis`
“Volume Visualization” on page 1-98 for related functions

Purpose Signum function

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

Description $Y = \text{sign}(X)$ returns an array Y the same size as X , where each element of Y is:

- 1 if the corresponding element of X is greater than zero
- 0 if the corresponding element of X equals zero
- -1 if the corresponding element of X is less than zero

For nonzero complex X , $\text{sign}(X) = X ./ \text{abs}(X)$.

See Also `abs`, `conj`, `imag`, `real`

sin

Purpose Sine of argument in radians

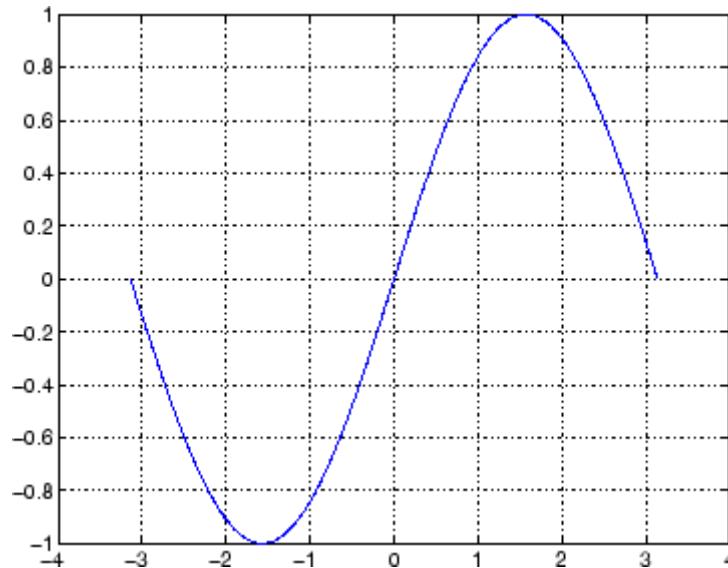
Syntax $Y = \sin(X)$

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

$Y = \sin(X)$ returns the circular sine of the elements of X .

Examples Graph the sine function over the domain $-\pi \leq x \leq \pi$.

```
x = -pi:0.01:pi;  
plot(x,sin(x)), grid on
```



The expression $\sin(\pi)$ is not exactly zero, but rather a value the size of the floating-point accuracy `eps`, because `pi` is only a floating-point approximation to the exact value of π .

Definition

The sine can be defined as

$$\sin(x + iy) = \sin(x)\cosh(y) + i\cos(x)\sinh(y)$$

$$\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}$$

Algorithm

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

See Also

sind, sinh, asin, asind, asinh

sind

Purpose Sine of argument in degrees

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

Description $Y = \text{sind}(X)$ is the sine of the elements of X , expressed in degrees. For integers n , $\text{sind}(n \cdot 180)$ is exactly zero, whereas $\sin(n \cdot \pi)$ reflects the accuracy of the floating point value of π .

See Also `sin`, `sinh`, `asin`, `asind`, `asinh`

Purpose Convert to single precision

Syntax `B = single(A)`

Description `B = single(A)` converts the matrix `A` to single precision, returning that value in `B`. `A` can be any numeric object (such as a double). If `A` is already single precision, `single` has no effect. Single-precision quantities require less storage than double-precision quantities, but have less precision and a smaller range.

The `single` class is primarily meant to be used to store single-precision values. Hence most operations that manipulate arrays without changing their elements are defined. Examples are `reshape`, `size`, the relational operators, subscripted assignment, and subscripted reference.

You can define your own methods for the `single` class by placing the appropriately named method in an `@single` directory within a directory on your path.

Examples

```
a = magic(4);  
b = single(a);
```

```
whos
```

Name	Size	Bytes	Class
a	4x4	128	double array
b	4x4	64	single array

See Also `double`

sinh

Purpose Hyperbolic sine of argument in radians

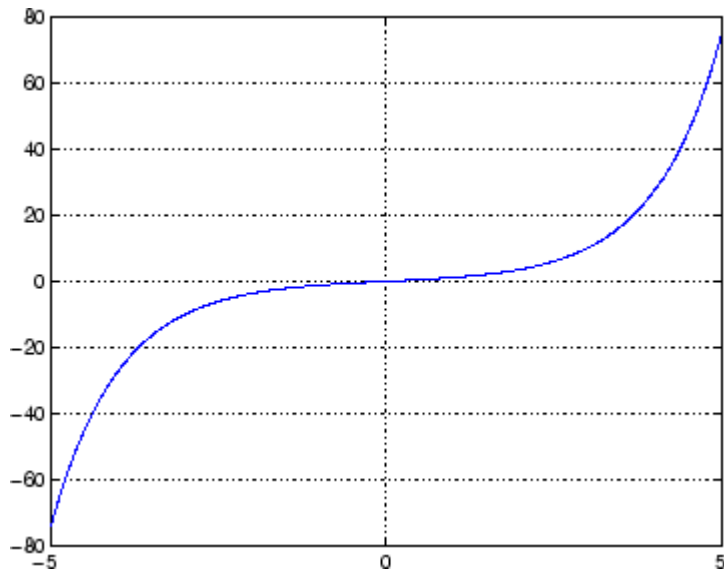
Syntax $Y = \sinh(X)$

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

$Y = \sinh(X)$ returns the hyperbolic sine of the elements of X .

Examples Graph the hyperbolic sine function over the domain $-5 \leq x \leq 5$.

```
x = -5:0.01:5;  
plot(x, sinh(x)), grid on
```



Definition The hyperbolic sine can be defined as

$$\sinh(z) = \frac{e^z - e^{-z}}{2}$$

Algorithm

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

See Also

`sin`, `sind`, `asin`, `asinh`, `asind`

size

Purpose Array dimensions

Syntax

```
d = size(X)
[m,n] = size(X)
m = size(X,dim)
[d1,d2,d3,...,dn] = size(X),
```

Description `d = size(X)` returns the sizes of each dimension of array `X` in a vector `d` with `ndims(X)` elements. If `X` is a scalar, which MATLAB regards as a 1-by-1 array, `size(X)` returns the vector `[1 1]`.

`[m,n] = size(X)` returns the size of matrix `X` in separate variables `m` and `n`.

`m = size(X,dim)` returns the size of the dimension of `X` specified by scalar `dim`.

`[d1,d2,d3,...,dn] = size(X)`, for $n > 1$, returns the sizes of the dimensions of the array `X` in the variables `d1,d2,d3,...,dn`, provided the number of output arguments `n` equals `ndims(X)`. If `n` does not equal `ndims(X)`, the following exceptions hold:

$n < \text{ndims}(X)$ `di` equals the size of the i th dimension of `X` for $1 \leq i < n$, but `dn` equals the product of the sizes of the remaining dimensions of `X`, that is, dimensions n through `ndims(X)`.

$n > \text{ndims}(X)$ `size` returns ones in the “extra” variables, that is, those corresponding to `ndims(X)+1` through `n`.

Note For a Java array, `size` returns the length of the Java array as the number of rows. The number of columns is always 1. For a Java array of arrays, the result describes only the top level array.

Examples

Example 1

The size of the second dimension of `rand(2,3,4)` is 3.

```
m = size(rand(2,3,4),2)
```

```
m =  
    3
```

Here the size is output as a single vector.

```
d = size(rand(2,3,4))
```

```
d =  
    2    3    4
```

Here the size of each dimension is assigned to a separate variable.

```
[m,n,p] = size(rand(2,3,4))
```

```
m =  
    2
```

```
n =  
    3
```

```
p =  
    4
```

Example 2

If $X = \text{ones}(3,4,5)$, then

```
[d1,d2,d3] = size(X)
```

```
d1 =    d2 =    d3 =  
    3    4    5
```

But when the number of output variables is less than $\text{ndims}(X)$:

```
[d1,d2] = size(X)
```

```
d1 =    d2 =  
    3    20
```

size

The “extra” dimensions are collapsed into a single product.

If $n > \text{ndims}(X)$, the “extra” variables all represent singleton dimensions:

```
[d1,d2,d3,d4,d5,d6] = size(X)
```

```
d1 =      d2 =      d3 =  
    3          4          5
```

```
d4 =      d5 =      d6 =  
    1          1          1
```

See Also

`exist`, `length`, `numel`, `whos`

Purpose	Size of timeseries object
Syntax	<code>size(ts)</code>
Description	<code>size(ts)</code> returns <code>[n 1]</code> , where <code>n</code> is the length of the time vector for timeseries object <code>ts</code> .
Remarks	<p>If you want the size of the whole data set, use the following syntax:</p> <pre>size(ts.data)</pre> <p>If you want the size of each data sample, use the following syntax:</p> <pre>getdatasamplesize(ts)</pre>
See Also	<code>getdatasamplesize</code> , <code>isempty (timeseries)</code> , <code>length (timeseries)</code>

size (tscollection)

Purpose	Size of tscollection object
Syntax	<code>size(tsc)</code>
Description	<code>size(tsc)</code> returns [n m], where n is the length of the time vector and m is the number of tscollection members.
See Also	<code>length (tscollection)</code> , <code>isempty (tscollection)</code> , <code>tscollection</code>

Purpose Size of serial port object array

Syntax

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,m3,...,mn] = size(obj)
m = size(obj,dim)
```

Arguments

obj	A serial port object or an array of serial port objects.
dim	The dimension of obj.
d	The number of rows and columns in obj.
m	The number of rows in obj, or the length of the dimension specified by dim.
n	The number of columns in obj.
m1,m2,...,mn	The length of the first N dimensions of obj.

Description

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in `obj`.

`[m,n] = size(obj)` returns the number of rows and columns in separate output variables.

`[m1,m2,m3,...,mn] = size(obj)` returns the length of the first `n` dimensions of `obj`.

`m = size(obj,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(obj,1)` returns the number of rows.

See Also **Functions**

length


slice

Purpose

Volumetric slice plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
slice(V, sx, sy, sz)
slice(X, Y, Z, V, sx, sy, sz)
slice(V, XI, YI, ZI)
slice(X, Y, Z, V, XI, YI, ZI)
slice(..., 'method')
slice(axes_handle, ...)
h = slice(...)
```

Description

`slice` displays orthogonal slice planes through volumetric data.

`slice(V, sx, sy, sz)` draws slices along the x , y , z directions in the volume V at the points in the vectors sx , sy , and sz . V is an m -by- n -by- p volume array containing data values at the default location $X = 1:n$, $Y = 1:m$, $Z = 1:p$. Each element in the vectors sx , sy , and sz defines a slice plane in the x -, y -, or z -axis direction.

`slice(X, Y, Z, V, sx, sy, sz)` draws slices of the volume V . X , Y , and Z are three-dimensional arrays specifying the coordinates for V . X , Y , and Z must be monotonic and orthogonally spaced (as if produced by the function `meshgrid`). The color at each point is determined by 3-D interpolation into the volume V .

`slice(V, XI, YI, ZI)` draws data in the volume V for the slices defined by XI , YI , and ZI . XI , YI , and ZI are matrices that define a surface, and the volume is evaluated at the surface points. XI , YI , and ZI must all be the same size.

`slice(X,Y,Z,V,XI,YI,ZI)` draws slices through the volume V along the surface defined by the arrays XI , YI , ZI .

`slice(..., 'method')` specifies the interpolation method. *'method'* is *'linear'*, *'cubic'*, or *'nearest'*.

- *linear* specifies trilinear interpolation (the default).
- *cubic* specifies tricubic interpolation.
- *nearest* specifies nearest-neighbor interpolation.

`slice(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes object (`gca`). The axes `clim` property is set to span the finite values of V .

`h = slice(...)` returns a vector of handles to surface graphics objects.

Remarks

The color drawn at each point is determined by interpolation into the volume V .

Examples

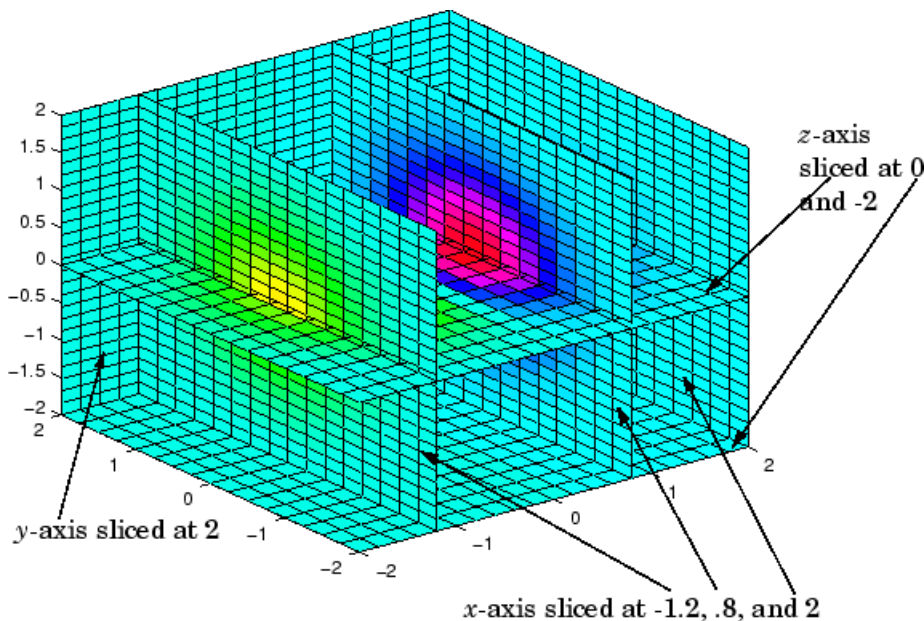
Visualize the function

$$v = xe^{(-x^2 - y^2 - z^2)}$$

over the range $-2 \leq x \leq 2$, $-2 \leq y \leq 2$, $-2 \leq z \leq 2$:

```
[x,y,z] = meshgrid(-2:.2:2,-2:.25:2,-2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2);
xslice = [-1.2,.8,2]; yslice = 2; zslice = [-2,0];
slice(x,y,z,v,xslice,yslice,zslice)
colormap hsv
```

slice



Slicing At Arbitrary Angles

You can also create slices that are oriented in arbitrary planes. To do this,

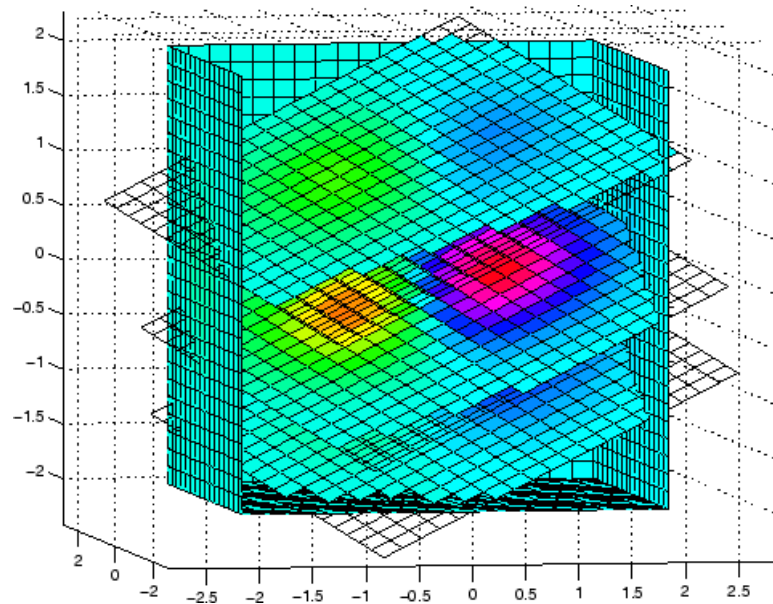
- Create a slice surface in the domain of the volume (`surf`, `linspace`).
- Orient this surface with respect to the axes (`rotate`).
- Get the `XData`, `YData`, and `ZData` of the surface (`get`).
- Use this data to draw the slice plane within the volume.

For example, these statements slice the volume in the first example with a rotated plane. Placing these commands within a `for` loop passes the plane through the volume along the `z`-axis.

```
for i = -2:.5:2
    hsp = surf(linspace(-2,2,20),linspace(-2,2,20),zeros(20)+i);
```

```
rotate(hsp,[1,-1,1],30)
xd = get(hsp,'XData');
yd = get(hsp,'YData');
zd = get(hsp,'ZData');
delete(hsp)
slice(x,y,z,v,[-2,2],2,-2) % Draw some volume boundaries
hold on
slice(x,y,z,v,xd,yd,zd)
hold off
axis tight
view(-5,10)
drawnow
end
```

The following picture illustrates three positions of the same slice surface as it passes through the volume.



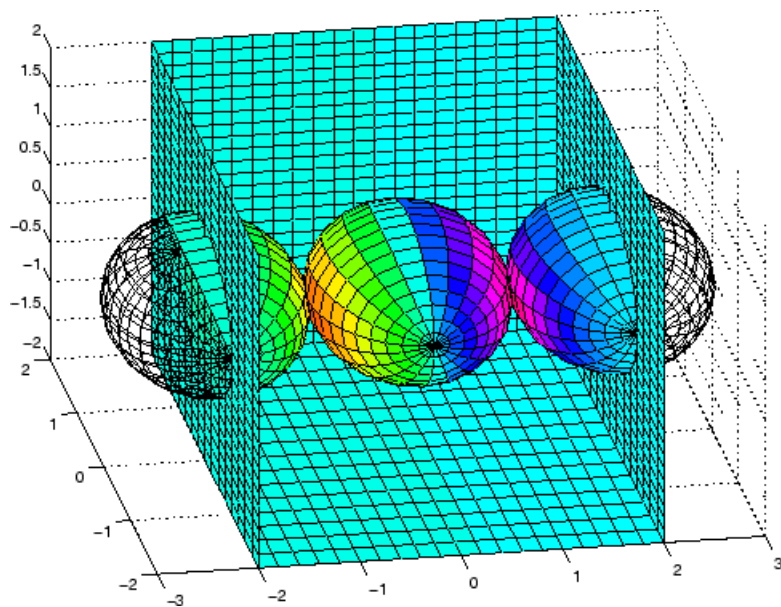
Slicing with a Nonplanar Surface

You can slice the volume with any surface. This example probes the volume created in the previous example by passing a spherical slice surface through the volume.

```
[xsp,ysp,zsp] = sphere;
slice(x,y,z,v,[-2,2],2,-2) % Draw some volume boundaries

for i = -3:.2:3
    hsp = surface(xsp+i,ysp,zsp);
    rotate(hsp,[1 0 0],90)
    xd = get(hsp,'XData');
    yd = get(hsp,'YData');
    zd = get(hsp,'ZData');
    delete(hsp)
    hold on
    hslicer = slice(x,y,z,v,xd,yd,zd);
    axis tight
    xlim([-3,3])
    view(-10,35)
    drawnow
    delete(hslicer)
    hold off
end
```

The following picture illustrates three positions of the spherical slice surface as it passes through the volume.

**See Also**

`interp3`, `meshgrid`

“Volume Visualization” on page 1-98 for related functions

Exploring Volumes with Slice Planes for more examples

smooth3

Purpose Smooth 3-D data

Syntax

Description `W = smooth3(V)` smooths the input data `V` and returns the smoothed data in `W`.

`W = smooth3(V, 'filter')` *filter* determines the convolution kernel and can be the strings

- 'gaussian'
- 'box' (default)

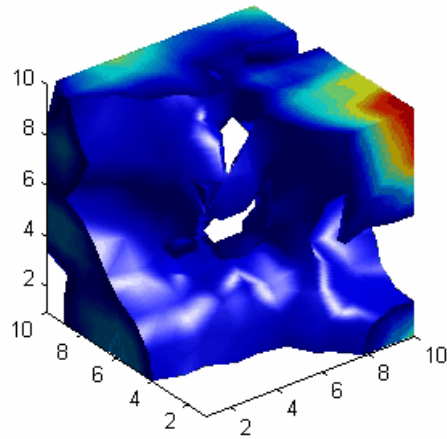
`W = smooth3(V, 'filter', size)` sets the size of the convolution kernel (default is [3 3 3]). If `size` is scalar, then `size` is interpreted as [size, size, size].

`W = smooth3(V, 'filter', size, sd)` sets an attribute of the convolution kernel. When *filter* is gaussian, `sd` is the standard deviation (default is .65).

Examples

This example smooths some random 3-D data and then creates an isosurface with end caps.

```
rand('seed',0)
data = rand(10,10,10);
data = smooth3(data,'box',5);
p1 = patch(isosurface(data,.5), ...
    'FaceColor','blue','EdgeColor','none');
p2 = patch(isocaps(data,.5), ...
    'FaceColor','interp','EdgeColor','none');
isonormals(data,p1)
view(3); axis vis3d tight
camlight; lighting phong
```

**See Also**

isocaps, isonormals, isosurface, patch

“Volume Visualization” on page 1-98 for related functions

See Displaying an Isosurface for another example.

sort

Purpose Sort array elements in ascending or descending order

Syntax
B = sort(A)
B = sort(A,dim)
B = sort(...,mode)
[B,IX] = sort(A,...)

Description B = sort(A) sorts the elements along different dimensions of an array, and arranges those elements in ascending order.

If A is a ...	sort(A) ...
Vector	Sorts the elements of A.
Matrix	Sorts each column of A.
Multidimensional array	Sorts A along the first non-singleton dimension, and returns an array of sorted vectors.
Cell array of strings	Sorts the strings in ASCII dictionary order.

Integer, floating-point, logical, and character arrays are permitted. Floating-point arrays can be complex. For elements of A with identical values, the order of these elements is preserved in the sorted list. When A is complex, the elements are sorted by magnitude, i.e., $\text{abs}(A)$, and where magnitudes are equal, further sorted by phase angle, i.e., $\text{angle}(A)$, on the interval $[-\pi, \pi]$. If A includes any NaN elements, sort places these at the high end.

B = sort(A,dim) sorts the elements along the dimension of A specified by a scalar dim.

B = sort(...,mode) sorts the elements in the specified direction, depending on the value of mode.

'ascend' Ascending order (default)

'descend' Descending order

[B,IX] = sort(A,...) also returns an array of indices IX, where size(IX) == size(A). If A is a vector, B = A(IX). If A is an m-by-n matrix, then each column of IX is a permutation vector of the corresponding column of A, such that

```
for j = 1:n
    B(:,j) = A(IX(:,j),j);
end
```

If A has repeated elements of equal value, the returned indices preserve the original ordering.

Sorting Complex Entries

If A has complex entries r and s, sort orders them according to the following rule: r appears before s in sort(A) if either of the following hold:

- $\text{abs}(r) < \text{abs}(s)$
- $\text{abs}(r) = \text{abs}(s)$ and $\text{angle}(r) < \text{angle}(s)$

where $-\pi < \text{angle}(r) \leq \pi$

For example,

```
v = [1 -1 i -i];
angle(v)

ans =

    0    3.1416    1.5708   -1.5708

sort(v)

ans =

    0 - 1.0000i    1.0000
    0 + 1.0000i   -1.0000
```

sort

Note sort uses a different rule for ordering complex numbers than do max and min, or the relational operators < and >. See the Relational Operators reference page for more information.

Examples

Example 1

This example sorts a matrix A in each dimension, and then sorts it a third time, returning an array of indices for the sorted result.

```
A = [ 3 7 5
      0 4 2 ];
```

```
sort(A,1)
```

```
ans =
     0     4     2
     3     7     5
```

```
sort(A,2)
```

```
ans =
     3     5     7
     0     2     4
```

```
[B,IX] = sort(A,2)
```

```
B =
     3     5     7
     0     2     4
```

```
IX =
     1     3     2
     1     3     2
```

Example 2

This example sorts each column of a matrix in descending order.

```
A = [ 3  7  5
      6  8  3
      0  4  2 ];
```

```
sort(A,1,'descend')
```

```
ans =
      6  8  5
      3  7  3
      0  4  2
```

This is equivalent to

```
sort(A,'descend')
```

```
ans =
      6  8  5
      3  7  3
      0  4  2
```

See Also

issorted, max, mean, median, min, sortrows

sortrows

Purpose Sort rows in ascending order

Syntax
B = sortrows(A)
B = sortrows(A,column)
[B,index] = sortrows(A,...)

Description B = sortrows(A) sorts the rows of A in ascending order. Argument A must be either a matrix or a column vector.

For strings, this is the familiar dictionary sort. When A is complex, the elements are sorted by magnitude, and, where magnitudes are equal, further sorted by phase angle on the interval $[-\pi, \pi]$.

B = sortrows(A,column) sorts the matrix based on the columns specified in the vector column. If an element of column is positive, MATLAB sorts the corresponding column of matrix A in ascending order; if an element of column is negative, MATLAB sorts the corresponding column in descending order. For example, sortrows(A,[2 -3]) sorts the rows of A first in ascending order for the second column, and then by descending order for the third column.

[B,index] = sortrows(A,...) also returns an index vector index.

If A is a column vector, then B = A(index). If A is an m-by-n matrix, then B = A(index,:).

Examples

Start with a mostly random matrix, A:

```
rand('state',0)
A = floor(rand(6,7) * 100);
A(1:4,1)=95; A(5:6,1)=76; A(2:4,2)=7; A(3,3)=73
A =
    95    45    92    41    13     1    84
    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

When called with only a single input argument, `sortrows` bases the sort on the first column of the matrix. For any rows that have equal elements in a particular column, (e.g., `A(1:4,1)` for this matrix), sorting is based on the column immediately to the right, (`A(1:4,2)` in this case):

```
sortrows(A)
ans =
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
    95     7    40    35    60    93    67
    95     7    73     5    19    44    20
    95     7    73    89    20    74    52
    95    45    92    41    13     1    84
```

When called with two input arguments, `sortrows` bases the sort entirely on the column specified in the second argument. Rows that have equal elements in this column are sorted; rows with equal elements in other columns are left in their original order:

```
sortrows(A,1)
ans =
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
    95    45    92    41    13     1    84
    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
```

This example specifies two columns to sort by: columns 1 and 7. This tells `sortrows` to sort by column 1 first, and then for any rows with equal values in column 1, to sort by column 7:

```
sortrows(A,[1 7])
ans =
    76    79    91     0    19    41     1
    76    61    93    81    27    46    83
    95     7    73     5    19    44    20
    95     7    73    89    20    74    52
```

sortrows

```
95    7    40    35    60    93    67
95   45    92    41    13     1    84
```

Sort the matrix using the values in column 4 this time and in reverse order:

```
sortrows(A, -4)
ans =
95     7    73    89    20    74    52
76    61    93    81    27    46    83
95    45    92    41    13     1    84
95     7    40    35    60    93    67
95     7    73     5    19    44    20
76    79    91     0    19    41     1
```

See Also

issorted, sort

Purpose Convert vector into sound

Syntax
sound(y,Fs)
sound(y)
sound(y,Fs,bits)

Description sound(y,Fs) sends the signal in vector y (with sample frequency Fs) to the speaker on PC and most UNIX platforms. Values in y are assumed to be in the range $-1.0 \leq y \leq 1.0$. Values outside that range are clipped. Stereo sound is played on platforms that support it when y is an n-by-2 matrix. The values in column 1 are assigned to the left channel, and those in column 2 to the right.

Note The playback duration that results from setting Fs depends on the sound card you have installed. Most sound cards support sample frequencies of approximately 5-10 kHz to 44.1 kHz. Sample frequencies outside this range can produce unexpected results.

sound(y) plays the sound at the default sample rate or 8192 Hz.

sound(y,Fs,bits) plays the sound using bits number of bits/sample, if possible. Most platforms support bits = 8 or bits = 16.

Remarks MATLAB supports all Windows-compatible sound devices. Additional sound acquisition and generation capability is available in the Data Acquisition Toolbox. The toolbox functionality includes the ability to buffer the acquisition so that you can analyze the data as it is being acquired.

See Also aread, auwrite, soundsc, audioplayer, wavread, wavwrite

soundsc

Purpose Scale data and play as sound

Syntax `soundsc(y,Fs)`
`soundsc(y)`
`soundsc(y,Fs,bits)`
`soundsc(y,...,slim)`

Description `soundsc(y,Fs)` sends the signal in vector `y` (with sample frequency `Fs`) to the speaker on PC and most UNIX platforms. The signal `y` is scaled to the range $-1.0 \leq y \leq 1.0$ before it is played, resulting in a sound that is played as loud as possible without clipping.

Note The playback duration that results from setting `Fs` depends on the sound card you have installed. Most sound cards support sample frequencies of approximately 5-10 kHz to 44.1 kHz. Sample frequencies outside this range can produce unexpected results.

`soundsc(y)` plays the sound at the default sample rate or 8192 Hz.

`soundsc(y,Fs,bits)` plays the sound using `bits` number of bits/sample if possible. Most platforms support `bits = 8` or `bits = 16`.

`soundsc(y,...,slim)`, where `slim = [slow shigh]`, maps the values in `y` between `slow` and `shigh` to the full sound range. The default value is `slim = [min(y) max(y)]`.

Remarks MATLAB supports all Windows-compatible sound devices.

See Also `auread`, `auwrite`, `sound`, `wavread`, `wavwrite`

Purpose Allocate space for sparse matrix

Syntax `S = spalloc(m,n,nzmax)`

Description `S = spalloc(m,n,nzmax)` creates an all zero sparse matrix `S` of size `m`-by-`n` with room to hold `nzmax` nonzeros. The matrix can then be generated column by column without requiring repeated storage allocation as the number of nonzeros grows.

`spalloc(m,n,nzmax)` is shorthand for

```
sparse([],[],[],m,n,nzmax)
```

Examples To generate efficiently a sparse matrix that has an average of at most three nonzero elements per column

```
S = spalloc(n,n,3*n);  
for j = 1:n  
    S(:,j) = [zeros(n-3,1)' round(rand(3,1))']';end
```

sparse

Purpose

Create sparse matrix

Syntax

```
S = sparse(A)
S = sparse(i, j, s, m, n, nzmax)
S = sparse(i, j, s, m, n)
S = sparse(i, j, s)
S = sparse(m, n)
```

Description

The `sparse` function generates matrices in the MATLAB sparse storage organization.

`S = sparse(A)` converts a full matrix to sparse form by squeezing out any zero elements. If `S` is already sparse, `sparse(S)` returns `S`.

`S = sparse(i, j, s, m, n, nzmax)` uses vectors `i`, `j`, and `s` to generate an `m`-by-`n` sparse matrix such that $S(i(k), j(k)) = s(k)$, with space allocated for `nzmax` nonzeros. Vectors `i`, `j`, and `s` are all the same length. Any elements of `s` that are zero are ignored, along with the corresponding values of `i` and `j`. Any elements of `s` that have duplicate values of `i` and `j` are added together.

Note If any value in `i` or `j` is larger than the maximum integer size, $2^{31} - 1$, then the sparse matrix cannot be constructed.

To simplify this six-argument call, you can pass scalars for the argument `s` and one of the arguments `i` or `j`—in which case they are expanded so that `i`, `j`, and `s` all have the same length.

`S = sparse(i, j, s, m, n)` uses `nzmax = length(s)`.

`S = sparse(i, j, s)` uses `m = max(i)` and `n = max(j)`. The maxima are computed before any zeros in `s` are removed, so one of the rows of `[i j s]` might be `[m n 0]`.

`S = sparse(m, n)` abbreviates `sparse([], [], [], m, n, 0)`. This generates the ultimate sparse matrix, an `m`-by-`n` all zero matrix.

Remarks

All of the MATLAB built-in arithmetic, logical, and indexing operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices.

In most cases, operations on mixtures of sparse and full matrices return full matrices. The exceptions include situations where the result of a mixed operation is structurally sparse, for example, $A * S$ is at least as sparse as S .

Examples

`S = sparse(1:n,1:n,1)` generates a sparse representation of the n -by- n identity matrix. The same S results from `S = sparse(eye(n,n))`, but this would also temporarily generate a full n -by- n matrix with most of its elements equal to zero.

`B = sparse(10000,10000,pi)` is probably not very useful, but is legal and works; it sets up a 10000-by-10000 matrix with only one nonzero element. Don't try `full(B)`; it requires 800 megabytes of storage.

This dissects and then reassembles a sparse matrix:

```
[i,j,s] = find(S);  
[m,n] = size(S);  
S = sparse(i,j,s,m,n);
```

So does this, if the last row and column have nonzero entries:

```
[i,j,s] = find(S);  
S = sparse(i,j,s);
```

See Also

`diag`, `find`, `full`, `issparse`, `nnz`, `nonzeros`, `nzmax`, `spones`, `sprandn`, `sprandsym`, `spy`

The `sparfun` directory

spaugment

Purpose Form least squares augmented system

Syntax $S = \text{spaugment}(A, c)$
 $S = \text{spaugment}(A)$

Description $S = \text{spaugment}(A, c)$ creates the sparse, square, symmetric indefinite matrix $S = [c \cdot I \ A; A' \ 0]$. The matrix S is related to the least squares problem

$$\min \text{norm}(b - A \cdot x)$$

by

$$r = b - A \cdot x$$
$$S \cdot [r/c; x] = [b; 0]$$

The optimum value of the residual scaling factor c , involves $\min(\text{svd}(A))$ and $\text{norm}(r)$, which are usually too expensive to compute.

$S = \text{spaugment}(A)$ without a specified value of c , uses $\max(\max(\text{abs}(A))) / 1000$.

Note In previous versions of MATLAB, the augmented matrix was used by sparse linear equation solvers, \backslash and $/$, for nonsquare problems. Now, MATLAB performs a least squares solve using the qr factorization of A instead.

See Also spparms

Purpose Import matrix from sparse matrix external format

Syntax `S = spconvert(D)`

Description `spconvert` is used to create sparse matrices from a simple sparse format easily produced by non-MATLAB sparse programs. `spconvert` is the second step in the process:

- 1 Load an ASCII data file containing `[i, j, v]` or `[i, j, re, im]` as rows into a MATLAB variable.
- 2 Convert that variable into a MATLAB sparse matrix.

`S = spconvert(D)` converts a matrix `D` with rows containing `[i, j, s]` or `[i, j, r, s]` to the corresponding sparse matrix. `D` must have an `nnz` or `nnz+1` row and three or four columns. Three elements per row generate a real matrix and four elements per row generate a complex matrix. A row of the form `[m n 0]` or `[m n 0 0]` anywhere in `D` can be used to specify `size(S)`. If `D` is already sparse, no conversion is done, so `spconvert` can be used after `D` is loaded from either a MAT-file or an ASCII file.

Examples Suppose the ASCII file `uphill.dat` contains

```

1   1   1.0000000000000000
1   2   0.5000000000000000
2   2   0.3333333333333333
1   3   0.3333333333333333
2   3   0.2500000000000000
3   3   0.2000000000000000
1   4   0.2500000000000000
2   4   0.2000000000000000
3   4   0.1666666666666667
4   4   0.142857142857143
4   4   0.0000000000000000

```

Then the statements

spconvert

```
load uphill.dat
H = spconvert(uphill)
```

```
H =
    (1,1)    1.0000
    (1,2)    0.5000
    (2,2)    0.3333
    (1,3)    0.3333
    (2,3)    0.2500
    (3,3)    0.2000
    (1,4)    0.2500
    (2,4)    0.2000
    (3,4)    0.1667
    (4,4)    0.1429
```

recreate `sparse(triu(hilb(4)))`, possibly with roundoff errors. In this case, the last line of the input file is not necessary because the earlier lines already specify that the matrix is at least 4-by-4.

Purpose

Extract and create sparse band and diagonal matrices

Syntax

```
B = spdiags(A)
[B,d] = spdiags(A)
B = spdiags(A,d)
A = spdiags(B,d,A)
A = spdiags(B,d,m,n)
```

Description

The `spdiags` function generalizes the function `diag`. Four different operations, distinguished by the number of input arguments, are possible.

`B = spdiags(A)` extracts all nonzero diagonals from the m -by- n matrix A . B is a $\min(m,n)$ -by- p matrix whose columns are the p nonzero diagonals of A .

`[B,d] = spdiags(A)` returns a vector d of length p , whose integer components specify the diagonals in A .

`B = spdiags(A,d)` extracts the diagonals specified by d .

`A = spdiags(B,d,A)` replaces the diagonals specified by d with the columns of B . The output is sparse.

`A = spdiags(B,d,m,n)` creates an m -by- n sparse matrix by taking the columns of B and placing them along the diagonals specified by d .

Note In this syntax, if a column of B is longer than the diagonal it is replacing, and $m \geq n$, `spdiags` takes elements of super-diagonals from the lower part of the column of B , and elements of sub-diagonals from the upper part of the column of B . However, if $m < n$, then super-diagonals are from the upper part of the column of B , and sub-diagonals from the lower part. (See “Example 5A” on page 2-2757 and “Example 5B” on page 2-2759, below).

Arguments

The `spdiags` function deals with three matrices, in various combinations, as both input and output.

- A An m -by- n matrix, usually (but not necessarily) sparse, with its nonzero or specified elements located on p diagonals.
- B A $\min(m, n)$ -by- p matrix, usually (but not necessarily) full, whose columns are the diagonals of A.
- d A vector of length p whose integer components specify the diagonals in A.

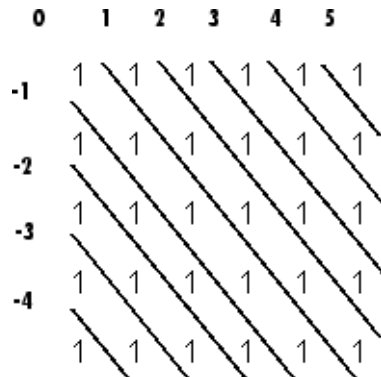
Roughly, A, B, and d are related by

```
for k = 1:p
    B(:,k) = diag(A,d(k))
end
```

Some elements of B, corresponding to positions outside of A, are not defined by these loops. They are not referenced when B is input and are set to zero when B is output.

How the Diagonals of A are Listed in the Vector d

An m -by- n matrix A has $m+n-1$ diagonals. These are specified in the vector d using indices from $-m+1$ to $n-1$. For example, if A is 5-by-6, it has 10 diagonals, which are specified in the vector d using the indices -4, -3, ..., 4, 5. The following diagram illustrates this for a vector of all ones.



Examples**Example 1**

For the following matrix,

```
A=[0 5 0 10 0 0;...
0 0 6 0 11 0;...
3 0 0 7 0 12;...
1 4 0 0 8 0;...
0 2 5 0 0 9]
```

A =

0	5	0	10	0	0
0	0	6	0	11	0
3	0	0	7	0	12
1	4	0	0	8	0
0	2	5	0	0	9

the command

```
[B, d] =spdiags(A)
```

returns

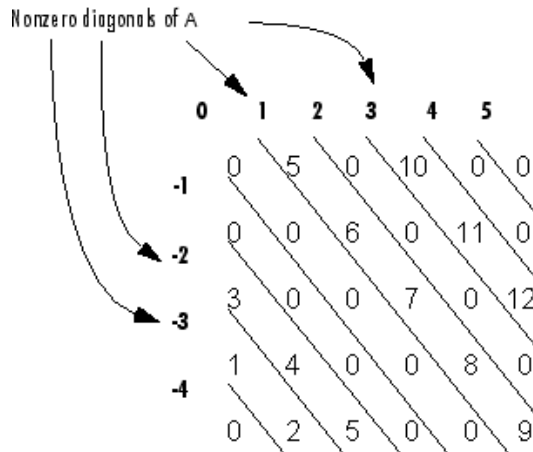
B =

0	0	5	10
0	0	6	11
0	3	7	12
1	4	8	0
2	5	9	0

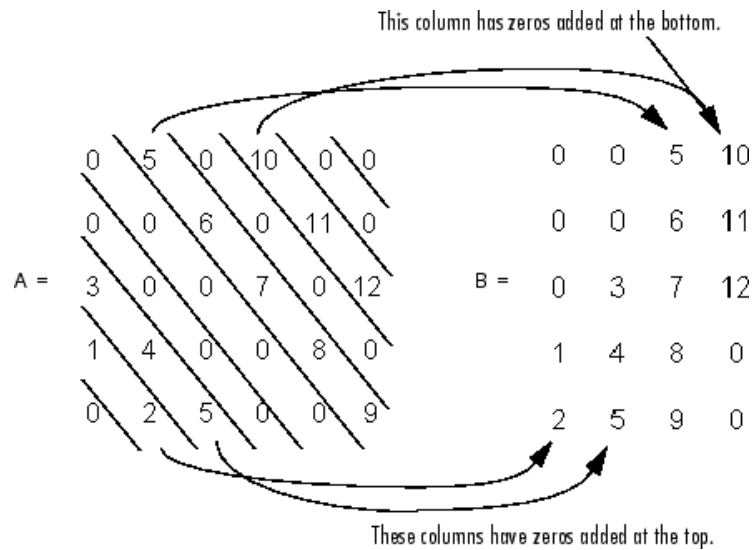
d =

```
-3
-2
1
```

The columns of the first output B contain the nonzero diagonals of A. The second output d lists the indices of the nonzero diagonals of A, as shown in the following diagram. See “How the Diagonals of A are Listed in the Vector d” on page 2-2752.



Note that the longest nonzero diagonal in A is contained in column 3 of B. The other nonzero diagonals of A have extra zeros added to their corresponding columns in B, to give all columns of B the same length. For the nonzero diagonals below the main diagonal of A, extra zeros are added at the tops of columns. For the nonzero diagonals above the main diagonal of A, extra zeros are added at the bottoms of columns. This is illustrated by the following diagram.



Example 2

This example generates a sparse tridiagonal representation of the classic second difference operator on n points.

```
e = ones(n,1);
A = spdiags([e -2*e e], -1:1, n, n)
```

Turn it into Wilkinson's test matrix (see gallery):

```
A = spdiags(abs(-(n-1)/2:(n-1)/2)', 0, A)
```

Finally, recover the three diagonals:

```
B = spdiags(A)
```

Example 3

The second example is not square.

```
A = [11 0 13 0
      0 22 0 24]
```

spdiags

```
0 0 33 0
41 0 0 44
0 52 0 0
0 0 63 0
0 0 0 74]
```

Here $m = 7$, $n = 4$, and $p = 3$.

The statement `[B,d] = spdiags(A)` produces `d = [-3 0 2]'` and

```
B = [41 11 0
52 22 0
63 33 13
74 44 24]
```

Conversely, with the above B and d, the expression `spdiags(B,d,7,4)` reproduces the original A.

Example 4

This example shows how `spdiags` creates the diagonals when the columns of B are longer than the diagonals they are replacing.

```
B = repmat((1:6)', [1 7])
```

```
B =
```

```
1 1 1 1 1 1 1
2 2 2 2 2 2 2
3 3 3 3 3 3 3
4 4 4 4 4 4 4
5 5 5 5 5 5 5
6 6 6 6 6 6 6
```

```
d = [-4 -2 -1 0 3 4 5];
```

```
A = spdiags(B,d,6,6);
```

```
full(A)
```

```
ans =
```

```

1  0  0  4  5  6
1  2  0  0  5  6
1  2  3  0  0  6
0  2  3  4  0  0
1  0  3  4  5  0
0  2  0  4  5  6

```

Example 5A

This example illustrates the use of the syntax `A = spdiags(B,d,m,n)`, under three conditions:

- `m` is equal to `n`
- `m` is greater than `n`
- `m` is less than `n`

The command used in this example is

```
A = full(spdiags(B, [-2 0 2], m, n))
```

where `B` is the 5-by-3 matrix shown below. The resulting matrix `A` has dimensions `m`-by-`n`, and has nonzero diagonals at `[-2 0 2]` (a sub-diagonal at `-2`, the main diagonal, and a super-diagonal at `2`).

```

B =
  1   6  11
  2   7  12
  3   8  13
  4   9  14
  5  10  15

```

The first and third columns of matrix `B` are used to create the sub- and super-diagonals of `A` respectively. In all three cases though, these two outer columns of `B` are longer than the resulting diagonals of `A`. Because of this, only a part of the columns is used in `A`.

When $m == n$ or $m > n$, `spdiags` takes elements of the super-diagonal in A from the lower part of the corresponding column of B, and elements of the sub-diagonal in A from the upper part of the corresponding column of B.

When $m < n$, `spdiags` does the opposite, taking elements of the super-diagonal in A from the upper part of the corresponding column of B, and elements of the sub-diagonal in A from the lower part of the corresponding column of B.

Part 1 – m is equal to n.

```
A = full(spdiags(B, [-2 0 2], 5, 5))
```

Matrix B				Matrix A				
1	6	11		6	0	13	0	0
2	7	12		0	7	0	14	0
3	8	13	== spdiags =>	1	0	8	0	15
4	9	14		0	2	0	9	0
5	10	15		0	0	3	0	10

A(3,1), A(4,2), and A(5,3) are taken from the upper part of B(:,1).

A(1,3), A(2,4), and A(3,5) are taken from the lower part of B(:,3).

Part 2 – m is greater than n.

```
A = full(spdiags(B, [-2 0 2], 5, 4))
```

Matrix B				Matrix A			
1	6	11		6	0	13	0
2	7	12		0	7	0	14
3	8	13	== spdiags =>	1	0	8	0
4	9	14		0	2	0	9
5	10	15		0	0	3	0

Same as in Part A.

Part 3 – m is less than n.

```
A = full(spdiags(B, [-2 0 2], 4, 5))
Matrix B                                Matrix A

1   6   11                                6   0   11   0   0
2   7   12                                0   7   0   12   0
3   8   13 == spdiags => 3   0   8   0   13
4   9   14                                0   4   0   9   0
5   10  15
```

A(3,1) and A(4,2) are taken from the lower part of B(:,1).

A(1,3), A(2,4), and A(3,5) are taken from the upper part of B(:,3).

Example 5B

Extract the diagonals from the first part of this example back into a column format using the command

```
B = spdiags(A)
```

You can see that in each case the original columns are restored (minus those elements that had overflowed the super- and sub-diagonals of matrix A).

Part 1.

```
Matrix A                                Matrix B

6   0   13   0   0                                1   6   0
0   7   0   14   0                                2   7   0
1   0   8   0   15 == spdiags => 3   8   13
0   2   0   9   0                                0   9   14
0   0   3   0   10                                0  10  15
```

Part 2.

```
Matrix A                                Matrix B
```

spdiags

```
6  0  13  0          1  6  0
0  7  0  14         2  7  0
1  0  8  0   == spdiags => 3  8  13
0  2  0  9          0  9  14
0  0  3  0
```

Part 3.

```
          Matrix A          Matrix B
6  0  11  0  0          0  6  11
0  7  0  12  0         0  7  12
3  0  8  0  13   == spdiags => 3  8  13
0  4  0  9  0          4  9  0
```

See Also

[diag](#)

Purpose Calculate specular reflectance

Syntax `R = specular(Nx,Ny,Nz,S,V)`

Description `R = specular(Nx,Ny,Nz,S,V)` returns the reflectance of a surface with normal vector components `[Nx,Ny,Nz]`. `S` and `V` specify the direction to the light source and to the viewer, respectively. You can specify these directions as three vectors `[x,y,z]` or two vectors `[Theta Phi]` (in spherical coordinates).

The specular highlight is strongest when the normal vector is in the direction of $(S+V)/2$ where `S` is the source direction, and `V` is the view direction.

The surface spread exponent can be specified by including a sixth argument as in `specular(Nx,Ny,Nz,S,V,spread)`.

speye

Purpose	Sparse identity matrix
Syntax	<code>S = speye(m,n)</code> <code>S = speye(n)</code>
Description	<code>S = speye(m,n)</code> forms an m-by-n sparse matrix with 1s on the main diagonal. <code>S = speye(n)</code> abbreviates <code>speye(n,n)</code> .
Examples	<code>I = speye(1000)</code> forms the sparse representation of the 1000-by-1000 identity matrix, which requires only about 16 kilobytes of storage. This is the same final result as <code>I = sparse(eye(1000,1000))</code> , but the latter requires eight megabytes for temporary storage for the full representation.
See Also	<code>spalloc</code> , <code>spones</code> , <code>spdiags</code> , <code>sprand</code> , <code>sprandn</code>

Purpose Apply function to nonzero sparse matrix elements

Syntax `f = spfun(fun,S)`

Description The `spfun` function selectively applies a function to only the *nonzero* elements of a sparse matrix `S`, preserving the sparsity pattern of the original matrix (except for underflow or if `fun` returns zero for some nonzero elements of `S`).

`f = spfun(fun,S)` evaluates `fun(S)` on the nonzero elements of `S`. `fun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information.

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

Remarks Functions that operate element-by-element, like those in the `elfun` directory, are the most appropriate functions to use with `spfun`.

Examples Given the 4-by-4 sparse diagonal matrix

```
S = spdiags([1:4] ',0,4,4)
```

```
S =
(1,1)    1
(2,2)    2
(3,3)    3
(4,4)    4
```

Because `fun` returns nonzero values for all nonzero element of `S`, `f = spfun(@exp,S)` has the same sparsity pattern as `S`.

```
f =
(1,1)    2.7183
(2,2)    7.3891
(3,3)   20.0855
(4,4)   54.5982
```

whereas `exp(S)` has 1s where `S` has 0s.

```
full(exp(S))
```

```
ans =
```

```
2.7183    1.0000    1.0000    1.0000
1.0000    7.3891    1.0000    1.0000
1.0000    1.0000   20.0855    1.0000
1.0000    1.0000    1.0000   54.5982
```

See Also

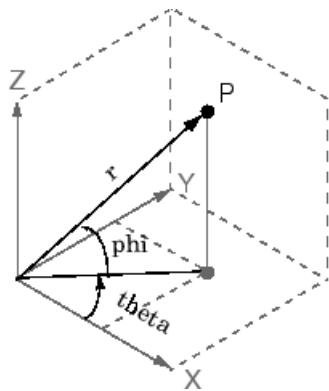
`function_handle` (@)

Purpose Transform spherical coordinates to Cartesian

Syntax `[x,y,z] = sph2cart(THETA,PHI,R)`

Description `[x,y,z] = sph2cart(THETA,PHI,R)` transforms the corresponding elements of spherical coordinate arrays to Cartesian, or xyz , coordinates. THETA, PHI, and R must all be the same size. THETA and PHI are angular displacements in radians from the positive x -axis and from the x - y plane, respectively.

Algorithm The mapping from spherical coordinates to three-dimensional Cartesian coordinates is



$$\begin{aligned}x &= r \cdot \cos(\text{phi}) \cdot \cos(\text{theta}) \\y &= r \cdot \cos(\text{phi}) \cdot \sin(\text{theta}) \\z &= r \cdot \sin(\text{phi})\end{aligned}$$

See Also `cart2pol`, `cart2sph`, `pol2cart`

sphere

Purpose

Generate sphere



Syntax

```
sphere
sphere(n)
[X,Y,Z] = sphere(n)
```

Description

The sphere function generates the x -, y -, and z -coordinates of a unit sphere for use with `surf` and `mesh`.

`sphere` generates a sphere consisting of 20-by-20 faces.

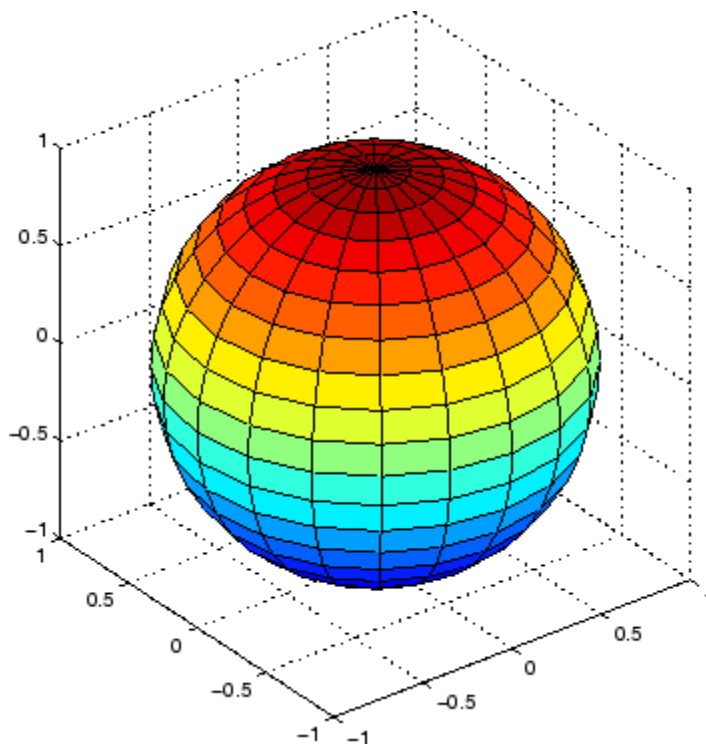
`sphere(n)` draws a `surf` plot of an n -by- n sphere in the current figure.

`[X,Y,Z] = sphere(n)` returns the coordinates of a sphere in three matrices that are $(n+1)$ -by- $(n+1)$ in size. You draw the sphere with `surf(X,Y,Z)` or `mesh(X,Y,Z)`.

Examples

Generate and plot a sphere.

```
sphere
axis equal
```



See Also

cylinder, axis equal

“Polygons and Surfaces” on page 1-86 for related functions

spinmap

Purpose Spin colormap

Syntax spinmap
spinmap(t)
spinmap(t,inc)
spinmap('inf')

Description The spinmap function shifts the colormap RGB values by some incremental value. For example, if the increment equals 1, color 1 becomes color 2, color 2 becomes color 3, etc.

spinmap cyclically rotates the colormap for approximately five seconds using an incremental value of 2.

spinmap(t) rotates the colormap for approximately 10*t seconds. The amount of time specified by t depends on your hardware configuration (e.g., if you are running MATLAB over a network).

spinmap(t,inc) rotates the colormap for approximately 10*t seconds and specifies an increment inc by which the colormap shifts. When inc is 1, the rotation appears smoother than the default (i.e., 2). Increments greater than 2 are less smooth than the default. A negative increment (e.g., -2) rotates the colormap in a negative direction.

spinmap('inf') rotates the colormap for an infinite amount of time. To break the loop, press **Ctrl+C**.

See Also colormap, colormapeditor

“Color Operations” on page 1-95 for related functions

Purpose Cubic spline data interpolation

Syntax
`pp = spline(x,Y)`
`yy = spline(x,Y,xx)`

Description `pp = spline(x,Y)` returns the piecewise polynomial form of the cubic spline interpolant for later use with `ppval` and the spline utility `unmkpp`. `x` must be a vector. `Y` can be a scalar, a vector, or an array of any dimension, subject to the following conditions:

- If `Y` is a scalar or vector, it must have the same length as `x`. A scalar value for `x` or `Y` is expanded to have the same length as the other. See [Exceptions \(1\)](#) for an exception to this rule, in which the not-a-knot end conditions are used.
- If `Y` is an array that is not a vector, the size of `Y` must have the form `[d1,d2,...,dk,n]`, where `n` is the length of `x`. The interpolation is performed for each `d1-by-d2-by-...-dk` value in `Y`. See [Exceptions \(2\)](#) for an exception to this rule.

`yy = spline(x,Y,xx)` is the same as `yy = ppval(spline(x,Y),xx)`, thus providing, in `yy`, the values of the interpolant at `xx`. `xx` can be a scalar, a vector, or a multidimensional array. The sizes of `xx` and `yy` are related as follows:

- If `Y` is a scalar or vector, `yy` has the same size as `xx`.
- If `Y` is an array that is not a vector,
 - If `xx` is a scalar or vector, `size(yy)` equals `[d1, d2, ..., dk, length(xx)]`.
 - If `xx` is an array of size `[m1,m2,...,mj]`, `size(yy)` equals `[d1,d2,...,dk,m1,m2,...,mj]`.

Exceptions

- 1 If Y is a vector that contains two more values than x has entries, the first and last value in Y are used as the endslopes for the cubic spline. If Y is a vector, this means
 - $f(x) = Y(2:\text{end}-1)$
 - $df(\min(x)) = Y(1)$
 - $df(\max(x)) = Y(\text{end})$
- 2 If Y is a matrix or an N -dimensional array with $\text{size}(Y,N)$ equal to $\text{length}(x)+2$, the following hold:
 - $f(x(j))$ matches the value $Y(:, \dots, :, j+1)$ for $j=1:\text{length}(x)$
 - $Df(\min(x))$ matches $Y(:, :, \dots, 1)$
 - $Df(\max(x))$ matches $Y(:, :, \dots, \text{end})$

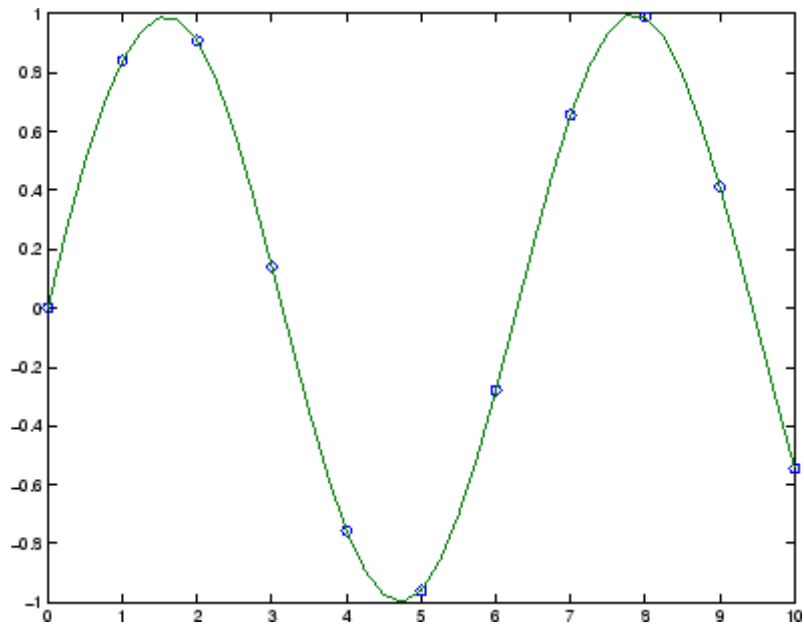
Note You can also perform spline interpolation using the `interp1` function with the command `interp1(x,y,xx,'spline')`. Note that while `spline` performs interpolation on rows of an input matrix, `interp1` performs interpolation on columns of an input matrix.

Examples

Example 1

This generates a sine curve, then samples the spline over a finer mesh.

```
x = 0:10;  
y = sin(x);  
xx = 0:.25:10;  
yy = spline(x,y,xx);  
plot(x,y,'o',xx,yy)
```

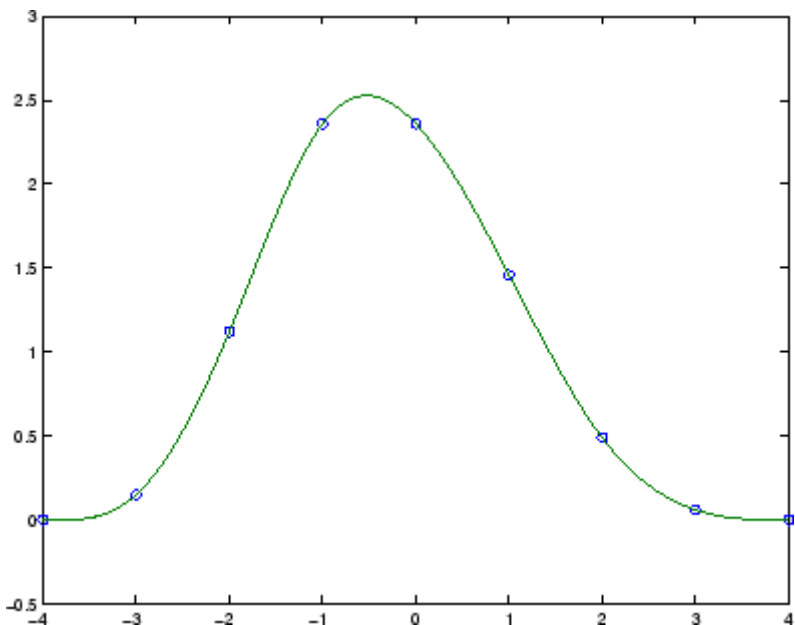


Example 2

This illustrates the use of clamped or complete spline interpolation where end slopes are prescribed. Zero slopes at the ends of an interpolant to the values of a certain distribution are enforced.

```
x = -4:4;  
y = [0 .15 1.12 2.36 2.36 1.46 .49 .06 0];  
cs = spline(x,[0 y 0]);  
xx = linspace(-4,4,101);  
plot(x,y,'o',xx,ppval(cs,xx),'-');
```

spline



Example 3

The two vectors

```
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 ];
```

represent the census years from 1900 to 1990 and the corresponding United States population in millions of people. The expression

```
spline(t,p,2000)
```

uses the cubic spline to extrapolate and predict the population in the year 2000. The result is

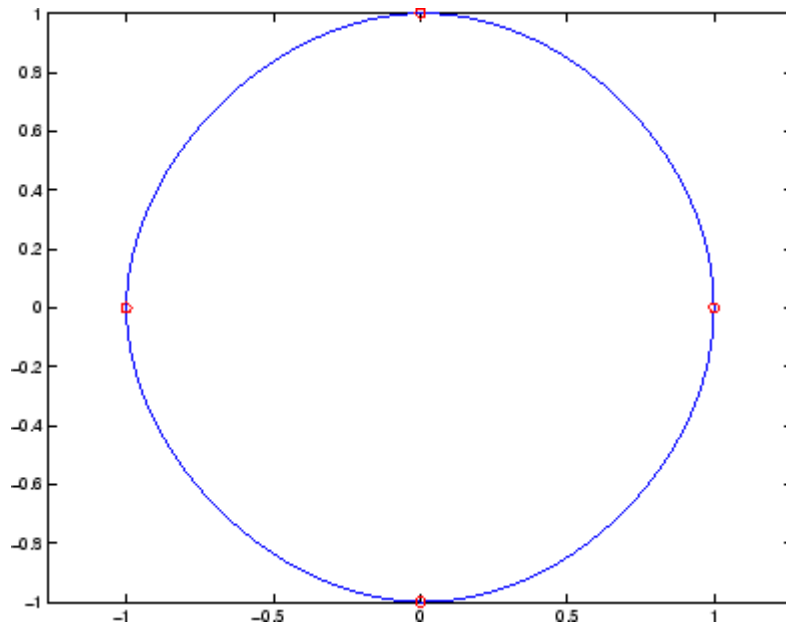
```
ans =  
    270.6060
```

Example 4

The statements

```
x = pi*[0:.5:2];
y = [0 1 0 -1 0 1 0;
     1 0 1 0 -1 0 1];
pp = spline(x,y);
yy = ppval(pp, linspace(0,2*pi,101));
plot(yy(1,:),yy(2:,:),'-b',y(1,2:5),y(2,2:5),'or'), axis equal
```

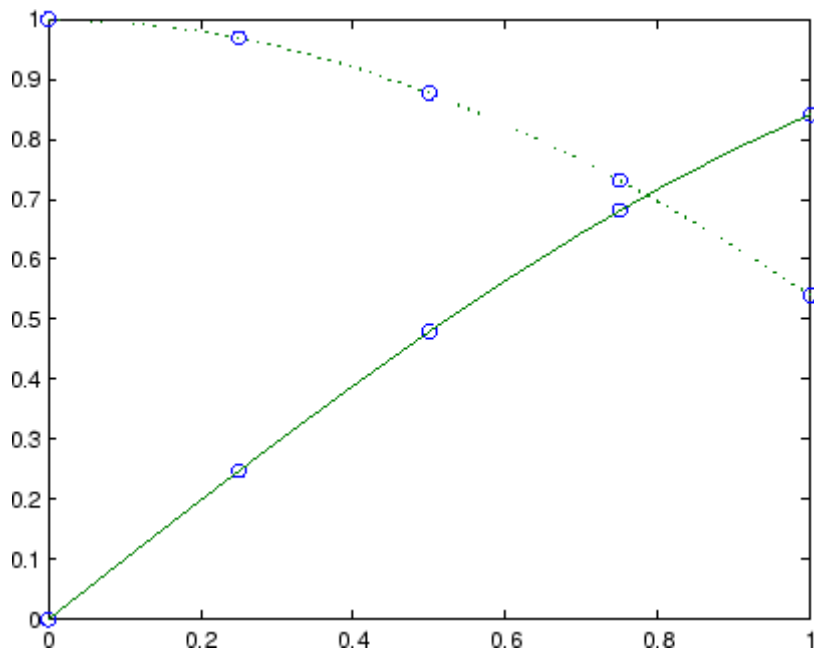
generate the plot of a circle, with the five data points $y(:,2), \dots, y(:,6)$ marked with o's. Note that this y contains two more values (i.e., two more columns) than does x , hence $y(:,1)$ and $y(:,end)$ are used as endslopes.



Example 5

The following code generates sine and cosine curves, then samples the splines over a finer mesh.

```
x = 0:.25:1;  
Y = [sin(x); cos(x)];  
xx = 0:.1:1;  
YY = spline(x,Y,xx);  
plot(x,Y(1,:), 'o',xx,YY(1,:), '-'); hold on;  
plot(x,Y(2,:), 'o',xx,YY(2,:), ':'); hold off;
```



Algorithm

A tridiagonal linear system (with, possibly, several right sides) is being solved for the information needed to describe the coefficients of the various cubic polynomials which make up the interpolating spline. spline uses the functions ppval, mkpp, and unmkpp. These routines

form a small suite of functions for working with piecewise polynomials. For access to more advanced features, see the M-file help for these functions and the Spline Toolbox.

See Also

`interp1`, `ppval`, `mkpp`, `pchip`, `unmkpp`

References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

spones

Purpose

Replace nonzero sparse matrix elements with ones

Syntax

`R = spones(S)`

Description

`R = spones(S)` generates a matrix `R` with the same sparsity structure as `S`, but with 1's in the nonzero positions.

Examples

`c = sum(spones(S))` is the number of nonzeros in each column.

`r = sum(spones(S'))'` is the number of nonzeros in each row.

`sum(c)` and `sum(r)` are equal, and are equal to `nnz(S)`.

See Also

`nnz`, `spalloc`, `spfun`

Purpose Set parameters for sparse matrix routines

Syntax

```
spparms('key',value)
spparms
values = spparms
[keys,values] = spparms
spparms(values)
value = spparms('key')
spparms('default')
spparms('tight')
```

Description spparms('key',value) sets one or more of the *tunable* parameters used in the sparse routines, particularly the minimum degree orderings, colmmd and symmmd, and also within sparse backslash. In ordinary use, you should never need to deal with this function.

The meanings of the key parameters are

'spumoni'	Sparse Monitor flag:
0	Produces no diagnostic output, the default
1	Produces information about choice of algorithm based on matrix structure, and about storage allocation
2	Also produces very detailed information about the sparse matrix algorithms
'thr_rel', 'thr_abs'	Minimum degree threshold is thr_rel*mindegree + thr_abs.
'exact_d'	Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.
'supernd'	If positive, minimum degree amalgamates the supernodes every supernd stages.

spparms

'rreduce'	If positive, minimum degree does row reduction every rreduce stages.
'wh_frac'	Rows with density > wh_frac are ignored in colmmd.
'autommd'	Nonzero to use minimum degree (MMD) orderings with QR-based \ and /.
'autoamd'	Nonzero to use colamd ordering with the UMFPACK LU-based \ and /, and to use amd with CHOLMOD Cholesky-based \ and /.
'piv_tol'	Pivot tolerance used by the UMFPACK LU-based \ and /.
'bandden'	Band density used by LAPACK-based \ and / for banded matrices. Band density is defined as (# nonzeros in the band)/(# nonzeros in a full band). If bandden = 1.0, never use band solver. If bandden = 0.0, always use band solver. Default is 0.5.
'umfpack'	Nonzero to use UMFPACK instead of the v4 LU-based solver in \ and /.

Note LU-based \ and / (UMFPACK) on square matrices use a modified colamd or amd. Cholesky-based \ and / (CHOLMOD) on symmetric positive definite matrices use amd. QR-based \ and / on rectangular matrices use colmmd.

spparms, by itself, prints a description of the current settings.

values = spparms returns a vector whose components give the current settings.

[keys, values] = spparms returns that vector, and also returns a character matrix whose rows are the keywords for the parameters.

spparms(values), with no output argument, sets all the parameters to the values specified by the argument vector.

value = spparms('key') returns the current setting of one parameter.

spparms('default') sets all the parameters to their default settings.

spparms('tight') sets the minimum degree ordering parameters to their *tight* settings, which can lead to orderings with less fill-in, but which make the ordering functions themselves use more execution time.

The key parameters for default and tight settings are

	Keyword	Default	Tight
values(1)	'spumoni'	0.0	
values(2)	'thr_rel'	1.1	1.0
values(3)	'thr_abs'	1.0	0.0
values(4)	'exact_d'	0.0	1.0
values(5)	'supernd'	3.0	1.0
values(6)	'rreduce'	3.0	1.0
values(7)	'wh_frac'	0.5	0.5
values(8)	'autommd'	1.0	
values(9)	'autoamd'	1.0	
values(10)	'piv_tol'	0.1	
values(11)	'bandden'	0.5	
values(12)	'umfpack'	1.0	

Notes

Sparse $A \setminus b$ on Symmetric Positive Definite A

Sparse $A \setminus b$ on symmetric positive definite A uses CHOLMOD in conjunction with the amd reordering routine.

The parameter 'autoamd' turns the amd reordering on or off within the solver.

Sparse $A \setminus b$ on General Square A

Sparse $A \setminus b$ on general square A usually uses UMFPACK in conjunction with amd or a modified colamd reordering routine.

The parameter 'umfpack' turns the use of the UMFPACK software on or off within the solver.

If UMFPACK is used,

- The parameter 'piv_tol' controls pivoting within the solver.
- The parameter 'autoamd' turns amd and the modified colamd on or off within the solver.

If UMFPACK is not used,

- An LU-based solver is used in conjunction with the colmmd reordering routine.
- If UMFPACK is not used, then the parameter 'autommd' turns the colmmd reordering routine on or off within the solver.
- If UMFPACK is not used and colmmd is used within the solver, then the minimum degree parameters affect the reordering routine within the solver.

Sparse $A \setminus b$ on Rectangular A

Sparse $A \setminus b$ on rectangular A uses a QR-based solve in conjunction with the colmmd reordering routine.

The parameter 'autommd' turns the colmmd reordering on or off within the solver.

If colmmd is used within the solver, then the minimum degree parameters affect the reordering routine within the solver.

See Also

\, chol, lu, qr, colamd, colmmd, symmmd

References

- [1] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis and Applications*, Vol. 13, 1992, pp. 333-356.
- [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.
- [3] Davis, T. A., *CHOLMOD Version 1.0 User Guide* (<http://www.cise.ufl.edu/research/sparse/cholmod/>), Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, 2005.

sprand

Purpose Sparse uniformly distributed random matrix

Syntax
R = sprand(S)
R = sprand(m,n,density)
R = sprand(m,n,density,rc)

Description R = sprand(S) has the same sparsity structure as S, but uniformly distributed random entries.

R = sprand(m,n,density) is a random, m-by-n, sparse matrix with approximately $\text{density} \times m \times n$ uniformly distributed nonzero entries ($0 \leq \text{density} \leq 1$).

R = sprand(m,n,density,rc) also has reciprocal condition number approximately equal to rc. R is constructed from a sum of matrices of rank one.

If rc is a vector of length lr, where $lr \leq \min(m,n)$, then R has rc as its first lr singular values, all others are zero. In this case, R is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

sprand uses the internal state information set with the rand function.

See Also sprandn, sprandsym

Purpose	Sparse normally distributed random matrix
Syntax	<pre>R = sprandn(S) R = sprandn(m,n,density) R = sprandn(m,n,density,rc)</pre>
Description	<p><code>R = sprandn(S)</code> has the same sparsity structure as <code>S</code>, but normally distributed random entries with mean 0 and variance 1.</p> <p><code>R = sprandn(m,n,density)</code> is a random, <code>m</code>-by-<code>n</code>, sparse matrix with approximately <code>density*m*n</code> normally distributed nonzero entries ($0 \leq \text{density} \leq 1$).</p> <p><code>R = sprandn(m,n,density,rc)</code> also has reciprocal condition number approximately equal to <code>rc</code>. <code>R</code> is constructed from a sum of matrices of rank one.</p> <p>If <code>rc</code> is a vector of length <code>lr</code>, where $lr \leq \min(m,n)$, then <code>R</code> has <code>rc</code> as its first <code>lr</code> singular values, all others are zero. In this case, <code>R</code> is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.</p> <p><code>sprandn</code> uses the internal state information set with the <code>randn</code> function.</p>
See Also	<code>sprand</code> , <code>sprandsym</code>

sprandsym

Purpose Sparse symmetric random matrix

Syntax

```
R = sprandsym(S)
R = sprandsym(n,density)
R = sprandsym(n,density,rc)
R = sprandsym(n,density,rc,kind)
```

Description

`R = sprandsym(S)` returns a symmetric random matrix whose lower triangle and diagonal have the same structure as `S`. Its elements are normally distributed, with mean 0 and variance 1.

`R = sprandsym(n,density)` returns a symmetric random, `n`-by-`n`, sparse matrix with approximately `density*n*n` nonzeros; each entry is the sum of one or more normally distributed random samples, and (`0 <= density <= 1`).

`R = sprandsym(n,density,rc)` returns a matrix with a reciprocal condition number equal to `rc`. The distribution of entries is nonuniform; it is roughly symmetric about 0; all are in `[-1, 1]`.

If `rc` is a vector of length `n`, then `R` has eigenvalues `rc`. Thus, if `rc` is a positive (nonnegative) vector then `R` is a positive definite matrix. In either case, `R` is generated by random Jacobi rotations applied to a diagonal matrix with the given eigenvalues or condition number. It has a great deal of topological and algebraic structure.

`R = sprandsym(n,density,rc,kind)` returns a positive definite matrix. Argument `kind` can be:

- 1 to generate `R` by random Jacobi rotation of a positive definite diagonal matrix. `R` has the desired condition number exactly.
- 2 to generate an `R` that is a shifted sum of outer products. `R` has the desired condition number only approximately, but has less structure.
- 3 to generate an `R` that has the same structure as the matrix `S` and approximate condition number `1/rc`. `density` is ignored.

See Also sprand, sprandn

Purpose Structural rank

Syntax `r = sprank(A)`

Description `r = sprank(A)` is the structural rank of the sparse matrix `A`. Also known as maximum traversal, maximum assignment, and size of a maximum matching in the bipartite graph of `A`.

Always `sprank(A) >= rank(full(A))`, and in exact arithmetic `sprank(A) == rank(full(sprandn(A)))` with probability one.

Examples

```
A = [1 0 2 0
      2 0 4 0];
```

```
A = sparse(A);
```

```
sprank(A)
```

```
ans =
      2
```

```
rank(full(A))
```

```
ans =
      1
```

See Also `dmperm`

sprintf

Purpose Write formatted data to string

Syntax `[s, errmsg] = sprintf(format, A, ...)`

Description `[s, errmsg] = sprintf(format, A, ...)` formats the data in matrix A (and in any additional matrix arguments) under control of the specified format string and returns it in the MATLAB string variable s. The `sprintf` function returns an error message string `errmsg` if an error occurred. `errmsg` is an empty matrix if no error occurred.

`sprintf` is the same as `fprintf` except that it returns the data in a MATLAB string variable rather than writing it to a file.

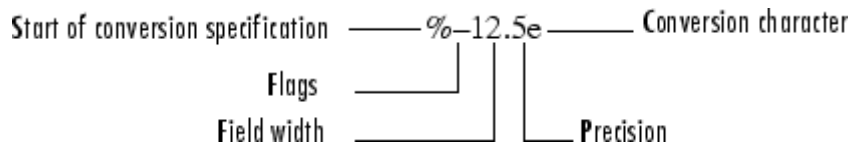
Format String

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
A minus sign (-)	Left-justifies the converted argument in its field	% 5.2d
A plus sign (+)	Always prints a sign character (+ or -)	%+5.2d
Zero (0)	Pad with zeros rather than spaces.	%05.2f

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
%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)

The following tables describe the nonalphanumeric characters found in format specification strings.

Escape Characters

This table lists the escape character sequences you use to specify non-printing characters in a format specification.

Character	Description
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return

Character	Description
\t	Horizontal tab
\\	Backslash
\”r” (two single quotes)	Single quotation mark
%%	Percent character

Remarks

The `sprintf` function behaves like its ANSI C language namesake with these exceptions and extensions.

- If you use `sprintf` to convert a MATLAB double into an integer, and the double contains a value that cannot be represented as an integer (for example, it contains a fraction), MATLAB ignores the specified conversion and outputs the value in exponential format. To successfully perform this conversion, use the `fix`, `floor`, `ceil`, or `round` functions to change the value in the double into a value that can be represented as an integer before passing it to `sprintf`.
- The following nonstandard subtype specifiers are supported for the conversion characters `%o`, `%u`, `%x`, and `%X`.

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 <code>'%bx'</code> .
t	The underlying C data type is a float rather than an unsigned integer.

For example, to print a double value in hexadecimal use the format `'%bx'`.

- The `sprintf` function is vectorized for nonscalar arguments. The function recycles the format string through the elements of `A` (columnwise) until all the elements are used up. The function

sprintf

then continues in a similar manner through any additional matrix arguments.

- If %s is used to print part of a nonscalar double argument, the following behavior occurs:
 - Successive values are printed as long as they are integers and in the range of a valid character. The first invalid character terminates the printing for this %s specifier and is used for a later specifier. For example, pi terminates the string below and is printed using %f format.

```
Str = [65 66 67 pi];
sprintf('%s %f', Str)
ans =
ABC 3.141593
```

- If the first value to print is not a valid character, then just that value is printed for this %s specifier using an e conversion as a warning to the user. For example, pi is formatted by %s below in exponential notation, and 65, though representing a valid character, is formatted as fixed-point (%f).

```
Str = [pi 65 66 67];
sprintf('%s %f %s', Str)
ans =
3.141593e+000 65.000000 BC
```

- One exception is zero, which is a valid character. If zero is found first, %s prints nothing and the value is skipped. If zero is found after at least one valid character, it terminates the printing for this %s specifier and is used for a later specifier.
- sprintf prints negative zero and exponents differently on some platforms, as shown in the following tables.

Negative Zero Printed with %e, %E, %f, %g, or %G

	Display of Negative Zero		
Platform	%e or %E	%f	%g or %G
PC	0.000000e+000	0.000000	0
Others	-0.000000e+00	-0.000000	-0

Exponents Printed with %e, %E, %g, or %G

Platform	Minimum Digits in Exponent	Example
PC	3	1.23e+004
UNIX	2	1.23e+04

You can resolve this difference in exponents by postprocessing the results of sprintf. For example, to make the PC output look like that of UNIX, use

```
a = sprintf('%e', 12345.678);
if ispc, a = strrep(a, 'e+0', 'e+'); end
```

Examples

Command	Result
<code>sprintf('%0.5g', (1+sqrt(5))/2)</code>	1.618
<code>sprintf('%0.5g', 1/eps)</code>	4.5036e+15
<code>sprintf('%15.5f', 1/eps)</code>	4503599627370496.00000
<code>sprintf('%d', round(pi))</code>	3
<code>sprintf('%s', 'hello')</code>	hello

sprintf

Command	Result
<code>sprintf('The array is %dx%d.',2,3)</code>	The array is 2x3
<code>sprintf('\n')</code>	Line termination character on all platforms

See Also

`int2str`, `num2str`, `sscanf`

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 Visualize sparsity pattern

Syntax

```
spy(S)
spy(S,markersize)
spy(S,'LineStyle')
spy(S,'LineStyle',markersize)
```

Description plots the `spy(S)` sparsity pattern of any matrix `S`.

`spy(S,markersize)`, where `markersize` is an integer, plots the sparsity pattern using markers of the specified point size.

`spy(S,'LineStyle')`, where `LineStyle` is a string, uses the specified plot marker type and color.

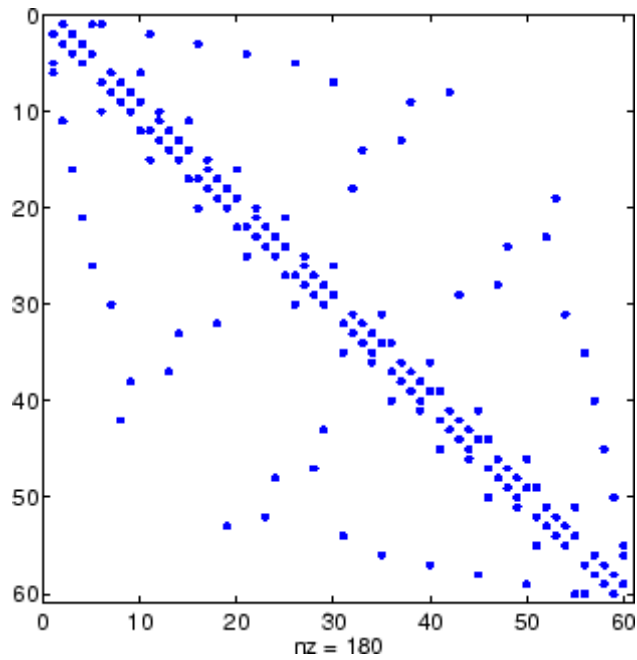
`spy(S,'LineStyle',markersize)` uses the specified type, color, and size for the plot markers.

`S` is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.

Note `spy` replaces `format +`, which takes much more space to display essentially the same information.

Examples This example plots the 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster Fuller geodesic dome. This matrix also represents the soccer ball and the carbon-60 molecule.

```
B = bucky;
spy(B)
```



See Also

find, gplot, LineSpec, symamd, symrcm

Purpose Square root

Syntax `B = sqrt(X)`

Description `B = sqrt(X)` returns the square root of each element of the array `X`. For the elements of `X` that are negative or complex, `sqrt(X)` produces complex results.

Remarks See `sqrtm` for the matrix square root.

Examples

```
sqrt((-2:2)')
ans =
    0 + 1.4142i
    0 + 1.0000i
    0
    1.0000
    1.4142
```

See Also `sqrtm`, `realsqrt`

sqrtm

Purpose Matrix square root

Syntax
 $X = \text{sqrtm}(A)$
 $[X, \text{resnorm}] = \text{sqrtm}(A)$
 $[X, \alpha, \text{condest}] = \text{sqrtm}(A)$

Description $X = \text{sqrtm}(A)$ is the principal square root of the matrix A , i.e. $X^2 = A$.

X is the unique square root for which every eigenvalue has nonnegative real part. If A has any eigenvalues with negative real parts then a complex result is produced. If A is singular then A may not have a square root. A warning is printed if exact singularity is detected.

$[X, \text{resnorm}] = \text{sqrtm}(A)$ does not print any warning, and returns the residual, $\text{norm}(A - X^2, 'fro') / \text{norm}(A, 'fro')$.

$[X, \alpha, \text{condest}] = \text{sqrtm}(A)$ returns a stability factor α and an estimate condest of the matrix square root condition number of X . The residual $\text{norm}(A - X^2, 'fro') / \text{norm}(A, 'fro')$ is bounded approximately by $n * \alpha * \text{eps}$ and the Frobenius norm relative error in X is bounded approximately by $n * \alpha * \text{condest} * \text{eps}$, where $n = \max(\text{size}(A))$.

Remarks If X is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.

Some matrices, like $X = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$, do not have any square roots, real or complex, and `sqrtm` cannot be expected to produce one.

Examples **Example 1**

A matrix representation of the fourth difference operator is

$$X = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ -4 & 6 & -4 & 1 & 0 \\ 1 & -4 & 6 & -4 & 1 \\ 0 & 1 & -4 & 6 & -4 \\ 0 & 0 & 1 & -4 & 5 \end{bmatrix}$$

This matrix is symmetric and positive definite. Its unique positive definite square root, $Y = \text{sqrtm}(X)$, is a representation of the second difference operator.

$$Y = \begin{bmatrix} 2 & -1 & -0 & -0 & -0 \\ -1 & 2 & -1 & 0 & -0 \\ 0 & -1 & 2 & -1 & 0 \\ -0 & 0 & -1 & 2 & -1 \\ -0 & -0 & -0 & -1 & 2 \end{bmatrix}$$

Example 2

The matrix

$$X = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

has four square roots. Two of them are

$$Y1 = \begin{bmatrix} 1.5667 & 1.7408 \\ 2.6112 & 4.1779 \end{bmatrix}$$

and

$$Y2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

The other two are $-Y1$ and $-Y2$. All four can be obtained from the eigenvalues and vectors of X .

$$[V,D] = \text{eig}(X);$$

$$D = \begin{bmatrix} 0.1386 & 0 \\ 0 & 28.8614 \end{bmatrix}$$

sqrtm

The four square roots of the diagonal matrix D result from the four choices of sign in

$$S = \begin{pmatrix} -0.3723 & 0 \\ 0 & -5.3723 \end{pmatrix}$$

All four Ys are of the form

$$Y = V * S / V$$

The sqrtm function chooses the two plus signs and produces Y1, even though Y2 is more natural because its entries are integers.

See Also

expm, funm, logm

Purpose Remove singleton dimensions

Syntax `B = squeeze(A)`

Description `B = squeeze(A)` returns an array `B` with the same elements as `A`, but with all singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`. Two-dimensional arrays are unaffected by `squeeze`; if `A` is a row or column vector or a scalar (1-by-1) value, then `B = A`.

Examples Consider the 2-by-1-by-3 array `Y = rand(2,1,3)`. This array has a singleton column dimension — that is, there's only one column per page.

`Y =`

<code>Y(:, :, 1) =</code>	<code>Y(:, :, 2) =</code>
0.5194	0.0346
0.8310	0.0535

<code>Y(:, :, 3) =</code>
0.5297
0.6711

The command `Z = squeeze(Y)` yields a 2-by-3 matrix:

<code>Z =</code>			
0.5194	0.0346	0.5297	
0.8310	0.0535	0.6711	

Consider the 1-by-1-by-5 array `mat= repmat(1,[1,1,5])`. This array has only one scalar value per page.

`mat =`

<code>mat(:, :, 1) =</code>	<code>mat(:, :, 2) =</code>
1	1

squeeze

```
mat(:,:,3) =   mat(:,:,4) =
```

```
    1    1
```

```
mat(:,:,5) =
```

```
    1
```

The command `squeeze(mat)` yields a 5-by-1 matrix:

```
squeeze(mat)
```

```
ans =
```

```
    1  
    1  
    1  
    1  
    1
```

```
size(squeeze(mat))
```

```
ans =
```

```
    5    1
```

See Also

`reshape`, `shiftdim`

Purpose Convert state-space filter parameters to transfer function form

Syntax `[b,a] = ss2tf(A,B,C,D,iu)`

Description `ss2tf` converts a state-space representation of a given system to an equivalent transfer function representation.

`[b,a] = ss2tf(A,B,C,D,iu)` returns the transfer function

$$H(s) = \frac{B(s)}{A(s)} = C(sI - A)^{-1}B + D$$

of the system

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

from the `iu`-th input. Vector `a` contains the coefficients of the denominator in descending powers of s . The numerator coefficients are returned in array `b` with as many rows as there are outputs y . `ss2tf` also works with systems in discrete time, in which case it returns the z -transform representation.

The `ss2tf` function is part of the standard MATLAB language.

Algorithm The `ss2tf` function uses `poly` to find the characteristic polynomial $\det(sI - A)$ and the equality:

$$H(s) = C(sI - A)^{-1}B = \frac{\det(sI - A + BC) - \det(sI - A)}{\det(sI - A)}$$

sscanf

Purpose Read formatted data from string

Syntax
`A = sscanf(s, format)`
`A = sscanf(s, format, size)`
`[A, count, errmsg, nextindex] = sscanf(...)`

Description `A = sscanf(s, format)` reads data from the MATLAB string `s`, converts it according to the specified format string, and returns it in matrix `A`. `format` is a string specifying the format of the data to be read. See Remarks for details. `sscanf` is the same as `fscanf` except that it reads the data from a MATLAB string rather than reading it from a file. If `s` is a character array with more than one row, `sscanf` reads the characters in column order.

`A = sscanf(s, format, size)` reads the amount of data specified by `size` and converts it according to the specified format string. `size` is an argument that determines how much data is read. Valid options are

<code>n</code>	Read at most <code>n</code> numbers, characters, or strings.
<code>inf</code>	Read to the end of the input string.
<code>[m,n]</code>	Read at most <code>(m*n)</code> 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 input string and on the `size` argument. If `sscanf` reads only numbers, and if `size` is not of the form `[m,n]`, matrix `A` is a column vector of numbers. If `sscanf` 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.

`sscanf` differs from its C language namesake `scanf()` in an important respect — it is *vectorized* to return a matrix argument. The format string is cycled through the input string until the first of these conditions occurs:

- The format string fails to match the data in the input string

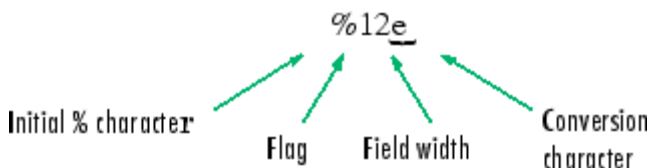
- The amount of data specified by size is read
- The end of the string is reached

[A, count, errmsg, nextindex] = sscanf(...) reads data from the MATLAB string (character array) s, converts it according to the specified format string, and returns it in matrix A. count is an optional output argument that returns the number of values successfully read. errmsg is an optional output argument that returns an error message string if an error occurred or an empty string if an error did not occur. nextindex is an optional output argument specifying one more than the number of characters scanned in s.

Remarks

When MATLAB reads a specified string, it attempts to match the data in the input string to the format string. If a match occurs, the data is written into the output matrix. If a partial match occurs, only the matching data is written to the matrix, and the read operation stops.

The format string consists of ordinary characters and/or conversion specifications. Conversion specifications indicate the type of data to be matched and involve the character %, optional width fields, and conversion characters, organized as shown below:



Add one or more of these characters between the % and the conversion character.

An asterisk (*)	Skip over the matched value and do not store it in the output matrix
A digit string	Maximum field width
A letter	The size of the receiving object; for example, h for short, as in %hd for a short integer, or l for long, as in %ld for a long integer or %lg for a double floating-point number

Valid conversion characters are as shown.

%c	Sequence of characters; number specified by field width
%d	Base 10 integers
%e, %f, %g	Floating-point numbers
%i	Defaults to signed base 10 integers. Data starting with 0 is read as base 8. Data starting with 0x or 0X is read as base 16.
%o	Signed octal integer returned as unsigned
%s	A series of non-white-space characters
%u	Signed decimal integer
%x	Signed hexadecimal integer returned as unsigned
[...]	Sequence of characters (scanlist)

Format specifiers %e, %f, and %g accept the text 'inf', '-inf', 'nan', and '-nan'. This text is not case sensitive. The sscanf function converts these to the numeric representation of Inf, -Inf, NaN, and -NaN.

Use %c to read space characters, or %s to skip all white space.

For more information about format strings, refer to the scanf() and fscanf() routines in a C language reference manual.

Output Characteristics: Only Numeric Values Read

Format characters that cause `sscanf` to read numbers from the input string are `%d`, `%e`, `%f`, `%g`, `%i`, `%o`, `%u`, and `%x`. When `sscanf` reads only numbers from the input string, the elements of the output matrix `A` are numbers.

When there is no `size` argument or the `size` argument is `inf`, `sscanf` reads to the end of the input string. The output matrix is a column vector with one element for each number read from the input.

When the `size` argument is a scalar `n`, `sscanf` reads at most `n` numbers from the input string. The output matrix is a column vector with one element for each number read from the input.

When the `size` argument is a matrix `[m,n]`, `sscanf` reads at most $(m*n)$ numbers from the input string. The output matrix contains at most `m` rows and `n` columns. `sscanf` fills the output matrix in column order, using as many columns as it needs to contain all the numbers read from the input. Any unfilled elements in the final column contain zeros.

Output Characteristics: Only Character Values Read

The format characters that cause `sscanf` to read characters and strings from the input string are `%c` and `%s`. When `sscanf` reads only characters and strings from the input string, the elements of the output matrix `A` are characters. When `sscanf` reads a string from the input, the output matrix includes one element for each character in the string.

When there is no `size` argument or the `size` argument is `inf`, `sscanf` reads to the end of the input string. The output matrix is a row vector with one element for each character read from the input.

When the `size` argument is a scalar `n`, `sscanf` reads at most `n` character or string values from the input string. The output matrix is a row vector with one element for each character read from the input. When string values are read from the input, the output matrix can contain more than `n` columns.

When the `size` argument is a matrix `[m,n]`, `sscanf` reads at most $(m*n)$ character or string values from the input string. The output

matrix contains at most m rows. `sscanf` fills the output matrix in column order, using as many columns as it needs to contain all the characters read from the input. When string values are read from the input, the output matrix can contain more than n columns. Any unfilled elements in the final column contain `char(0)`.

Output Characteristics: Both Numeric and Character Values Read

When `sscanf` reads a combination of numbers and either characters or strings from the input string, the elements of the output matrix A are numbers. When `sscanf` reads a string from the input, the output matrix includes one element for each character in the string. All characters are converted to their numeric equivalents in the output matrix.

When there is no `size` argument or the `size` argument is `inf`, `sscanf` reads to the end of the input string. The output matrix is a column vector with one element for each character read from the input.

When the `size` argument is a scalar n , `sscanf` reads at most n number, character, or string values from the input string. The output matrix contains at most n rows. `sscanf` fills the output matrix in column order, using as many columns as it needs to represent all the numbers and characters read from the input. When string values are read from the input, the output matrix can contain more than one column. Any unfilled elements in the final column contain zeros.

When the `size` argument is a matrix $[m,n]$, `sscanf` reads at most $(m*n)$ number, character, or string values from the input string. The output matrix contains at most m rows. `sscanf` fills the output matrix in column order, using as many columns as it needs to represent all the numbers and characters read from the input. When string values are read from the input, the output matrix can contain more than n columns. Any unfilled elements in the final column contain zeros.

Note This section applies only when `sscanf` actually reads a combination of numbers and either characters or strings from the input string. Even if the format string has both format characters that would result in numbers (such as `%d`) and format characters that would result in characters or strings (such as `%s`), `sscanf` might actually read only numbers or only characters or strings. If `sscanf` reads only numbers, see “Output Characteristics: Only Numeric Values Read” on page 2-2805. If `sscanf` reads only characters or strings, see “Output Characteristics: Only Character Values Read” on page 2-2805.

Examples

Example 1

The statements

```
s = '2.7183 3.1416';
A = sscanf(s, '%f')
```

create a two-element vector containing poor approximations to `e` and `pi`.

Example 2

When using the `%i` conversion specifier, `sscanf` reads data starting with 0 as base 8 and returns the converted value as signed:

```
sscanf('-010', '%i')
ans =
    -8
```

When using `%o`, on the other hand, `sscanf` returns the converted value as unsigned:

```
sscanf('-010', '%o')
ans =
    4.2950e+009
```

Example 3

Create character array `A` representing both character and numeric data:

sscanf

```
A = ['abc 46 6 ghi'; 'def 7 89 jkl']
A =
    abc 46 6 ghi
    def 7 89 jkl
```

Read A into 2-by-N matrix B, ignoring the character data. As stated in the Description section, sscanf reads the characters in A in column order, filling matrix B in column order:

```
B = sscanf(A, '%*s %d %d %*s', [2, inf])
B =
    476
    869
```

If you want sscanf to return the numeric data in B in the same order as in A, you can use this technique:

```
for k = 1:2
    C(k,:) = sscanf(A(k, :)', '%*s %d %d %*s', [1, inf]);
end

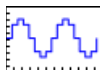
C
C =
    46     6
     7    89
```


See Also

eval, sprintf, textread

Purpose

Stairstep graph

**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

```
stairs(Y)
stairs(X,Y)
stairs(...,LineStyle)
stairs(...,'PropertyName',propertyvalue)
stairs(axes_handles,...)
h = stairs(...)
[xb,yb] = stairs(Y,...)
hlines = stairs('v6',...)
```

Description

Stairstep graphs are useful for drawing time-history graphs of digitally sampled data.

`stairs(Y)` draws a stairstep graph of the elements of `Y`, drawing one line per column for matrices. The axes `ColorOrder` property determines the color of the lines.

When `Y` is a vector, the x -axis scale ranges from 1 to `length(Y)`. When `Y` is a matrix, the x -axis scale ranges from 1 to the number of rows in `Y`.

`stairs(X,Y)` plots the elements in `Y` at the locations specified in `X`.

`X` must be the same size as `Y` or, if `Y` is a matrix, `X` can be a row or a column vector such that

```
length(X) = size(Y,1)
```

stairs

`stairs(...,LineStyle)` specifies a line style, marker symbol, and color for the graph (see `LineStyle` for more information).

`stairs(..., 'PropertyName',propertyvalue)` creates the stairstep graph, applying the specified property settings. See `Stairseries` properties for a description of properties.

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

`h = stairs(...)` returns the handles of the stairseries objects created (one per matrix column).

`[xb,yb] = stairs(Y,...)` does not draw graphs, but returns vectors `xb` and `yb` such that `plot(xb,yb)` plots the stairstep graph.

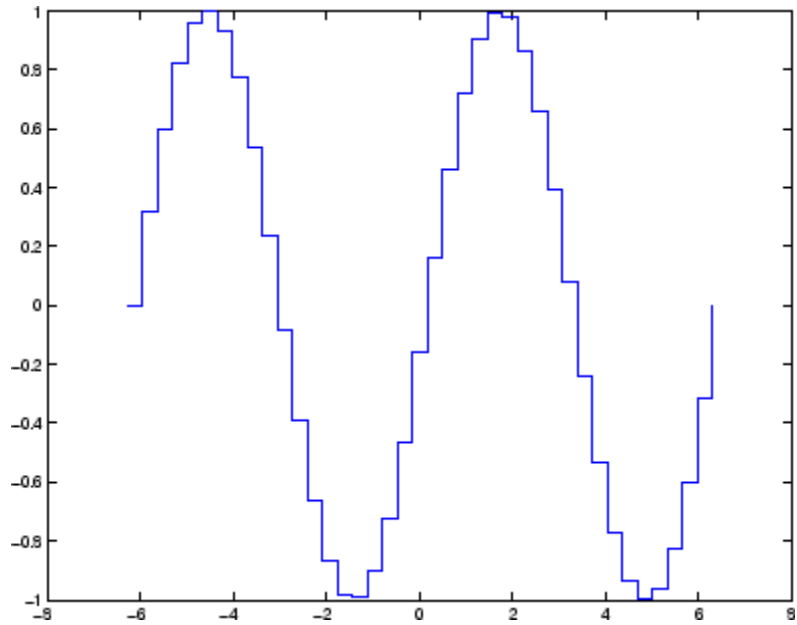
Backward-Compatible Version

`hlines = stairs('v6',...)` returns the handles of line objects instead of stairseries objects for compatibility with MATLAB 6.5 and earlier.

Examples

Create a stairstep plot of a sine wave.

```
x = linspace(-2*pi,2*pi,40);
stairs(x,sin(x))
```



See Also

bar, hist, stem

“Discrete Data Plots” on page 1-85 for related functions

See Stairseries Properties for property descriptions

Stairseries Properties

Purpose

Defines the stairseries properties

Modifying Properties

You can set and query graphics object properties using the set and get commands or the Property Editor (propertyeditor).

Note that you cannot define default property values for stairseries objects.

See Plot Objects for information on stairseries objects.

Stairseries Property Descriptions

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

BeingDeleted
on | {off} Read Only

This object is being deleted. The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the BeingDeleted property to on when the object's delete function callback is called (see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

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

BusyAction
cancel | {queue}

Callback routine interruption. The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over the stairseries object.

This property can be

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

The expression executes in the MATLAB workspace.

See [Function Handle Callbacks](#) for information on how to use function handles to define the callbacks.

`Children`

array of graphics object handles

Children of the stairseries object. An array containing the handles of all line objects parented to the stairseries object (whether visible or not).

Stairseries Properties

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in the `stairs Children` property unless you set the root `ShowHiddenHandles` property to `on`:

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

Clipping
{on} | off

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

Color
ColorSpec

Color of lines. A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

CreateFcn
string or function handle

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

```
stairs(1:10, 'CreateFcn', @CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other `stairs` properties. Setting this property on an existing `stairs` object has no effect.

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

See [Function Handle Callbacks](#) for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when the stairseries object is deleted (e.g., this might happen when you issue a `delete` command on the stairseries object, its parent axes object, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

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

See [Function Handle Callbacks](#) for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string

Label used by plot legends. The legend and the plot browser use this text for labels for any stairseries objects appearing in these legends.

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

Erase mode. This property controls the technique MATLAB uses to draw and erase stairs child objects. Alternative erase modes

Stairseries Properties

are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- **none** — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- **xor** — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- **background** — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR operation on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting

to obtain greater rendering speed. However, these techniques are not applied to the printed output.

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

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

`HandleVisibility`
{on} | callback | off

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

- on — Handles are always visible when `HandleVisibility` is on.
- callback — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- off — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching

Stairseries Properties

the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

```
HitTest  
{on} | off
```

Selectable by mouse click. `HitTest` determines if the stairseries object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the line objects that compose the stairs plot. If `HitTest` is `off`, clicking the stairseries object selects the object below it (which is usually the axes object containing it).

```
HitTestArea  
on | {off}
```

Select stairseries object on lines or area of extent. This property enables you to select stairseries objects in two ways:

- Select by clicking on lines (default).
- Select by clicking anywhere in the extent of the stairstep graph.

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

Interruptible

{on} | off

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

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to on allows any graphics object's callback to interrupt callback routines originating from a stairs property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

LineStyle

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

Line style. This property specifies the line style used for the stairstep lines. Available line styles are shown in the table.

Stairseries Properties

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 stairs lines. Specify this value in points (1 point = $\frac{1}{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`
character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at the end of the stairs lines. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
+	Plus sign
o	Circle
*	Asterisk
.	Point

Marker Specifier	Description
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 stairs Color property.

MarkerFaceColor

ColorSpec | {none} | auto

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the axes color, or the

Stairseries Properties

figure color, if the axes `Color` property is set to none (which is the factory default for axes).

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of axes, `hggroup`, or `hgtransform`

Parent of stairseries object. This property contains the handle of the stairseries object's parent. The parent of a stairseries object is the axes, `hggroup`, or `hgtransform` object that contains it.

See [Objects That Can Contain Other Objects](#) for more information on parenting graphics objects.

`Selected`
on | {off}

Is object selected. When you set this property to on, MATLAB displays selection handles at the corners and midpoints if the `SelectionHighlight` property is also on (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to on, thereby indicating that the stairseries object is selected.

`SelectionHighlight`
{on} | off

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

Tag

string

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

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

```
t = stairs(Y,'Tag','stairs1')
```

When you want to access the stairseries object, you can use `findobj` to find the stairseries object's handle. The following statement changes the `MarkerFaceColor` property of the object whose Tag is `stairs1`.

```
set(findobj('Tag','stairs1'),'MarkerFaceColor','red')
```

Type

string (read only)

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

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

UIContextMenu

handle of a `uicontextmenu` object

Associate a context menu with the stairseries object. Assign this property the handle of a `uicontextmenu` object created in the stairseries object's parent figure. Use the `uicontextmenu` function

Stairseries Properties

to create the context menu. MATLAB displays the context menu whenever you right-click over the stairseries object.

UserData
array

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

Visible
{on} | off

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

XData
array

X-axis location of stairs. The stairs function uses XData to label the x -axis. XData can be either a matrix equal in size to YData or a vector equal in length to the number of rows in YData. That is, $\text{length}(\text{XData}) == \text{size}(\text{YData}, 1)$.

If you do not specify XData (i.e., the input argument x), the stairs function uses the indices of YData to create the stairstep graph. See the XDataMode property for related information.

XDataMode
{auto} | manual

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

If you set `XDataMode` to `auto` after having specified `XData`, the `stairs` function resets the stairs locations and x tick-mark labels to the indices of the `YData`, overwriting any previous values.

`XDataSource`
string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

`YData`
scalar, vector, or matrix

Stairs plot data. `YData` contains the data plotted in the stairstep graph. Each value in `YData` is represented by a marker in the stairstep graph. If `YData` is a matrix, the `stairs` function creates a line for each column in the matrix.

Stairseries Properties

The input argument `y` in the `stairs` function calling syntax assigns values to `YData`.

`YDataSource`
string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

Purpose Start timer(s) running

Syntax start(obj)

Description start(obj) starts the timer running, represented by the timer object, obj. If obj is an array of timer objects, start starts all the timers. Use the timer function to create a timer object.

start sets the Running property of the timer object, obj, to 'on', initiates TimerFcn callbacks, and executes the StartFcn callback.

The timer stops running if one of the following conditions apply:

- The first TimerFcn callback completes, if ExecutionMode is 'singleShot'.
- The number of TimerFcn callbacks specified in TasksToExecute have been executed.
- The stop(obj) command is issued.
- An error occurred while executing a TimerFcn callback.

See Also timer, stop

startat

Purpose Start timer(s) running at specified time

Syntax

```
startat(obj,time)
startat(obj,S)
startat(obj,S,pivotyear)
startat(obj,Y,M,D)
startat(obj,[Y,M,D])
startat(obj,Y,M,D,H,MI,S)
startat(obj,[Y,M,D,H,MI,S])
```

Description `startat(obj,time)` starts the timer running, represented by the timer object `obj`, at the time specified by the serial date number `time`. If `obj` is an array of timer objects, `startat` starts all the timers running at the specified time. Use the `timer` function to create the timer object.

`startat` sets the `Running` property of the timer object, `obj`, to 'on', initiates `TimerFcn` callbacks, and executes the `StartFcn` callback.

The serial date number, `time`, indicates the number of days that have elapsed since 1-Jan-0000 (starting at 1). See `datenum` for additional information about serial date numbers.

`startat(obj,S)` starts the timer running at the time specified by the date string `S`. The date string must use date format 0, 1, 2, 6, 13, 14, 15, 16, or 23, as defined by the `datestr` function. Date strings with two-character years are interpreted to be within the 100 years centered on the current year.

`startat(obj,S,pivotyear)` uses the specified pivot year as the starting year of the 100-year range in which a two-character year resides. The default pivot year is the current year minus 50 years.

`startat(obj,Y,M,D)` `startat(obj,[Y,M,D])` start the timer at the year (`Y`), month (`M`), and day (`D`) specified. `Y`, `M`, and `D` must be arrays of the same size (or they can be a scalar).

`startat(obj,Y,M,D,H,MI,S)` `startat(obj,[Y,M,D,H,MI,S])` start the timer at the year (`Y`), month (`M`), day (`D`), hour (`H`), minute (`MI`), and second (`S`) specified. `Y`, `M`, `D`, `H`, `MI`, and `S` must be arrays of the same size (or they can be a scalar). Values outside the normal range of each array

are automatically carried to the next unit (for example, month values greater than 12 are carried to years). Month values less than 1 are set to be 1; all other units can wrap and have valid negative values.

The timer stops running if one of the following conditions apply:

- The number of `TimerFcn` callbacks specified in `TasksToExecute` have been executed.
- The `stop(obj)` command is issued.
- An error occurred while executing a `TimerFcn` callback.

Examples

This example uses a timer object to execute a function at a specified time.

```
t1=timer('TimerFcn','disp(''it is 10 o''''clock'')');
startat(t1,'10:00:00');
```

This example uses a timer to display a message when an hour has elapsed.

```
t2=timer('TimerFcn','disp(''It has been an hour now.'')');
startat(t2,now+1/24);
```

See Also

`datetime`, `datestr`, `now`, `timer`, `start`, `stop`

startup

Purpose MATLAB startup M-file for user-defined options

Syntax startup

Description startup automatically executes the master M-file matlabrc.m and, if it exists, startup.m, when MATLAB starts. On multiuser or networked systems, matlabrc.m is reserved for use by the system manager. The file matlabrc.m invokes the file startup.m if it exists on the MATLAB search path.

You can create a startup.m file in your own MATLAB startup directory. The file can include physical constants, Handle Graphics defaults, engineering conversion factors, or anything else you want predefined in your workspace.

There are other ways to predefine aspects of MATLAB. See Startup Options and About Preferences in the MATLAB Desktop Tools and Development Environment documentation.

Algorithm Only matlabrc.m is actually invoked by MATLAB at startup. However, matlabrc.m contains the statements

```
if exist('startup')==2
    startup
end
```

that invoke startup.m. You can extend this process to create additional startup M-files, if required.

See Also matlabrc, matlabroot, path, quit

Purpose Standard deviation

Syntax
`s = std(X)`
`s = std(X,flag)`
`s = std(X,flag,dim)`

Definition There are two common textbook definitions for the standard deviation s of a data vector X .

$$(1) \quad s = \left(\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^{\frac{1}{2}}$$

$$(2) \quad s = \left(\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^{\frac{1}{2}}$$

where

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

and n is the number of elements in the sample. The two forms of the equation differ only in $n - 1$ versus n in the divisor.

Description `s = std(X)`, where X is a vector, returns the standard deviation using (1) above. The result s is the square root of an unbiased estimator of the variance of the population from which X is drawn, as long as X consists of independent, identically distributed samples.

If X is a matrix, `std(X)` returns a row vector containing the standard deviation of the elements of each column of X . If X is a multidimensional array, `std(X)` is the standard deviation of the elements along the first nonsingleton dimension of X .

std

`s = std(X,flag)` for `flag = 0`, is the same as `std(X)`. For `flag = 1`, `std(X,1)` returns the standard deviation using (2) above, producing the second moment of the set of values about their mean.

`s = std(X,flag,dim)` computes the standard deviations along the dimension of `X` specified by scalar `dim`. Set `flag` to 0 to normalize `Y` by $n-1$; set `flag` to 1 to normalize by n .

Examples

For matrix `X`

```
X =
     1     5     9
     7    15    22
s = std(X,0,1)
s =
  4.2426   7.0711   9.1924
s = std(X,0,2)
s =
  4.000
  7.5056
```

See Also

`corrcoef`, `cov`, `mean`, `median`, `var`

Purpose Standard deviation of timeseries data

Syntax

```
ts_std = std(ts)
ts_std = std(ts, 'PropertyName1', PropertyValue1, ...)
```

Description `ts_std = std(ts)` returns the standard deviation of the time-series data. When `ts.Data` is a vector, `ts_std` is the standard deviation of `ts.Data` values. When `ts.Data` is a matrix, `ts_std` is the standard deviation 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, `std` always operates along the first nonsingleton dimension of `ts.Data`.

```
ts_std = std(ts, 'PropertyName1', PropertyValue1, ...)
```

specifies the following optional input arguments:

- 'MissingData' property has two possible values, 'remove' (default) or 'interpolate', indicating how to treat missing data during the calculation.
- 'Quality' values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- 'Weighting' property has two possible values, 'none' (default) or 'time'. When you specify 'time', larger time values correspond to larger weights.

Examples 1 Load a 24-by-3 data array.

```
load count.dat
```

2 Create a timeseries object with 24 time values.

```
count_ts = timeseries(count,1:24,'Name','CountPerSecond')
```

std (timeseries)

- 3** Calculate the standard deviation of each data column for this `timeseries` object.

```
std(count_ts)

ans =

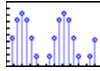
    25.3703    41.4057    68.0281
```

The standard deviation is calculated independently for each data column in the `timeseries` object.


See Also

```
iqr (timeseries), mean (timeseries), median (timeseries), var  
(timeseries), timeseries
```

Purpose Plot discrete sequence data



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
stem(Y)
stem(X,Y)
stem(...,'fill')
stem(...,LineStyle)
stem(axes_handles,...)
h = stem(...)
hlines = stem('v6',...)
```

Description

A two-dimensional stem plot displays data as lines extending from a baseline along the x -axis. A circle (the default) or other marker whose y -position represents the data value terminates each stem.

`stem(Y)` plots the data sequence Y as stems that extend from equally spaced and automatically generated values along the x -axis. When Y is a matrix, `stem` plots all elements in a row against the same x value.

`stem(X,Y)` plots X versus the columns of Y . X and Y must be vectors or matrices of the same size. Additionally, X can be a row or a column vector and Y a matrix with `length(X)` rows.

`stem(...,'fill')` specifies whether to color the circle at the end of the stem.

`stem(...,LineStyle)` specifies the line style, marker symbol, and color for the stem and top marker (the baseline is not affected). See `LineStyle` for more information.

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

`h = stem(...)` returns a vector of stemseries object handles in `h`, one handle per column of data in `Y`.

Backward-Compatible Version

`hlines = stem('v6',...)` returns the handles of line objects instead of stemseries objects for compatibility with MATLAB 6.5 and earlier.

`hlines` contains the handles to three line graphics objects:

- `hlines(1)` — The marker symbol at the top of each stem
- `hlines(2)` — The stem line
- `hlines(3)` — The baseline handle

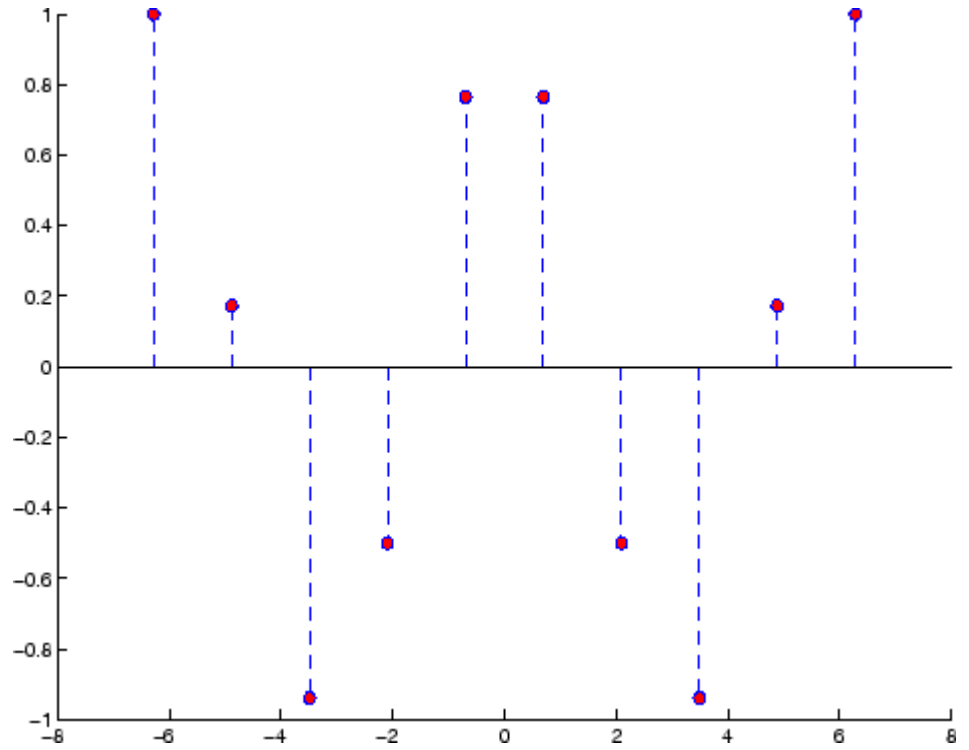
See Plot Objects and Backward Compatibility for more information.

Examples

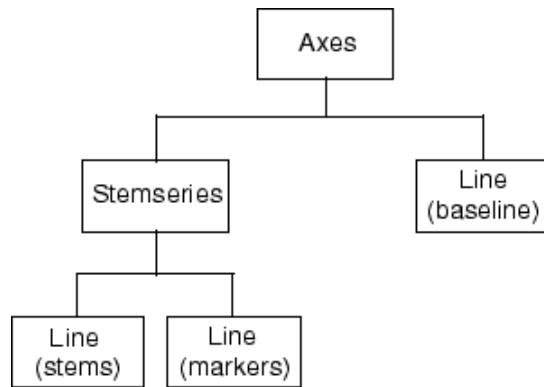
Single Series of Data

This example creates a stem plot representing the cosine of 10 values linearly spaced between 0 and 2π . Note that the line style of the baseline is set by first getting its handle from the stemseries object's `BaseLine` property.

```
t = linspace(-2*pi,2*pi,10);
h = stem(t,cos(t),'fill','--');
set(get(h,'BaseLine'),'LineStyle',':')
set(h,'MarkerFaceColor','red')
```



The following diagram illustrates the parent-child relationship in the previous stem plot. Note that the stemsseries object contains two line objects used to draw the stem lines and the end markers. The baseline is a separate line object.

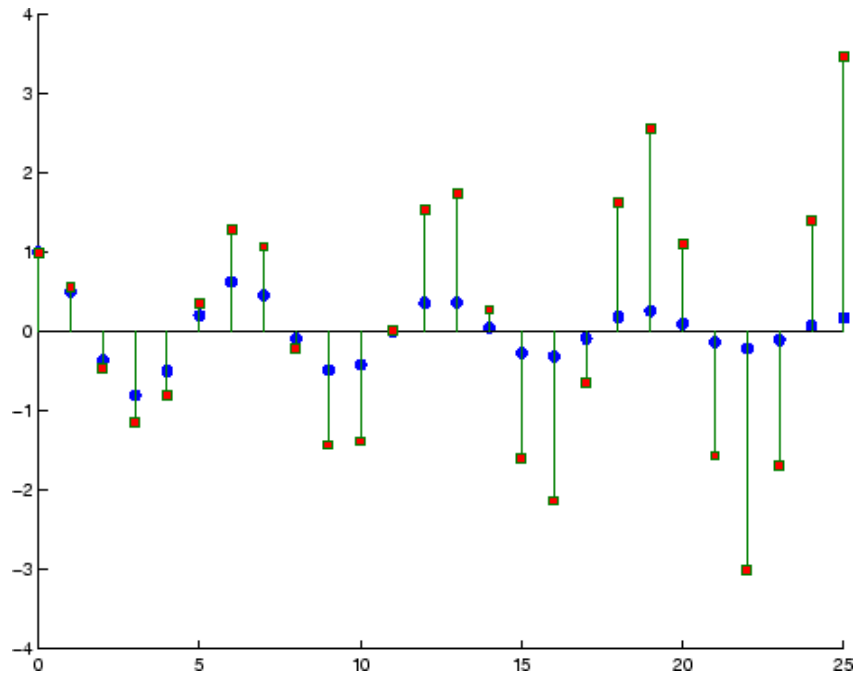


Two Series of Data on One Graph

The following example creates a stem plot from a two-column matrix. In this case, the `stem` function creates two `stemseries` objects, one of each column of data. Both objects' handles are returned in the output argument `h`.

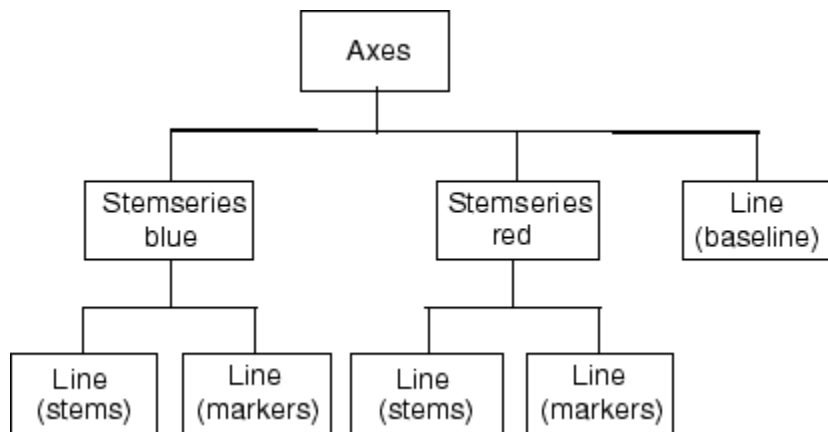
- `h(1)` is the handle to the `stemseries` object plotting the expression `exp(-.07*x).*cos(x)`.
- `h(2)` is the handle to the `stemseries` object plotting the expression `exp(.05*x).*cos(x)`.

```
x = 0:25;  
y = [exp(-.07*x).*cos(x);exp(.05*x).*cos(x)]';  
h = stem(x,y);  
set(h(1),'MarkerFaceColor','blue')  
set(h(2),'MarkerFaceColor','red','Marker','square')
```



The following diagram illustrates the parent-child relationship in the previous stem plot. Note that each column in the input matrix y results in the creation of a stemseries object, which contains two line objects (one for the stems and one for the markers). The baseline is shared by both stemseries objects.

stem



See Also


bar, plot, stairs

See Stemseries properties for property descriptions

Purpose Plot 3-D discrete sequence data



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
stem3(Z)
stem3(X,Y,Z)
stem3(...,'fill')
stem3(...,LineStyle)
h = stem3(...)
hlines = stem3('v6',...)
```

Description

Three-dimensional stem plots display lines extending from the x - y plane. A circle (the default) or other marker symbol whose z -position represents the data value terminates each stem.

`stem3(Z)` plots the data sequence Z as stems that extend from the x - y plane. x and y are generated automatically. When Z is a row vector, `stem3` plots all elements at equally spaced x values against the same y value. When Z is a column vector, `stem3` plots all elements at equally spaced y values against the same x value.

`stem3(X,Y,Z)` plots the data sequence Z at values specified by X and Y . X , Y , and Z must all be vectors or matrices of the same size.

`stem3(...,'fill')` specifies whether to color the interior of the circle at the end of the stem.

`stem3(...,LineStyle)` specifies the line style, marker symbol, and color for the stems. See `LineStyle` for more information.

`h = stem3(...)` returns handles to stems series graphics objects.

stem3

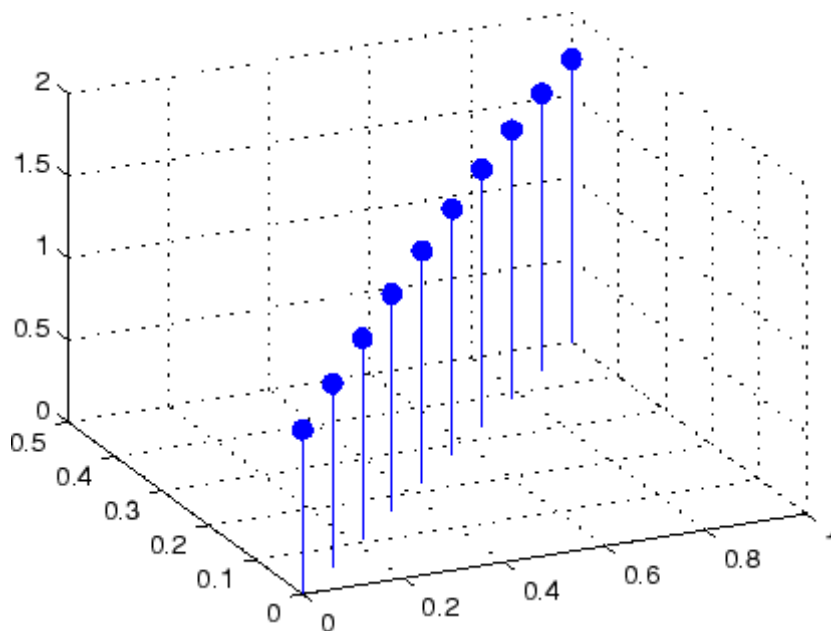
Backward-Compatible Version

`hlines = stem3('v6',...)` returns the handles of line objects instead of stemseries objects for compatibility with MATLAB 6.5 and earlier.

Examples

Create a three-dimensional stem plot to visualize a function of two variables.

```
X = linspace(0,1,10);  
Y = X./2;  
Z = sin(X) + cos(Y);  
stem3(X,Y,Z,'fill')  
view(-25,30)
```



See Also

`bar`, `plot`, `stairs`, `stem`

“Discrete Data Plots” on page 1-85 for related functions

Stemseries Properties for descriptions of properties

Three-Dimensional Stem Plots for more examples

Stemseries Properties

Purpose

Defines the stemseries properties.

Modifying Properties

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

Note that you cannot define default properties for stemseries objects.

See Plot Objects for information on stemseries objects.

Stemseries Property Descriptions

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

BaseLine

handle of baseline

Handle of the baseline object. This property contains the handle of the line object used as the baseline. You can set the properties of this line using its handle. For example, the following statements create a stem plot, obtain the handle of the baseline from the stemseries object, and then set line properties that make the baseline a dashed, red line.

```
stem_handle = stem(randn(10,1));  
baseline_handle = get(stem_handle,'BaseLine');  
set(baseline_handle,'LineStyle','--','Color','red')
```

BaseValue

y-axis value

Y-axis value where baseline is drawn. You can specify the value along the y-axis at which MATLAB draws the baseline.

BeingDeleted

on | {off} Read Only

This object is being deleted. The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the BeingDeleted

property to on when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to on while the delete function executes, after which the object no longer exists.

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

`BusyAction`
cancel | {queue}

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is off, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over the stemseries object.

Stemseries Properties

This property can be

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

The expression executes in the MATLAB workspace.

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

Children

array of graphics object handles

Children of the stemseries object. An array containing the handles of all line objects parented to the stemseries object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in the stem `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

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

Clipping

{on} | off

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

Color

ColorSpec

Color of stem lines. A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

For example, the following statement would produce a stem plot with red lines.

```
h = stem(randn(10,1), 'Color', 'r');
```

CreateFcn

string or function handle

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

```
stem(x,y, 'CreateFcn', @CallbackFcn)
```

where @CallbackFcn is a function handle that references the callback function.

MATLAB executes this routine after setting all other stemseries properties. Setting this property on an existing stemseries object has no effect.

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

See [Function Handle Callbacks](#) for information on how to use function handles to define the callback function.

DeleteFcn

string or function handle

Callback executed during object deletion. A callback that executes when the stemseries object is deleted (e.g., this might happen when you issue a delete command on the stemseries object, its parent axes object, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

Stemseries Properties

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

See [Function Handle Callbacks](#) for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string

Label used by plot legends. The legend and the plot browser use this text for labels for any stemseries objects appearing in these legends.

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

Erase mode. This property controls the technique MATLAB uses to draw and erase stem child objects (the lines used to construct the stem plot). Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.

- `xor`— Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to none). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color (or the figure background color if the axes `Color` property is set to none). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

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

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

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

`HandleVisibility`
{on} | callback | off

Control access to object's handle by command-line users and GUIs.
This property determines when an object's handle is visible in

Stemseries Properties

its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the stemseries object.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

`HitTest`
`{on} | off`

Selectable by mouse click. `HitTest` determines whether the `stemseries` object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the line objects that compose the stem plot. If `HitTest` is `off`, clicking the `stemseries` object selects the object below it (which is usually the axes object containing it).

`HitTestArea`
`on | {off}`

Select stemseries object on stem lines or area of extent. This property enables you to select `stemseries` objects in two ways:

- Select by clicking stem lines (default).
- Select by clicking anywhere in the extent of the stem graph.

When `HitTestArea` is `off`, you must click the stem lines (excluding the baseline) to select the `stemseries` object. When `HitTestArea` is `on`, you can select the `stemseries` object by clicking anywhere within the extent of the stem plot (i.e., anywhere within a rectangle that encloses all the stem lines).

`Interruptible`
`{on} | off`

Stemseries Properties

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

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a stem property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

LineStyle

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

Line style. This property specifies the line style used for the stem lines. Available line styles are shown in the table.

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

Width of the stem lines. 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 the end of the stem lines. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

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)

Stemseries Properties

MarkerEdgeColor

ColorSpec | none | {auto}

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none specifies no color, which makes nonfilled markers invisible. auto sets MarkerEdgeColor to the same color as the stem Color property.

MarkerFaceColor

ColorSpec | {none} | auto

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the axes color, or the figure color, if the axes Color property is set to none (which is the factory default for axes).

MarkerSize

size in points

Marker size. A scalar specifying the size of the marker in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the ' .' symbol) at one-third the specified size.

Parent

handle of axes, hggroup, or hgtransform object

Parent of stemseries object. This property contains the handle of the stemseries object's parent. The parent of a stemseries object is the axes, hggroup, or hgtransform object that contains it.

See [Objects That Can Contain Other Objects](#) for more information on parenting graphics objects.

Selected

on | {off}

Is object selected? When you set this property to on, MATLAB displays selection handles at the corners and midpoints if the SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this property to on, thereby indicating that the stemseries object has been selected.

SelectionHighlight

{on} | off

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

ShowBaseLine

{on} | off

Turn baseline display on or off. This property determines whether stem plots display a baseline from which the stems are drawn. By default, the baseline is displayed.

Tag

string

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

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

```
t = stem(Y,'Tag','stem1')
```

Stemseries Properties

When you want to access the stemseries object, you can use `findobj` to find the stemseries object's handle. The following statement changes the `MarkerFaceColor` property of the object whose `Tag` is `stem1`.

```
set(findobj('Tag','stem1'),'MarkerFaceColor','red')
```

Type

string (read only)

Type of graphics object. This property contains a string that identifies the class of the graphics object. For stemseries objects, `Type` is `'hgroup'`. The following statement finds all the `hgroup` objects in the current axes object.

```
t = findobj(gca,'Type','hgroup');
```

UIContextMenu

handle of a `uicontextmenu` object

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

UserData

array

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

Visible

{on} | off

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

XData

array

X-axis location of stems. The stem function draws an individual stem at each x -axis location in the XData array. XData can be either a matrix equal in size to YData or a vector equal in length to the number of rows in YData. That is, `length(XData) == size(YData, 1)`. XData does not need to be monotonically increasing.

If you do not specify XData (i.e., the input argument x), the stem function uses the indices of YData to create the stem plot. See the XDataMode property for related information.

XDataMode

{auto} | manual

Use automatic or user-specified x -axis values. If you specify XData, MATLAB sets this property to manual.

If you set XDataMode to auto after having specified XData, MATLAB resets the x -axis ticks and x -tick labels to the column indices of the ZData, overwriting any previous values for XData.

XDataSource

string (MATLAB variable)

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

Stemseries Properties

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

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

See the `refreshdata` reference page for more information.

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

YData

scalar, vector, or matrix

Stem plot data. YData contains the data plotted as stems. Each value in YData is represented by a marker in the stem plot. If YData is a matrix, MATLAB creates a series of stems for each column in the matrix.

The input argument `y` in the `stem` function calling syntax assigns values to YData.

YDataSource

string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

ZData

vector of coordinates

Z-coordinates. A data defining the stems for 3-D stem graphs. XData and YData (if specified) must be the same size.

ZDataSource

string (MATLAB variable)

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

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

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the

Stemseries Properties

data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

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

Purpose Stop timer(s)

Syntax stop(obj)

Description stop(obj) stops the timer, represented by the timer object, obj. If obj is an array of timer objects, the stop function stops them all. Use the timer function to create a timer object.

The stop function sets the Running property of the timer object, obj, to 'off', halts further TimerFcn callbacks, and executes the StopFcn callback.

See Also timer, start

stopasync

Purpose	Stop asynchronous read and write operations
Syntax	<code>stopasync(obj)</code>
Arguments	<code>obj</code> A serial port object or an array of serial port objects.
Description	<code>stopasync(obj)</code> stops any asynchronous read or write operation that is in progress for <code>obj</code> .
Remarks	<p>You can write data asynchronously using the <code>fprintf</code> or <code>fwrite</code> function. You can read data asynchronously using the <code>readasync</code> function, or by configuring the <code>ReadAsyncMode</code> property to <code>continuous</code>. In-progress asynchronous operations are indicated by the <code>TransferStatus</code> property.</p> <p>If <code>obj</code> is an array of serial port objects and one of the objects cannot be stopped, the remaining objects in the array are stopped and a warning is returned. After an object stops:</p> <ul style="list-style-type: none">• Its <code>TransferStatus</code> property is configured to <code>idle</code>.• Its <code>ReadAsyncMode</code> property is configured to <code>manual</code>.• The data in its output buffer is flushed. <p>Data in the input buffer is not flushed. You can return this data to the MATLAB workspace using any of the synchronous read functions. If you execute the <code>readasync</code> function, or configure the <code>ReadAsyncMode</code> property to <code>continuous</code>, then the new data is appended to the existing data in the input buffer.</p>
See Also	Functions <code>fprintf</code> , <code>fwrite</code> , <code>readasync</code>

Properties

ReadAsyncMode, TransferStatus

str2double

Purpose Convert string to double-precision value

Syntax
`X = str2double('str')`
`X = str2double(C)`

Description `X = str2double('str')` converts the string `str`, which should be an ASCII character representation of a real or complex scalar value, to the MATLAB double-precision representation. The string can contain digits, a comma (thousands separator), a decimal point, a leading + or - sign, an e preceding a power of 10 scale factor, and an i for a complex unit.

If `str` does not represent a valid scalar value, `str2double` returns NaN.

`X = str2double(C)` converts the strings in the cell array of strings `C` to double precision. The matrix `X` returned will be the same size as `C`.

Examples Here are some valid `str2double` conversions.

```
str2double('123.45e7')
str2double('123 + 45i')
str2double('3.14159')
str2double('2.7i - 3.14')
str2double({'2.71' '3.1415'})
str2double('1,200.34')
```

See Also `char`, `hex2num`, `num2str`, `str2num`

Purpose	Construct function handle from function name string
Syntax	<code>str2func('str')</code>
Description	<p><code>str2func('str')</code> constructs a function handle <code>fhandle</code> for the function named in the string <code>'str'</code>.</p> <p>You can create a function handle using either the <code>@function</code> syntax or the <code>str2func</code> command. You can create an array of function handles from strings by creating the handles individually with <code>str2func</code>, and then storing these handles in a cellarray.</p>

Examples

Example 1

To convert the string, `'sin'`, into a handle for that function, type

```
fh = str2func('sin')
fh =
    @sin
```

Example 2

If you pass a function name string in a variable, the function that receives the variable can convert the function name to a function handle using `str2func`. The example below passes the variable, `funcname`, to function `makeHandle`, which then creates a function handle. Here is the function M-file:

```
function fh = makeHandle(funcname)
fh = str2func(funcname);
```

This is the code that calls `makeHandle` to construct the function handle:

```
makeHandle('sin')
ans =
    @sin
```

Example 3

To call `str2func` on a cell array of strings, use the `cellfun` function. This returns a cell array of function handles:

```
fh_array = cellfun(@str2func, {'sin' 'cos' 'tan'}, ...  
                    'UniformOutput', false);
```

```
fh_array{2}(5)  
ans =  
    0.2837
```

Example 4

In the following example, the `myminbnd` function expects to receive either a function handle or string in the first argument. If you pass a string, `myminbnd` constructs a function handle from it using `str2func`, and then uses that handle in a call to `fminbnd`:

```
function myminbnd(fhandle, lower, upper)  
if ischar(fhandle)  
    disp 'converting function string to function handle ...'  
    fhandle = str2func(fhandle);  
end  
fminbnd(fhandle, lower, upper)
```

Whether you call `myminbnd` with a function handle or function name string, the function can handle the argument appropriately:

```
myminbnd('humps', 0.3, 1)  
converting function string to function handle ...  
ans =  
    0.6370
```

See Also

`function_handle`, `func2str`, `functions`

Purpose Form blank-padded character matrix from strings

Syntax `S = str2mat(T1, T2, T3, ...)`

Description `S = str2mat(T1, T2, T3, ...)` forms the matrix `S` containing the text strings `T1`, `T2`, `T3`, ... as rows. The function automatically pads each string with blanks in order to form a valid matrix. Each text parameter, `Ti`, can itself be a string matrix. This allows the creation of arbitrarily large string matrices. Empty strings are significant.

Note This routine will become obsolete in a future version. Use `char` instead.

Remarks `str2mat` differs from `strvcat` in that empty strings produce blank rows in the output. In `strvcat`, empty strings are ignored.

Examples `x = str2mat('36842', '39751', '38453', '90307');`

```
whos x
  Name      Size      Bytes  Class
   x         4x5         40  char array

x(2,3)

ans =

     7
```

See Also `char`, `strvcat`

str2num

Purpose Convert string to number

Syntax
`x = str2num('str')`
`[x status] = str2num('str')`

Description `x = str2num('str')` converts the string `str`, which is an ASCII character representation of a numeric value, to numeric representation. `str2num` also converts string matrices to numeric matrices. If the input string does not represent a valid number or matrix, `str2num(str)` returns the empty matrix in `x`.

The input string can contain

- Digits
- A decimal point
- A leading + or - sign
- A letter e or d preceding a power of 10 scale factor
- A letter i or j indicating a complex or imaginary number.

`[x status] = str2num('str')` returns the status of the conversion in logical status, where `status` equals logical 1 (true) if the conversion succeeds, and logical 0 (false) otherwise. If the input string `str` does not represent a valid number or matrix, MATLAB sets `x` to the empty matrix. If the conversions fails, `status` is set to 0.

Space characters can be significant. For instance, `str2num('1+2i')` and `str2num('1 + 2i')` produce `x = 1+2i`, while `str2num('1 +2i')` produces `x = [1 2i]`. You can avoid these problems by using the `str2double` function.

Note `str2num` uses the `eval` function to convert the input argument, so side effects can occur if the string contains calls to functions. Use `str2double` to avoid such side effects or when `S` contains a single number.

Examples

`str2num('3.14159e0')` is approximately π .

To convert a string matrix,

```
str2num(['1 2'; '3 4'])
```

```
ans =
```

```
1    2  
3    4
```

See Also

`num2str`, `hex2num`, `sscanf`, `sparse`, `special characters`

strcat

Purpose Concatenate strings horizontally

Syntax `t = strcat(s1, s2, s3, ...)`

Description `t = strcat(s1, s2, s3, ...)` horizontally concatenates corresponding rows of the character arrays `s1`, `s2`, `s3`, etc. All input arrays must have the same number of rows (or any can be a single string). When the inputs are all character arrays, the output is also a character array.

When any of the inputs is a cell array of strings, `strcat` returns a cell array of strings formed by concatenating corresponding elements of `s1`, `s2`, etc. The inputs must all have the same size (or any can be a scalar). Any of the inputs can also be character arrays.

Trailing spaces in character array inputs are ignored and do not appear in the output. This is not true for inputs that are cell arrays of strings. Use the concatenation syntax `[s1 s2 s3 ...]` to preserve trailing spaces.

Remarks `strcat` and matrix operation are different for strings that contain trailing spaces:

```
a = 'hello '  
b = 'goodbye'  
strcat(a, b)  
ans =  
hellogoodbye  
[a b]  
ans =  
hello goodbye
```

Examples Given two 1-by-2 cell arrays `a` and `b`,

```
a =          b =  
    'abcde'    'fghi'    'jkl'    'mn'
```

the command `t = strcat(a,b)` yields

```
t =  
    'abcdeijkl'    'fghimn'
```

Given the 1-by-1 cell array `c = {'Q'}`, the command `t = strcat(a,b,c)` yields

```
t =  
    'abcdeijklQ'    'fghimnQ'
```

See Also

`strvcat`, `cat`, `cellstr`

strcmp, strcmpi

Purpose Compare strings

Syntax
TF = strcmp('str1', 'str2')
TF = strcmp('str', C)
TF = strcmp(C1, C2)

Each of these syntaxes apply to both strcmp and strcmpi. The strcmp function is case sensitive in matching strings, while strcmpi is not:

Description Although the following descriptions show only strcmp, they apply to strcmpi as well. The two functions are the same except that strcmpi compares strings without sensitivity to letter case:

TF = strcmp('str1', 'str2') compares the strings str1 and str2 and returns logical 1 (true) if they are identical, and returns logical 0 (false) otherwise.

TF = strcmp('str', C) compares string str to the each element of cell array C, where str is a character vector (or a 1-by-1 cell array) and C is a cell array of strings. The function returns TF, a logical array that is the same size as C and contains logical 1 (true) for those elements of C that are a match, and logical 0 (false) for those elements that are not. The order of the first two input arguments is not important.

TF = strcmp(C1, C2) compares each element of C1 to the same element in C2, where C1 and C2 are equal-size cell arrays of strings. Input C1 and/or C2 can also be a character array with the right number of rows. The function returns TF, a logical array that is the same size as C1 and C2, and contains logical 1 (true) for those elements of C1 and C2 that are a match, and logical 0 (false) for those elements that are not.

Remarks These functions are intended for comparison of character data. When used to compare numeric data, they return logical 0.

Any leading and trailing blanks in either of the strings are explicitly included in the comparison.

The value returned by strcmp and strcmpi is not the same as the C language convention.

strcmp and strcmpi support international character sets.

Examples

Perform a simple comparison of two strings:

```
strcmp('Yes', 'No') =  
    0  
strcmp('Yes', 'Yes') =  
    1
```

Create 3 cell arrays of strings:

```
A = {'MATLAB', 'SIMULINK'; ...  
    'Toolboxes', 'The MathWorks'};  
  
B = {'Handle Graphics', 'Real Time Workshop'; ...  
    'Toolboxes', 'The MathWorks'};  
  
C = {'handle graphics', 'Signal Processing'; ...  
    ' Toolboxes', 'The MATHWORKS'};
```

Perform a comparison of two cell arrays of strings. Compare cell arrays A and B with sensitivity to case:

```
strcmp(A, B)  
ans =  
    0    0  
    1    1
```

Compare cell arrays B and C without sensitivity to case. Note that 'Toolboxes' doesn't match because of the leading space characters in C{2,1} that do not appear in B{2,1}:

```
strcmpi(B, C)  
ans =  
    1    0  
    0    1
```

strcmp, strcmpi

See Also

`strncmp`, `strncmpi`, `strmatch`, `strfind`, `findstr`, `regexp`, `regexpi`,
`regexprep`, `regexpttranslate`

Purpose Compute 2-D streamline data

Syntax

```
XY = stream2(x,y,u,v,startx,starty)
XY = stream2(u,v,startx,starty)
XY = stream2(...,options)
```

Description `XY = stream2(x,y,u,v,startx,starty)` computes streamlines from vector data `u` and `v`. The arrays `x` and `y` define the coordinates for `u` and `v` and must be monotonic and 2-D plaid (such as the data produced by `meshgrid`). `startx` and `starty` define the starting positions of the streamlines. The section [Specifying Starting Points for Stream Plots](#) provides more information on defining starting points.

The returned value `XY` contains a cell array of vertex arrays.

`XY = stream2(u,v,startx,starty)` assumes the arrays `x` and `y` are defined as `[x,y] = meshgrid(1:n,1:m)` where `[m,n] = size(u)`.

`XY = stream2(...,options)` specifies the options used when creating the streamlines. Define options as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify a value, MATLAB uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

Use the `streamline` command to plot the data returned by `stream2`.

Examples This example draws 2-D streamlines from data representing air currents over regions of North America.

stream2

```
load wind
[sx,sy] = meshgrid(80,20:10:50);
streamline(stream2(x(:, :, 5),y(:, :, 5),u(:, :, 5),v(:, :, 5),sx,sy));
```

See Also

coneplot, stream3, streamline

“Volume Visualization” on page 1-98 for related functions

Specifying Starting Points for Stream Plots for related information

Purpose Compute 3-D streamline data

Syntax

```
XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)
XYZ = stream3(U,V,W,startx,starty,startz)
XYZ = stream3(...,options)
```

Description XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz) computes streamlines from vector data U, V, W. The arrays X, Y, Z define the coordinates for U, V, W and must be monotonic and 3-D plaid (such as the data produced by meshgrid). startx, starty, and startz define the starting positions of the streamlines. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The returned value XYZ contains a cell array of vertex arrays.

XYZ = stream3(U,V,W,startx,starty,startz) assumes the arrays X, Y, and Z are defined as [X,Y,Z] = meshgrid(1:N,1:M,1:P) where [M,N,P] = size(U).

XYZ = stream3(...,options) specifies the options used when creating the streamlines. Define options as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

Use the streamline command to plot the data returned by stream3.

stream3

Examples

This example draws 3-D streamlines from data representing air currents over regions of North America.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamline(stream3(x,y,z,u,v,w,sx,sy,sz))
view(3)
```

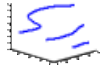
See Also

coneplot, stream2, streamline


“Volume Visualization” on page 1-98 for related functions

Specifying Starting Points for Stream Plots for related information

Purpose Plot streamlines from 2-D or 3-D vector data



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
streamline(X,Y,Z,U,V,W,startx,starty,startz)
streamline(U,V,W,startx,starty,startz)
streamline(XYZ)
streamline(X,Y,U,V,startx,starty)
streamline(U,V,startx,starty)
streamline(XY)
streamline(...,options)
streamline(axes_handles,...)
h = streamline(...)
```

Description

`streamline(X,Y,Z,U,V,W,startx,starty,startz)` draws streamlines from 3-D vector data U, V, W . The arrays X, Y, Z define the coordinates for U, V, W and must be monotonic and 3-D plaid (such as the data produced by `meshgrid`). `startx, starty, startz` define the starting positions of the streamlines. The section [Specifying Starting Points for Stream Plots](#) provides more information on defining starting points.

`streamline(U,V,W,startx,starty,startz)` assumes the arrays $X, Y,$ and Z are defined as $[X,Y,Z] = \text{meshgrid}(1:N,1:M,1:P)$ where $[M,N,P] = \text{size}(U)$.

`streamline(XYZ)` assumes XYZ is a precomputed cell array of vertex arrays (as produced by `stream3`).

streamline

`streamline(X,Y,U,V,startx,starty)` draws streamlines from 2-D vector data `U, V`. The arrays `X, Y` define the coordinates for `U, V` and must be monotonic and 2-D plaid (such as the data produced by `meshgrid`). `startx` and `starty` define the starting positions of the streamlines. The output argument `h` contains a vector of line handles, one handle for each streamline.

`streamline(U,V,startx,starty)` assumes the arrays `X` and `Y` are defined as `[X,Y] = meshgrid(1:N,1:M)` where `[M,N] = size(U)`.

`streamline(XY)` assumes `XY` is a precomputed cell array of vertex arrays (as produced by `stream2`).

`streamline(...,options)` specifies the options used when creating the streamlines. Define options as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

`[stepsize]`

or

`[stepsize, max_number_vertices]`

If you do not specify values, MATLAB uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 1000

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

`h = streamline(...)` returns a vector of line handles, one handle for each streamline.

Examples

This example draws streamlines from data representing air currents over a region of North America. Loading the wind data set creates the variables `x, y, z, u, v,` and `w` in the MATLAB workspace.

The plane of streamlines indicates the flow of air from the west to the east (the x -direction) beginning at $x = 80$ (which is close to the minimum value of the x coordinates). The y - and z -coordinate starting points are multivalued and approximately span the range of these coordinates. `meshgrid` generates the starting positions of the streamlines.

```
load wind
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
h = streamline(x,y,z,u,v,w,sx,sy,sz);
set(h,'Color','red')
view(3)
```

See Also

`coneplot`, `stream2`, `stream3`, `streamparticles`


“Volume Visualization” on page 1-98 for related functions

Specifying Starting Points for Stream Plots for related information

Stream Line Plots of Vector Data for another example

streamparticles

Purpose Plot stream particles

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

```
streamparticles(vertices)
streamparticles(vertices,n)
streamparticles(...,'PropertyName',PropertyValue,...)
streamparticles(line_handle,...)
h = streamparticles(...)
```

Description `streamparticles(vertices)` draws stream particles of a vector field. Stream particles are usually represented by markers and can show the position and velocity of a streamline. `vertices` is a cell array of 2-D or 3-D vertices (as if produced by `stream2` or `stream3`).

`streamparticles(vertices,n)` uses `n` to determine how many stream particles to draw. The `ParticleAlignment` property controls how `n` is interpreted.

- If `ParticleAlignment` is set to `off` (the default) and `n` is greater than 1, then approximately `n` particles are drawn evenly spaced over the streamline vertices.

If `n` is less than or equal to 1, `n` is interpreted as a fraction of the original stream vertices; for example, if `n` is 0.2, approximately 20% of the vertices are used.

`n` determines the upper bound for the number of particles drawn. Note that the actual number of particles can deviate from `n` by as much as a factor of 2.

- If `ParticleAlignment` is on, `n` determines the number of particles on the streamline having the most vertices and sets the spacing on the other streamlines to this value. The default value is `n = 1`.

`streamparticles(..., 'PropertyName', PropertyValue, ...)`
controls the stream particles using named properties and specified values. Any unspecified properties have default values. MATLAB ignores the case of property names.

Stream Particle Properties

`Animate` — Stream particle motion [nonnegative integer]

The number of times to animate the stream particles. The default is 0, which does not animate. `Inf` animates until you enter **Ctrl+C**.

`FrameRate` — Animation frames per second [nonnegative integer]

This property specifies the number of frames per second for the animation. `Inf`, the default, draws the animation as fast as possible. Note that the speed of the animation might be limited by the speed of the computer. In such cases, the value of `FrameRate` cannot necessarily be achieved.

`ParticleAlignment` — Align particles with streamlines [on | {off}]

Set this property to `on` to draw particles at the beginning of each streamline. This property controls how `streamparticles` interprets the argument `n` (number of stream particles).

Stream particles are line objects. In addition to stream particle properties, you can specify any line object property, such as `Marker` and `EraseMode`. `streamparticles` sets the following line properties when called.

Line Property	Value Set by <code>streamparticles</code>
<code>EraseMode</code>	<code>xor</code>
<code>LineStyle</code>	<code>none</code>
<code>Marker</code>	<code>o</code>

streamparticles

Line Property	Value Set by streamparticles
MarkerEdgeColor	none
MarkerFaceColor	red

You can override any of these properties by specifying a property name and value as arguments to `streamparticles`. For example, this statement uses RGB values to set the `MarkerFaceColor` to medium gray:

```
streamparticles(vertices, 'MarkerFaceColor', [.5 .5 .5])
```

`streamparticles(line_handle, ...)` uses the line object identified by `line_handle` to draw the stream particles.

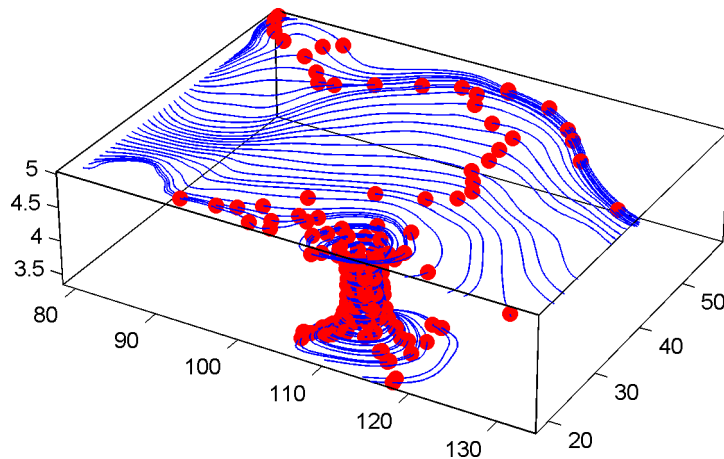
`h = streamparticles(...)` returns a vector of handles to the line objects it creates.

Examples

This example combines streamlines with stream particle animation. The `interpstreamspeed` function determines the vertices along the streamlines where stream particles will be drawn during the animation, thereby controlling the speed of the animation. Setting the axes `DrawMode` property to `fast` provides faster rendering.

```
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
sl = streamline(verts);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.025);
axis tight; view(30,30); daspect([1 1 .125])
camproj perspective; camva(8)
set(gca, 'DrawMode', 'fast')
box on
streamparticles(iverts,35, 'animate', 10, 'ParticleAlignment', 'on')
```

The following picture is a static view of the animation.



This example uses the streamlines in the $z = 5$ plane to animate the flow along these lines with streamparticles.

```
load wind
daspect([1 1 1]); view(2)
[verts averts] = streamslice(x,y,z,u,v,w,[],[],[5]);
sl = streamline([verts averts]);
axis tight off;
set(sl,'Visible','off')
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.05);
set(gca,'DrawMode','fast','Position',[0 0 1 1],'ZLim',[4.9 5.1])
set(gcf,'Color','black')
streamparticles(iverts, 200, ...
    'Animate',100,'FrameRate',40, ...
    'MarkerSize',10,'MarkerFaceColor','yellow')
```

See Also

[interpstreamspeed](#), [stream3](#), [streamline](#)

“Volume Visualization” on page 1-98 for related functions

[Creating Stream Particle Animations](#) for more details


[Specifying Starting Points for Stream Plots](#) for related information

streamribbon

Purpose 3-D stream ribbon plot from vector volume data



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
streamribbon(X,Y,Z,U,V,W,startx,starty,startz)
streamribbon(U,V,W,startx,starty,startz)
streamribbon(vertices,X,Y,Z,cav,speed)
streamribbon(vertices,cav,speed)
streamribbon(vertices,twistangle)
streamribbon(...,width)
streamribbon(axes_handles,...)
h = streamribbon(...)
```

Description

`streamribbon(X,Y,Z,U,V,W,startx,starty,startz)` draws stream ribbons from vector volume data `U, V, W`. The arrays `X, Y, Z` define the coordinates for `U, V, W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`). `startx, starty, and startz` define the starting positions of the stream ribbons at the center of the ribbons. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The twist of the ribbons is proportional to the curl of the vector field. The width of the ribbons is calculated automatically.

Generally, you should set the `DataAspectRatio (daspect)` before calling `streamribbon`.

`streamribbon(U,V,W,startx,starty,startz)` assumes `X, Y, and Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamribbon(vertices,X,Y,Z,cav,speed)` assumes precomputed streamline vertices, curl angular velocity, and flow speed. `vertices` is a cell array of streamline vertices (as produced by `stream3`). `X`, `Y`, `Z`, `cav`, and `speed` are 3-D arrays.

`streamribbon(vertices,cav,speed)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(cav)`.

`streamribbon(vertices,twistangle)` uses the cell array of vectors `twistangle` for the twist of the ribbons (in radians). The size of each corresponding element of `vertices` and `twistangle` must be equal.

`streamribbon(...,width)` sets the width of the ribbons to `width`.

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

`h = streamribbon(...)` returns a vector of handles (one per start point) to surface objects.

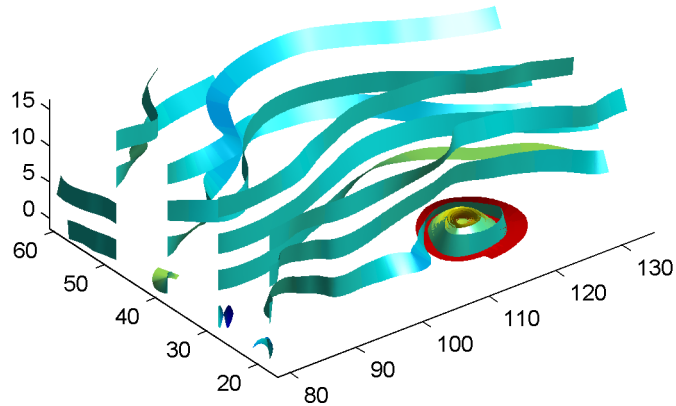
Examples

This example uses stream ribbons to indicate the flow in the wind data set. Inputs include the coordinates, vector field components, and starting location for the stream ribbons.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
streamribbon(x,y,z,u,v,w,sx,sy,sz);
%-----Define viewing and lighting
axis tight
shading interp;
view(3);
```

streamribbon

```
camlight; lighting gouraud
```

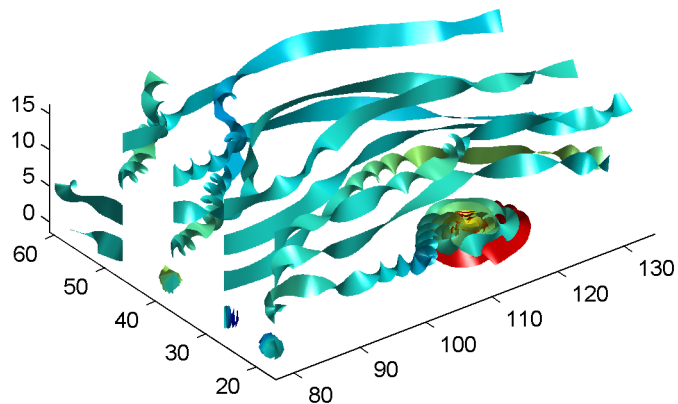


This example uses precalculated vertex data (`stream3`), curl average velocity (`curl`), and speed ($\sqrt{u^2 + v^2 + w^2}$). Using precalculated data enables you to use values other than those calculated from the single data source. In this case, the speed is reduced by a factor of 10 compared to the previous example.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
cav = curl(x,y,z,u,v,w);
spd = sqrt(u.^2 + v.^2 + w.^2).*0.1;
streamribbon(verts,x,y,z,cav,spd);
%-----Define viewing and lighting
axis tight
shading interp
```



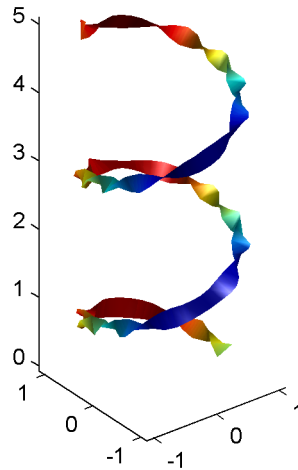
```
view(3)
camlight; lighting gouraud
```



This example specifies a twist angle for the stream ribbon.

```
t = 0:.15:15;
verts = {[cos(t)' sin(t)' (t/3)']};
twistangle = {cos(t)'};
daspect([1 1 1])
streamribbon(verts,twistangle);
%-----Define viewing and lighting
axis tight
shading interp;
view(3);
camlight; lighting gouraud
```

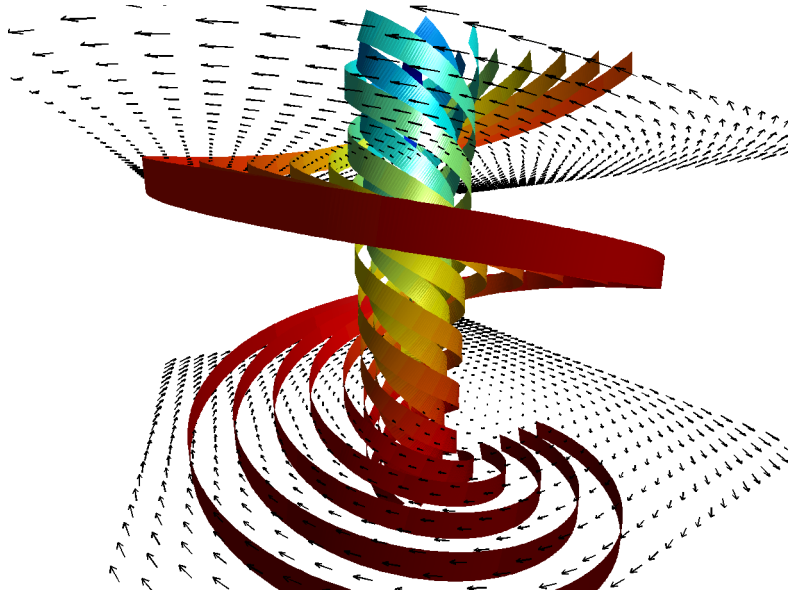
streamribbon



This example combines cone plots (coneplot) and stream ribbon plots in one graph.

```
%-----Define 3-D arrays x, y, z, u, v, w
xmin = -7; xmax = 7;
ymin = -7; ymax = 7;
zmin = -7; zmax = 7;
x = linspace(xmin,xmax,30);
y = linspace(ymin,ymax,20);
z = linspace(zmin,zmax,20);
[x y z] = meshgrid(x,y,z);
u = y; v = -x; w = 0*x+1;
daspect([1 1 1]);
[cx cy cz] = meshgrid(linspace(xmin,xmax,30),...
    linspace(ymin,ymax,30),[-3 4]);
h = coneplot(x,y,z,u,v,w,cx,cy,cz,'quiver');
set(h,'color','k');
%-----Plot two sets of streamribbons
```

```
[sx sy sz] = meshgrid([-1 0 1],[-1 0 1],-6);  
streamribbon(x,y,z,u,v,w,sx,sy,sz);  
[sx sy sz] = meshgrid([1:6],[0],-6);  
streamribbon(x,y,z,u,v,w,sx,sy,sz);  
%-----Define viewing and lighting  
shading interp  
view(-30,10) ; axis off tight  
camproj perspective; camva(66); camlookat;  
camdolly(0,0,.5,'fixtarget')  
camlight
```

**See Also**

`curl`, `streamtube`, `streamline`, `stream3`

“Volume Visualization” on page 1-98 for related functions

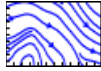
Displaying Curl with Stream Ribbons for another example

Specifying Starting Points for Stream Plots for related information


streamslice

Purpose

Plot streamlines in slice planes



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
streamslice(X,Y,Z,U,V,W,startx,starty,startz)
streamslice(U,V,W,startx,starty,startz)
streamslice(X,Y,U,V)
streamslice(U,V)
streamslice(...,density)
streamslice(...,'arrowsmode')
streamslice(...,'method')
streamslice(axes_handles,...)
h = streamslice(...)
[vertices arrowvertices] = streamslice(...)
```

Description

`streamslice(X,Y,Z,U,V,W,startx,starty,startz)` draws well spaced streamlines (with direction arrows) from vector data U, V, W in axis aligned x -, y -, z -planes starting at the points in the vectors `startx`, `starty`, `startz`. (The section Specifying Starting Points for Stream Plots provides more information on defining starting points.) The arrays X, Y, Z define the coordinates for U, V, W and must be monotonic and 3-D plaid (as if produced by `meshgrid`). U, V, W must be m -by- n -by- p volume arrays.

You should not assume that the flow is parallel to the slice plane. For example, in a stream slice at a constant z , the z component of the vector field W is ignored when you are calculating the streamlines for that plane.

Stream slices are useful for determining where to start streamlines, stream tubes, and stream ribbons. It is good practice is to set the axes `DataAspectRatio` to `[1 1 1]` when using `streamslice`.

`streamslice(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamslice(X,Y,U,V)` draws well spaced streamlines (with direction arrows) from vector volume data `U`, `V`. The arrays `X`, `Y` define the coordinates for `U`, `V` and must be monotonic and 2-D plaid (as if produced by `meshgrid`).

`streamslice(U,V)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamslice(...,density)` modifies the automatic spacing of the streamlines. `density` must be greater than 0. The default value is 1; higher values produce more streamlines on each plane. For example, 2 produces approximately twice as many streamlines, while 0.5 produces approximately half as many.

`streamslice(...,'arrowmode')` determines if direction arrows are present or not. `arrowmode` can be

- `arrows` — Draw direction arrows on the streamlines (default).
- `noarrows` — Do not draw direction arrows.

`streamslice(...,'method')` specifies the interpolation method to use. `method` can be

- `linear` — Linear interpolation (default)

streamslice

- `cubic` — Cubic interpolation
- `nearest` — Nearest-neighbor interpolation

See `interp3` for more information on interpolation methods.

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

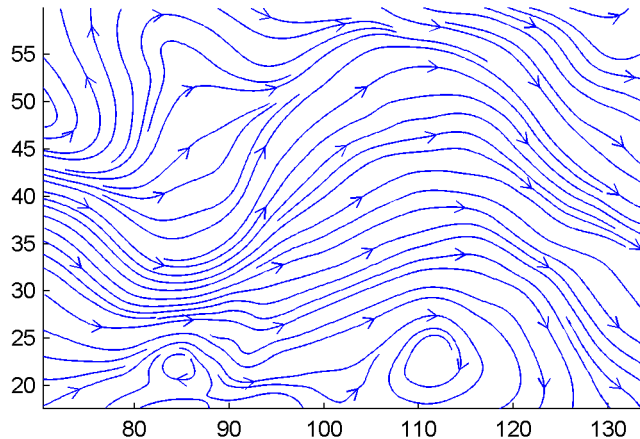
`h = streamslice(...)` returns a vector of handles to the line objects created.

`[vertices arrowvertices] = streamslice(...)` returns two cell arrays of vertices for drawing the streamlines and the arrows. You can pass these values to any of the streamline drawing functions (`streamline`, `streamribbon`, `streamtube`).

Examples

This example creates a stream slice in the wind data set at `z = 5`.

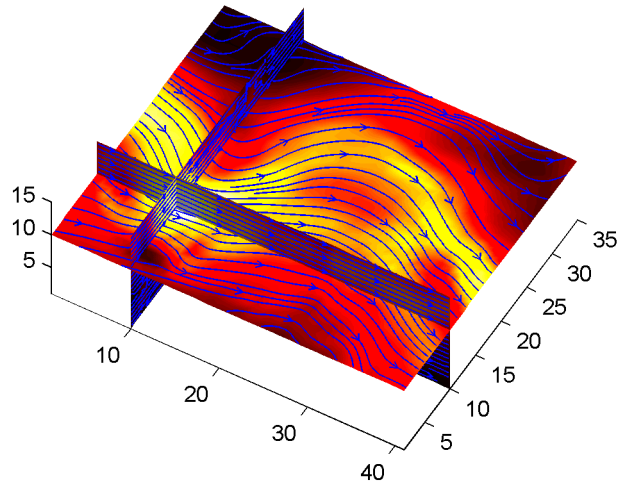
```
load wind
daspect([1 1 1])
streamslice(x,y,z,u,v,w,[],[],[5])
axis tight
```



This example uses `streamslice` to calculate vertex data for the streamlines and the direction arrows. This data is then used by `streamline` to plot the lines and arrows. Slice planes illustrating with color the wind speed ($\sqrt{u^2 + v^2 + w^2}$) are drawn by `slice` in the same planes.

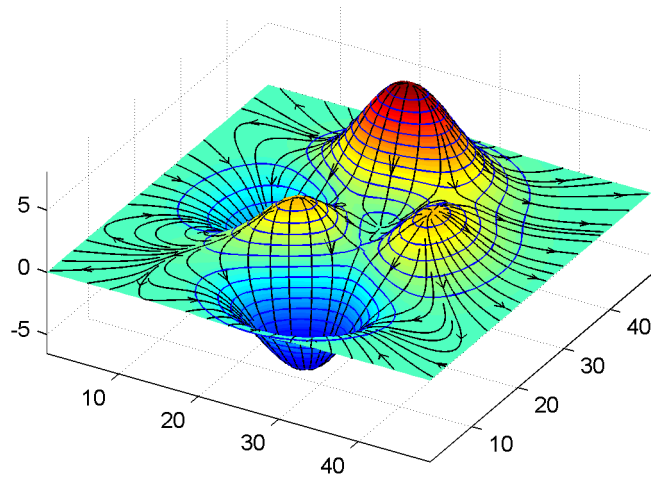
```
load wind
daspect([1 1 1])
[verts averts] = streamslice(u,v,w,10,10,10);
streamline([verts averts])
spd = sqrt(u.^2 + v.^2 + w.^2);
hold on;
slice(spd,10,10,10);
colormap(hot)
shading interp
view(30,50); axis(volumebounds(spd));
camlight; material([.5 1 0])
```

streamslice



This example superimposes contour lines on a surface and then uses `streamslice` to draw lines that indicate the gradient of the surface. `interp2` is used to find the points for the lines that lie on the surface.

```
z = peaks;  
surf(z)  
shading interp  
hold on  
[c ch] = contour3(z,20); set(ch,'edgecolor','b')  
[u v] = gradient(z);  
h = streamslice(-u,-v);  
set(h,'color','k')  
for i=1:length(h);  
    zi = interp2(z,get(h(i),'xdata'),get(h(i),'ydata'));  
    set(h(i),'zdata',zi);  
end  
view(30,50); axis tight
```

See Also

`contourslice`, `slice`, `streamline`, `volumebounds`

“Volume Visualization” on page 1-98 for related functions


Specifying Starting Points for Stream Plots for related information

streamtube

Purpose Create 3-D stream tube plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
streamtube(X,Y,Z,U,V,W,startx,starty,startz)
streamtube(U,V,W,startx,starty,startz)
streamtube(vertices,X,Y,Z,divergence)
streamtube(vertices,divergence)
streamtube(vertices,width)
streamtube(vertices)
streamtube(...,[scale n])
streamtube(axes_handles,...)
h = streamtube(...z)
```

Description

`streamtube(X,Y,Z,U,V,W,startx,starty,startz)` draws stream tubes from vector volume data `U, V, W`. The arrays `X, Y, Z` define the coordinates for `U, V, W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`). `startx, starty, and startz` define the starting positions of the streamlines at the center of the tubes. The section [Specifying Starting Points for Stream Plots](#) provides more information on defining starting points.

The width of the tubes is proportional to the normalized divergence of the vector field.

Generally, you should set the `DataAspectRatio (daspect)` before calling `streamtube`.

`streamtube(U,V,W,startx,starty,startz)` assumes `X, Y, and Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamtube(vertices,X,Y,Z,divergence)` assumes precomputed streamline vertices and divergence. `vertices` is a cell array of streamline vertices (as produced by `stream3`). `X`, `Y`, `Z`, and `divergence` are 3-D arrays.

`streamtube(vertices,divergence)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(divergence)`.

`streamtube(vertices,width)` specifies the width of the tubes in the cell array of vectors, `width`. The size of each corresponding element of `vertices` and `width` must be equal. `width` can also be a scalar, specifying a single value for the width of all stream tubes.

`streamtube(vertices)` selects the width automatically.

`streamtube(...,[scale n])` scales the width of the tubes by `scale`. The default is `scale = 1`. When the stream tubes are created, using start points or divergence, specifying `scale = 0` suppresses automatic scaling. `n` is the number of points along the circumference of the tube. The default is `n = 20`.

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

`h = streamtube(...z)` returns a vector of handles (one per start point) to surface objects used to draw the stream tubes.

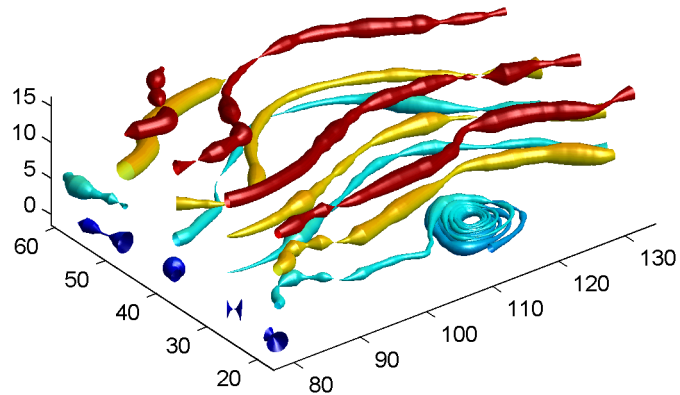
Examples

This example uses stream tubes to indicate the flow in the wind data set. Inputs include the coordinates, vector field components, and starting location for the stream tubes.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
```

streamtube

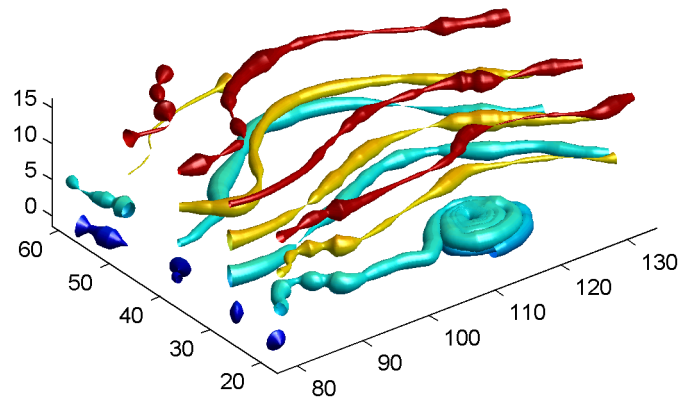
```
daspect([1 1 1])
streamtube(x,y,z,u,v,w,sx,sy,sz);
%-----Define viewing and lighting
view(3)
axis tight
shading interp;
camlight; lighting gouraud
```



This example uses precalculated vertex data (stream3) and divergence (divergence).

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
div = divergence(x,y,z,u,v,w);
streamtube(verts,x,y,z,-div);
%-----Define viewing and lighting
```

```
view(3)
axis tight
shading interp
camlight; lighting gouraud
```

**See Also**

`divergence`, `streamribbon`, `streamline`, `stream3`

“Volume Visualization” on page 1-98 for related functions

Displaying Divergence with Stream Tubes for another example

Specifying Starting Points for Stream Plots for related information

strfind

Purpose Find one string within another

Syntax
`k = strfind(str, pattern)`
`k = strfind(cellstr, pattern)`

Description `k = strfind(str, pattern)` searches the string `str` for occurrences of a shorter string, `pattern`, and returns the starting index of each such occurrence in the double array `k`. If `pattern` is not found in `str`, or if `pattern` is longer than `str`, then `strfind` returns the empty array `[]`.

`k = strfind(cellstr, pattern)` searches each string in cell array of strings `cellstr` for occurrences of a shorter string, `pattern`, and returns the starting index of each such occurrence in cell array `k`. If `pattern` is not found in a string or if `pattern` is longer than all strings in the cell array, then `strfind` returns the empty array `[]`, for that string in the cell array.

The search performed by `strfind` is case sensitive. Any leading and trailing blanks in `pattern` or in the strings being searched are explicitly included in the comparison.

Examples

Use `strfind` to find a two-letter pattern in string `S`:

```
S = 'Find the starting indices of the pattern string';
strfind(S, 'in')
ans =
     2     15     19     45

strfind(S, 'In')
ans =
     []

strfind(S, ' ')
ans =
     5     9     18     26     29     33     41
```

Use `strfind` on a cell array of strings:

```
cstr = {'How much wood would a woodchuck chuck';  
       'if a woodchuck could chuck wood?'};  
  
idx = strfind(cstr, 'wood');  
  
idx{:,:}  
ans =  
    10    23  
ans =  
     6    28
```

This means that 'wood' occurs at indices 10 and 23 in the first string and at indices 6 and 28 in the second.

See Also

findstr, strmatch, strtok, strcmp, strncmp, strcmpi, strncmpi, regexp, regexpi, regexprep

strings

Purpose MATLAB string handling

Syntax
S = 'Any Characters'
S = [S1 S2 ...]
S = strcat(S1, S2, ...)

Description S = 'Any Characters' creates a character array, or string. The string is actually a vector whose components are the numeric codes for the characters (the first 127 codes are ASCII). The actual characters displayed depend on the character encoding scheme for a given font. The length of S is the number of characters. A quotation within the string is indicated by two quotes.

S = [S1 S2 ...] concatenates character arrays S1, S2, etc. into a new character array, S.

S = strcat(S1, S2, ...) concatenates S1, S2, etc., which can be character arrays or “Cell Arrays of Strings”. When the inputs are all character arrays, the output is also a character array. When any of the inputs is a cell array of strings, strcat returns a cell array of strings.

Trailing spaces in strcat character array inputs are ignored and do not appear in the output. This is not true for strcat inputs that are cell arrays of strings. Use the S = [S1 S2 ...] concatenation syntax, shown above, to preserve trailing spaces.

S = char(X) can be used to convert an array that contains positive integers representing numeric codes into a MATLAB character array.

X = double(S) converts the string to its equivalent double-precision numeric codes.

A collection of strings can be created in either of the following two ways:

- As the rows of a character array via strvcat
- As a cell array of strings via the curly braces

You can convert between character array and cell array of strings using char and cellstr. Most string functions support both types.

`ischar(S)` tells if `S` is a string variable. `iscellstr(S)` tells if `S` is a cell array of strings.

Examples

Create a simple string that includes a single quote.

```
msg = 'You're right!'

msg =
You're right!
```

Create the string name using two methods of concatenation.

```
name = ['Thomas' ' R.' ' Lee']
name = strcat('Thomas',' R.',' Lee')
```

Create a vertical array of strings.

```
C = strvcat('Hello','Yes','No','Goodbye')

C =
Hello
Yes
No
Goodbye
```

Create a cell array of strings.

```
S = {'Hello' 'Yes' 'No' 'Goodbye'}

S =
'Hello'      'Yes'      'No'      'Goodbye'
```

See Also

`char`, `isstrprop`, `cellstr`, `ischar`, `isletter`, `isspace`, `iscellstr`, `strvcat`, `sprintf`, `sscanf`, `text`, `input`

strjust

Purpose Justify character array

Syntax
T = strjust(S)
T = strjust(S, 'right')
T = strjust(S, 'left')
T = strjust(S, 'center')

Description T = strjust(S) or T = strjust(S, 'right') returns a right-justified version of the character array S.

T = strjust(S, 'left') returns a left-justified version of S.

T = strjust(S, 'center') returns a center-justified version of S.

See Also deblank, strtrim

Purpose Find possible matches for string

Syntax
`x = strmatch('str', STRS)`
`x = strmatch('str', STRS, 'exact')`

Description `x = strmatch('str', STRS)` looks through the rows of the character array or cell array of strings `STRS` to find strings that begin with string `str`, returning the matching row indices. `strmatch` is fastest when `STRS` is a character array.

`x = strmatch('str', STRS, 'exact')` returns only the indices of the strings in `STRS` matching `str` exactly.

Examples The statement

```
x = strmatch('max', strvcat('max', 'minimax', 'maximum'))
```

returns `x = [1; 3]` since rows 1 and 3 begin with 'max'. The statement

```
x = strmatch('max', strvcat('max', 'minimax', 'maximum'),'exact')
```

returns `x = 1`, since only row 1 matches 'max' exactly.

See Also `strcmp`, `strcmpi`, `strncmp`, `strncmpi`, `strfind`, `findstr`, `strvcat`, `regexp`, `regexpi`, `regexprep`

strncmp, strncmpi

Purpose Compare first *n* characters of strings

Syntax
TF = strncmp('str1', 'str2', n)
TF = strncmp('str', C, n)
TF = strncmp(C1, C2, n)

Each of these syntaxes apply to both `strncmp` and `strncmpi`. The `strncmp` function is case sensitive in matching strings, while `strncmpi` is not:

Description Although the following descriptions show only `strncmp`, they apply to `strncmpi` as well. The two functions are the same except that `strncmpi` compares strings without sensitivity to letter case:

TF = strncmp('str1', 'str2', n) compares the first *n* characters of strings `str1` and `str2` and returns logical 1 (true) if they are identical, and returns logical 0 (false) otherwise.

TF = strncmp('str', C, n) compares the first *n* characters of `str` to the first *n* characters of each element of cell array `C`, where `str` is a character vector (or a 1-by-1 cell array), and `C` is a cell array of strings. The function returns TF, a logical array that is the same size as `C` and contains logical 1 (true) for those elements of `C` that are a match, and logical 0 (false) for those elements that are not. The order of the first two input arguments is not important.

TF = strncmp(C1, C2, n) compares each element of `C1` to the same element in `C2`, where `C1` and `C2` are equal-size cell arrays of strings. Input `C1` and/or `C2` can also be a character array with the right number of rows. The function attempts to match only the first *n* characters of each string. The function returns TF, a logical array that is the same size as `C1` and `C2`, and contains logical 1 (true) for those elements of `C1` and `C2` that are a match, and logical 0 (false) for those elements that are not.

Remarks These functions are intended for comparison of character data. When used to compare numeric data, they return logical 0.

Any leading and trailing blanks in either of the strings are explicitly included in the comparison.

The value returned by `strncmp` and `strncmpi` is not the same as the C language convention.

`strncmp` and `strncmpi` support international character sets.

Examples

From a list of 10 MATLAB functions, find those that apply to using a camera:

```
function_list = {'calendar' 'case' 'camdolly' 'circshift' ...  
                'caxis' 'camtarget' 'cast' 'camorbit' ...  
                'callib' 'cart2sph'};
```

```
strncmp(function_list, 'cam', 3)  
ans =  
    0    0    1    0    0    1    0    1    0    0
```

```
function_list{strncmp(function_list, 'cam', 3)}  
ans =  
    camdolly  
ans =  
    camtarget  
ans =  
    camorbit
```

See Also

`strcmp`, `strcmpi`, `strmatch`, `strfind`, `findstr`, `regexp`, `regexpi`, `regexprep`, `regextranslate`

strread

Purpose

Read formatted data from string

Note The `textscan` function is intended as a replacement for both `strread` and `textread`.

Syntax

```
A = strread('str')
[A, B, ...] = strread('str')
[A, B, ...] = strread('str', 'format')
[A, B, ...] = strread('str', 'format', N)
[A, B, ...] = strread('str', 'format', N, param, value, ...)
```

Description

`A = strread('str')` reads numeric data from input string `str` into a 1-by-`N` vector `A`, where `N` equals the number of whitespace-separated numbers in `str`. Use this form only with strings containing numeric data. See “Example 1” on page 2-2914 below.

`[A, B, ...] = strread('str')` reads numeric data from the string input `str` into scalar output variables `A`, `B`, and so on. The number of output variables must equal the number of whitespace-separated numbers in `str`. Use this form only with strings containing numeric data. See “Example 2” on page 2-2914 below.

`[A, B, ...] = strread('str', 'format')` reads data from `str` into variables `A`, `B`, and so on using the specified format. The number of output variables `A`, `B`, etc. must be equal to the number of format specifiers (e.g., `%s` or `%d`) in the `format` argument. You can read all of the data in `str` to a single output variable as long as you use only one format specifier in the command. See “Example 4” on page 2-2915 and “Example 5” on page 2-2915 below.

The table `Formats for strread` on page 2-2911 lists the valid format specifiers. More information on using formats is available under “Formats” on page 2-2913 in the Remarks section below.

`[A, B, ...] = strread('str', 'format', N)` reads data from `str` reusing the format string `N` times, where `N` is an integer greater than zero. If `N` is `-1`, `strread` reads the entire string. When `str` contains

only numeric data, you can set format to the empty string (''). See “Example 3” on page 2-2915 below.

[A, B, ...] = stread('str', 'format', N, param, value, ...) customizes stread using param/value pairs, as listed in the table Parameters and Values for stread on page 2-2912 below. When str contains only numeric data, you can set format to the empty string (''). The N argument is optional and may be omitted entirely. See “Example 7” on page 2-2916 below.

Formats for stread

Format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a string that has Dept followed by a number (for department number), to skip the Dept and read only the number, use 'Dept' in the format string.	None
%d	Read a signed integer value.	Double array
%u	Read an integer value.	Double array
%f	Read a floating-point value.	Double array
%s	Read a white-space separated string.	Cell array of strings
%q	Read a double quoted string, ignoring the quotes.	Cell array of strings
%c	Read characters, including white space.	Character array
%[...]	Read the longest string containing characters specified in the brackets.	Cell array of strings

strread

Format	Action	Output
%[^...]	Read the longest nonempty string containing characters that are not specified in the brackets.	Cell array of strings
%*...	Ignore the characters following *. See “Example 8” on page 2-2916 below.	No output
%w...	Read field width specified by w. The %f format supports %w.pf, where w is the field width and p is the precision.	

Parameters and Values for strread

param	value	Action
whitespace	* where * can be	Treats vector of characters, *, as white space. Default is \b\r\n\t.
	b	Backspace
	f	Form feed
	n	New line
	r	Carriage return
	t	Horizontal tab
	\\	Backslash
	\' or \'	Single quotation mark
	%%	Percent sign

param	value	Action
delimiter	Delimiter character	Specifies delimiter character. Default is one or more whitespace characters.
expchars	Exponent characters	Default is eEdD.
bufsize	Positive integer	Specifies the maximum string length, in bytes. Default is 4095.
commentstyle	matlab	Ignores characters after %.
commentstyle	shell	Ignores characters after #.
commentstyle	c	Ignores characters between /* and */.
commentstyle	c++	Ignores characters after //.

Remarks

Delimiters

If your data uses a character other than a space as a delimiter, you must use the `stread` parameter `'delimiter'` to specify the delimiter. For example, if the string `str` used a semicolon as a delimiter, you would use this command:

```
[names, types, x, y, answer] = stread(str, '%s %s %f ...
    %d %s', 'delimiter', ';')
```

Formats

The format string determines the number and types of return arguments. The number of return arguments must match the number of conversion specifiers in the format string.

The `stread` function continues reading `str` until the entire string is read. If there are fewer format specifiers than there are entities in `str`, `stread` reapplies the format specifiers, starting over at the beginning. See “Example 5” on page 2-2915 below.

strread

The format string supports a subset of the conversion specifiers and conventions of the C language `fscanf` routine. White-space characters in the format string are ignored.

Preserving White-Space

If you want to preserve leading and trailing spaces in a string, use the `whitespace` parameter as shown here:

```
str = '  An example      of preserving      spaces      ';  
  
strread(str, '%s', 'whitespace', '')  
ans =  
      '  An example      of preserving      spaces      '
```

Examples

Example 1

Read numeric data into a 1-by-5 vector:

```
a = strread('0.41 8.24 3.57 6.24 9.27')  
a =  
    0.4100    8.2400    3.5700    6.2400    9.2700
```

Example 2

Read numeric data into separate scalar variables:

```
[a b c d e] = strread('0.41 8.24 3.57 6.24 9.27')  
a =  
    0.4100  
b =  
    8.2400  
c =  
    3.5700  
d =  
    6.2400  
e =  
    9.2700
```

Example 3

Read the only first three numbers in the string, also formatting as floating point:

```
a = strread('0.41 8.24 3.57 6.24 9.27', '%4.2f', 3)
```

```
a =
    0.4100
    8.2400
    3.5700
```

Example 4

Truncate the data to one decimal digit by specifying format %3.1f. The second specifier, %*1d, tells strread not to read in the remaining decimal digit:

```
a = strread('0.41 8.24 3.57 6.24 9.27', '%3.1f %*1d')
```

```
a =
    0.4000
    8.2000
    3.5000
    6.2000
    9.2000
```

Example 5

Read six numbers into two variables, reusing the format specifiers:

```
[a b] = strread('0.41 8.24 3.57 6.24 9.27 3.29', '%f %f')
```

```
a =
    0.4100
    3.5700
    9.2700
b =
    8.2400
```

strread

```
6.2400
3.2900
```

Example 6

Read string and numeric data to two output variables. Ignore commas in the input string:

```
str = 'Section 4, Page 7, Line 26';

[name value] = strread(str, '%s %d,')
name =
    'Section'
    'Page'
    'Line'
value =
     4
     7
    26
```

Example 7

Read the string used in the last example, but this time delimiting with commas instead of spaces:

```
str = 'Section 4, Page 7, Line 26';

[a b c] = strread(str, '%s %s %s', 'delimiter', ',')
a =
    'Section 4'
b =
    'Page 7'
c =
    'Line 26'
```

Example 8

Read selected portions of the input string:

```
str = '<table border=5 width=100% cellspacing=0>';
```

```
[border width space] = strread(str, ...
    '%*s%*s %c %*s %4s %*s %c', 'delimiter', '= ')
border =
    5
width =
    '100%'
space =
    0
```

Example 9

Read the string into two vectors, restricting the Answer values to T and F. Also note that two delimiters (comma and space) are used here:

```
str = 'Answer_1: T, Answer_2: F, Answer_3: F';

[a b] = strread(str, '%s %[TF]', 'delimiter', ', ', ' ')
a =
    'Answer_1:'
    'Answer_2:'
    'Answer_3:'
b =
    'T'
    'F'
    'F'
```

See Also

textscan, textread, sscanf

strrep

Purpose Find and replace substring

Syntax `str = strrep(str1, str2, str3)`

Description `str = strrep(str1, str2, str3)` replaces all occurrences of the string `str2` within string `str1` with the string `str3`.

`strrep(str1, str2, str3)`, when any of `str1`, `str2`, or `str3` is a cell array of strings, returns a cell array the same size as `str1`, `str2`, and `str3` obtained by performing a `strrep` using corresponding elements of the inputs. The inputs must all be the same size (or any can be a scalar cell). Any one of the strings can also be a character array with the right number of rows.

Examples

```
s1 = 'This is a good example.';
str = strrep(s1, 'good', 'great')
str =
This is a great example.
A =
    'MATLAB'      'SIMULINK'
    'Toolboxes'  'The MathWorks'
B =
    'Handle Graphics'  'Real Time Workshop'
    'Toolboxes'        'The MathWorks'
C =
    'Signal Processing'  'Image Processing'
    'MATLAB'             'SIMULINK'
strrep(A, B, C)
ans =
    'MATLAB' 'SIMULINK'
    'MATLAB' 'SIMULINK'
```

See Also `strfind`

Purpose

Selected parts of string

Syntax

```
token = strtok('str')
token = strtok('str', delimiter)
[token, remain] = strtok('str', ...)
```

Description

`token = strtok('str')` returns in `token` that part of the input string `str` that precedes the first white-space character (the default delimiter). Parsing of the string begins at the first nondelimiting (i.e., nonwhite-space) character and continues to the right until MATLAB either locates a delimiter or reaches the end of the string. If no delimiters are found in the body of the input string, then the entire string (excluding any leading delimiting characters) is returned.

White-space characters include space (ASCII 32), tab (ASCII 9), and carriage return (ASCII 13).

If `str` is a cell array of strings, `token` is a cell array of tokens.

`token = strtok('str', delimiter) [4]` is the same as the above syntax except that you can specify one or more nondefault delimiters in the character vector, `delimiter`. Ignoring any leading delimiters, MATLAB returns in `token` that part of the input string that precedes one of the characters from the given delimiter vector.

`[token, remain] = strtok('str', ...)` returns in `remain` a substring of the input string that begins immediately after the `token` substring and ends with the last character in `str`. If no delimiters are found in the body of the input string, then the entire string (excluding any leading delimiting characters) is returned in `token`, and `remain` is an empty string ('').

If `str` is a cell array of strings, `token` is a cell array of tokens and `remain` is a character array.

Examples**Example 1**

This example uses the default white-space delimiter:

```
s = ' This is a simple example.';
```

strtok

```
[token, remain] = strtok(s)
token =
    This
remain =
    is a simple example.
```

Example 2

Take a string of HTML code and break it down into segments delimited by the < and > characters. Write a while loop to parse the string and print each segment:

```
s = sprintf('%s%s%s', ...
    '<ul class=continued><li class=continued><pre>', ...
    '<a name=13474></a>token = strtok(''str'', delimiter)', ...
    '<a name=13475></a>token = strtok(''str'')');

remain = s;

while true
    [str, remain] = strtok(remain, '<>');
    if isempty(str), break; end
    disp(sprintf('%s', str))
end
```

Here is the output:

```
ul class=continued
li class=continued
pre
a name=13474
/a
token = strtok('str', delimiter)
a name=13475
/a
token = strtok('str')
```


Example 3

Using `strtok` on a cell array of strings returns a cell array of strings in `token` and a character array in `remain`:

```
s = {'all in good time'; ...  
    'my dog has fleas'; ...  
    'leave no stone unturned'};  
  
remain = s;  
  
for k = 1:4  
    [token, remain] = strtok(remain);  
    token  
end
```

Here is the output:

```
token =  
    'all'  
    'my'  
    'leave'  
token =  
    'in'  
    'dog'  
    'no'  
token =  
    'good'  
    'has'  
    'stone'  
token =  
    'time'  
    'fleas'  
    'unturned'
```

See Also

`findstr`, `strmatch`

strtrim

Purpose Remove leading and trailing white space from string

Syntax S = strtrim(str)
C = strtrim(cstr)

Description S = strtrim(str) returns a copy of string str with all leading and trailing white-space characters removed. A white-space character is one for which the isspace function returns logical 1 (true).

C = strtrim(cstr) returns a copy of the cell array of strings cstr with all leading and trailing white-space characters removed from each string in the cell array.

Examples Remove the leading white-space characters (spaces and tabs) from str:

```
str = sprintf(' \t Remove leading white-space')
str =
    Remove leading white-space
```

```
str = strtrim(str)
str =
Remove leading white-space
```

Remove leading and trailing white-space from the cell array of strings:

```
cstr = {' Trim leading white-space';
        'Trim trailing white-space '};

cstr = strtrim(cstr)
cstr =
    'Trim leading white-space'
    'Trim trailing white-space'
```

See Also isspace, cellstr, deblank, strjust

Purpose

Create structure array

Syntax

```
s = struct('field1', values1, 'field2', values2, ...)  
s = struct('field1', {}, 'field2', {}, ...)  
struct  
struct([])  
struct(obj)
```

Description

`s = struct('field1', values1, 'field2', values2, ...)` creates a structure array with the specified fields and values. Each value input (`values1`, `values2`, etc.), can either be a cell array or a scalar value. Those that are cell arrays must all have the same dimensions.

The size of the resulting structure is the same size as the value cell arrays, or 1-by-1 if none of the values is a cell array. Elements of the value array inputs are placed into corresponding structure array elements.

Note If any of the values fields is an empty cell array {}, MATLAB creates an empty structure array in which all fields are also empty.

Structure field names must begin with a letter, and are case-sensitive. The rest of the name may contain letters, numerals, and underscore characters. Use the `namelengthmax` function to determine the maximum length of a field name.

`s = struct('field1', {}, 'field2', {}, ...)` creates an empty structure with fields `field1`, `field2`, ...

`struct` creates a 1-by-1 structure with no fields.

`struct([])` creates an empty structure with no fields.

`struct(obj)` converts the object `obj` into its equivalent structure. The class information is lost.

Remarks

Two Ways to Access Fields

The most common way to access the data in a structure is by specifying the name of the field that you want to reference. Another means of accessing structure data is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run-time.

Fields That Are Cell Arrays

To create fields that contain cell arrays, place the cell arrays within a value cell array. For instance, to create a 1-by-1 structure, type

```
s = struct('strings',{ 'hello', 'yes' }, 'lengths', [5 3])
s =
    strings: { 'hello'  'yes' }
    lengths: [5 3]
```

Specifying Cell Versus Noncell Values

When using the syntax

```
s = struct('field1', values1, 'field2', values2, ...)
```

the values inputs can be cell arrays or scalar values. For those values that are specified as a cell array, MATLAB assigns each element of values{m,n,...} to the corresponding field in each element of structure s:

```
s(m,n,...).fieldN = valuesN{m,n,...}
```

For those values that are scalar, MATLAB assigns that single value to the corresponding field for all elements of structure s:

```
s(m,n,...).fieldN = valuesN
```

See Example 3, below.

Examples

Example 1

The command

```
s = struct('type', {'big','little'}, 'color', {'red'}, ...  
         'x', {3 4})
```

produces a structure array s:

```
s =  
1x2 struct array with fields:  
    type  
    color  
    x
```

The value arrays have been distributed among the fields of s:

```
s(1)  
ans =  
    type: 'big'  
    color: 'red'  
    x: 3  
  
s(2)  
ans =  
    type: 'little'  
    color: 'red'  
    x: 4
```

Example 2

Similarly, the command

```
a.b = struct('z', {});
```

produces an empty structure a.b with field z.

```
a.b  
ans =  
    0x0 struct array with fields:  
    z
```

Example 3

This example initializes one field `f1` using a cell array, and the other `f2` using a scalar value:

```
s = struct('f1', {1 3; 2 4}, 'f2', 25)
s =
2x2 struct array with fields:
    f1
    f2
```

Field `f1` in each element of `s` is assigned the corresponding value from the cell array `{1 3; 2 4}`:

```
s.f1
ans =
     1
ans =
     2
ans =
     3
ans =
     4
```

Field `f2` for all elements of `s` is assigned one common value because the values input for this field was specified as a scalar:

```
s.f2
ans =
    25
ans =
    25
ans =
    25
ans =
    25
```

See Also

isstruct, fieldnames, isfield, orderfields, getfield, setfield, rmfield, substruct, deal, cell2struct, struct2cell, namelengthmax, dynamic field names

struct2cell

Purpose Convert structure to cell array

Syntax `c = struct2cell(s)`

Description `c = struct2cell(s)` converts the m -by- n structure `s` (with p fields) into a p -by- m -by- n cell array `c`.

If structure `s` is multidimensional, cell array `c` has size `[p size(s)]`.

Examples The commands

```
clear s, s.category = 'tree';  
s.height = 37.4; s.name = 'birch';
```

create the structure

```
s =  
    category: 'tree'  
    height: 37.4000  
    name: 'birch'
```

Converting the structure to a cell array,

```
c = struct2cell(s)  
  
c =  
    'tree'  
    [37.4000]  
    'birch'
```

See Also `cell2struct`, `cell`, `iscell`, `struct`, `isstruct`, `fieldnames`, “Using Dynamic Field Names”

Purpose

Apply function to each field of scalar structure

Syntax

```
A = structfun(fun, S)
[A, B, ...] = structfun(fun, S)
[A, ...] = structfun(fun, S, 'param1', value1, ...)
```

Description

`A = structfun(fun, S)` applies the function specified by `fun` to each field of scalar structure `S`, and returns the results in array `A`. `fun` is a function handle to a function that takes one input argument and returns a scalar value. Return value `A` is a column vector that has one element for each field in input structure `S`. The `N`th element of `A` is the result of applying `fun` to the `N`th field of `S`, and the order of the fields is the same as that returned by a call to `fieldnames`.

`fun` must return values of the same class each time it is called. If `fun` is a handle to an overloaded function, then `structfun` follows MATLAB dispatching rules in calling the function.

`[A, B, ...] = structfun(fun, S)` returns arrays `A, B, ...`, each array corresponding to one of the output arguments of `fun`. `structfun` calls `fun` each time with as many outputs as there are in the call to `structfun`. `fun` can return output arguments having different classes, but the class of each output must be the same each time `fun` is called.

`[A, ...] = structfun(fun, S, 'param1', value1, ...)` enables you to specify optional parameter name/parameter value pairs. Parameters are

structfun

Parameter	Value
'UniformOutput'	<p>Logical value indicating whether or not the outputs of fun can be returned without encapsulation in a structure. The default value is true.</p> <p>If equal to logical 1 (true), fun must return scalar values that can be concatenated into an array. The outputs can be any of the following types: numeric, logical, char, struct, or cell.</p> <p>If equal to logical 0 (false), structfun returns a scalar structure or multiple scalar structures having fields that are the same as the fields of the input structure S. The values in the output structure fields are the results of calling fun on the corresponding values in the input structure B. In this case, the outputs can be of any data type.</p>
'ErrorHandler'	<p>Function handle specifying the function MATLAB is to call if the call to fun fails. MATLAB calls the error handling function with the following input arguments:</p> <ul style="list-style-type: none">• A structure, with the fields 'identifier', 'message', and 'index', respectively containing the identifier of the error that occurred, the text of the error message, and the number of the field (in the same order as returned by field names) at which the error occurred.• The input argument at which the call to the function failed. <p>The error handling function should either rethrow an error or return the same number of outputs as fun. These outputs are then returned as the outputs of structfun. If</p>

'UniformOutput' is true, the outputs of the error handler must also be scalars of the same type as the outputs of fun.

For example,

```
function [A, B] = errorFunc(S, ...  
    varargin)  
    warning(S.identifier, S.message);  
    A = NaN; B = NaN;
```

If an error handler is not specified, the error from the call to fun is rethrown.

Examples

To create shortened weekday names from the full names, for example:
Create a structure with strings in several fields:

```
s.f1 = 'Sunday';  
s.f2 = 'Monday';  
s.f3 = 'Tuesday';  
s.f4 = 'Wednesday';  
s.f5 = 'Thursday';  
s.f6 = 'Friday';  
s.f7 = 'Saturday';
```

```
shortNames = structfun(@(x) ( x(1:3) ), s, ...  
    'UniformOutput', false);
```

See Also

cellfun, arrayfun, function_handle, cell2mat, spfun

strvcat

Purpose Concatenate strings vertically

Syntax `S = strvcat(t1, t2, t3, ...)`
`S = strvcat(c)`

Description `S = strvcat(t1, t2, t3, ...)` forms the character array `S` containing the text strings (or string matrices) `t1, t2, t3, ...` as rows. Spaces are appended to each string as necessary to form a valid matrix. Empty arguments are ignored.

`S = strvcat(c)` when `c` is a cell array of strings, passes each element of `c` as an input to `strvcat`. Empty strings in the input are ignored.

Remarks If each text parameter, `ti`, is itself a character array, `strvcat` appends them vertically to create arbitrarily large string matrices.

Examples The command `strvcat('Hello', 'Yes')` is the same as `['Hello'; 'Yes']`, except that `strvcat` performs the padding automatically.

```
t1 = 'first'; t2 = 'string'; t3 = 'matrix'; t4 = 'second';
```

```
S1 = strvcat(t1, t2, t3)      S2 = strvcat(t4, t2, t3)
```

```
S1 =                          S2 =
```

```
first                          second
string                          string
matrix                          matrix
```

```
S3 = strvcat(S1, S2)
```

```
S3 =
first
string
matrix
second
string
```

matrix

See Also

strcat, cat, int2str, mat2str, num2str, strings

sub2ind

Purpose Single index from subscripts

Syntax
`IND = sub2ind(siz,I,J)`
`IND = sub2ind(siz,I1,I2,...,In)`

Description The `sub2ind` command determines the equivalent single index corresponding to a set of subscript values.

`IND = sub2ind(siz,I,J)` returns the linear index equivalent to the row and column subscripts `I` and `J` for a matrix of size `siz`. `siz` is a 2-element vector, where `siz(1)` is the number of rows and `siz(2)` is the number of columns.

`IND = sub2ind(siz,I1,I2,...,In)` returns the linear index equivalent to the `n` subscripts `I1,I2,...,In` for an array of size `siz`. `siz` is an `n`-element vector that specifies the size of each array dimension.

Examples Create a 3-by-4-by-2 array, `A`.

```
A = [17 24 1 8; 2 22 7 14; 4 6 13 20];  
A(:,:,2) = A - 10
```

```
A(:,:,1) =  
  
    17    24     1     8  
     2    22     7    14  
     4     6    13    20
```

```
A(:,:,2) =  
  
     7    14    -9    -2  
    -8    12    -3     4  
    -6    -4     3    10
```

The value at row 2, column 1, page 2 of the array is -8.

```
A(2,1,2)
```

```
ans =
```

```
-8
```

To convert `A(2,1,2)` into its equivalent single subscript, use `sub2ind`.

```
sub2ind(size(A),2,1,2)
```

```
ans =
```

```
14
```

You can now access the same location in `A` using the single subscripting method.

```
A(14)
```

```
ans =
```

```
-8
```

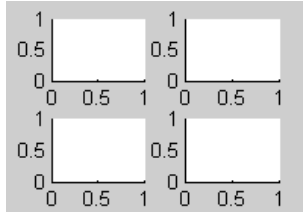
See Also

`ind2sub`, `find`, `size`

subplot

Purpose

Create axes in tiled positions



GUI Alternatives

To add subplots to figures, click one of the *New Subplot* icons in the Figure Palette, and slide right to select an arrangement of subplots. For details, see [Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation](#).

Syntax

```
h = subplot(m,n,p) or subplot(mnp)
subplot(m,n,p,'replace')
subplot(m,n,p,'v6')
subplot(h)
subplot('Position',[left bottom width height])
h = subplot(...)
```

Description

`subplot` divides the current figure into rectangular panes that are numbered rowwise. Each pane contains an axes object. Subsequent plots are output to the current pane.

`h = subplot(m,n,p)` or `subplot(mnp)` breaks the figure window into an m -by- n matrix of small axes, selects the p th axes object for the current plot, and returns the axes handle. The axes are counted along the top row of the figure window, then the second row, etc. For example,

```
subplot(2,1,1), plot(income)
subplot(2,1,2), plot(outgo)
```

plots `income` on the top half of the window and `outgo` on the bottom half. If the `CurrentAxes` is nested in a `uipanel`, the panel is used as

the parent for the subplot instead of the current figure. The new axes object becomes the current axes.

If `p` is a vector, it specifies an axes object having a position that covers all the subplot positions listed in `p`.

`subplot(m,n,p, 'replace')`, If the specified axes object already exists, delete it and create a new axes.

`subplot(m,n,p, 'v6')` places the axes so that the plot boxes are aligned, but does not prevent the labels and ticks from overlapping. Saved subplots created with the `v6` option are compatible with MATLAB 6.5 and earlier versions.

`subplot(h)` makes the axes object with handle `h` current for subsequent plotting commands.

`subplot('Position',[left bottom width height])` creates an axes at the position specified by a four-element vector. `left`, `bottom`, `width`, and `height` are in normalized coordinates in the range from 0.0 to 1.0.

`h = subplot(...)` returns the handle to the new axes object.

Backwards Compatibility

Use the subplot `'v6'` option and save the figure with the `'v6'` option when you want to be able to load a FIG-file containing subplots into MATLAB Version 6.5 or earlier.

Remarks

If a subplot specification causes a new axes object to overlap any existing axes, then `subplot` deletes the existing axes object and `uicontrol` objects. However, if the subplot specification exactly matches the position of an existing axes object, then the matching axes object is not deleted and it becomes the current axes.

`subplot(1,1,1)` or `clf` deletes all axes objects and returns to the default `subplot(1,1,1)` configuration.

You can omit the parentheses and specify subplot as

```
subplot mnp
```

subplot

where m refers to the row, n refers to the column, and p specifies the pane.

Be aware when creating subplots from scripts that the `Position` property of subplots is not finalized until either

- A `drawnow` command is issued.
- MATLAB returns to await a user command.

That is, the value obtained for subplot i by the command

```
get(h(i), 'position')
```

will not be correct until the script refreshes the plot or exits.

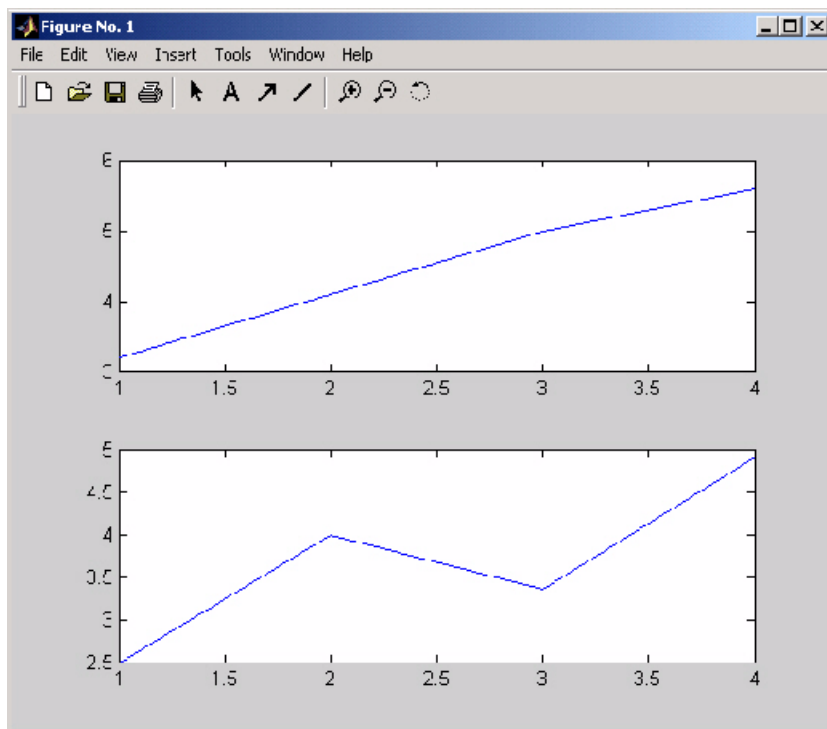
Special Case: `subplot(111)`

The command `subplot(111)` is not identical in behavior to `subplot(1,1,1)` and exists only for compatibility with previous releases. This syntax does not immediately create an axes object, but instead sets up the figure so that the next graphics command executes a `clf reset` (deleting all figure children) and creates a new axes object in the default position. This syntax does not return a handle, so it is an error to specify a return argument. (MATLAB implements this behavior by setting the figure's `NextPlot` property to `replace`.)

Examples

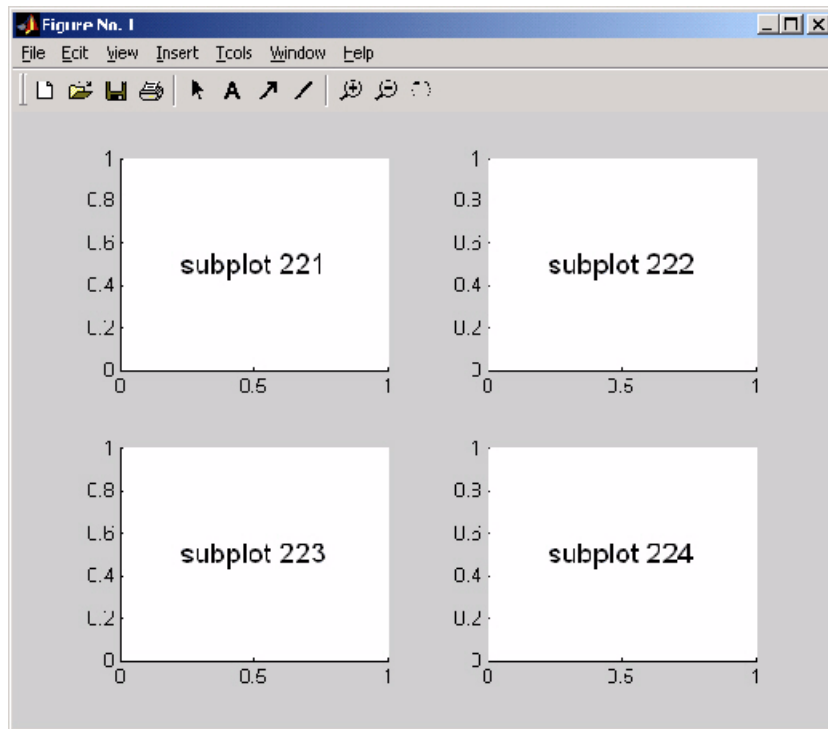
To plot `income` in the top half of a figure and `outgo` in the bottom half,

```
income = [3.2 4.1 5.0 5.6];  
outgo = [2.5 4.0 3.35 4.9];  
subplot(2,1,1); plot(income)  
subplot(2,1,2); plot(outgo)
```



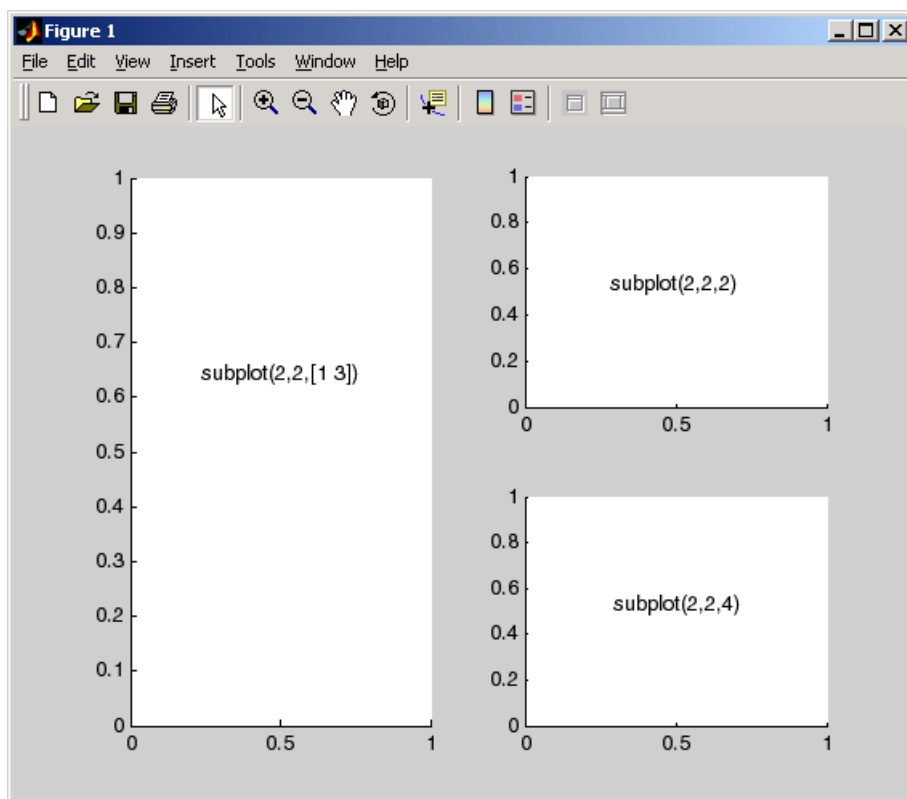
The following illustration shows four subplot regions and indicates the command used to create each.

subplot



The following combinations produce asymmetrical arrangements of subplots.

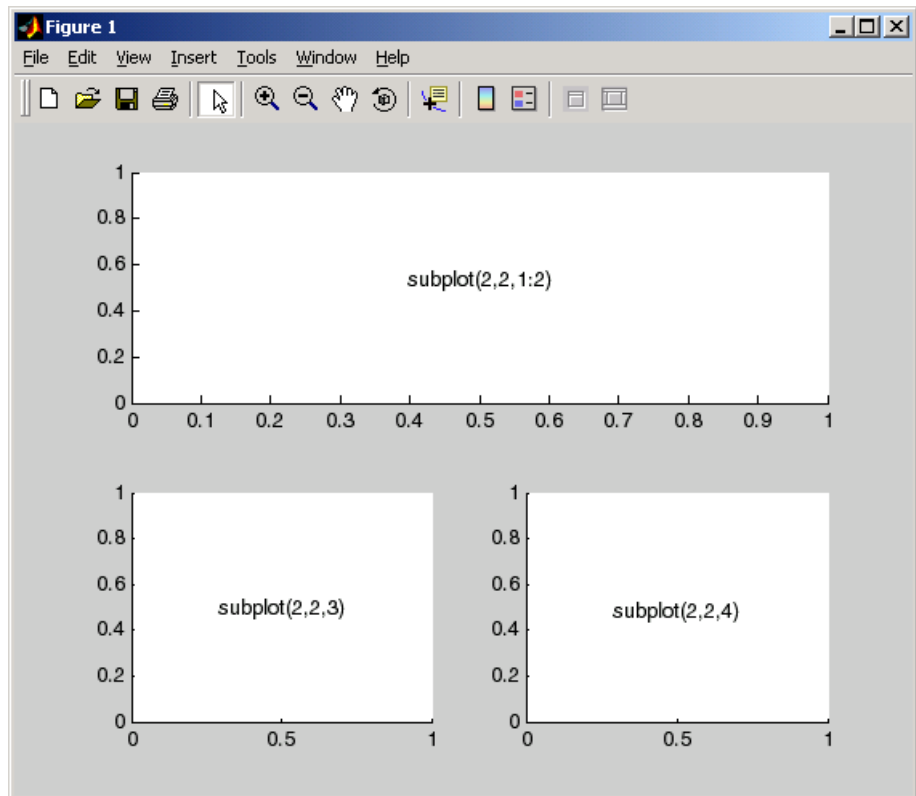
```
subplot(2,2,[1 3])  
subplot(2,2,2)  
subplot(2,2,4)
```



You can also use the colon operator to specify multiple locations if they are in sequence.

```
subplot(2,2,1:2)  
subplot(2,2,3)  
subplot(2,2,4)
```

subplot



See Also

`axes`, `cla`, `clf`, `figure`, `gca`

“Basic Plots and Graphs” on page 1-82 for more information

Purpose	Subscripted assignment for objects
Syntax	$A = \text{subsasgn}(A, S, B)$
Description	<p>$A = \text{subsasgn}(A, S, B)$ is called for the syntax $A(i)=B$, $A\{i\}=B$, or $A.i=B$ when A is an object. S is a structure array with the fields</p> <ul style="list-style-type: none">• type: A string containing '()', '{}', or '.', where '()' specifies integer subscripts, '{}' specifies cell array subscripts, and '.' specifies subscripted structure fields.• subs: A cell array or string containing the actual subscripts.
Remarks	<p><code>subsasgn</code> is designed to be used by the MATLAB interpreter to handle indexed assignments to objects. Calling <code>subsasgn</code> directly as a function is not recommended. If you do use <code>subsasgn</code> in this way, it conforms to the formal MATLAB dispatching rules and can yield unexpected results.</p> <p>In the assignment $A(J,K,\dots) = B(M,N,\dots)$, subscripts J, K, M, N, etc. may be scalar, vector, or array, provided that all of the following are true:</p> <ul style="list-style-type: none">• The number of subscripts specified for B, excluding trailing subscripts equal to 1, does not exceed <code>ndims(B)</code>.• The number of nonscalar subscripts specified for A equals the number of nonscalar subscripts specified for B. For example, $A(5, 1:4, 1, 2) = B(5:8)$ is valid because both sides of the equation use one nonscalar subscript.• The order and length of all nonscalar subscripts specified for A matches the order and length of nonscalar subscripts specified for B. For example, $A(1:4, 3, 3:9) = B(5:8, 1:7)$ is valid because both sides of the equation (ignoring the one scalar subscript 3) use a 4-element subscript followed by a 7-element subscript. <p>See the Remarks section of the <code>numel</code> reference page for information concerning the use of <code>numel</code> with regards to the overloaded <code>subsasgn</code> function.</p>

subsasgn

If `A` is an array of one of the fundamental MATLAB data types, then assigning a value to `A` with indexed assignment calls the builtin MATLAB `subsasgn` method. It does not call any `subsasgn` method that you may have overloaded for that data type. For example, if `A` is an array of type `double`, and there is an `@double/subsasgn` method on your MATLAB path, the statement `A(I) = B` does not call this method, but calls the MATLAB builtin `subsasgn` method instead.

Examples

The syntax `A(1:2,:)=B` calls `A=subsasgn(A,S,B)` where `S` is a 1-by-1 structure with `S.type=()'` and `S.subs = {1:2, ':'}`. A colon used as a subscript is passed as the string `' : '`.

The syntax `A{1:2}=B` calls `A=subsasgn(A,S,B)` where `S.type='{}'`.

The syntax `A.field=B` calls `subsasgn(A,S,B)` where `S.type='.'` and `S.subs='field'`.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)=B` calls `A=subsasgn(A,S,B)` where `S` is a 3-by-1 structure array with the following values:

<code>S(1).type=()'</code>	<code>S(2).type='.'</code>	<code>S(3).type=()'</code>
<code>S(1).subs={1,2}</code>	<code>S(2).subs='name'</code>	<code>S(3).subs={3:5}</code>

See Also

`subsref`, `substruct`

See “Handling Subscripted Assignment” for more information about overloaded methods and `subsasgn`.

Purpose Subscripted indexing for objects

Syntax `ind = subsindex(A)`

Description `ind = subsindex(A)` is called for the syntax '`X(A)`' when `A` is an object. `subsindex` must return the value of the object as a zero-based integer index. (`ind` must contain integer values in the range 0 to `prod(size(X))-1`.) `subsindex` is called by the default `subsref` and `subsasgn` functions, and you can call it if you overload these functions.

See Also `subsasgn`, `subsref`

subspace

Purpose Angle between two subspaces

Syntax `theta = subspace(A,B)`

Description `theta = subspace(A,B)` finds the angle between two subspaces specified by the columns of A and B. If A and B are column vectors of unit length, this is the same as $\text{acos}(A' * B)$.

Remarks If the angle between the two subspaces is small, the two spaces are nearly linearly dependent. In a physical experiment described by some observations A, and a second realization of the experiment described by B, `subspace(A,B)` gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.

Examples Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.

```
H = hadamard(8);
A = H(:,2:4);
B = H(:,5:8);
```

Note that matrices A and B are different sizes — A has three columns and B four. It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

```
theta = subspace(A,B)
theta =
    1.5708
```

That A and B are orthogonal is shown by the fact that theta is equal to $\pi/2$.

```
theta - pi/2
ans =
    0
```

Purpose	Subscripted reference for objects
Syntax	<code>B = subsref(A, S)</code>
Description	<p><code>B = subsref(A, S)</code> is called for the syntax <code>A(i)</code>, <code>A{i}</code>, or <code>A.i</code> when <code>A</code> is an object. <code>S</code> is a structure array with the fields</p> <ul style="list-style-type: none">• <code>type</code>: A string containing <code>'()'</code>, <code>'{}'</code>, or <code>'.'</code>, where <code>'()'</code> specifies integer subscripts, <code>'{}'</code> specifies cell array subscripts, and <code>'.'</code> specifies subscripted structure fields.• <code>subs</code>: A cell array or string containing the actual subscripts.
Remarks	<p><code>subsref</code> is designed to be used by the MATLAB interpreter to handle indexed references to objects. Calling <code>subsref</code> directly as a function is not recommended. If you do use <code>subsref</code> in this way, it conforms to the formal MATLAB dispatching rules and can yield unexpected results.</p> <p>See the Remarks section of the <code>numel</code> reference page for information concerning the use of <code>numel</code> with regards to the overloaded <code>subsref</code> function.</p> <p>If <code>A</code> is an array of one of the fundamental MATLAB data types, then referencing a value of <code>A</code> using an indexed reference calls the builtin MATLAB <code>subsref</code> method. It does not call any <code>subsref</code> method that you may have overloaded for that data type. For example, if <code>A</code> is an array of type <code>double</code>, and there is an <code>@double/subsref</code> method on your MATLAB path, the statement <code>B = A(I)</code> does not call this method, but calls the MATLAB builtin <code>subsref</code> method instead.</p>
Examples	<p>The syntax <code>A(1:2,:)</code> calls <code>subsref(A,S)</code> where <code>S</code> is a 1-by-1 structure with <code>S.type='()'</code> and <code>S.subs={1:2, ':'}</code>. A colon used as a subscript is passed as the string <code>':'</code>.</p> <p>The syntax <code>A{1:2}</code> calls <code>subsref(A,S)</code> where <code>S.type='{}'</code> and <code>S.subs={1:2}</code>.</p> <p>The syntax <code>A.field</code> calls <code>subsref(A,S)</code> where <code>S.type='.'</code> and <code>S.subs='field'</code>.</p>

subsref

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)` calls `subsref(A,S)` where `S` is a 3-by-1 structure array with the following values:

```
S(1).type='()'      S(2).type='.'      S(3).type='()'
S(1).subs={1,2}    S(2).subs='name'   S(3).subs={3:5}
```

See Also

`subsasgn`, `substruct`

See “Handling Subscripted Reference” for more information about overloaded methods and `subsref`.

Purpose Create structure argument for subsasgn or subsref

Syntax `S = substruct(type1, subs1, type2, subs2, ...)`

Description `S = substruct(type1, subs1, type2, subs2, ...)` creates a structure with the fields required by an overloaded subsref or subsasgn method. Each type string must be one of `'.'`, `'()'` , or `'{'}`'. The corresponding subs argument must be either a field name (for the `'.'` type) or a cell array containing the index vectors (for the `'()'` or `'{'}` types).

The output `S` is a structure array containing the fields

- `type`: one of `'.'`, `'()'` , or `'{'}`'
- `subs`: subscript values (field name or cell array of index vectors)

Examples To call subsref with parameters equivalent to the syntax

```
B = A(3,5).field
```

you can use

```
S = substruct('()', {3,5}, '.', 'field');
B = subsref(A, S);
```

The structure created by substruct in this example contains the following:

```
S(1)

ans =

    type: '()'
    subs: {[3] [5]}

S(2)
```

substruct

```
ans =  
    type: '.'  
    subs: 'field'
```

See Also

subsasgn, subsref

Purpose Extract subset of volume data set

Syntax

```
[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)
[Nx,Ny,Nz,Nv] = subvolume(V,limits)
Nv = subvolume(...)
```

Description [Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits) extracts a subset of the volume data set V using the specified axis-aligned limits. limits = [xmin,xmax,ymin,ymax,zmin,zmax] (Any NaNs in the limits indicate that the volume should not be cropped along that axis.)

The arrays X, Y, and Z define the coordinates for the volume V. The subvolume is returned in NV and the coordinates of the subvolume are given in NX, NY, and NZ.

[Nx,Ny,Nz,Nv] = subvolume(V,limits) assumes the arrays X, Y, and Z are defined as

```
[X,Y,Z] = meshgrid(1:N,1:M,1:P)
```

where [M,N,P] = size(V).

Nv = subvolume(...) returns only the subvolume.

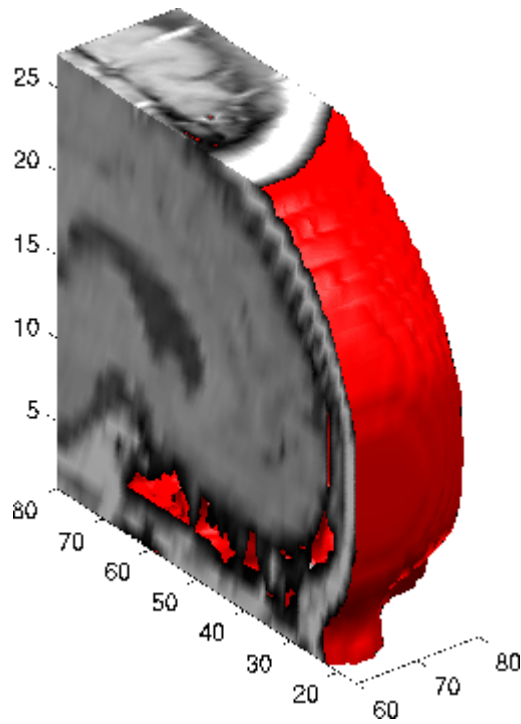
Examples This example uses a data set that is a collection of MRI slices of a human skull. The data is processed in a variety of ways:

- The 4-D array is squeezed (squeeze) into three dimensions and then a subset of the data is extracted (subvolume).
- The outline of the skull is an isosurface generated as a patch (p1) whose vertex normals are recalculated to improve the appearance when lighting is applied (patch, isosurface, isonormals).
- A second patch (p2) with interpolated face color draws the end caps (FaceColor, isocaps).
- The view of the object is set (view, axis, daspect).

subvolume

- A 100-element grayscale colormap provides coloring for the end caps (colormap).
- Adding lights to the right and left of the camera illuminates the object (camlight, lighting).

```
load mri
D = squeeze(D);
[x,y,z,D] = subvolume(D,[60,80,nan,80,nan,nan]);
p1 = patch(isosurface(x,y,z,D, 5),...
           'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
           'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight right; camlight left; lighting gouraud
```

See Also

isocaps, isonormals, isosurface, reducepatch, reducevolume, smooth3

“Volume Visualization” on page 1-98 for related functions

sum

Purpose Sum of array elements

Syntax

```
B = sum(A)
B = sum(A,dim)
B = sum(..., 'double')
B = sum(..., dim,'double')
B = sum(..., 'native')
B = sum(..., dim,'native')
```

Description `B = sum(A)` returns sums along different dimensions of an array.

If `A` is a vector, `sum(A)` returns the sum of the elements.

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

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

`B = sum(A,dim)` sums along the dimension of `A` specified by scalar `dim`. The `dim` input is an integer value from 1 to `N`, where `N` is the number of dimensions in `A`. Set `dim` to 1 to compute the sum of each column, 2 to sum rows, etc.

`B = sum(..., 'double')` and `B = sum(..., dim,'double')` performs additions in double-precision and return an answer of type `double`, even if `A` has data type `single` or an integer data type. This is the default for integer data types.

`B = sum(..., 'native')` and `B = sum(..., dim,'native')` performs additions in the native data type of `A` and return an answer of the same data type. This is the default for `single` and `double`.

Remarks `sum(diag(X))` is the trace of `X`.

Examples The magic square of order 3 is

```
M = magic(3)
M =
```

```

      8     1     6
      3     5     7
      4     9     2

```

This is called a magic square because the sums of the elements in each column are the same.

```

sum(M) =
      15     15     15

```

as are the sums of the elements in each row, obtained either by:

- Transposing

```

sum(M') =
      15     15     15

```

- Using the `dim` argument

```

sum(M,1) =
      15
      15
      15

```

transposing:

Nondouble Data Type Support

This section describes the support of `sum` for data types other than `double`.

Data Type `single`

You can apply `sum` to an array of type `single` and MATLAB returns an answer of type `single`. For example,

```

sum(single([2 5 8]))

ans =

      15

```

sum

```
class(ans)
```

```
ans =
```

```
single
```

Integer Data Types

When you apply `sum` to any of the following integer data types, MATLAB returns an answer of type double:

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

For example,

```
sum(single([2 5 8]));
```

```
class(ans)
```

```
ans =
```

```
single
```

If you want MATLAB to perform additions on an integer data type in the same integer type as the input, use the syntax

```
sum(int8([2 5 8], 'native');
```

```
class(ans)
```

```
ans =
```

```
int8
```

See Also

`accumarray`, `cumsum`, `diff`, `isfloat`, `prod`

Purpose Sum of timeseries data

Syntax

```
ts_sm = sum(ts)
ts_sm = sum(ts, 'PropertyName1', PropertyValue1, ...)
```

Description `ts_sm = sum(ts)` returns the sum of the time-series data. When `ts.Data` is a vector, `ts_sm` is the sum of `ts.Data` values. When `ts.Data` is a matrix, `ts_sm` is a row vector containing the sum of each column of `ts.Data` (when `IsTimeFirst` is true and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `sum` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_sm = sum(ts, 'PropertyName1', PropertyValue1, ...)` specifies the following optional input arguments:

- 'MissingData' property has two possible values, 'remove' (default) or 'interpolate', indicating how to treat missing data during the calculation.
- 'Quality' values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- 'Weighting' property has two possible values, 'none' (default) or 'time'. When you specify 'time', larger time values correspond to larger weights.

Examples 1 Load a 24-by-3 data array.

```
load count.dat
```

2 Create a timeseries object with 24 time values.

```
count_ts = timeseries(count,1:24,'Name','CountPerSecond')
```

3 Calculate the sum of each data column for this timeseries object.

```
sum(count_ts)
```

sum (timeseries)

```
ans =
```

```
       768       1117       1574
```

The sum is calculated independently for each data column in the `timeseries` object.

See Also

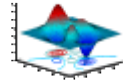
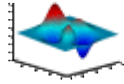
```
iqr (timeseries), mean (timeseries), median (timeseries), std  
(timeseries), var (timeseries), timeseries
```

Purpose	Establish superior class relationship
Syntax	<code>superiorto('class1', 'class2', ...)</code>
Description	<p>The <code>superiorto</code> function establishes a hierarchy that determines the order in which MATLAB calls object methods.</p> <p><code>superiorto('class1', 'class2', ...)</code> invoked within a class constructor method (say <code>myclass.m</code>) indicates that <code>myclass</code>'s method should be invoked if a function is called with an object of class <code>myclass</code> and one or more objects of class <code>class1</code>, <code>class2</code>, and so on.</p>
Remarks	<p>Suppose A is of class <code>'class_a'</code>, B is of class <code>'class_b'</code> and C is of class <code>'class_c'</code>. Also suppose the constructor <code>class_c.m</code> contains the statement <code>superiorto('class_a')</code>. Then <code>e = fun(a,c)</code> or <code>e = fun(c,a)</code> invokes <code>class_c/fun</code>.</p> <p>If a function is called with two objects having an unspecified relationship, the two objects are considered to have equal precedence, and the leftmost object's method is called. So <code>fun(b,c)</code> calls <code>class_b/fun</code>, while <code>fun(c,b)</code> calls <code>class_c/fun</code>.</p>
See Also	<code>inferiorto</code>


support

Purpose	Open MathWorks Technical Support Web page
Syntax	support
Description	<p>support opens the MathWorks Technical Support Web page, http://www.mathworks.com/support, in the MATLAB Web browser.</p> <p>This Web page contains resources including</p> <ul style="list-style-type: none">• A search engine, including an option for solutions to common problems• Information about installation and licensing• A patch archive for bug fixes you can download• Other useful resources
See Also	doc, web

Purpose 3-D shaded surface plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
surf(Z)
surf(Z,C)
surf(X,Y,Z)
surf(X,Y,Z,C)
surf(...,'PropertyName',PropertyValue)
surf(axes_handles,...)
surfc(...)
h = surf(...)
hsurface = surf('v6',...)
```

Description

Use `surf` and `surfc` to view mathematical functions over a rectangular region. `surf` and `surfc` create colored parametric surfaces specified by X , Y , and Z , with color specified by Z or C .

`surf(Z)` creates a three-dimensional shaded surface from the z components in matrix Z , using $x = 1:n$ and $y = 1:m$, where $[m,n] = \text{size}(Z)$. The height, Z , is a single-valued function defined over a geometrically rectangular grid. Z specifies the color data as well as surface height, so color is proportional to surface height.

`surf(Z,C)` plots the height of Z , a single-valued function defined over a geometrically rectangular grid, and uses matrix C , assumed to be the same size as Z , to color the surface.

`surf(X,Y,Z)` creates a shaded surface using Z for the color data as well as surface height. X and Y are vectors or matrices defining the x and y components of a surface. If X and Y are vectors, $\text{length}(X) = n$ and $\text{length}(Y) = m$, where $[m,n] = \text{size}(Z)$. In this case, the vertices of the surface faces are $(X(j), Y(i), Z(i,j))$ triples.

`surf(X,Y,Z,C)` creates a shaded surface, with color defined by C . MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`surf(..., 'PropertyName', PropertyValue)` specifies surface properties along with the data.

`surf(axes_handles,...)` and `surfc(axes_handles,...)` plot into the axes with handle `axes_handle` instead of the current axes (`gca`).

`surfc(...)` draws a contour plot beneath the surface.

`h = surf(...)` and `h = surfc(...)` return a handle to a surfaceplot graphics object.

Backward-Compatible Version

`hsurface = surf('v6',...)` and `hsurface = surfc('v6',...)` return the handles of surface objects instead of surfaceplot objects for compatibility with MATLAB 6.5 and earlier.

Algorithm

Abstractly, a parametric surface is parameterized by two independent variables, i and j , which vary continuously over a rectangle; for example, $1 \leq i \leq m$ and $1 \leq j \leq n$. The three functions $x(i,j)$, $y(i,j)$, and $z(i,j)$ specify the surface. When i and j are integer values, they define a rectangular grid with integer grid points. The functions $x(i,j)$, $y(i,j)$, and $z(i,j)$ become three m -by- n matrices, X , Y , and Z . Surface color is a fourth function, $c(i,j)$, denoted by matrix C .

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

$$\begin{array}{ccccc} & & i & 1, & j \\ & & & | & \\ i, & j & 1 & i, & j & i, & j+1 \end{array}$$

$$\begin{array}{c} | \\ i+1, j \end{array}$$

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, `[X(:) Y(:) Z(:)]` returns a list of triples specifying points in 3-space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors; the four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

Surface color can be specified in two different ways: at the vertices or at the centers of each patch. In this general setting, the surface need not be a single-valued function of x and y . Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The shading function sets the shading. If the shading is `interp`, `C` must be the same size as `X`, `Y`, and `Z`; it specifies the colors at the vertices. The color within a surface patch is a bilinear function of the local coordinates. If the shading is `faceted` (the default) or `flat`, `C(i, j)` specifies the constant color in the surface patch:

$$\begin{array}{ccc} (i, j) & & (i, j+1) \\ | & C(i, j) & | \\ (i+1, j) & & (i+1, j+1) \end{array}$$

In this case, `C` can be the same size as `X`, `Y`, and `Z` and its last row and column are ignored. Alternatively, its row and column dimensions can be one less than those of `X`, `Y`, and `Z`.

The `surf` and `surfc` functions specify the viewpoint using `view(3)`.

The range of `X`, `Y`, and `Z` or the current setting of the axes `XLimMode`, `YLimMode`, and `ZLimMode` properties (also set by the `axis` function) determines the axis labels.

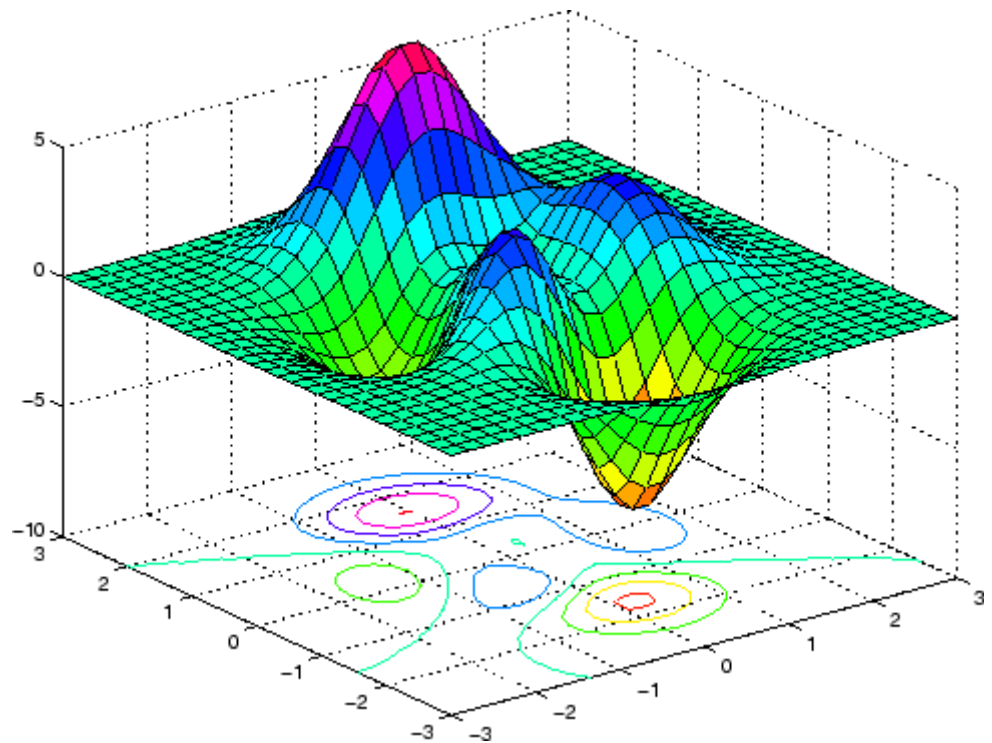
The range of `C` or the current setting of the axes `CLim` and `CLimMode` properties (also set by the `axis` function) determines the color scaling. The scaled color values are used as indices into the current `colormap`.

surf, surfc

Examples

Display a surfaceplot and contour plot of the peaks surface.

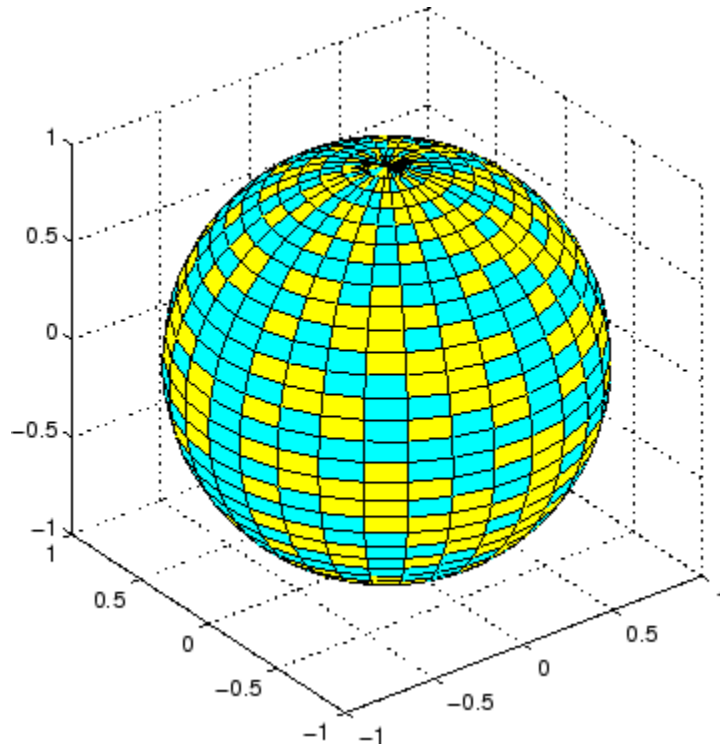
```
[X,Y,Z] = peaks(30);  
surfc(X,Y,Z)  
colormap hsv  
axis([-3 3 -3 3 -10 5])
```



Color a sphere with the pattern of +1s and -1s in a Hadamard matrix.

```
k = 5;  
n = 2^k - 1;  
[x,y,z] = sphere(n);  
c = hadamard(2^k);  
surf(x,y,z,c);
```

```
colormap([1 1 0; 0 1 1])  
axis equal
```



See Also

[axis](#), [caxis](#), [colormap](#), [contour](#), [delaunay](#), [mesh](#), [pcolor](#), [shading](#), [trisurf](#), [view](#)

Properties for surfaceplot graphics objects

“Creating Surfaces and Meshes” on page 1-94 for related functions

Representing a Matrix as a Surface for more examples

Coloring Mesh and Surface Plots for information about how to control the coloring of surfaces

surf2patch

Purpose Convert surface data to patch data

Syntax

```
fvc = surf2patch(Z)
fvc = surf2patch(Z,C)
fvc = surf2patch(X,Y,Z)
fvc = surf2patch(X,Y,Z,C)
fvc = surf2patch(...,'triangles')
[f,v,c] = surf2patch(...)
```

Description `fvc = surf2patch(h)` converts the geometry and color data from the surface object identified by the handle `h` into patch format and returns the face, vertex, and color data in the struct `fvc`. You can pass this struct directly to the `patch` command.

`fvc = surf2patch(Z)` calculates the patch data from the surface's ZData matrix `Z`.

`fvc = surf2patch(Z,C)` calculates the patch data from the surface's ZData and CData matrices `Z` and `C`.

`fvc = surf2patch(X,Y,Z)` calculates the patch data from the surface's XData, YData, and ZData matrices `X`, `Y`, and `Z`.

`fvc = surf2patch(X,Y,Z,C)` calculates the patch data from the surface's XData, YData, ZData, and CData matrices `X`, `Y`, `Z`, and `C`.

`fvc = surf2patch(...,'triangles')` creates triangular faces instead of the quadrilaterals that compose surfaces.

`[f,v,c] = surf2patch(...)` returns the face, vertex, and color data in the three arrays `f`, `v`, and `c` instead of a struct.

Examples The first example uses the `sphere` command to generate the XData, YData, and ZData of a surface, which is then converted to a patch. Note that the ZData (`z`) is passed to `surf2patch` as both the third and fourth arguments — the third argument is the ZData and the fourth argument is taken as the CData. This is because the `patch` command does not

automatically use the z -coordinate data for the color data, as does the `surface` command.

Also, because `patch` is a low-level command, you must set the view to 3-D and shading to faceted to produce the same results produced by the `surf` command.

```
[x y z] = sphere;  
patch(surf2patch(x,y,z,z));  
shading faceted; view(3)
```

In the second example `surf2patch` calculates face, vertex, and color data from a surface whose handle has been passed as an argument.

```
s = surf(peaks);  
pause  
patch(surf2patch(s));  
delete(s)  
shading faceted; view(3)
```

See Also

`patch`, `reducepatch`, `shrinkfaces`, `surface`, `surf`

“Volume Visualization” on page 1-98 for related functions

surface

Purpose

Create surface object

Syntax

```
surface(Z)
surface(Z,C)
surface(X,Y,Z)
surface(X,Y,Z,C)
surface(x,y,Z)
surface(... 'PropertyName',PropertyValue,...)
h = surface(...)
```

Description

`surface` is the low-level function for creating surface graphics objects. Surfaces are plots of matrix data created using the row and column indices of each element as the x - and y -coordinates and the value of each element as the z -coordinate.

`surface(Z)` plots the surface specified by the matrix Z . Here, Z is a single-valued function, defined over a geometrically rectangular grid.

`surface(Z,C)` plots the surface specified by Z and colors it according to the data in C (see Examples).

`surface(X,Y,Z)` uses $C = Z$, so color is proportional to surface height above the x - y plane.

`surface(X,Y,Z,C)` plots the parametric surface specified by X , Y , and Z , with color specified by C .

`surface(x,y,Z)`, `surface(x,y,Z,C)` replaces the first two matrix arguments with vectors and must have $\text{length}(x) = n$ and $\text{length}(y) = m$ where $[m,n] = \text{size}(Z)$. In this case, the vertices of the surface facets are the triples $(x(j), y(i), Z(i,j))$. Note that x corresponds to the columns of Z and y corresponds to the rows of Z . For a complete discussion of parametric surfaces, see the `surf` function.

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

`h = surface(...)` returns a handle to the created surface object.

Remarks

surface does not respect the settings of the figure and axes NextPlot properties. It simply adds the surface object to the current axes.

If you do not specify separate color data (C), MATLAB uses the matrix (Z) to determine the coloring of the surface. In this case, color is proportional to values of Z. You can specify a separate matrix to color the surface independently of the data defining the area of the surface.

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

surface provides convenience forms that allow you to omit the property name for the XData, YData, ZData, and CData properties. For example,

```
surface('XData',X,'YData',Y,'ZData',Z,'CData',C)
```

is equivalent to

```
surface(X,Y,Z,C)
```

When you specify only a single matrix input argument,

```
surface(Z)
```

MATLAB assigns the data properties as if you specified

```
surface('XData',[1:size(Z,2)],...  
        'YData',[1:size(Z,1)],...  
        'ZData',Z,...  
        'CData',Z)
```

The axis, caxis, colormap, hold, shading, and view commands set graphics properties that affect surfaces. You can also set and query surface property values after creating them using the set and get commands.

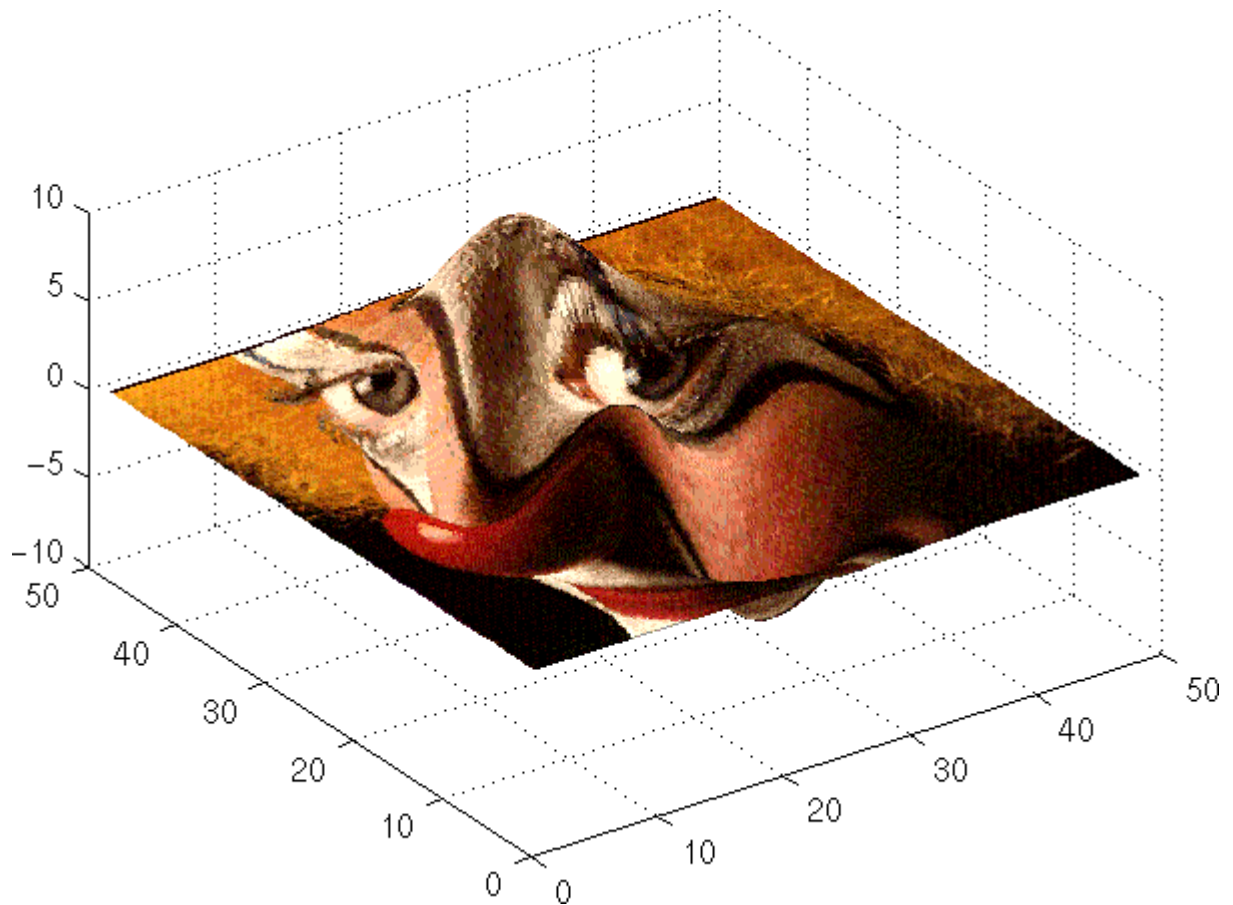
Example

This example creates a surface using the peaks M-file to generate the data, and colors it using the clown image. The ZData is a 49-by-49

surface

element matrix, while the CData is a 200-by-320 matrix. You must set the surface's FaceColor to texturemap to use ZData and CData of different dimensions.

```
load clown
surface(peaks,flipud(X),...
        'FaceColor','texturemap',...
        'EdgeColor','none',...
        'CDataMapping','direct')
colormap(map)
view(-35,45)
```

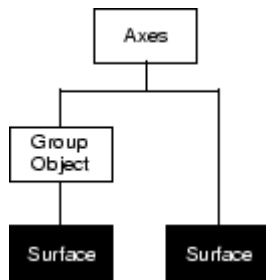


Note the use of the `surface(Z,C)` convenience form combined with property name/property value pairs.

Since the clown data (`X`) is typically viewed with the `image` command, which MATLAB normally displays with 'ij' axis numbering and `direct CDataMapping`, this example reverses the data in the vertical direction using `flipud` and sets the `CDataMapping` property to `direct`.

surface

Object Hierarchy



Setting Default Properties

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

```
set(0, 'DefaultSurfaceProperty', PropertyValue...)  
set(gcf, 'DefaultSurfaceProperty', PropertyValue...)  
set(gca, 'DefaultSurfaceProperty', PropertyValue...)
```

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

See Also

`ColorSpec`, `patch`, `pcolor`, `surf`

Representing a Matrix as a Surface for examples

“Creating Surfaces and Meshes” on page 1-94 and “Object Creation Functions” on page 1-90 for related functions

Surface Properties for property descriptions

Purpose

Defines the surface properties.

Modifying Properties

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

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

To change the default values of properties, see [Setting Default Property Values](#).

See “Core Graphics Objects” for general information about this type of object.

Surface Property Descriptions

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

AlphaData

m-by-n matrix of double or uint8

The transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The AlphaData can be of class double or uint8.

MATLAB determines the transparency in one of three ways:

- Using the elements of AlphaData as transparency values (AlphaDataMapping set to none)
- Using the elements of AlphaData as indices into the current alphamap (AlphaDataMapping set to direct)
- Scaling the elements of AlphaData to range between the minimum and maximum values of the axes ALim property (AlphaDataMapping set to scaled, the default)

AlphaDataMapping

none | direct | {scaled}

Surface Properties

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

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

AmbientStrength
scalar ≥ 0 and ≤ 1

Strength of ambient light. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes AmbientLightColor property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of light objects. See the surface DiffuseStrength and SpecularStrength properties.

BackFaceLighting
unlit | lit | reverselit

Face lighting control. This property determines how faces are lit when their vertex normals point away from the camera.

- `unlit` — Face is not lit.
- `lit` — Face is lit in normal way.
- `reverselit` — Face is lit as if the vertex pointed towards the camera.

This property is useful for discriminating between the internal and external surfaces of an object. See “Back Face Lighting” for an example.

`BeingDeleted`
`on` | `{off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

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

`BusyAction`
`cancel` | `{queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then

Surface Properties

interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is off, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
string or function handle

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the surface object.

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

Set this property to a function handle that references the callback. You can also use a string that is a valid MATLAB expression or the name of an M-file. The expressions execute in the MATLAB workspace.

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

`CData`
matrix (of type `double`)

Vertex colors. A matrix containing values that specify the color at every point in `ZData`.

Mapping `CData` to a `Colormap`

You can specify color as indexed values or true color. Indexed color data specifies a single value for each vertex. These values are

either scaled to map linearly into the current colormap (see `caxis`) or interpreted directly as indices into the colormap, depending on the setting of the `CDataMapping` property.

CData as True Color

True color defines an RGB value for each vertex. If the coordinate data (`XData`, for example) are contained in m -by- n matrices, then `CData` must be an m -by- n -3 array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

Texturemapping the Surface FaceColor

If you set the `FaceColor` property to `texturemap`, `CData` does not need to be the same size as `ZData`, but must be of type `double` or `uint8`. In this case, MATLAB maps `CData` to conform to the surface defined by `ZData`.

`CDataMapping`
{scaled} | direct

Direct or scaled color mapping. This property determines how MATLAB interprets indexed color data used to color the surface. (If you use true color specification for `CData`, this property has no effect.)

- `scaled` — Transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` reference page for more information on this mapping.
- `direct` — Use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to `length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

Surface Properties

Children
matrix of handles

Always the empty matrix; surface objects have no children.

Clipping
{on} | off

Clipping to axes rectangle. When Clipping is on, MATLAB does not display any portion of the surface that is outside the axes rectangle.

CreateFcn
string or function handle

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a surface object. You must define this property as a default value for surfaces or set the CreateFcn property during object creation.

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

```
surface(x,y,z,c,'CreateFcn',@myCreateFcn)
```

MATLAB executes this routine after setting all surface properties. Setting this property on an existing surface object has no effect.

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

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

DeleteFcn
string or function handle

Delete surface callback routine. A callback routine that executes when you delete the surface object (e.g., when you issue a delete command or clear the axes or figure). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

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

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

DiffuseStrength

scalar ≥ 0 and ≤ 1

Intensity of diffuse light. This property sets the intensity of the diffuse component of the light falling on the surface. Diffuse light comes from light objects in the axes.

You can also set the intensity of the ambient and specular components of the light on the surface object. See the AmbientStrength and SpecularStrength properties.

EdgeAlpha

{scalar = 1} | flat | interp

Transparency of the surface edges. This property can be any of the following:

- **scalar** — A single non-Nan scalar value between 0 and 1 that controls the transparency of all the edges of the object. 1 (the default) means fully opaque and 0 means completely transparent.
- **flat** — The alpha data (AlphaData) value for the first vertex of the face determines the transparency of the edges.

Surface Properties

- `interp` — Linear interpolation of the alpha data (`AlphaData`) values at each vertex determines the transparency of the edge.

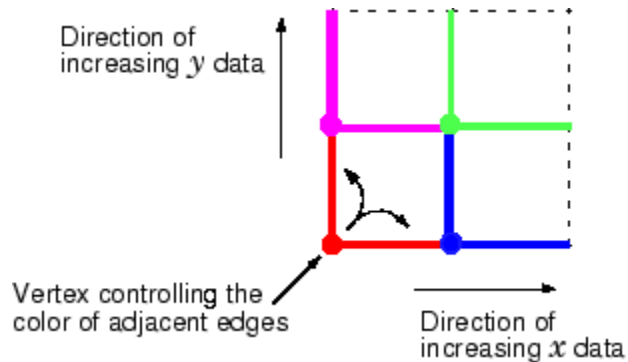
Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

EdgeColor

`{ColorSpec} | none | flat | interp`

Color of the surface edge. This property determines how MATLAB colors the edges of the individual faces that make up the surface:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default `EdgeColor` is black. See `ColorSpec` for more information on specifying color.
- `none` — Edges are not drawn.
- `flat` — The `CData` value of the first vertex for a face determines the color of each edge.



- `interp` — Linear interpolation of the `CData` values at the face vertices determines the edge color.

EdgeLighting

`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on surface edges. Choices are

- none — Lights do not affect the edges of this object.
- flat — The effect of light objects is uniform across each edge of the surface.
- gouraud — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- phong — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

EraseMode

{normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase surface objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- normal — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- none — Do not erase the surface when it is moved or destroyed. While the object is still visible on the screen after erasing with EraseMode none, you cannot print it because MATLAB stores no information about its former location.
- xor — Draw and erase the surface by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing

Surface Properties

the surface does not damage the color of the objects behind it. However, surface color depends on the color of the screen behind it and is correctly colored only when over the axes background Color, or the figure background Color if the axes Color is set to none.

- background — Erase the surface by drawing it in the axes background Color, or the figure background Color if the axes Color is set to none. This damages objects that are behind the erased object, but surface objects are always properly colored.

Printing with Nonnormal Erase Modes

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

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

FaceAlpha

{scalar = 1} | flat | interp | texturemap

Transparency of the surface faces. This property can be any of the following:

- scalar — A single non-NaN scalar value between 0 and 1 that controls the transparency of all the faces of the object. 1 (the default) means fully opaque and 0 means completely transparent (invisible).

- `flat` — The values of the alpha data (`AlphaData`) determine the transparency for each face. The alpha data at the first vertex determine the transparency of the entire face.
- `interp` — Bilinear interpolation of the alpha data (`AlphaData`) at each vertex determines the transparency of each face.
- `texturemap` — Use transparency for the texture map.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `FaceAlpha`.

`FaceColor`

`ColorSpec` | `none` | `{flat}` | `interp` | `texturemap`

Color of the surface face. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See `ColorSpec` for more information on specifying color.
- `none` — Do not draw faces. Note that edges are drawn independently of faces.
- `flat` — The values of `CData` determine the color for each face of the surface. The color data at the first vertex determine the color of the entire face.
- `interp` — Bilinear interpolation of the values at each vertex (the `CData`) determines the coloring of each face.
- `texturemap` — Texture map the `CData` to the surface. MATLAB transforms the color data so that it conforms to the surface. (See the texture mapping example.)

`FaceLighting`

`{none}` | `flat` | `gouraud` | `phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on the surface. Choices are

Surface Properties

- none — Lights do not affect the faces of this object.
- flat — The effect of light objects is uniform across the faces of the surface. Select this choice to view faceted objects.
- gouraud — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- phong — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

HandleVisibility

{on} | callback | off

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. This property is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when HandleVisibility is on.

Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

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

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

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

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

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

`HitTest`
{on} | off

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

`Interruptible`
{on} | off

Callback routine interruption mode. The `Interruptible` property controls whether a surface callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback

Surface Properties

routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

LineStyle

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

Edge line type. This property determines the line style used to draw surface edges. The available line styles are shown in this table.

Symbol	Line Style
	Solid line (default)
	Dashed line
:	Dotted line
.	Dash-dot line
none	No line

LineWidth

scalar

Edge line width. The width of the lines in points used to draw surface edges. The default width is 0.5 points (1 point = 1/72 inch).

Marker

marker symbol (see table)

Marker symbol. The `Marker` property specifies symbols that are displayed at vertices. You can set values for the `Marker` property independently from the `LineStyle` property.

You can specify these markers.

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

none | {auto} | flat | ColorSpec

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- none specifies no color, which makes nonfilled markers invisible.
- auto uses the same color as the EdgeColor property.
- flat uses the CData value of the vertex to determine the color of the maker edge.

Surface Properties

- ColorSpec defines a single color to use for the edge (see ColorSpec for more information).

MarkerFaceColor

{none} | auto | flat | ColorSpec

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- none makes the interior of the marker transparent, allowing the background to show through.
- auto uses the axes Color for the marker face color.
- flat uses the CData value of the vertex to determine the color of the face.
- ColorSpec defines a single color to use for all markers on the surface (see ColorSpec for more information).

MarkerSize

size in points

Marker size. A scalar specifying the marker size, in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker at 1/3 the specified marker size.

MeshStyle

{both} | row | column

Row and column lines. This property specifies whether to draw all edge lines or just row or column edge lines.

- both draws edges for both rows and columns.
- row draws row edges only.
- column draws column edges only.

NormalMode
{auto} | manual

MATLAB generated or user-specified normal vectors. When this property is auto, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB sets this property to manual and does not generate its own data. See also the VertexNormals property.

Parent
handle of axes, hggroup, or hgtransform

Parent of surface object. This property contains the handle of the surface object's parent. The parent of a surface object is the axes, hggroup, or hgtransform object that contains it.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

Selected
on | {off}

Is object selected? When this property is on, MATLAB displays a dashed bounding box around the surface if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

SelectionHighlight
{on} | off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing a dashed bounding box around the surface. When SelectionHighlight is off, MATLAB does not draw the handles.

SpecularColorReflectance
scalar in the range 0 to 1

Surface Properties

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

SpecularExponent
scalar ≥ 1

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

SpecularStrength
scalar ≥ 0 and ≤ 1

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

You can also set the intensity of the ambient and diffuse components of the light on the surface object. See the AmbientStrength and DiffuseStrength properties. Also see the material function.

Tag
string

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

Type
string (read only)

Class of the graphics object. The class of the graphics object. For surface objects, Type is always the string 'surface'.

UIContextMenu

handle of a uicontextmenu object

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

UserData

matrix

User-specified data. Any matrix you want to associate with the surface object. MATLAB does not use this data, but you can access it using the set and get commands.

VertexNormals

vector or matrix

Surface normal vectors. This property contains the vertex normals for the surface. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

Visible

{on} | off

Surface object visibility. By default, all surfaces are visible. When set to off, the surface is not visible, but still exists, and you can query and set its properties.

XData

vector or matrix

Surface Properties

X-coordinates. The x -position of the surface points. If you specify a row vector, surface replicates the row internally until it has the same number of columns as ZData.

YData

vector or matrix

Y-coordinates. The y -position of the surface points. If you specify a row vector, surface replicates the row internally until it has the same number of rows as ZData.

ZData

matrix

Z-coordinates. The z -position of the surfaceplot data points. See the Description section for more information.

Purpose

Defines the surfaceplot properties.

Modifying Properties

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

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

Note that you cannot define default properties for surfaceplot objects.

See Plot Objects for information on surfaceplot objects.

Surfaceplot Property Descriptions

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

AlphaData

m-by-n matrix of double or uint8

The transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The AlphaData can be of class double or uint8.

MATLAB determines the transparency in one of three ways:

- Using the elements of AlphaData as transparency values (AlphaDataMapping set to none)
- Using the elements of AlphaData as indices into the current alphamap (AlphaDataMapping set to direct)
- Scaling the elements of AlphaData to range between the minimum and maximum values of the axes ALim property (AlphaDataMapping set to scaled, the default)

AlphaDataMapping

none | direct | {scaled}

Surfaceplot Properties

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

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

AmbientStrength
scalar ≥ 0 and ≤ 1

Strength of ambient light. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes AmbientLightColor property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of light objects. See the surfaceplot DiffuseStrength and SpecularStrength properties.

BackFaceLighting
unlit | lit | reverselit

Face lighting control. This property determines how faces are lit when their vertex normals point away from the camera.

- `unlit` — Face is not lit.
- `lit` — Face is lit in normal way.
- `reverselit` — Face is lit as if the vertex pointed towards the camera.

This property is useful for discriminating between the internal and external surfaces of an object. See [Back Face Lighting](#) for an example.

`BeingDeleted`
`on` | `{off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

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

`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

Surfaceplot Properties

processed. If the `Interruptible` property is off, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

string or function handle

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the surfaceplot object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

See [Function Handle Callbacks](#) for information on how to use function handles to define the callback function.

`CData`

matrix

Vertex colors. A matrix containing values that specify the color at every point in `ZData`. If you set the `FaceColor` property to `texturemap`, `CData` does not need to be the same size as `ZData`. In this case, MATLAB maps `CData` to conform to the surfaceplot defined by `ZData`.

You can specify color as indexed values or true color. Indexed color data specifies a single value for each vertex. These values are either scaled to map linearly into the current colormap (see [caxis](#)) or interpreted directly as indices into the colormap, depending on the setting of the `CDataMapping` property. Note that any non-texture data passed as an input argument must be of type `double`.

True color defines an RGB value for each vertex. If the coordinate data (`XData`, for example) are contained in m -by- n matrices, then `CData` must be an m -by- n -by-3 array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

`CDataMapping`
{scaled} | direct

Direct or scaled color mapping. This property determines how MATLAB interprets indexed color data used to color the surfaceplot. (If you use true color specification for `CData`, this property has no effect.)

- `scaled` — Transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` reference page for more information on this mapping.
- `direct` — Use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to `length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

`CDataMode`
{auto} | manual

Use automatic or user-specified color data values. If you specify `CData`, MATLAB sets this property to `manual` and uses the `CData` values to color the surfaceplot.

If you set `CDataMode` to `auto` after having specified `CData`, MATLAB resets the color data of the surfaceplot to that defined by `ZData`, overwriting any previous values for `CData`.

`CDataSource`
string (MATLAB variable)

Surfaceplot Properties

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

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

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

See the `refreshdata` reference page for more information.

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

Children
matrix of handles

Always the empty matrix; surfaceplot objects have no children.

Clipping
{on} | off

Clipping to axes rectangle. When Clipping is on, MATLAB does not display any portion of the surfaceplot that is outside the axes rectangle.

CreateFcn
string or function handle

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

```
surf(peaks,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other surfaceplot properties. Setting this property on an existing surfaceplot object has no effect.

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

See [Function Handle Callbacks](#) for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

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

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

See [Function Handle Callbacks](#) for information on how to use function handles to define the callback function.

Surfaceplot Properties

DiffuseStrength

scalar ≥ 0 and ≤ 1

Intensity of diffuse light. This property sets the intensity of the diffuse component of the light falling on the surface. Diffuse light comes from light objects in the axes.

You can also set the intensity of the ambient and specular components of the light on the surfaceplot object. See the AmbientStrength and SpecularStrength properties.

EdgeAlpha

{scalar = 1} | flat | interp

Transparency of the surfaceplot edges. This property can be any of the following:

- **scalar** — A single non-Nan scalar value between 0 and 1 that controls the transparency of all the edges of the object. 1 (the default) means fully opaque and 0 means completely transparent.
- **flat** — The alpha data (AlphaData) value for the first vertex of the face determines the transparency of the edges.
- **interp** — Linear interpolation of the alpha data (AlphaData) values at each vertex determines the transparency of the edge.

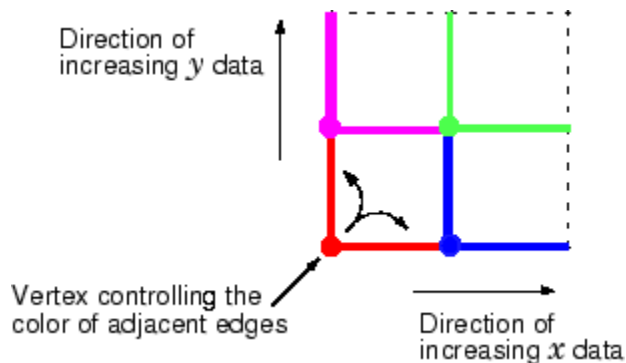
Note that you must specify AlphaData as a matrix equal in size to ZData to use flat or interp EdgeAlpha.

EdgeColor

{ColorSpec} | none | flat | interp

Color of the surfaceplot edge. This property determines how MATLAB colors the edges of the individual faces that make up the surface:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default `EdgeColor` is black. See `ColorSpec` for more information on specifying color.
- `none` — Edges are not drawn.
- `flat` — The `CData` value of the first vertex for a face determines the color of each edge.



- `interp` — Linear interpolation of the `CData` values at the face vertices determines the edge color.

EdgeLighting

{none} | flat | gouraud | phong

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on surfaceplot edges. Choices are

- `none` — Lights do not affect the edges of this object.
- `flat` — The effect of light objects is uniform across each edge of the surface.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.

Surfaceplot Properties

- phong — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

EraseMode

{normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase surfaceplot objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- normal — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- none — Do not erase the surfaceplot when it is moved or destroyed. While the object is still visible on the screen after erasing with EraseMode none, you cannot print it because MATLAB stores no information about its former location.
- xor — Draw and erase the surfaceplot by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the surfaceplot does not damage the color of the objects behind it. However, surfaceplot color depends on the color of the screen behind it and is correctly colored only when over the axes background Color, or the figure background Color if the axes Color is set to none.
- background — Erase the surfaceplot by drawing it in the axes background Color, or the figure background Color if the axes Color is set to none. This damages objects that are behind

the erased object, but surfaceplot objects are always properly colored.

Printing with Nonnormal Erase Modes

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

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

FaceAlpha

`{scalar = 1} | flat | interp | texturemap`

Transparency of the surfaceplot faces. This property can be any of the following:

- `scalar` — A single non-NaN scalar value between 0 and 1 that controls the transparency of all the faces of the object. 1 (the default) means fully opaque and 0 means completely transparent (invisible).
- `flat` — The values of the alpha data (`AlphaData`) determine the transparency for each face. The alpha data at the first vertex determine the transparency of the entire face.
- `interp` — Bilinear interpolation of the alpha data (`AlphaData`) at each vertex determines the transparency of each face.
- `texturemap` — Use transparency for the texture map.

Surfaceplot Properties

Note that you must specify AlphaData as a matrix equal in size to ZData to use flat or interp FaceAlpha.

FaceColor

ColorSpec | none | {flat} | interp

Color of the surfaceplot face. This property can be any of the following:

- **ColorSpec** — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See ColorSpec for more information on specifying color.
- **none** — Do not draw faces. Note that edges are drawn independently of faces.
- **flat** — The values of CData determine the color for each face of the surface. The color data at the first vertex determine the color of the entire face.
- **interp** — Bilinear interpolation of the values at each vertex (the CData) determines the coloring of each face.
- **texturemap** — Texture map the Cdata to the surface. MATLAB transforms the color data so that it conforms to the surface. (See the texture mapping example.)

FaceLighting

{none} | flat | gouraud | phong

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on the surface. Choices are

- **none** — Lights do not affect the faces of this object.
- **flat** — The effect of light objects is uniform across the faces of the surface. Select this choice to view faceted objects.

- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. This property is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

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

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching

Surfaceplot Properties

the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

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

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

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

`HitTest`
{on} | off

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

`Interruptible`
{on} | off

Callback routine interruption mode. The `Interruptible` property controls whether a surfaceplot callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`,

or pause command in the routine. See the `BusyAction` property for related information.

LineStyle

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

Edge line type. This property determines the line style used to draw surfaceplot edges. The available line styles are shown in this table.

Symbol	Line Style
	Solid line (default)
	Dashed line
:	Dotted line
.	Dash-dot line
none	No line

LineWidth

scalar

Edge line width. The width of the lines in points used to draw surfaceplot edges. The default width is 0.5 points (1 point = 1/72 inch).

Marker

marker symbol (see table)

Marker symbol. The `Marker` property specifies symbols that are displayed at vertices. You can set values for the `Marker` property independently from the `LineStyle` property.

You can specify these markers.

Surfaceplot Properties

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

none | {auto} | flat | ColorSpec

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- none specifies no color, which makes nonfilled markers invisible.
- auto uses the same color as the EdgeColor property.
- flat uses the CData value of the vertex to determine the color of the maker edge.

- ColorSpec defines a single color to use for the edge (see ColorSpec for more information).

MarkerFaceColor

{none} | auto | flat | ColorSpec

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- none makes the interior of the marker transparent, allowing the background to show through.
- auto uses the axes Color for the marker face color.
- flat uses the CData value of the vertex to determine the color of the face.
- ColorSpec defines a single color to use for all markers on the surfaceplot (see ColorSpec for more information).

MarkerSize

size in points

Marker size. A scalar specifying the marker size, in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker at 1/3 the specified marker size.

MeshStyle

{both} | row | column

Row and column lines. This property specifies whether to draw all edge lines or just row or column edge lines.

- both draws edges for both rows and columns.
- row draws row edges only.
- column draws column edges only.

Surfaceplot Properties

NormalMode
{auto} | manual

MATLAB generated or user-specified normal vectors. When this property is auto, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB sets this property to manual and does not generate its own data. See also the VertexNormals property.

Parent
handle of axes, hggroup, or hgtransform object

Parent of surfaceplot object. This property contains the handle of the surfaceplot object's parent. The parent of a surfaceplot object is the axes, hggroup, or hgtransform object that contains it.

See Objects That Can Contain Other Objects for more information on parenting graphics objects.

Selected
on | {off}

Is object selected? When this property is on, MATLAB displays a dashed bounding box around the surfaceplot if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

SelectionHighlight
{on} | off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing a dashed bounding box around the surface. When SelectionHighlight is off, MATLAB does not draw the handles.

SpecularColorReflectance
scalar in the range 0 to 1

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

SpecularExponent

scalar ≥ 1

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

SpecularStrength

scalar ≥ 0 and ≤ 1

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

You can also set the intensity of the ambient and diffuse components of the light on the surfaceplot object. See the AmbientStrength and DiffuseStrength properties. Also see the material function.

Tag

string

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

Type

string (read only)

Surfaceplot Properties

Class of the graphics object. The class of the graphics object. For surfaceplot objects, Type is always the string 'surface'.

UIContextMenu
handle of a uicontextmenu object

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

UserData
matrix

User-specified data. Any matrix you want to associate with the surfaceplot object. MATLAB does not use this data, but you can access it using the set and get commands.

VertexNormals
vector or matrix

Surfaceplot normal vectors. This property contains the vertex normals for the surfaceplot. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

Visible
{on} | off

Surfaceplot object visibility. By default, all surfaceplots are visible. When set to off, the surfaceplot is not visible, but still exists, and you can query and set its properties.

XData
vector or matrix

X-coordinates. The *x*-position of the surfaceplot data points. If you specify a row vector, MATLAB replicates the row internally until it has the same number of columns as ZData.

XDataMode
{auto} | manual

Use automatic or user-specified x-axis values. If you specify XData, MATLAB sets this property to manual.

If you set XDataMode to auto after having specified XData, MATLAB resets the *x*-axis ticks and *x*-tick labels to the column indices of the ZData, overwriting any previous values for XData.

XDataSource
string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

Surfaceplot Properties

YData

vector or matrix

Y-coordinates. The *y*-position of the surfaceplot data points. If you specify a row vector, MATLAB replicates the row internally until it has the same number of rows as ZData.

YDataMode

{auto} | manual

Use automatic or user-specified x-axis values. If you specify XData, MATLAB sets this property to manual.

If you set YDataMode to auto after having specified YData, MATLAB resets the *y*-axis ticks and *y*-tick labels to the row indices of the ZData, overwriting any previous values for YData.

YDataSource

string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

ZData

matrix

Z-coordinates. The *z*-position of the surfaceplot data points. See the Description section for more information.

ZDataSource

string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.


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

Purpose

Surface plot with colormap-based lighting



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

```
surf1(Z)
surf1(...,'light')
surf1(...,s)
surf1(X,Y,Z,s,k)
h = surf1(...)
```

Description

The `surf1` function displays a shaded surface based on a combination of ambient, diffuse, and specular lighting models.

`surf1(Z)` and `surf1(X,Y,Z)` create three-dimensional shaded surfaces using the default direction for the light source and the default lighting coefficients for the shading model. `X`, `Y`, and `Z` are vectors or matrices that define the x , y , and z components of a surface.

`surf1(...,'light')` produces a colored, lighted surface using a MATLAB light object. This produces results different from the default lighting method, `surf1(...,'cdata')`, which changes the color data for the surface to be the reflectance of the surface.

`surf1(...,s)` specifies the direction of the light source. `s` is a two- or three-element vector that specifies the direction from a surface to a light source. `s = [sx sy sz]` or `s = [azimuth elevation]`. The default `s` is 45° counterclockwise from the current view direction.

`surf1(X,Y,Z,s,k)` specifies the reflectance constant. `k` is a four-element vector defining the relative contributions of ambient light,

diffuse reflection, specular reflection, and the specular shine coefficient. $k = [k_a \ k_d \ k_s \ \text{shine}]$ and defaults to $[.55, .6, .4, 10]$.

`h = surf1(...)` returns a handle to a surface graphics object.

Remarks

For smoother color transitions, use colormaps that have linear intensity variations (e.g., gray, copper, bone, pink).

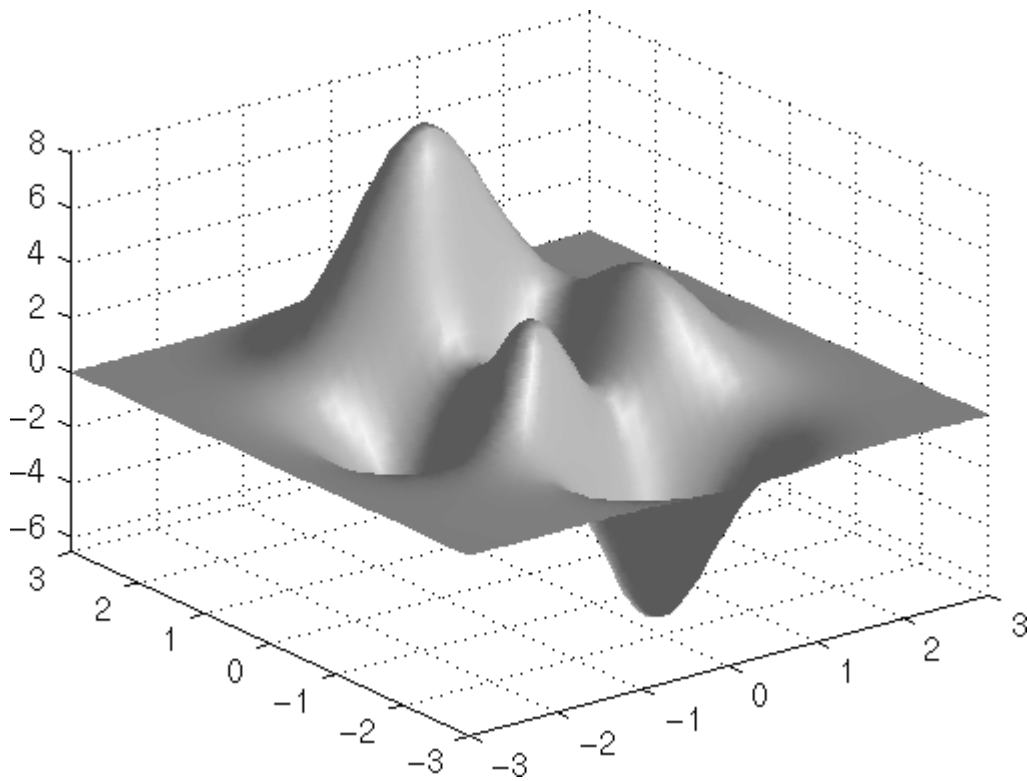
The ordering of points in the X, Y, and Z matrices defines the inside and outside of parametric surfaces. If you want the opposite side of the surface to reflect the light source, use `surf1(X',Y',Z')`. Because of the way surface normal vectors are computed, `surf1` requires matrices that are at least 3-by-3.

Examples

View peaks using colormap-based lighting.

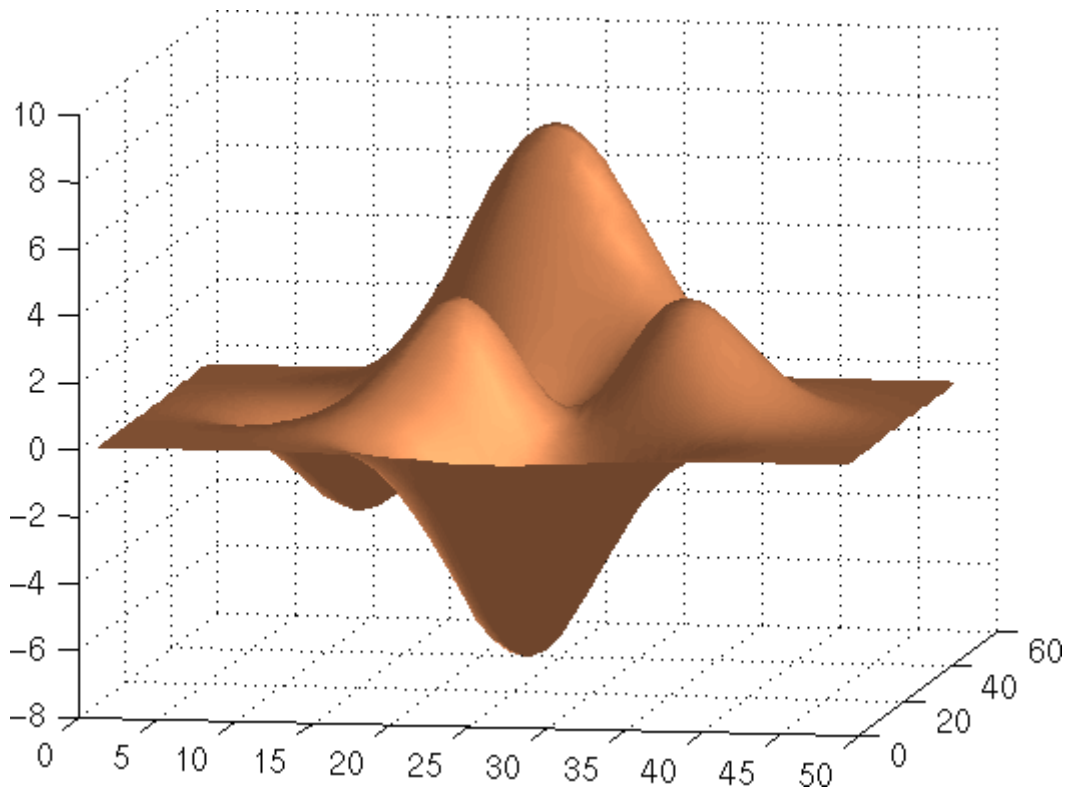
```
[x,y] = meshgrid( 3:1/8:3);  
z = peaks(x,y);  
surf1(x,y,z);  
shading interp  
colormap(gray);  
axis([ 3 3 3 3 8 8])
```

surf1



To plot a lighted surface from a view direction other than the default,

```
view([10 10])  
grid on  
hold on  
surf1(peaks)  
shading interp  
colormap copper  
hold off
```

**See Also**

`colormap`, `shading`, `light`

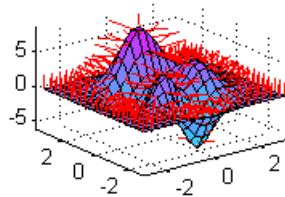
“Creating Surfaces and Meshes” on page 1-94 for functions related to surfaces

“Lighting” on page 1-98 for functions related to lighting

surfnorm

Purpose

Compute and display 3-D surface normals



Syntax

```
surfnorm(Z)  
[Nx,Ny,Nz] = surfnorm(...)
```

Description

The `surfnorm` function computes surface normals for the surface defined by X , Y , and Z . The surface normals are unnormalized and valid at each vertex. Normals are not shown for surface elements that face away from the viewer.

`surfnorm(Z)` and `surfnorm(X,Y,Z)` plot a surface and its surface normals. Z is a matrix that defines the z component of the surface. X and Y are vectors or matrices that define the x and y components of the surface.

`[Nx,Ny,Nz] = surfnorm(...)` returns the components of the three-dimensional surface normals for the surface.

Remarks

The direction of the normals is reversed by calling `surfnorm` with transposed arguments:

```
surfnorm(X',Y',Z')
```

`surf1` uses `surfnorm` to compute surface normals when calculating the reflectance of a surface.

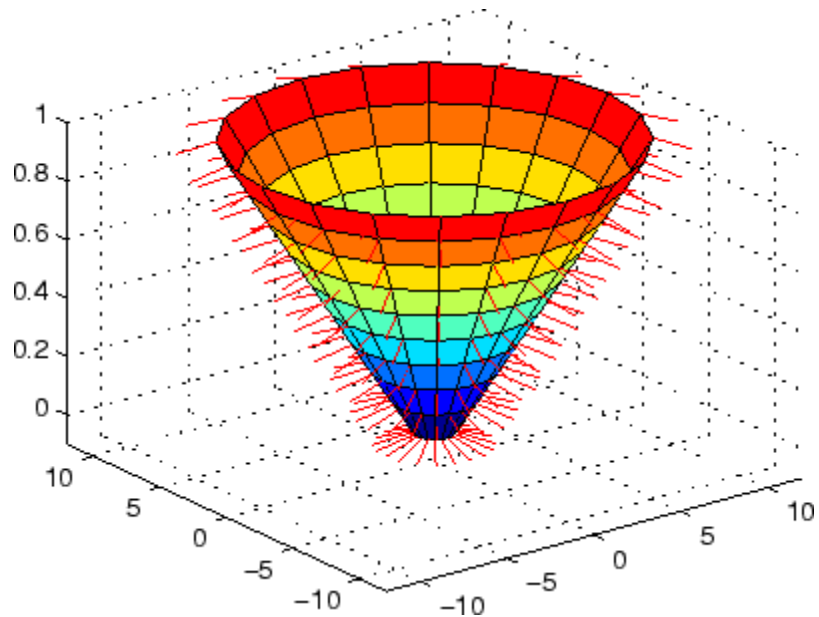
Algorithm

The surface normals are based on a bicubic fit of the data in X , Y , and Z . For each vertex, diagonal vectors are computed and crossed to form the normal.

Examples

Plot the normal vectors for a truncated cone.

```
[x,y,z] = cylinder(1:10);  
surfnorm(x,y,z)  
axis([-12 12 -12 12 -0.1 1])
```

**See Also**

surf, quiver3

“Colormaps” on page 1-96 for related functions

svd

Purpose Singular value decomposition

Syntax

```
s = svd(X)
[U,S,V] = svd(X)
[U,S,V] = svd(X,0)
[U,S,V] = svd(X,'econ')
```

Description The svd command computes the matrix singular value decomposition.

`s = svd(X)` returns a vector of singular values.

`[U,S,V] = svd(X)` produces a diagonal matrix S of the same dimension as X , with nonnegative diagonal elements in decreasing order, and unitary matrices U and V so that $X = U*S*V'$.

`[U,S,V] = svd(X,0)` produces the “economy size” decomposition. If X is m -by- n with $m > n$, then `svd` computes only the first n columns of U and S is n -by- n .

`[U,S,V] = svd(X,'econ')` also produces the “economy size” decomposition. If X is m -by- n with $m \geq n$, it is equivalent to `svd(X,0)`. For $m < n$, only the first m columns of V are computed and S is m -by- m .

Examples

For the matrix

```
X =
     1     2
     3     4
     5     6
     7     8
```

the statement

```
[U,S,V] = svd(X)
```

produces

```
U =
-0.1525  -0.8226  -0.3945  -0.3800
```

-0.3499	-0.4214	0.2428	0.8007
-0.5474	-0.0201	0.6979	-0.4614
-0.7448	0.3812	-0.5462	0.0407

S =

14.2691	0
0	0.6268
0	0
0	0

V =

-0.6414	0.7672
-0.7672	-0.6414

The economy size decomposition generated by

$$[U, S, V] = \text{svd}(X, 0)$$

produces

U =

-0.1525	-0.8226
-0.3499	-0.4214
-0.5474	-0.0201
-0.7448	0.3812

S =

14.2691	0
0	0.6268

V =

-0.6414	0.7672
-0.7672	-0.6414

Algorithm

svd uses the LAPACK routines listed in the following table to compute the singular value decomposition.

	Real	Complex
X double	DGESVD	ZGESVD
X single	SGESVD	CGESVD

Diagnostics

If the limit of 75 QR step iterations is exhausted while seeking a singular value, this message appears:

Solution will not converge.

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), Third Edition, SIAM, Philadelphia, 1999.

Purpose Find singular values and vectors

Syntax

```
s = svds(A)
s = svds(A,k)
s = svds(A,k,sigma)
s = svds(A,k,'L')
s = svds(A,k,sigma,options)
[U,S,V] = svds(A,...)
[U,S,V,flag] = svds(A,...)
```

Description `s = svds(A)` computes the six largest singular values and associated singular vectors of matrix A. If A is m-by-n, `svds(A)` manipulates eigenvalues and vectors returned by `eigs(B)`, where `B = [sparse(m,m) A; A' sparse(n,n)]`, to find a few singular values and vectors of A. The positive eigenvalues of the symmetric matrix B are the same as the singular values of A.

`s = svds(A,k)` computes the k largest singular values and associated singular vectors of matrix A.

`s = svds(A,k,sigma)` computes the k singular values closest to the scalar shift sigma. For example, `s = svds(A,k,0)` computes the k smallest singular values and associated singular vectors.

`s = svds(A,k,'L')` computes the k largest singular values (the default).

`s = svds(A,k,sigma,options)` sets some parameters (see `eigs`):

Option Structure Fields and Descriptions

Field name	Parameter	Default
<code>options.tol</code>	Convergence tolerance: $\text{norm}(AV-US,1) \leq \text{tol} * \text{norm}(A,1)$	1e-10
<code>options.maxit</code>	Maximum number of iterations	300
<code>options.disp</code>	Number of values displayed each iteration	0

`[U,S,V] = svds(A,...)` returns three output arguments, and if A is m -by- n :

- U is m -by- k with orthonormal columns
- S is k -by- k diagonal
- V is n -by- k with orthonormal columns
- $U*S*V'$ is the closest rank k approximation to A

`[U,S,V,flag] = svds(A,...)` returns a convergence flag. If `eigs` converged then $\text{norm}(A*V-U*S,1) \leq \text{tol}*\text{norm}(A,1)$ and `flag` is 0. If `eigs` did not converge, then `flag` is 1.

Note `svds` is best used to find a few singular values of a large, sparse matrix. To find all the singular values of such a matrix, `svd(full(A))` will usually perform better than `svds(A,min(size(A)))`.

Algorithm

`svds(A,k)` uses `eigs` to find the k largest magnitude eigenvalues and corresponding eigenvectors of $B = [0 \ A; \ A' \ 0]$.

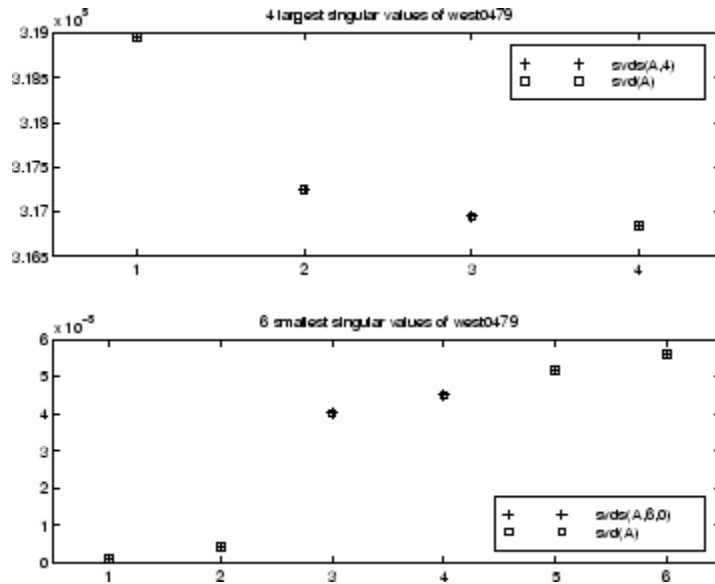
`svds(A,k,0)` uses `eigs` to find the $2k$ smallest magnitude eigenvalues and corresponding eigenvectors of $B = [0 \ A; \ A' \ 0]$, and then selects the k positive eigenvalues and their eigenvectors.

Example

`west0479` is a real 479-by-479 sparse matrix. `svd` calculates all 479 singular values. `svds` picks out the largest and smallest singular values.

```
load west0479
s = svd(full(west0479))
s1 = svds(west0479,4)
ss = svds(west0479,6,0)
```

These plots show some of the singular values of `west0479` as computed by `svd` and `svds`.



The largest singular value of west0479 can be computed a few different ways:

```
svds(west0479, 1) =
  3.189517598808622e+05
max(svd(full(west0479))) =
  3.18951759880862e+05
norm(full(west0479)) =
  3.189517598808623e+05
```

and estimated:

```
normest(west0479) =
  3.189385666549991e+05
```

See Also

`svd`, `eigs`

swapbytes

Purpose Swap byte ordering

Syntax `Y = swapbytes(X)`

Description `Y = swapbytes(X)` reverses the byte ordering of each element in array `X`, converting little-endian values to big-endian (and vice versa). The input array must contain all full, noncomplex, numeric elements.

Examples

Example 1

Reverse the byte order for a scalar 32-bit value, changing hexadecimal 12345678 to 78563412:

```
A = uint32(hex2dec('12345678'));  
  
B = dec2hex(swapbytes(A))  
B =  
    78563412
```

Example 2

Reverse the byte order for each element of a 1-by-4 matrix:

```
X = uint16([0 1 128 65535])  
X =  
     0     1    128 65535  
  
Y = swapbytes(X);  
Y =  
     0    256 32768 65535
```

Examining the output in hexadecimal notation shows the byte swapping:

```
format hex  
  
X, Y  
X =  
    0000    0001    0080    ffff
```

```
Y =  
    0000    0100    8000    ffff
```

Example 3

Create a three-dimensional array A of 16-bit integers and then swap the bytes of each element:

```
format hex  
  
A = uint16(magic(3) * 150);  
A(:,:,2) = A * 40;  
  
A  
A(:,:,1) =  
    04b0    0096    0384  
    01c2    02ee    041a  
    0258    0546    012c  
A(:,:,2) =  
    bb80    1770    8ca0  
    4650    7530    a410  
    5dc0    d2f0    2ee0  
  
swapbytes(A)  
ans(:,:,1) =  
    b004    9600    8403  
    c201    ee02    1a04  
    5802    4605    2c01  
ans(:,:,2) =  
    80bb    7017    a08c  
    5046    3075    10a4  
    c05d    f0d2    e02e
```

See Also

typecast

switch

Purpose Switch among several cases, based on expression

Syntax

```
switch switch_expr
  case case_expr
    statement, ..., statement
  case {case_expr1, case_expr2, case_expr3, ...}
    statement, ..., statement
  otherwise
    statement, ..., statement
end
```

Discussion The switch statement syntax is a means of conditionally executing code. In particular, switch executes one set of statements selected from an arbitrary number of alternatives. Each alternative is called a case, and consists of

- The case statement
- One or more case expressions
- One or more statements

In its basic syntax, switch executes the statements associated with the first case where *switch_expr* == *case_expr*. When the case expression is a cell array (as in the second case above), the *case_expr* matches if any of the elements of the cell array matches the switch expression. If no case expression matches the switch expression, then control passes to the otherwise case (if it exists). After the case is executed, program execution resumes with the statement after the end.

The *switch_expr* can be a scalar or a string. A scalar *switch_expr* matches a *case_expr* if *switch_expr*==*case_expr*. A string *switch_expr* matches a *case_expr* if strcmp(*switch_expr*,*case_expr*) returns logical 1 (true).

Note for C Programmers Unlike the C language switch construct, the MATLAB switch does not “fall through.” That is, switch executes only the first matching case; subsequent matching cases do not execute. Therefore, break statements are not used.

Examples

To execute a certain block of code based on what the string, method, is set to,

```
method = 'Bilinear';

switch lower(method)
    case {'linear','bilinear'}
        disp('Method is linear')
    case 'cubic'
        disp('Method is cubic')
    case 'nearest'
        disp('Method is nearest')
    otherwise
        disp('Unknown method.')
end

Method is linear
```

See Also

case, otherwise, end, if, else, elseif, while

symamd

Purpose Symmetric approximate minimum degree permutation

Syntax

```
p = symamd(S)
p = symamd(S,knobs)
[p,stats] = symamd(...)
```

Description `p = symamd(S)` for a symmetric positive definite matrix `S`, returns the permutation vector `p` such that `S(p,p)` tends to have a sparser Cholesky factor than `S`. To find the ordering for `S`, `symamd` constructs a matrix `M` such that `spones(M'*M) = spones(S)`, and then computes `p = colamd(M)`. The `symamd` function may also work well for symmetric indefinite matrices.

`S` must be square; only the strictly lower triangular part is referenced.

`p = symamd(S,knobs)` where `knobs` is a scalar. If `S` is `n`-by-`n`, rows and columns with more than `knobs*n` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs = spparms('wh_frac')`.

`[p,stats] = symamd(...)` produces the optional vector `stats` that provides data about the ordering and the validity of the matrix `S`.

<code>stats(1)</code>	Number of dense or empty rows ignored by <code>symamd</code>
<code>stats(2)</code>	Number of dense or empty columns ignored by <code>symamd</code>
<code>stats(3)</code>	Number of garbage collections performed on the internal data structure used by <code>symamd</code> (roughly of size $8.4 * \text{nnz}(\text{tril}(S, -1)) + 9n$ integers)
<code>stats(4)</code>	0 if the matrix is valid, or 1 if invalid
<code>stats(5)</code>	Rightmost column index that is unsorted or contains duplicate entries, or 0 if no such column exists
<code>stats(6)</code>	Last seen duplicate or out-of-order row index in the column index given by <code>stats(5)</code> , or 0 if no such row index exists
<code>stats(7)</code>	Number of duplicate and out-of-order row indices

Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to symamd. For this reason, symamd verifies that S is valid:

- If a row index appears two or more times in the same column, symamd ignores the duplicate entries, continues processing, and provides information about the duplicate entries in stats(4:7).
- If row indices in a column are out of order, symamd sorts each column of its internal copy of the matrix S (but does not repair the input matrix S), continues processing, and provides information about the out-of-order entries in stats(4:7).
- If S is invalid in any other way, symamd cannot continue. It prints an error message, and returns no output arguments (p or stats).

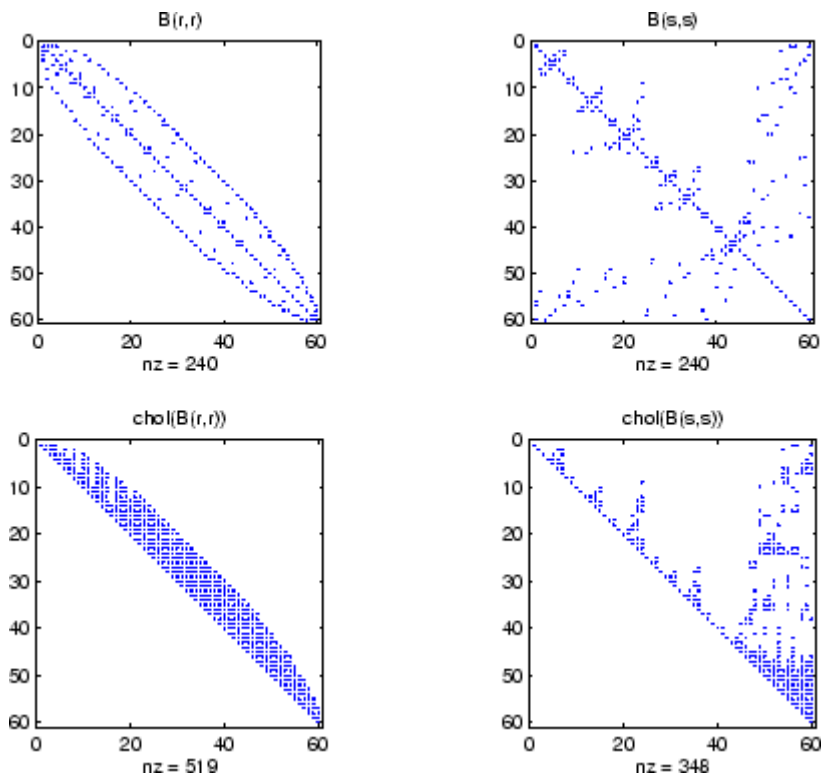
The ordering is followed by a symmetric elimination tree post-ordering.

Note symamd tends to be faster than symmmd and tends to return a better ordering.

Examples

Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the symrcm reference page.

```
B = bucky+4*speye(60);
r = symrcm(B);
p = symamd(B);
R = B(r,r);
S = B(p,p);
subplot(2,2,1), spy(R,4), title('B(r,r)')
subplot(2,2,2), spy(S,4), title('B(s,s)')
subplot(2,2,3), spy(chol(R),4), title('chol(B(r,r))')
subplot(2,2,4), spy(chol(S),4), title('chol(B(s,s))')
```



Even though this is a very small problem, the behavior of both orderings is typical. RCM produces a matrix with a narrow bandwidth which fills in almost completely during the Cholesky factorization. Minimum degree produces a structure with large blocks of contiguous zeros which do not fill in during the factorization. Consequently, the minimum degree ordering requires less time and storage for the factorization.

See Also

`colamd`, `colperm`, `spparms`, `symrcm`

References

The authors of the code for `symamd` are Stefan I. Larimore and Timothy A. Davis (davis@cise.ufl.edu), University of Florida. The algorithm was developed in collaboration with John Gilbert,

Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory.
Sparse Matrix Algorithms Research at the University of Florida:
<http://www.cise.ufl.edu/research/sparse/>

symbfact

Purpose Symbolic factorization analysis

Syntax

```
count = symbfact(A)
count = symbfact(A, 'sym')
count = symbfact(A, 'col')
count = symbfact(A, 'row')
count = symbfact(A, 'lo')
[count,h,parent,post,R] = symbfact(...)
[count,h,parent,post,L] = symbfact(A,type,'lower')
```

Description `count = symbfact(A)` returns the vector of row counts for the upper triangular Cholesky factor of a symmetric matrix whose upper triangle is that of `A`, assuming no cancellation during the factorization. `symbfact` should be much faster than `chol(A)`.

`count = symbfact(A, 'sym')` is the same as `count = symbfact(A)`.

`count = symbfact(A, 'col')` returns row counts of `R=chol(A'*A)'` *`A` (without forming it explicitly).

`count = symbfact(A, 'row')` returns row counts of `R=chol(A*A')`.

`count = symbfact(A, 'lo')` is the same as `count = symbfact(A)` and uses `tril(A)`.

`[count,h,parent,post,R] = symbfact(...)` has several optional return values.

Return Value	Description
h	Height of the elimination tree
parent	The elimination tree itself
post	Postordering permutation of the elimination tree
R	0-1 matrix having the structure of <code>chol(A)</code> for the symmetric case, <code>chol(A'*A)</code> for the 'col' case, or <code>chol(A*A')</code> for the 'row' case.

`[count,h,parent,post,L] = symbfact(A,type,'lower')` where `type` is one of `'sym'`, `'col'`, `'row'`, or `'lo'` returns a lower triangular symbolic factor `L=R'`. This form is quicker and requires less memory.

`symbfact(A)` and `symbfact(A,'sym')` use the upper triangular part of `A` (`TRIU(A)`) and assume the lower triangular part is the transpose of the upper triangular part. `symbfact(A,'lo')` uses `TRIL(A)` instead.

See Also

`chol`, `etree`, `treelayout`

symmlq

Purpose

Symmetric LQ method

Syntax

```
x = symmlq(A,b)
symmlq(A,b,tol)
symmlq(A,b,tol,maxit)
symmlq(A,b,tol,maxit,M)
symmlq(A,b,tol,maxit,M1,M2)
symmlq(A,b,tol,maxit,M1,M2,x0)
[x,flag] = symmlq(A,b,...)
[x,flag,relres] = symmlq(A,b,...)
[x,flag,relres,iter] = symmlq(A,b,...)
[x,flag,relres,iter,resvec] = symmlq(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...)
```

Description

`x = symmlq(A,b)` attempts to solve the system of linear equations $A*x=b$ for x . The n -by- n coefficient matrix A must be symmetric but need not be positive definite. It should also be large and sparse. The column vector b must have length n . A can be a function handle `afun` such that `afun(x)` returns $A*x$. See “Function Handles” in the MATLAB Programming documentation for more information.

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

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

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

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

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

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

`[x,flag] = symmlq(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	<code>symmlq</code> converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.
1	<code>symmlq</code> iterated <code>maxit</code> times but did not converge.
2	Preconditioner M was ill-conditioned.
3	<code>symmlq</code> stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during <code>symmlq</code> became too small or too large to continue computing.
5	Preconditioner M was not symmetric positive definite.

Whenever `flag` is not 0, the solution x returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = symmlq(A,b,...)` also returns the relative residual $\text{norm}(b-A*x)/\text{norm}(b)$. If `flag` is 0, `relres` \leq `tol`.

`[x,flag,relres,iter] = symmlq(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] = symmlq(A,b,...)` also returns a vector of estimates of the `symmlq` residual norms at each iteration, including $\text{norm}(b-A*x0)$.

`[x,flag,relres,iter,resvec,resvecg] = symmlq(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 = symmlq(A,b,tol,maxit,M1);
symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015
```

Example 2

This example replaces the matrix *A* in Example 1 with a handle to a matrix-vector product function *afun*. The example is contained in an M-file `run_symmlq` that

- Calls `symmlq` with the function handle `@afun` as its first argument.
- Contains *afun* as a nested function, so that all variables in `run_symmlq` are available to *afun*.

The following shows the code for `run_symmlq`:

```
function x1 = run_symmlq
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
```



```

x1 = symmlq(@afun,b,tol,maxit,M1);

function y = afun(x)
    y = 4 * x;
    y(2:n) = y(2:n) - 2 * x(1:n-1);
    y(1:n-1) = y(1:n-1) - 2 * x(2:n);
end
end

```

When you enter

```
x1=run_symmlq;
```

MATLAB displays the message

```

symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015

```

Example 3

Use a symmetric indefinite matrix that fails with pcg.

```

A = diag([20:-1:1,-1:-1:-20]);
b = sum(A,2);           % The true solution is the vector of all ones.
x = pcg(A,b);          % Errors out at the first iteration.
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1

```

However, symmlq can handle the indefinite matrix A.

```

x = symmlq(A,b,1e-6,40);
symmlq converged at iteration 39 to a solution with relative
residual 1.3e-007

```

See Also

bicg, bicgstab, cgs, lsqr, gmres, minres, pcg, qmr
function_handle (@), mldivide (\)

References

[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 Sparse symmetric minimum degree ordering

Syntax `p = symmmd(S)`

Note `symmmd` is obsolete and will be removed from a future version of MATLAB. Use `symamd` instead.

Description `p = symmmd(S)` returns a symmetric minimum degree ordering of S . For a symmetric positive definite matrix S , this is a permutation p such that $S(p, p)$ tends to have a sparser Cholesky factor than S . Sometimes `symmmd` works well for symmetric indefinite matrices too.

Algorithm The symmetric minimum degree algorithm is based on the column minimum degree algorithm. In fact, `symmmd(A)` just creates a nonzero structure K such that $K' * K$ has the same nonzero structure as A and then calls the column minimum degree code for K .

See Also `colamd`, `colmmd`, `colperm`, `symamd`, `symrcm`

References [1] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis and Applications* 13, 1992, pp. 333-356.

Purpose Sparse reverse Cuthill-McKee ordering

Syntax `r = symrcm(S)`

Description `r = symrcm(S)` returns the symmetric reverse Cuthill-McKee ordering of S . This is a permutation r such that $S(r, r)$ tends to have its nonzero elements closer to the diagonal. This is a good reordering for LU or Cholesky factorization of matrices that come from long, skinny problems. The ordering works for both symmetric and nonsymmetric S . For a real, symmetric sparse matrix, S , the eigenvalues of $S(r, r)$ are the same as those of S , but `eig(S(r, r))` probably takes less time to compute than `eig(S)`.

Algorithm The algorithm first finds a pseudoperipheral vertex of the graph of the matrix. It then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudoperipheral vertex. The implementation is based closely on the SPARSPAK implementation described by George and Liu.

Examples The statement

```
B = bucky;
```

uses an M-file in the demos toolbox to generate the adjacency graph of a truncated icosahedron. This is better known as a soccer ball, a Buckminster Fuller geodesic dome (hence the name bucky), or, more recently, as a 60-atom carbon molecule. There are 60 vertices. The vertices have been ordered by numbering half of them from one hemisphere, pentagon by pentagon; then reflecting into the other hemisphere and gluing the two halves together. With this numbering, the matrix does not have a particularly narrow bandwidth, as the first spy plot shows

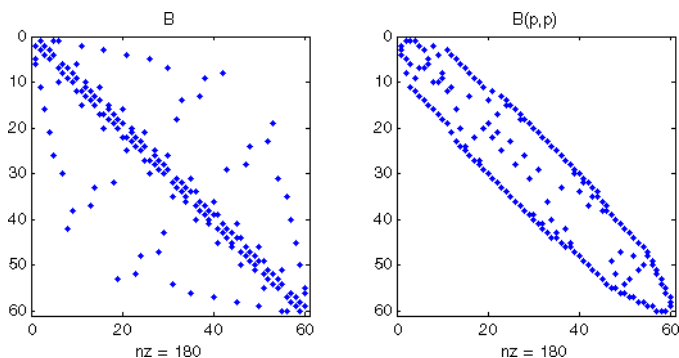
```
subplot(1,2,1), spy(B), title('B')
```

The reverse Cuthill-McKee ordering is obtained with

```
p = symrcm(B);
R = B(p,p);
```

The spy plot shows a much narrower bandwidth.

```
subplot(1,2,2), spy(R), title('B(p,p)')
```



This example is continued in the reference pages for symamd.

The bandwidth can also be computed with

```
[i,j] = find(B);
bw = max(i-j) + 1;
```

The bandwidths of B and R are 35 and 12, respectively.

See Also

colamd, colperm, symamd

References

[1] George, Alan and Joseph Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[2] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis*, 1992. A slightly expanded version is also available as a technical report from the Xerox Palo Alto Research Center.

symvar

Purpose Determine symbolic variables in expression

Syntax `symvar 'expr'`
`s = symvar('expr')`

Description `symvar 'expr'` searches the expression, `expr`, for identifiers other than `i`, `j`, `pi`, `inf`, `nan`, `eps`, and common functions. `symvar` displays those variables that it finds or, if no such variable exists, displays an empty cell array, `{}`.

`s = symvar('expr')` returns the variables in a cell array of strings, `s`. If no such variable exists, `s` is an empty cell array.

Examples `symvar` finds variables `beta1` and `x`, but skips `pi` and the `cos` function.

```
symvar 'cos(pi*x - beta1)'
```

```
ans =
```

```
    'beta1'
```

```
    'x'
```

See Also `findstr`

Purpose Synchronize and resample two timeseries objects using common time vector

Syntax `[ts1 ts2] = synchronize(ts1,ts2,'SynchronizeMethod')`

Description `[ts1 ts2] = synchronize(ts1,ts2,'SynchronizeMethod')` creates two new timeseries objects by synchronizing `ts1` and `ts2` using a common time vector. The string `'SynchronizeMethod'` defines the method for synchronizing the timeseries and can be one of the following:

- `'Union'` — Resample timeseries objects using a time vector that is a union of the time vectors of `ts1` and `ts2` on the time range where the two time vectors overlap.
- `'Intersection'` — Resample timeseries objects on a time vector that is the intersection of the time vectors of `ts1` and `ts2`.
- `'Uniform'` — Requires an additional argument as follows:

```
[ts1 ts2] = synchronize(ts1,ts2,'Uniform','Interval',value)
```

This method resamples time series on a uniform time vector, where `value` specifies the time interval between the two samples. The uniform time vector is the overlap of the time vectors of `ts1` and `ts2`. The interval units are assumed to be the smaller units of `ts1` and `ts2`.

You can specify additional arguments by using property-value pairs:

- `'InterpMethod'`: Forces the specified interpolation method (over the default method) for this `synchronize` operation. Can be either a string, `'linear'` or `'zoh'`, or a `tsdata.interpolation` object that contains a user-defined interpolation method.
- `'QualityCode'`: Integer (between -128 and 127) used as the quality code for both time series after the synchronization.

synchronize

- 'KeepOriginalTimes': Logical value (true or false) indicating whether the new time series should keep the original time values. For example,

```
ts1 = timeseries([1 2],[datestr(now); datestr(now+1)]);  
ts2 = timeseries([1 2],[datestr(now-1); datestr(now)]);
```

Note that `ts1.timeinfo.StartDate` is one day after `ts2.timeinfo.StartDate`. If you use

```
[ts1 ts2] = synchronize(ts1,ts2,'union');
```

the `ts1.timeinfo.StartDate` is changed to match `ts2.TimeInfo.StartDate` and `ts1.Time` changes to 1.

But if you use

```
[ts1 ts2] =  
synchronize(ts1,ts2,'union','KeepOriginalTimes',true);
```

`ts1.timeinfo.StartDate` is unchanged and `ts1.Time` is still 0.

- 'tolerance': Real number used as the tolerance for differentiating two time values when comparing the `ts1` and `ts2` time vectors. The default tolerance is $1e-10$. For example, when the sixth time value in `ts1` is $5+(1e-12)$ and the sixth time value in `ts2` is $5-(1e-13)$, both values are treated as 5 by default. To differentiate those two times, you can set 'tolerance' to a smaller value such as $1e-15$, for example.

See Also

`timeseries`

Purpose

Two ways to call MATLAB functions

Description

You can call MATLAB functions using either *command syntax* or *function syntax*, as described below.

Command Syntax

A function call in this syntax consists of the function name followed by one or more arguments separated by spaces:

```
functionname arg1 arg2 ... argn
```

Command syntax does not allow you to obtain any values that might be returned by the function. Attempting to assign output from the function to a variable using command syntax generates an error. Use function syntax instead.

Examples of command syntax:

```
save mydata.mat x y z
import java.awt.Button java.lang.String
```

Arguments are treated as string literals. See the examples below, under “Argument Passing” on page 2-3050.

Function Syntax

A function call in this syntax consists of the function name followed by one or more arguments separated by commas and enclosed in parentheses:

```
functionname(arg1, arg2, ..., argn)
```

You can assign the output of the function to one or more output values. When assigning to more than one output variable, separate the variables by commas or spaces and enclose them in square brackets ([]):

```
[out1, out2, ..., outn] = functionname(arg1, arg2, ..., argn)
```

Examples of function syntax:

```
copyfile('srcfile', '..\mytests', 'writable')
[x1, x2, x3, x4] = deal(A{:})
```

Arguments are passed to the function by value. See the examples below, under “Argument Passing” on page 2-3050.

Argument Passing

When calling a function using command syntax, MATLAB passes the arguments as string literals. When using function syntax, arguments are passed by value.

In the following example, assign a value to `A` and then call `disp` on the variable to display the value passed. Calling `disp` with command syntax passes the variable name, `'A'`:

```
A = pi;
disp A
A
```

while function syntax passes the value assigned to `A`:

```
A = pi;
disp(A)
3.1416
```

The next example passes two strings to `strcmp` for comparison. Calling the function with command syntax compares the variable names, `'str1'` and `'str2'`:

```
str1 = 'one';    str2 = 'one';
strcmp str1 str2
ans =
    0           (unequal)
```

while function syntax compares the values assigned to the variables, `'one'` and `'one'`:

```
str1 = 'one';    str2 = 'one';
strcmp(str1, str2)
```

```
ans =  
    1      (equal)
```

Passing Strings

When using the function syntax to pass a string literal to a function, you must enclose the string in single quotes, ('string'). For example, to create a new directory called myapptests, use

```
mkdir('myapptests')
```

On the other hand, variables that contain strings do not need to be enclosed in quotes:

```
dirname = 'myapptests';  
mkdir(dirname)
```

See Also

`mlint`

system

Purpose Execute operating system command and return result

Syntax `system('command')`
`[status, result] = system('command')`

Description `system('command')` calls upon the operating system to run `command`, for example `dir` or `ls` or a UNIX shell script, and directs the output to MATLAB. If `command` runs successfully, `ans` is 0. If `command` fails or does not exist on your operating system, `ans` is a nonzero value and an explanatory message appears.

`[status, result] = system('command')` calls upon the operating system to run `command`, and directs the output to MATLAB. If `command` runs successfully, `status` is 0 and `result` contains the output from `command`. If `command` fails or does not exist on your operating system, `status` is a nonzero value and `result` contains an explanatory message.

Note Running `system` on Windows with a command that relies upon the current directory will fail when the current directory is specified using a UNC pathname. This is because DOS does not support UNC pathnames. In that event, MATLAB returns this error:

```
??? Error using ==> system DOS commands may not be  
executed when the current directory is a UNC pathname.
```

To work around this limitation, change the directory to a mapped drive prior to running `system` or a function that calls `system`.

Examples On a Windows system, display the current directory by accessing the operating system.

```
[status currdir] = system('cd')  
status =  
    0  
currdir =
```

D:\work\matlab\test

See Also

! (bang), computer, dos, perl, unix, winopen

“Running External Programs” in the MATLAB Desktop Tools and Development Environment documentation

tan

Purpose Tangent of argument in radians

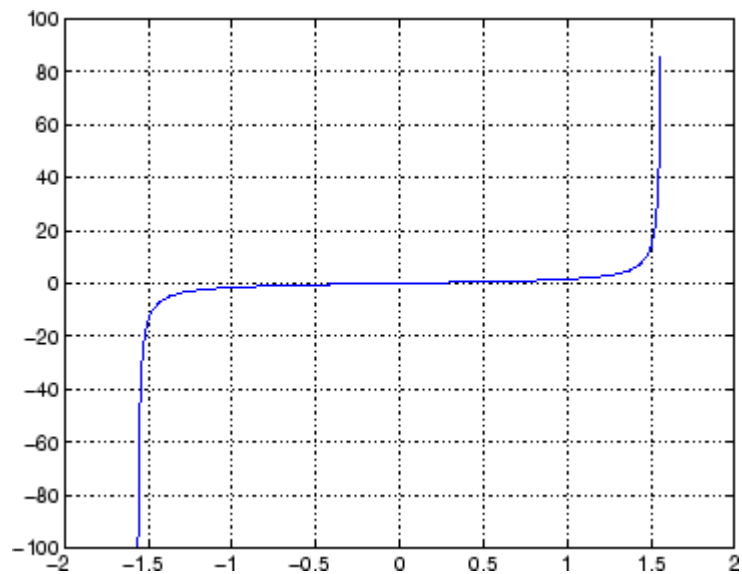
Syntax $Y = \tan(X)$

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

$Y = \tan(X)$ returns the circular tangent of each element of X .

Examples Graph the tangent function over the domain $-\pi/2 < x < \pi/2$.

```
x = (-pi/2)+0.01:0.01:(pi/2)-0.01;  
plot(x,tan(x)), grid on
```



The expression $\tan(\pi/2)$ does not evaluate as infinite but as the reciprocal of the floating point accuracy `eps` since `pi` is only a floating-point approximation to the exact value of π .

Definition The tangent can be defined as

$$\tan(z) = \frac{\sin(z)}{\cos(z)}$$

Algorithm

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

See Also

tand, tanh, atan, atan2, atand, atanh

tand

Purpose Tangent of argument in degrees

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

Description $Y = \text{tand}(X)$ is the tangent of the elements of X , expressed in degrees. For odd integers n , $\text{tand}(n*90)$ is infinite, whereas $\tan(n*\pi/2)$ is large but finite, reflecting the accuracy of the floating point value of π .

See Also `tan`, `tanh`, `atan`, `atan2`, `atand`, `atanh`

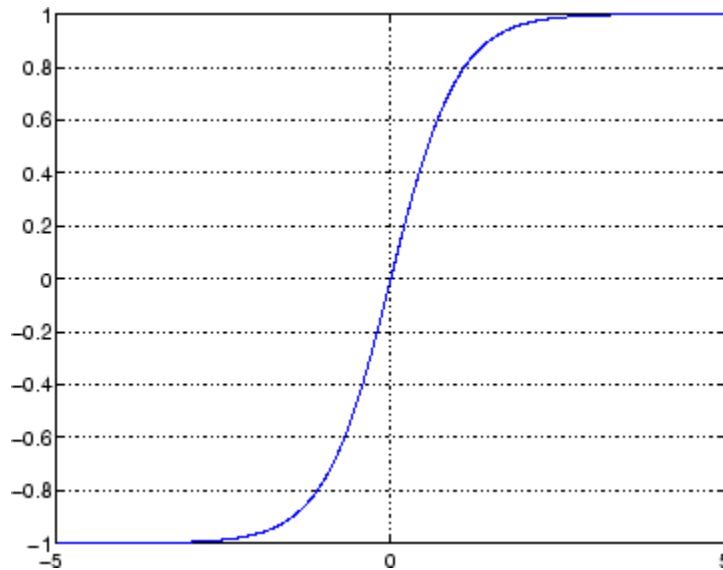
Purpose Hyperbolic tangent

Syntax $Y = \tanh(X)$

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

Examples Graph the hyperbolic tangent function over the domain $-5 \leq x \leq 5$.

```
x = -5:0.01:5;  
plot(x,tanh(x)), grid on
```



Definition The hyperbolic tangent can be defined as

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$$

tanh

Algorithm

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

See Also

atan, atan2, tan

Purpose Compress files into tar file

Syntax

```
tar(tarfilename,files)
tar(tarfilename,files,rootdir)
entrynames = tar(...)
```

Description `tar(tarfilename,files)` creates a tar file with the name `tarfilename` from the list of files and directories specified in `files`. Relative paths are stored in the tar file, but absolute paths are not. Directories recursively include all of their content.

`tarfilename` is a string specifying the name of the tar file. The `.tar` extension is appended to `tarfilename` if omitted. The `tarfilename` extension can end in `.tgz` or `.gz`. In this case, `tarfilename` is gzipped.

`files` is a string or cell array of strings containing the list of files or directories included in `tarfilename`. Individual files that are on the MATLAB path can be specified as partial pathnames. Otherwise an individual file can be specified relative to the current directory or with an absolute path. Directories must be specified relative to the current directory or with absolute paths. On UNIX systems, directories can also start with `~/` or `~username/`, which expands to the current user's home directory or the specified user's home directory, respectively. The wildcard character `*` can be used when specifying files or directories, except when relying on the MATLAB path to resolve a filename or partial pathname.

`tar(tarfilename,files,rootdir)` allows the path for `files` to be specified relative to `rootdir` rather than the current directory.

`entrynames = tar(...)` returns a string cell array of the relative path entry names contained in `tarfilename`.

Example Tar all files in the current directory to the file `backup.tgz`:

```
tar('backup.tgz','.');
```

See Also `gzip`, `gunzip`, `untar`, `unzip`, `zip`

tempdir

Purpose	Name of system's temporary directory
Syntax	<code>tmp_dir = tempdir</code>
Description	<code>tmp_dir = tempdir</code> returns the name of the system's temporary directory, if one exists. This function does not create a new directory. See "Opening Temporary Files and Directories" for more information.
See Also	<code>tempname</code>

Purpose Unique name for temporary file

Syntax tmp_nam = tempname

Description tmp_nam = tempname returns a unique string, tmp_nam, suitable for use as a temporary filename.

Note The filename that tempname generates is not guaranteed to be unique; however, it is likely to be so.

See “Opening Temporary Files and Directories” for more information.

See Also tmpdir

tetramesh

Purpose Tetrahedron mesh plot

Syntax

```
tetramesh(T,X,c)
tetramesh(T,X)
h = tetramesh(...)
tetramesh(...,'param','value','param','value'...)
```

Description `tetramesh(T,X,c)` displays the tetrahedrons defined in the m -by-4 matrix T as mesh. T is usually the output of `delaunayn`. A row of T contains indices into X of the vertices of a tetrahedron. X is an n -by-3 matrix, representing n points in 3 dimension. The tetrahedron colors are defined by the vector C , which is used as indices into the current colormap.

Note If T is the output of `delaunay3`, then X is the concatenation of the `delaunay3` input arguments x , y , z interpreted as column vectors, i.e., $X = [x(:) \ y(:) \ z(:)]$.

`tetramesh(T,X)` uses $C = 1:m$ as the color for the m tetrahedrons. Each tetrahedron has a different color (modulo the number of colors available in the current colormap).

`h = tetramesh(...)` returns a vector of tetrahedron handles. Each element of h is a handle to the set of patches forming one tetrahedron. You can use these handles to view a particular tetrahedron by turning the patch 'Visible' property 'on' or 'off'.

`tetramesh(...,'param','value','param','value'...)` allows additional patch property name/property value pairs to be used when displaying the tetrahedrons. For example, the default transparency parameter is set to 0.9. You can overwrite this value by using the property name/property value pair ('FaceAlpha',value) where value is a number between 0 and 1. See Patch Properties for information about the available properties.

Examples

Generate a 3-dimensional Delaunay tessellation, then use tetramesh to visualize the tetrahedrons that form the corresponding simplex.

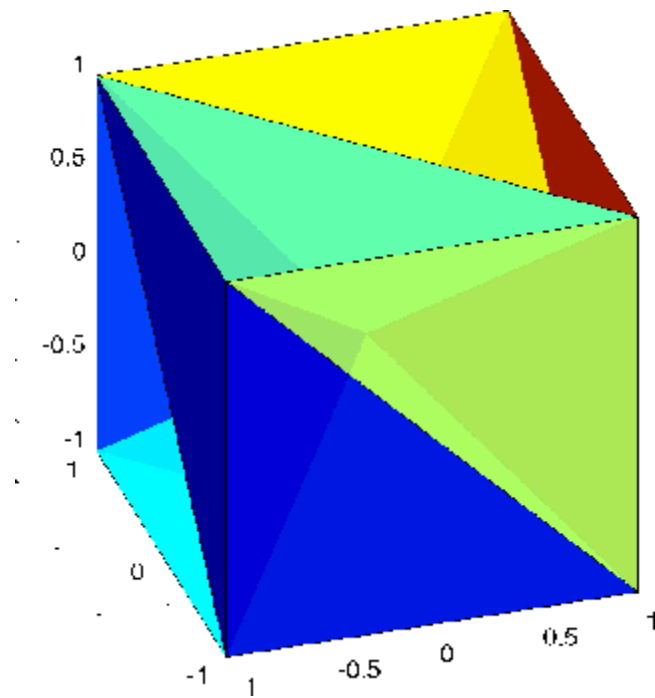
```
d = [-1 1];  
[x,y,z] = meshgrid(d,d,d); % A cube  
x = [x(:);0];  
y = [y(:);0];  
z = [z(:);0];  
% [x,y,z] are corners of a cube plus the center.  
X = [x(:) y(:) z(:)];  
Tes = delaunayn(X)
```

```
Tes =
```

```
 9  1  5  6  
 3  9  1  5  
 2  9  1  6  
 2  3  9  4  
 2  3  9  1  
 7  9  5  6  
 7  3  9  5  
 8  7  9  6  
 8  2  9  6  
 8  2  9  4  
 8  3  9  4  
 8  7  3  9
```

```
tetramesh(Tes,X);camorbit(20,0)
```

tetramesh



See Also

`delaunayn`, `patch`, `Patch Properties`, `trimesh`, `trisurf`

Purpose Produce TeX format from character string

Syntax `texlabel(f)`
`texlabel(f, 'literal')`

Description `texlabel(f)` converts the MATLAB expression `f` into the TeX equivalent for use in text strings. It processes Greek variable names (e.g., `lambda`, `delta`, etc.) into a string that is displayed as actual Greek letters.

`texlabel(f, 'literal')` prints Greek variable names as literals.

If the string is too long to fit into a figure window, then the center of the expression is replaced with a tilde ellipsis (~~~).

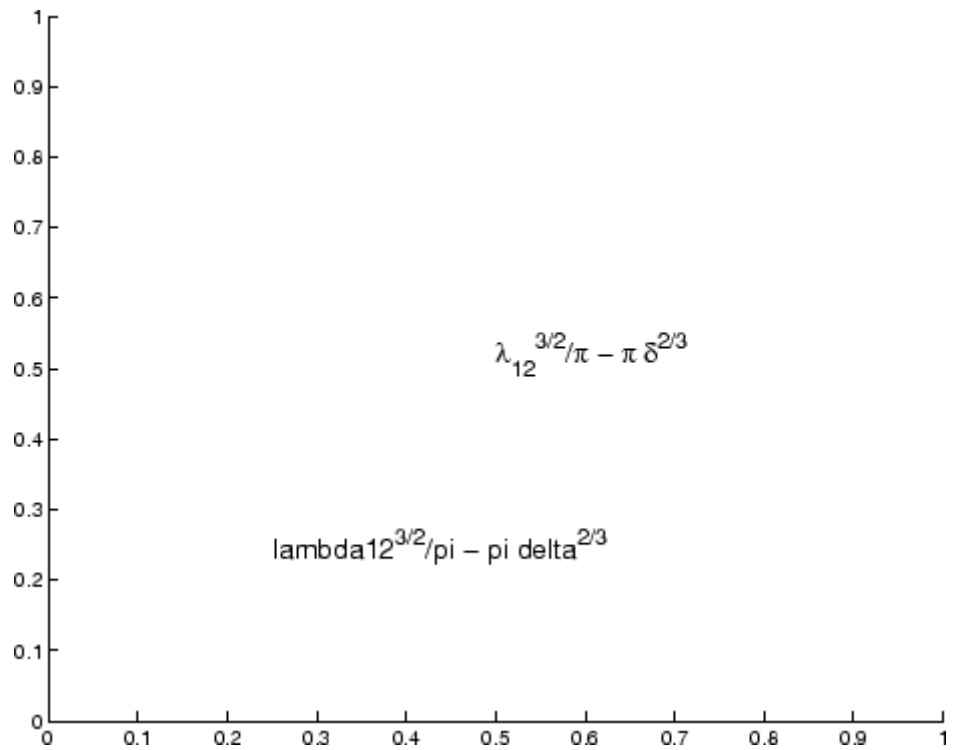
Examples You can use `texlabel` as an argument to the `title`, `xlabel`, `ylabel`, `zlabel`, and `text` commands. For example,

```
title(texlabel('sin(sqrt(x^2 + y^2))/sqrt(x^2 + y^2)'))
```

By default, `texlabel` translates Greek variable names to the equivalent Greek letter. You can select literal interpretation by including the `literal` argument. For example, compare these two commands.

```
text(.5,.5,...
      texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)'))
text(.25,.25,...
      texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)', 'literal'))
```

textlabel



See Also

text, title, xlabel, ylabel, zlabel, the text String property
“Annotating Plots” on page 1-83 for related functions

Purpose

Create text object in current axes

Syntax

```
text(x,y,'string')
text(x,y,z,'string')
text(x,y,z,'string','PropertyName',PropertyValue....)
text('PropertyName',PropertyValue....)
h = text(...)
```

Description

`text` is the low-level function for creating text graphics objects. Use `text` to place character strings at specified locations.

`text(x,y,'string')` adds the string in quotes to the location specified by the point (x,y) .

`text(x,y,z,'string')` adds the string in 3-D coordinates.

`text(x,y,z,'string','PropertyName',PropertyValue....)` adds the string in quotes to the location defined by the coordinates and uses the values for the specified text properties. See the text property list section at the end of this page for a list of text properties.

`text('PropertyName',PropertyValue....)` omits the coordinates entirely and specifies all properties using property name/property value pairs.

`h = text(...)` returns a column vector of handles to text objects, one handle per object. All forms of the `text` function optionally return this output argument.

See the String property for a list of symbols, including Greek letters.

Remarks**Position Text Within the Axes**

The default text units are the units used to plot data in the graph. Specify the text location coordinates (the x , y , and z arguments) in the data units of the current graph (see “Example”). You can use other units to position the text by set the text `Units` property to `normalized` or one of the nonrelative units (pixels, inches, centimeters, points).

Note that the Axes Units property controls the positioning of the Axes within the figure and is not related to the axes data units used for graphing.

The Extent, VerticalAlignment, and HorizontalAlignment properties control the positioning of the character string with regard to the text location point.

If the coordinates are vectors, text writes the string at all locations defined by the list of points. If the character string is an array the same length as x, y, and z, text writes the corresponding row of the string array at each point specified.

Multiline Text

When specifying strings for multiple text objects, the string can be

- A cell array of strings
- A padded string matrix
- A string vector using vertical slash characters ('|') as separators.

Each element of the specified string array creates a different text object.

When specifying the string for a single text object, cell arrays of strings and padded string matrices result in a text object with a multiline string, while vertical slash characters are not interpreted as separators and result in a single line string containing vertical slashes.

Behavior of the Text Function

text is a low-level function that accepts property name/property value pairs as input arguments. However, the convenience form,

```
text(x,y,z,'string')
```

is equivalent to

```
text('Position',[x,y,z],'String','string')
```

You can specify other properties only as property name/property value pairs. See the text property list at the end of this page for a description of each property. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the set and get reference pages for examples of how to specify these data types).

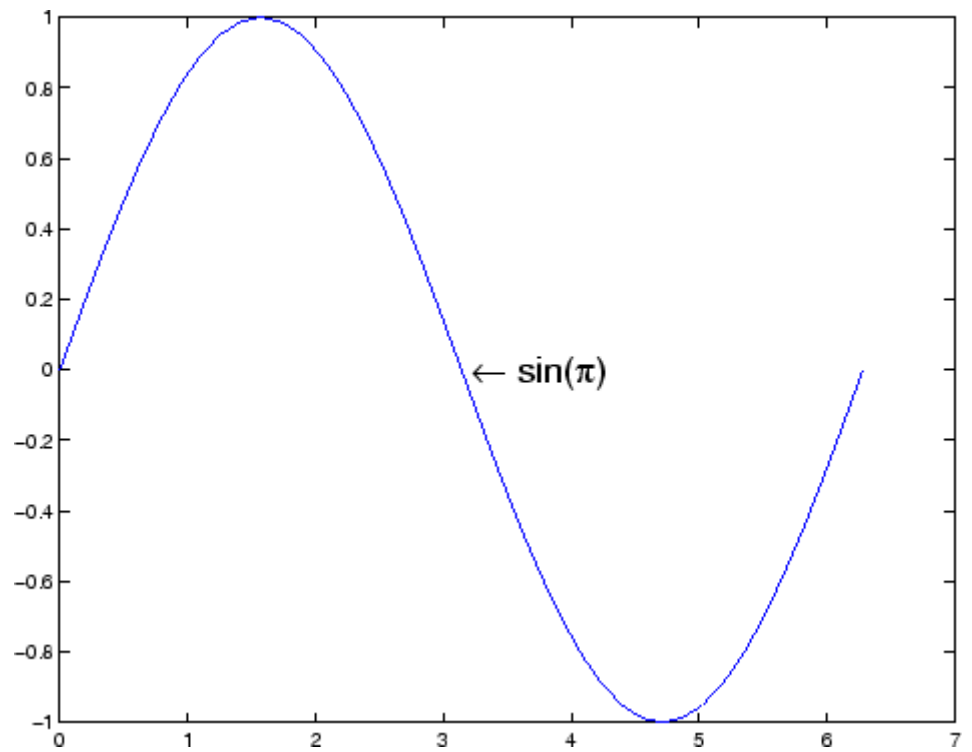
text does not respect the setting of the figure or axes NextPlot property. This allows you to add text objects to an existing axes without setting hold to on.

Examples

The statements

```
plot(0:pi/20:2*pi,sin(0:pi/20:2*pi))
text(pi,0,' \leftarrow sin(\pi)', 'FontSize',18)
```

annotate the point at $(\pi,0)$ with the string $\sin(\pi)$



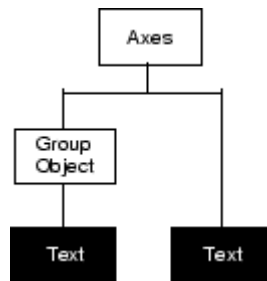
The statement

```
text(x,y,'\ite^{i\omega\tau} = cos(\omega\tau) + i sin(\omega\tau)')
```

uses embedded TeX sequences to produce

$$e^{i\omega\tau} = \cos(\omega\tau) + i \sin(\omega\tau)$$

Object Hierarchy



Setting Default Properties

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

```
set(0, 'DefaulttextProperty',PropertyValue...)  
set(gcf, 'DefaulttextProperty',PropertyValue...)  
set(gca, 'DefaulttextProperty',PropertyValue...)
```

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

See Also

`annotation`, `gtext`, `int2str`, `num2str`, `title`, `xlabel`, `ylabel`, `zlabel`, `strings`

“Object Creation Functions” on page 1-90 for related functions

Text Properties for property descriptions

Text Properties

Purpose Defines the stext properties

Modifying Properties You can set and query graphics object properties using the property editor or the set and get commands.

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

To change the default values of properties, see [Setting Default Property Values](#).

See [Core Objects](#) for general information about this type of object.

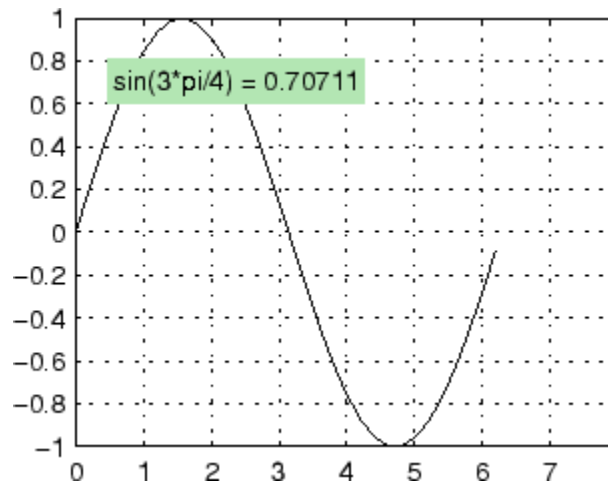
Text Property Descriptions

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

BackgroundColor
ColorSpec | {none}

Color of text extent rectangle. This property enables you to define a color for the rectangle that encloses the text Extent. For example, the following code creates a text object that labels a plot and sets the background color to light green.

```
text(3*pi/4,sin(3*pi/4),...  
 ['sin(3*pi/4) = ',num2str(sin(3*pi/4))],...  
 'HorizontalAlignment','center',...  
 'BackgroundColor',[.7 .9 .7]);
```

For additional features, see the following properties:

- `EdgeColor` — Color of the rectangle's edge (none by default).
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increase the size of the rectangle by adding a margin to the existing text extent rectangle.

See also [Drawing Text in a Box](#) in the MATLAB Graphics documentation for an example using background color with contour labels.

`BeingDeleted`
on | {off} read only

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

Text Properties

(see the `DeleteFcn` property) It remains set to on while the delete function executes, after which the object no longer exists.

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

`BusyAction`
cancel | {queue}

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is set to off, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
string or function handle

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the text object.

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

Set this property to a function handle that references the callback. You can also use a string that is a valid MATLAB expression or the name of an M-file. The expressions execute in the MATLAB workspace.

See `SelectionType` for information on how to use function handles to define the callback function.

Children

matrix (read only)

The empty matrix; text objects have no children.

Clipping

on | {off}

Clipping mode. When Clipping is on, MATLAB does not display any portion of the text that is outside the axes.

Color

ColorSpec

Text color. A three-element RGB vector or one of the predefined names, specifying the text color. The default value for Color is white. See `ColorSpec` for more information on specifying color.

CreateFcn

string or function handle

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a text object. You must define this property as a default value for text or in a call to the text function that creates a new text object. For example, the statement

```
set(0, 'DefaultTextCreateFcn', ...  
    'set(gcf, ''Pointer'', ''crosshair'')')
```

Text Properties

defines a default value on the root level that sets the figure Pointer property to crosshairs whenever you create a text object. MATLAB executes this routine after setting all text properties. Setting this property on an existing text object has no effect.

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

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

DeleteFcn
string or function handle

Delete text callback routine. A callback routine that executes when you delete the text object (e.g., when you issue a delete command or clear the axes or figure). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

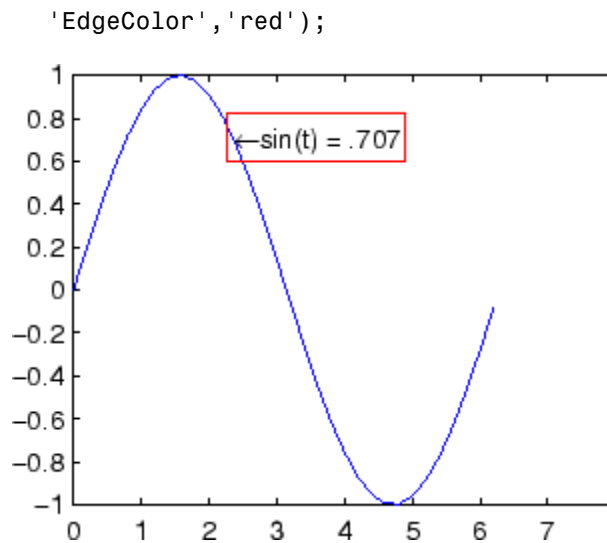
The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.

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

EdgeColor
ColorSpec | {none}

Color of edge drawn around text extent rectangle. This property enables you to specify the color of a box drawn around the text Extent. For example, the following code draws a red rectangle around text that labels a plot.

```
text(3*pi/4,sin(3*pi/4),...  
'\leftarrowsin(t) = .707',...
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (none by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increases the size of the rectangle by adding a margin to the existing text extent rectangle

Editing

on | {off}

Enable or disable editing mode. When this property is set to the default off, you cannot edit the text string interactively (i.e., you must change the `String` property to change the text). When this property is set to on, MATLAB places an insert cursor at the end of the text string and enables editing. To apply the new text string,

Text Properties

- 1 Press the **Esc** key.
- 2 Click in any figure window (including the current figure).
- 3 Reset the Editing property to off.

MATLAB then updates the String property to contain the new text and resets the Editing property to off. You must reset the Editing property to on to resume editing.

EraseMode

{normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase text objects. Alternative erase modes are useful for creating animated sequences where controlling the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- **none** — Do not erase the text when it is moved or destroyed. While the object is still visible on the screen after erasing with EraseMode none, you cannot print it because MATLAB stores no information about its former location.
- **xor** — Draw and erase the text by performing an exclusive OR (XOR) with each pixel index of the screen beneath it. When the text is erased, it does not damage the objects beneath it. However, when text is drawn in xor mode, its color depends on the color of the screen beneath it. It is correctly colored only when it is over axes background Color, or the figure background Color if the axes Color is set to none.
- **background** — Erase the text by drawing it in the axes background Color, or the figure background Color if the axes Color is set to none.

This damages objects that are behind the erased text, but text is always properly colored.

Printing with Nonnormal Erase Modes

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

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

Extent

`position` rectangle (read only)

Position and size of text. A four-element read-only vector that defines the size and position of the text string

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

If the `Units` property is set to `data` (the default), `left` and `bottom` are the x - and y -coordinates of the lower left corner of the text Extent.

For all other values of `Units`, `left` and `bottom` are the distance from the lower left corner of the axes position rectangle to the lower left corner of the text Extent. `width` and `height` are the dimensions of the Extent rectangle. All measurements are in units specified by the `Units` property.

FontAngle

```
{normal} | italic | oblique
```

Text Properties

Character slant. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to italic or oblique selects a slanted font.

FontName

A name, such as Courier, or the string FixedWidth

Font family. A string specifying the name of the font to use for the text object. To display and print properly, this must be a font that your system supports. The default font is Helvetica.

Specifying a Fixed-Width Font

If you want text to use a fixed-width font that looks good in any locale, you should set FontName to the string FixedWidth:

```
set(text_handle, 'FontName', 'FixedWidth')
```

This eliminates the need to hard-code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan where multibyte character sets are used). A properly written MATLAB application that needs to use a fixed-width font should set FontName to FixedWidth (note that this string is case sensitive) and rely on FixedWidthFontName to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root FixedWidthFontName property to the appropriate value for that locale from startup.m.

Note that setting the root FixedWidthFontName property causes an immediate update of the display to use the new font.

FontSize

size in FontUnits

Font size. A value specifying the font size to use for text in units determined by the FontUnits property. The default point size is 10 (1 point = 1/72 inch).

FontWeight

light | {normal} | demi | bold

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

FontUnits

{points} | normalized | inches |
centimeters | pixels

Font size units. MATLAB uses this property to determine the units used by the FontSize property. Normalized units interpret FontSize as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen FontSize accordingly. pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).

Note that if you are setting both the FontSize and the FontUnits in one function call, you must set the FontUnits property first so that MATLAB can correctly interpret the specified FontSize.

HandleVisibility

{on} | callback | off

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. HandleVisibility is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when HandleVisibility is set to on.

Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from

Text Properties

the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

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

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

When a handle's visibility is restricted using `callback` or `off`,

- The object's handle does not appear in its parent's `Children` property.
- Figures do not appear in the root's `CurrentFigure` property.
- Objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property.
- Axes do not appear in their parent's `CurrentAxes` property.

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

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

`HitTest`
{on} | off

Selectable by mouse click. `HitTest` determines if the text can become the current object (as returned by the `gco` command and

the figure `CurrentObject` property) as a result of a mouse click on the text. If `HitTest` is set to `off`, clicking the text selects the object below it (which is usually the axes containing it).

For example, suppose you define the button down function of an image (see the `ButtonDownFcn` property) to display text at the location you click with the mouse.

First define the callback routine.

```
function bd_function
pt = get(gca,'CurrentPoint');
text(pt(1,1),pt(1,2),pt(1,3),...
    '\fontsize{20}\oplus The spot to label',...
    'HitTest','off')
```

Now display an image, setting its `ButtonDownFcn` property to the callback routine.

```
load earth
image(X,'ButtonDownFcn','bd_function'); colormap(map)
```

When you click the image, MATLAB displays the text string at that location. With `HitTest` set to `off`, existing text cannot intercept any subsequent button down events that occur over the text. This enables the image's button down function to execute.

`HorizontalAlignment`
{left} | center | right

Horizontal alignment of text. This property specifies the horizontal justification of the text string. It determines where MATLAB places the string with regard to the point specified by the `Position` property. The following picture illustrates the alignment options.

Text Properties

HorizontalAlignment viewed with the VerticalAlignment set to middle (the default).



See the Extent property for related information.

Interpreter

latex | {tex} | none

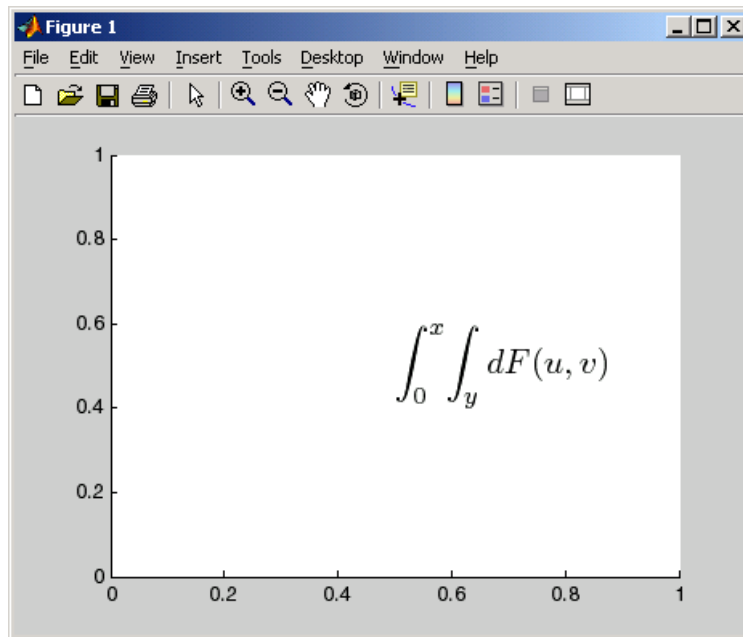
Interpret T_EX instructions. This property controls whether MATLAB interprets certain characters in the String property as T_EX instructions (default) or displays all characters literally. The options are:

- latex — Supports the full L_AT_EX markup language.
- tex — Supports a subset of plain T_EX markup language. See the String property for a list of supported T_EX instructions.
- none — Displays literal characters.

Latex Interpreter

To enable the L_AT_EX interpreter for text objects, set the Interpreter property to latex. For example, the following statement displays an equation in a figure at the point [.5 .5], and enlarges the font to 16 points.

```
text('Interpreter','latex',...  
    'String','$$\int_0^x \! \int_y dF(u,v)$$',...  
    'Position',[.5 .5],...  
    'FontSize',16)
```



Information About Using TEX

The following references may be useful to people who are not familiar with T_EX.

- Donald E. Knuth, *The T_EXbook*, Addison Wesley, 1986.
- The T_EX Users Group home page: <http://www.tug.org>

Interruptible
{on} | off

Callback routine interruption mode. The Interruptible property controls whether a text callback routine can be interrupted by subsequently invoked callback routines. Text objects have three properties that define callback routines: ButtonDownFcn,

Text Properties

CreateFcn, and DeleteFcn. See the BusyAction property for information on how MATLAB executes callback routines.

LineStyle

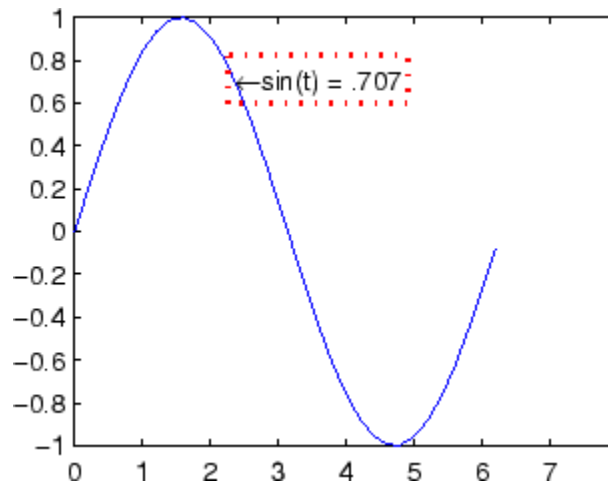
{-} | -- | : | -. | none

Edge line type. This property determines the line style used to draw the edges of the text Extent. The available line styles are shown in the following table.

Symbol	Line Style
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line
none	No line

For example, the following code draws a red rectangle with a dotted line style around text that labels a plot.

```
text(3*pi/4,sin(3*pi/4),...  
     '\leftarrow sin(t) = .707',...  
     'EdgeColor','red',...  
     'LineWidth',2,...  
     'LineStyle',':');
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (none by default)
- `EdgeColor` — Color of the rectangle's edge (none by default)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increases the size of the rectangle by adding a margin to the existing text extent rectangle

`LineWidth`

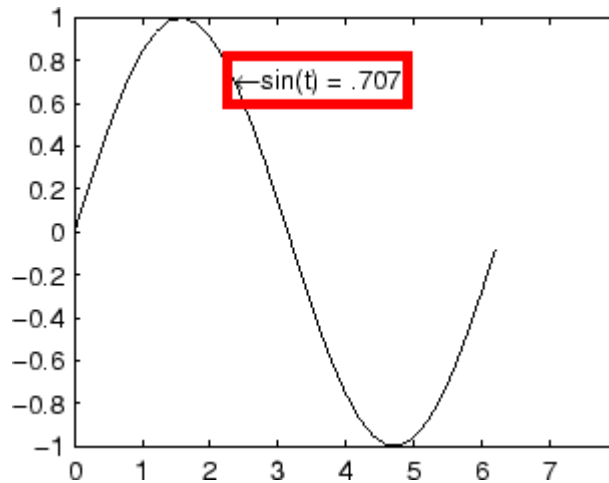
scalar (points)

Width of line used to draw text extent rectangle. When you set the text `EdgeColor` property to a color (the default is none), MATLAB displays a rectangle around the text `Extent`. Use the `LineWidth` property to specify the width of the rectangle edge. For example, the following code draws a red rectangle around text that labels a plot and specifies a line width of 3 points:

```
text(3*pi/4,sin(3*pi/4),...  
'\leftarrow sin(t) = .707',...
```

Text Properties

```
'EdgeColor','red',...  
'LineWidth',3);
```



For additional features, see the following properties:

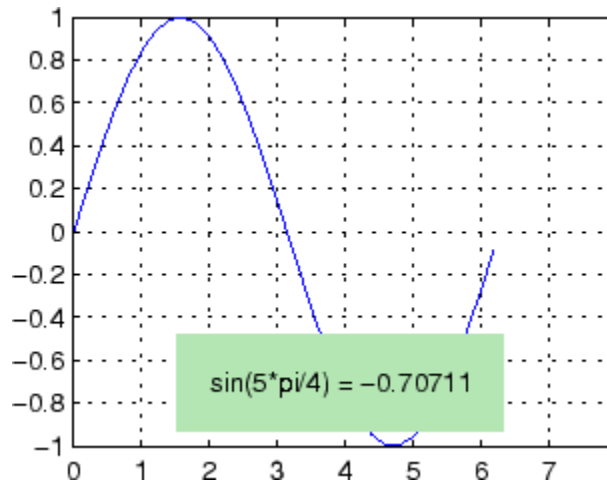
- `BackgroundColor` — Color of the rectangle's interior (none by default)
- `EdgeColor` — Color of the rectangle's edge (none by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increases the size of the rectangle by adding a margin to the existing text extent rectangle

`Margin`

scalar (pixels)

Distance between the text extent and the rectangle edge. When you specify a color for the `BackgroundColor` or `EdgeColor` text properties, MATLAB draws a rectangle around the area defined by the text `Extent` plus the value specified by the `Margin`. For example, the following code displays a light green rectangle with a 10-pixel margin.


```
text(5*pi/4,sin(5*pi/4),...  
    ['sin(5*pi/4) = ',num2str(sin(5*pi/4))],...  
    'HorizontalAlignment','center',...  
    'BackgroundColor',[.7 .9 .7],...  
    'Margin',10);
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (none by default)
- `EdgeColor` — Color of the rectangle's edge (none by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)

Parent

handle of axes, `hggroup`, or `hgtransform`

Parent of text object. This property contains the handle of the text object's parent. The parent of a text object is the axes, `hggroup`, or `hgtransform` object that contains it.

Text Properties

See [Objects That Can Contain Other Objects](#) for more information on parenting graphics objects.

Position

[x,y,[z]]

Location of text. A two- or three-element vector, [x y [z]], that specifies the location of the text in three dimensions. If you omit the z value, it defaults to 0. All measurements are in units specified by the Units property. Initial value is [0 0 0].

Rotation

scalar (default = 0)

Text orientation. This property determines the orientation of the text string. Specify values of rotation in degrees (positive angles cause counterclockwise rotation).

Selected

on | {off}

Is object selected? When this property is set to on, MATLAB displays selection handles if the SelectionHighlight property is also set to on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

SelectionHighlight

{on} | off

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

String

string

The text string. Specify this property as a quoted string for single-line strings, or as a cell array of strings, or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as line breaks in text strings, and are drawn as part of the text string. See Mathematical Symbols, Greek Letters, and TeX Characters for an example.

When the text Interpreter property is set to Tex (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequences used to define them.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
<code>\alpha</code>	α	<code>\upsilon</code>	υ	<code>\sim</code>	\sim
<code>\beta</code>	β	<code>\phi</code>	Φ	<code>\leq</code>	\leq
<code>\gamma</code>	γ	<code>\chi</code>	χ	<code>\infty</code>	∞
<code>\delta</code>	δ	<code>\psi</code>	ψ	<code>\clubsuit</code>	\clubsuit
<code>\epsilon</code>	ϵ	<code>\omega</code>	ω	<code>\diamondsuit</code>	\diamondsuit
<code>\zeta</code>	ζ	<code>\Gamma</code>	Γ	<code>\heartsuit</code>	\heartsuit
<code>\eta</code>	η	<code>\Delta</code>	Δ	<code>\spadesuit</code>	\spadesuit
<code>\theta</code>	Θ	<code>\Theta</code>	Θ	<code>\leftrightarrow</code>	\leftrightarrow
<code>\vartheta</code>	ϑ	<code>\Lambda</code>	Λ	<code>\leftarrow</code>	\rightarrow
<code>\iota</code>	ι	<code>\Xi</code>	Ξ	<code>\uparrow</code>	\uparrow
<code>\kappa</code>	κ	<code>\Pi</code>	Π	<code>\rightarrow</code>	\leftrightarrow
<code>\lambda</code>	λ	<code>\Sigma</code>	Σ	<code>\downarrow</code>	\downarrow

Text Properties

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
<code>\mu</code>	μ	<code>\Upsilon</code>	Υ	<code>\circ</code>	\circ
<code>\nu</code>	ν	<code>\Phi</code>	Φ	<code>\pm</code>	\pm
<code>\xi</code>	ξ	<code>\Psi</code>	Ψ	<code>\geq</code>	\geq
<code>\pi</code>	π	<code>\Omega</code>	Ω	<code>\propto</code>	\propto
<code>\rho</code>	ρ	<code>\forall</code>	\forall	<code>\partial</code>	∂
<code>\sigma</code>	σ	<code>\exists</code>	\exists	<code>\bullet</code>	\bullet
<code>\varsigma</code>	ς	<code>\ni</code>	\ni	<code>\div</code>	\div
<code>\tau</code>	τ	<code>\cong</code>	\cong	<code>\neq</code>	\neq
<code>\equiv</code>	\equiv	<code>\approx</code>	\approx	<code>\aleph</code>	\aleph
<code>\Im</code>	\Im	<code>\Re</code>	\Re	<code>\wp</code>	\wp
<code>\otimes</code>	\otimes	<code>\oplus</code>	\oplus	<code>\oslash</code>	\oslash
<code>\cap</code>	\cap	<code>\cup</code>	\cup	<code>\supseteq</code>	\supseteq
<code>\supset</code>	\supset	<code>\subseteq</code>	\subseteq	<code>\subset</code>	\subset
<code>\int</code>	\int	<code>\in</code>	\in	<code>\o</code>	\circ
<code>\rfloor</code>	\rfloor	<code>\lceil</code>	\lceil	<code>\nabla</code>	∇
<code>\lfloor</code>	\lfloor	<code>\cdot</code>	\cdot	<code>\ldots</code>	\dots
<code>\perp</code>	\perp	<code>\neg</code>	\neg	<code>\prime</code>	$'$
<code>\wedge</code>	\wedge	<code>\times</code>	\times	<code>\emptyset</code>	\emptyset
<code>\rceil</code>	\rceil	<code>\surd</code>	\surd	<code>\mid</code>	$ $
<code>\vee</code>	\vee	<code>\varpi</code>	ϖ	<code>\copyright</code>	\copyright
<code>\langle</code>	\langle	<code>\rangle</code>	\rangle		

You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However,

you can use `\fontname` in combination with one of the other modifiers:

- `\bf` — Bold font
- `\it` — Italic font
- `\sl` — Oblique font (rarely available)
- `\rm` — Normal font
- `\fontname{fontname}` — Specify the name of the font family to use.
- `\fontsize{fontsize}` — Specify the font size in FontUnits.
- `\color{colorSpec}` — Specify color for succeeding characters

Stream modifiers remain in effect until the end of the string or only within the context defined by braces `{ }`.

Specifying Text Color in TeX Strings

Use the `\color` modifier to change the color of characters following it from the previous color (which is black by default). Syntax is:

- `\color{colorname}` for the eight basic named colors (red, green, yellow, magenta, blue, black, white), and plus the four Simulink colors (gray, darkGreen, orange, and lightBlue)

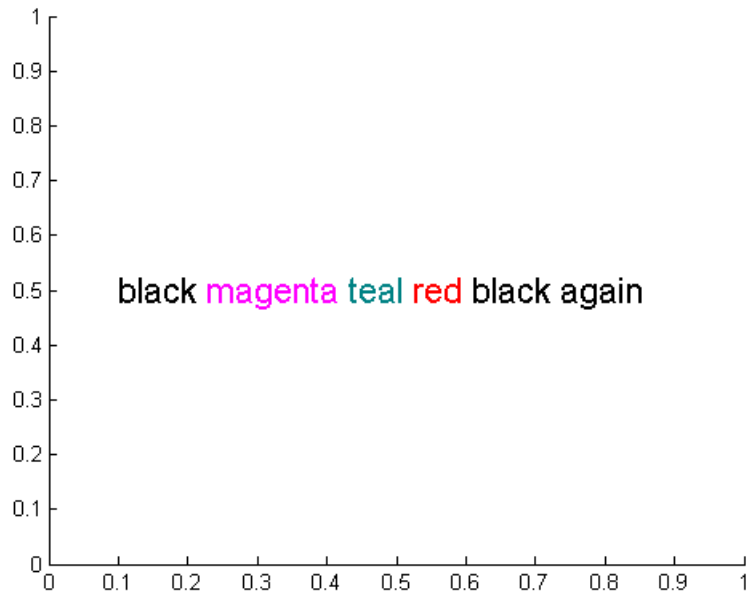
Note that short names (one-letter abbreviations) for colors are not supported by the `\color` modifier.

- `\color[rgb]{r g b}` to specify an RGB triplet with values between 0 and 1 as a cell array

For example,

```
text(.1,.5,['\fontsize{16}black {\color{magenta}magenta '...
'\color[rgb]{0 .5 .5}teal \color{red}red} black again'])
```

Text Properties



Specifying Subscript and Superscript Characters

The subscript character “`_`” and the superscript character “`^`” modify the character or substring defined in braces immediately following.

To print the special characters used to define the TeX strings when Interpreter is `Tex`, prefix them with the backslash “`\`” character: `\{`, `\}`, `_`, `\^`.

See the “Examples” on page 2-3069 in the text reference page for more information.

When Interpreter is set to `none`, no characters in the String are interpreted, and all are displayed when the text is drawn.

When Interpreter is set to `latex`, MATLAB provides a complete $\text{LaT}_\text{E}\text{X}$ interpreter for text objects. See the Interpreter property for more information.

Tag

string

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

Type

string (read only)

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

Units

pixels | normalized | inches |
centimeters | points | {data}

Units of measurement. This property specifies the units MATLAB uses to interpret the Extent and Position properties. All units are measured from the lower left corner of the axes plot box.

- Normalized units map the lower left corner of the rectangle defined by the axes to (0,0) and the upper right corner to (1.0,1.0).
- pixels, inches, centimeters, and points are absolute units (1 point = $1/72$ inch).
- data refers to the data units of the parent axes as determined by the data graphed (not the axes Units property, which controls the positioning of the within the figure window).

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

Text Properties

UserData
matrix

User-specified data. Any data you want to associate with the text object. MATLAB does not use this data, but you can access it using `set` and `get`.

UIContextMenu
handle of a `uicontextmenu` object

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

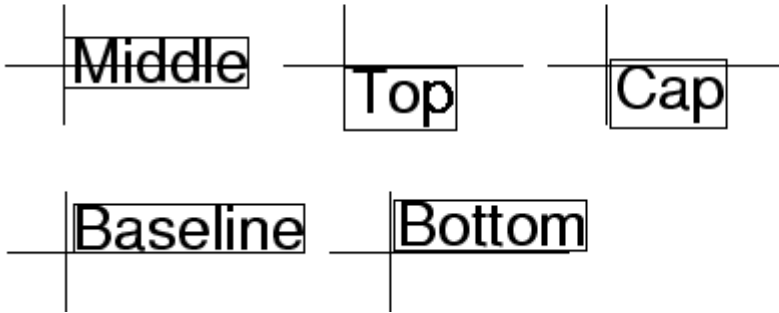
VerticalAlignment
top | cap | {middle} | baseline |
bottom

Vertical alignment of text. This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the `Position` property. The possible values mean

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

The following picture illustrates the alignment options.

Text VerticalAlignment property viewed with the **HorizontalAlignment** property set to left (the default).



Visible
{on} | off

Text visibility. By default, all text is visible. When set to off, the text is not visible, but still exists, and you can query and set its properties.

textread

Purpose

Read data from text file; write to multiple outputs

Note The textscan function is intended as a replacement for both textread and streadd.

Graphical Interface

As an alternative to textread, use the Import Wizard. To activate the Import Wizard, select **Import Data** from the **File** menu.

Syntax

```
[A,B,C,...] = textread(filename,format)
[A,B,C,...] = textread('filename','format',N)
[...] = textread(...,'param','value',...)
```

Description

[A,B,C,...] = textread(filename,format) reads data from the file 'filename' into the variables A,B,C, and so on, using the specified format, until the entire file is read. The filename and format inputs are strings, each enclosed in single quotes. textread is useful for reading text files with a known format. textread handles both fixed and free format files.

Note When reading large text files, reading from a specific point in a file, or reading file data into a cell array rather than multiple outputs, you might prefer to use the textscan function.

textread matches and converts groups of characters from the input. Each input field is defined as a string of non-white-space characters that extends to the next white-space or delimiter character, or to the maximum field width. Repeated delimiter characters are significant, while repeated white-space characters are treated as one.

The format string determines the number and types of return arguments. The number of return arguments is the number of items in the format string. The format string supports a subset of the conversion specifiers and conventions of the C language fscanf routine.

Values for the format string are listed in the table below. White-space characters in the format string are ignored.

format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a file that has Dept followed by a number (for department number), to skip the Dept and read only the number, use 'Dept' in the format string.	None
%d	Read a signed integer value.	Double array
%u	Read an integer value.	Double array
%f	Read a floating-point value.	Double array
%s	Read a white-space or delimiter-separated string.	Cell array of strings
%q	Read a double quoted string, ignoring the quotes.	Cell array of strings
%c	Read characters, including white space.	Character array
%[...]	Read the longest string containing characters specified in the brackets.	Cell array of strings
%[^...]	Read the longest nonempty string containing characters that are not specified in the brackets.	Cell array of strings
%*... instead of %	Ignore the matching characters specified by *.	No output
%w... instead of %	Read field width specified by w. The %f format supports %w.pf, where w is the field width and p is the precision.	

[A,B,C,...] = textread('filename','format',N) reads the data, reusing the format string N times, where N is an integer greater than zero. If N is smaller than zero, textread reads the entire file.

textread

[...] = textread(..., 'param', 'value', ...) customizes textread using param/value pairs, as listed in the table below.

param	value	Action
	' '\b \n \r \t	Space Backspace Newline Carriage return Horizontal tab
bufsize	Positive integer	Specifies the maximum string length, in bytes. Default is 4095.
commentstyle	matlab	Ignores characters after %.
commentstyle	shell	Ignores characters after #.
commentstyle	c	Ignores characters between /* and */.
commentstyle	c++	Ignores characters after //.
delimiter	One or more characters	Act as delimiters between elements. Default is none.
emptyvalue	Scalar double	Value given to empty cells when reading delimited files. Default is 0.
endofline	Single character or '\r\n'	Character that denotes the end of a line. Default is determined from file
expchars	Exponent characters	Default is eEdD.
headerlines	Positive integer	Ignores the specified number of lines at the beginning of the file.
whitespace	Any from the list below:	Treats vector of characters as white space. Default is ' \b\t'.

Note When textread reads a consecutive series of whitespace values, it treats them as one white space. When it reads a consecutive series of delimiter values, it treats each as a separate delimiter.

Remarks

If you want to preserve leading and trailing spaces in a string, use the whitespace parameter as shown here:

```
textread('myfile.txt', '%s', 'whitespace', '')
ans =
    '  An  example      of preserving      spaces  '
```

Examples

Example 1 – Read All Fields in Free Format File Using %

The first line of mydata.dat is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a free format file using the % format.

```
[names, types, x, y, answer] = textread('mydata.dat', ...
    '%s %s %f %d %s', 1)
```

returns

```
names =
    'Sally'
types =
    'Level1'
x =
    12.340000000000000
y =
    45
answer =
    'Yes'
```

Example 2 – Read as Fixed Format File, Ignoring the Floating Point Value

The first line of mydata.dat is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a fixed format file, ignoring the floating-point value.

```
[names, types, y, answer] = textread('mydata.dat', ...  
'%9c %5s %*f %2d %3s', 1)
```

returns

```
names =  
Sally  
types =  
    'Level1'  
y =  
    45  
answer =  
    'Yes'
```

`%*f` in the format string causes `textread` to ignore the floating point value, in this case, 12.34.

Example 3 – Read Using Literal to Ignore Matching Characters

The first line of mydata.dat is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file, ignoring the characters Type in the second field.

```
[names, typenum, x, y, answer] = textread('mydata.dat', ...  
'%s Type%d %f %d %s', 1)
```

returns

```
names =
    'Sally'
typenum =
    1
x =
    12.340000000000000
y =
    45
answer =
    'Yes'
```

Type%d in the format string causes the characters Type in the second field to be ignored, while the rest of the second field is read as a signed integer, in this case, 1.

Example 4 – Specify Value to Fill Empty Cells

For files with empty cells, use the emptyvalue parameter. Suppose the file data.csv contains:

```
1,2,3,4,,6
7,8,9,,11,12
```

Read the file using NaN to fill any empty cells:

```
data = textread('data.csv', '', 'delimiter', ',', ...
    'emptyvalue', NaN);
```

Example 5 – Read M-File into a Cell Array of Strings

Read the file fft.m into cell array of strings.

```
file = textread('fft.m', '%s', 'delimiter', '\n', 'whitespace', '');
```

See Also

textscan, dlmread, csvread, strread, fscanf

Purpose Read formatted data from text file or string

Syntax

```
C = textscan(fid, 'format')
C = textscan(fid, 'format', N)
C = textscan(fid, 'format', param, value, ...)
C = textscan(fid, 'format', N, param, value, ...)
C = textscan(str, ...)
[C, position] = textscan(...)
```

Description Before reading a file with `textscan`, you must open the file with the `fopen` function. `fopen` supplies the `fid` input required by `textscan`. When you are finished reading from the file, you should close the file by calling `fclose(fid)`.

`C = textscan(fid, 'format')` reads data from an open text file identified by file identifier `fid` into cell array `C`. MATLAB parses the data into fields and converts it according to the conversion specifiers in `format`. The `format` input is a string enclosed in single quotes. These conversion specifiers determine the type of each cell in the output cell array. The number of specifiers determines the number of cells in the cell array.

`C = textscan(fid, 'format', N)` reads data from the file, reusing the `format` conversion specifier `N` times, where `N` is a positive integer. You can resume reading from the file after `N` cycles by calling `textscan` again using the original `fid`.

`C = textscan(fid, 'format', param, value, ...)` reads data from the file using nondefault parameter settings specified by one or more pairs of `param` and `value` arguments. The section “User Configurable Options” on page 2-3113 lists all valid parameter strings, value descriptions, and defaults.

`C = textscan(fid, 'format', N, param, value, ...)` reads data from the file, reusing the `format` conversion specifier `N` times, and using nondefault parameter settings specified by pairs of `param` and `value` arguments.

`C = textscan(str, ...)` reads data from string `str` in exactly the same way as it does when reading from a file. You can use the `format`, `N`, and `parameter/value` arguments described above with this syntax. Unlike when reading from a file, if you call `textscan` more than once on the same string, it does not resume reading where the last call left off but instead reads from the beginning of the string each time.

`[C, position] = textscan(...)` returns the location of the file or string position as the second output argument. For a file, this is exactly equivalent to calling `ftell(fid)` after making the call to `TEXTSCAN`. For a string, it indicates how many characters were read.

The Difference Between the `textscan` and `textread` Functions

The `textscan` function differs from `textread` in the following ways:

- The `textscan` function offers better performance than `textread`, making it a better choice when reading large files.
- With `textscan`, you can start reading at any point in the file. Once the file is open, (`textscan` requires that you open the file first), you can `fseek` to any position in the file and begin the `textscan` at that point. The `textread` function requires that you start reading from the beginning of the file.
- Subsequent `textscans` start reading the file at the point where the last `textscan` left off. The `textread` function always begins at the start of the file, regardless of any prior `textread`.
- `textscan` returns a single cell array regardless of how many fields you read. With `textscan`, you don't need to match the number of output arguments to the number of fields being read as you would with `textread`.
- `textscan` offers more choices in how the data being read is converted.
- `textscan` offers more user-configurable options.

Field Delimiters

The `textscan` function regards a text file as consisting of blocks. Each block consists of a number of internally consistent fields. Each field consists of a group of characters delimited by a field delimiter character. Fields can span a number of rows. Each row is delimited by an end-of-line (EOL) character sequence.

The default field delimiter is the white-space character, (i.e., any character that returns `true` from a call to the `isspace` function). You can set the delimiter to a different character by specifying a `'delimiter'` parameter in the `textscan` command (see “User Configurable Options” on page 2-3113). If a nondefault delimiter is specified, repeated delimiter characters are treated as separate delimiters. When using the default delimiter, repeated white-space characters are treated as a single delimiter.

The default end-of-line character sequence depends on which operating system you are using. You can set end-of-line to a different character sequence by specifying an `'endofline'` parameter in the `textscan` command (see “User Configurable Options” on page 2-3113).

Conversion Specifiers

This table shows the conversion type specifiers supported by `textscan`.

Specifier	Description
<code>%n</code>	Read a number and convert to double.
<code>%d</code>	Read a number and convert to <code>int32</code> .
<code>%d8</code>	Read a number and convert to <code>int8</code> .
<code>%d16</code>	Read a number and convert to <code>int16</code> .
<code>%d32</code>	Read a number and convert to <code>int32</code> .
<code>%d64</code>	Read a number and convert to <code>int64</code> .
<code>%u</code>	Read a number and convert to <code>uint32</code> .
<code>%u8</code>	Read a number and convert to <code>uint8</code> .

Specifier	Description
%u16	Read a number and convert to uint16.
%u32	Read a number and convert to uint32.
%u64	Read a number and convert to uint64.
%f	Read a number and convert to double.
%f32	Read a number and convert to single.
%f64	Read a number and convert to double.
%s	Read a string.
%q	Read a (possibly double-quoted) string.
%c	Read one character, including white space.
%[...]	Read characters that match characters between the brackets. Stop reading at the first nonmatching character. Use %[...] to include] in the set.
%[^...]	Read characters that do not match characters between the brackets. Stop reading at the first matching character. Use %[^...] to exclude] from the set.

Specifying Field Length

To read a certain number of characters or digits from a field, specify that number directly following the percent sign. For example, if the file you are reading contains the string

```
'Blackbird singing in the dead of night'
```

then the following command returns only five characters of the first field:

```
C = textscan(fid, '%5s', 1);
C{:}
ans =
    'Black'
```

If you continue reading from the file, `textscan` resumes the operation at the point in the string where you left off. It applies the next format specifier to that portion of the field. For example, execute this command on the same file:

```
C = textscan(fid, '%s %s', 1);
```

Note Spaces between the conversion specifiers are shown only to make the example easier to read. They are not required.

`textscan` reads starting from where it left off and continues to the next whitespace, returning `'bird'`. The second `%s` reads the word `'singing'`.

The results are

```
C{:}
ans =
    'bird'
ans =
    'singing'
```

Skipping Fields

To skip any field, put an asterisk directly after the percent sign. MATLAB does not create an output cell for any fields that are skipped.

Refer to the example from the last section, where the file you are reading contains the string

```
'Blackbird singing in the dead of night'
```

Seek to the beginning of the file and then reread the line, this time skipping the second, fifth, and sixth fields:

```
fseek(fid, 0, -1);
C = textscan(fid, '%s %*s %s %s %*s %*s %s', 1);
```

C is a cell array of cell arrays, each containing a string. Piece together the string and display it:

```
str = '';
for k = 1:length(C)
    str = [str char(C{k}) ' '];
    if k == 4, disp(str), end
end
```

Blackbird in the night

Skipping Literal Strings

In addition to skipping entire fields, you can have textscan skip leading literal characters in a string. Reading a file containing the following data,

```
Sally    Level1  12.34
Joe      Level2  23.54
Bill     Level3  34.90
```

this command removes the substring 'Level' from the output and converts the level number to a uint8:

```
C = textscan(fid, '%s Level%u8 %f');
```

This returns a cell array C with the second cell containing only the unsigned integers:

```
C{1} = {'Sally'; 'Joe'; 'Bill'}           class cell
C{2} = [1; 2; 3]                          class uint8
C{3} = [12.34; 23.54; 34.90]              class double
```

Specifying Numeric Field Length and Decimal Digits

With numeric fields, you can specify the number of digits to read in the same manner described for strings in the section “Specifying Field Length” on page 2-3107. The next example uses a file containing the line

```
'405.36801 551.94387 298.00752 141.90663'
```

This command returns the starting 7 digits of each number in the line. Note that the decimal point counts as a digit.

```
C = textscan(fid, '%7f32 %*n');  
C{:} =  
    [405.368; 551.943; 298.007; 141.906]
```

You can also control the number of digits that are read to the right of the decimal point for any numeric field of type %f, %f32, or %f64. The format specifier in this command uses a %9.1 prefix to cause textscan to read the first 9 digits of each number, but only include 1 digit of the decimal value in the number it returns:

```
C = textscan(fid, '%9.1f32 %*n');  
C{:} =  
    [405.3; 551.9; 298.0; 141.9]
```

Conversion of Numeric Fields

This table shows how textscan interprets the numeric field specifiers.

Format Specifier	Action Taken
%n, %d, %u, %f, and variants thereof	Read to the first delimiter. Example: %n reads '473.238 ' as 473.238.
%Nn, %Nd, %Nu, %Nf, and variants thereof	Read N digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first. Example: %5f32 reads '473.238 ' as 473.2.
Specifiers that start with %N.Df	Read N digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first. Return D decimal digits in the output. Example: %7.2f reads '473.238 ' as 473.23.

Conversion specifiers %n, %d, %u, %f, or any variant thereof (e.g., %d16) return a K-by-1 MATLAB numeric vector of the type indicated by the

conversion specifier, where *K* is the number of times that specifier was found in the file. `textscan` converts the numeric fields from the field content to the output type according to the conversion specifier and MATLAB rules regarding overflow and truncation. `NaN`, `Inf`, and `-Inf` are converted according to applicable MATLAB rules.

`textscan` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type. Valid forms for a complex number are

Form	Example
<code>–<real>–<imag>i j</code>	<code>5.7-3.1i</code>
<code>–<imag>i j</code>	<code>-7j</code>

Embedded white-space in a complex number is invalid and is regarded as a field delimiter.

Conversion of Strings

This table shows how `textscan` interprets the string field specifiers.

Format Specifier	Action Taken
<code>%s</code> or <code>%q</code>	Read to the first delimiter. Example: <code>%s</code> reads 'summer' as 'summer'.
<code>%Ns</code> or <code>%Nq</code>	Read <i>N</i> characters, or to the first delimiter, whichever comes first. Example: <code>%3s</code> reads 'summer' as 'sum'.
<code>%[abc]</code>	Read up to the first character not specified within the brackets (i.e., read up to the first character that is not an <i>a</i> , <i>b</i> , or <i>c</i>). Example: <code>%[mus]</code> reads 'summer' as 'summ'.

Format Specifier	Action Taken
<code>%N[abc]</code>	Read N characters, or up to the first character not specified within the brackets, whichever comes first. Example: <code>%2[mus]</code> reads 'summer' as 'su'.
<code>%[^abc]</code>	Read up to the first character that is specified within the brackets, (i.e., read up to the first occurrence of an a, b, or c). Example: <code>%[^xrg]</code> reads 'summer ' as 'summe'.
<code>%N[^abc]</code>	Read N characters, or up to the first character that is specified within the brackets, whichever comes first. Example: <code>%2[^xrg]</code> reads 'summer ' as 'su'.

Conversion specifiers `%s`, `%q`, `%[...]`, and `%[^...]` return a K-by-1 MATLAB cell vector of strings, where K is the number of times that specifier was found in the file. If you set the delimiter parameter to a non-white-space character, or set the whitespace parameter to `' '`, `textscan` returns all characters in the string field, including white-space. Otherwise each string terminates at the beginning of white-space.

Conversion of Characters

This table shows how `textscan` interprets the character field specifiers.

Format Specifier	Action Taken
<code>%c</code>	Read one character. Example: <code>%c</code> reads 'Let's go!' as 'L'.
<code>%Nc</code>	Read N characters, including delimiter characters. Example: <code>%9c</code> reads 'Let's go!' as 'Let's go!'.

Conversion specifier %Nc returns a K-by-N MATLAB character array, where K is the number of times that specifier was found in the file. textscan returns all characters, including white-space but excluding the delimiter.

Conversion of Empty Fields

An empty field in the text file is defined by two adjacent delimiters indicating an empty set of characters, or, in all cases except %c, white-space. The empty field is returned as NaN by default, but is user definable. In addition, you may specify custom strings to be used as empty values, in *numeric fields only*. textscan does not examine nonnumeric fields for custom empty values. See “User Configurable Options” on page 2-3113.

Note MATLAB represents integer NaN as zero. If textscan reads an empty field that is assigned an integer format specifier (one that starts with %d or %u), it returns the empty value as zero rather than as NaN. (See the value returned in C{5} in Example 6 — Using a Nondefault Empty Value.)

User Configurable Options

This table shows the valid param-value options and their default values.

Parameter	Value	Default
bufSize	Maximum string length in bytes	4095
commentStyle	Symbol(s) designating text to be ignored (see “Values for commentStyle” on page 2-3114, below)	None
delimiter	Delimiter characters	None

Parameter	Value	Default
emptyValue	Empty cell value in delimited files	NaN
endOfLine	End-of-line character	Determined from the file
expChars	Exponent characters	'eEdD'
headerLines	Number of lines at beginning of file to skip	0
multipleDelimsAsOne	If set to 1, textread treats consecutive delimiters as a single delimiter. If set to 0, textread treats them as separate delimiters. Only valid if the delimiter option is specified.	0
returnOnError	Behavior on failing to read or convert (1=true or 0)	1
treatAsEmpty	String(s) to be treated as an empty value. A single string or cell array of strings can be used.	None
whitespace	White-space characters	' \b\t'

White-Space Characters

Leading white-space characters are not included in the processing of any of the data fields. When processing numeric data, trailing whitespace is also assumed to have no significance.

Values for commentStyle

Possible values for the commentStyle parameter are

Value	Description	Example
Single string, S	Ignore any characters that follow string S and are on the same line.	'%', '//'
Cell array of two strings, C	Ignore any characters that lie between the opening and closing strings in C.	{'/*', '*/'}, {'/%', '%/'}

Resuming a Text Scan

If `textscan` fails to convert a data field, it stops reading and returns all fields read before the failure. When reading from a file, you can resume reading from the same file by calling `textscan` again using the same file identifier, `fid`. When reading from a string, the two-output argument syntax enables you to resume reading from the string at the point where the last read terminated. The following command is an example of how you can do this:

```
textscan(str(position+1:end), ...)
```

Remarks

For information on how to use `textscan` to import large data sets, see “Reading Files with Large Data Sets” in the MATLAB Programming documentation.

Examples

Example 1 – Reading Different Types of Data

Text file `scan1.dat` contains data in the following form:

```
Sally Level1 12.34 45 1.23e10 inf NaN Yes
Joe Level2 23.54 60 9e19 -inf 0.001 No
Bill Level3 34.90 12 2e5 10 100 No
```

Read each column into a variable:

```
fid = fopen('scan1.dat');
C = textscan(fid, '%s %s %f32 %d8 %u %f %f %s');
```

```
fclose(fid);
```

Note Spaces between the conversion specifiers are shown only to make the example easier to read. They are not required.

textscan returns a 1-by-8 cell array C with the following cells:

```
C{1} = {'Sally'; 'Joe'; 'Bill'}           class cell
C{2} = {'Level1'; 'Level2'; 'Level3'}    class cell
C{3} = [12.34; 23.54; 34.9]              class single
C{4} = [45; 60; 12]                      class int8
C{5} = [4294967295; 4294967295; 200000]  class uint32
C{6} = [Inf; -Inf; 10]                   class double
C{7} = [NaN; 0.001; 100]                 class double
C{8} = {'Yes'; 'No'; 'No'}               class cell
```

The first two elements of C{5} are the maximum values for a 32-bit unsigned integer, or `intmax('uint32')`.

Example 2 – Reading All But One Field

Read the file as a fixed-format file, skipping the third field:

```
fid = fopen('scan1.dat');
C = textscan(fid, '%7c %6s %*f %d8 %u %f %f %s');
fclose(fid);
```

textscan returns a 1-by-8 cell array C with the following cells:

```
C{1} = ['Sally  '; 'Joe   '; 'Bill  ']   class char
C{2} = {'Level1'; 'Level2'; 'Level3'}    class cell
C{3} = [45; 60; 12]                      class int8
C{4} = [4294967295; 4294967295; 200000]  class uint32
C{5} = [Inf; -Inf; 10]                   class double
C{6} = [NaN; 0.001; 100]                 class double
C{7} = {'Yes'; 'No'; 'No'}               class cell
```

Example 3 – Reading Only the First Field

Read the first column into a cell array, skipping the rest of the line:

```
fid = fopen('scan1.dat');
names = textscan(fid, '%s*[\n]');
fclose(fid);
```

textscan returns a 1-by-1 cell array names:

```
size(names)
ans =
     1     1
```

The one cell contains

```
names{1} = {'Sally'; 'Joe'; 'Bill'}      class cell
```

Example 4 – Removing a Literal String in the Output

The second format specifier in this example, %sLevel, tells textscan to read the second field from a line in the file, but to ignore the initial string 'Level' within that field. All that is left of the field is a numeric digit. textscan assigns the next specifier, %f, to that digit, converting it to a double.

See C{2} in the results:

```
fid = fopen('scan1.dat');
C = textscan(fid, '%s Level%u8 %f32 %d8 %u %f %f %s');fclose(fid);
```

textscan returns a 1-by-8 cell array, C, with cells

```
C{1} = {'Sally'; 'Joe'; 'Bill'}      class cell
C{2} = [1; 2; 3]                     class uint8
C{3} = [12.34; 23.54; 34.90]         class single
C{4} = [45; 60; 12]                  class int8
C{5} = [4294967295; 4294967295; 200000] class uint32
C{6} = [Inf; -Inf; 10]                class double
C{7} = [NaN; 0.001; 100]              class double
```

```
C{8} = {'Yes'; 'No'; 'No'}          class cell
```

Example 5 – Using a Nondefault Delimiter and White-Space

Read the M-file into a cell array of strings:

```
fid = fopen('fft.m');  
file = textscan(fid, '%s', 'delimiter', '\n', 'whitespace', '');  
fclose(fid);
```

textscan returns a 1-by-1 cell array, file, that contains a 37-by-1 cell array:

```
file =  
    {37x1 cell}
```

Show the first three lines of the file:

```
lines = file{1};  
lines{1:3, :}  
ans =  
    'function [varargout] = fft(varargin)'  
ans =  
    '%FFT Discrete Fourier transform.'  
ans =  
    '%   FFT(X) is the discrete Fourier transform (DFT) of vector  
X. For'
```

Example 6 – Using a Nondefault Empty Value

Read files with empty cells, setting the emptyvalue parameter. The file data.csv contains

```
1, 2, 3, 4, , 6  
7, 8, 9, , 11, 12
```

Read the file as shown here, using -Inf in empty cells:

```
fid = fopen('data.csv');  
C = textscan(fid, '%f%f%f%f%u32%f', 'delimiter', ',', '...', 'emptyvalue', -Inf);
```

```
        'emptyValue', -Inf);
fclose(fid);
```

textscan returns a 1-by-6 cell array C with the following cells:

```
C{1} = [1; 7]           class double
C{2} = [2; 8]           class double
C{3} = [3; 9]           class double
C{4} = [4; NaN]         class double
C{5} = [-Inf; 11]       class uint32 (-Inf converted to 0)
C{6} = [6; 12]         class double
```

Example 7 – Using Custom Empty Values and Comments

You have a file data.csv that contains the lines

```
abc, 2, NA, 3, 4
// Comment Here
def, na, 5, 6, 7
```

Designate what should be treated as empty values and as comments. Read in all other values from the file:

```
fid = fopen('data5.csv');
C = textscan(fid, '%s%n%n%n%n', 'delimiter', ',', ...
            'treatAsEmpty', {'NA', 'na'}, ...
            'commentStyle', '//');
fclose(fid);
```

This returns the following data in cell array C:

```
C{:}
ans =
    'abc'
    'def'
ans =
     2
    NaN
ans =
```

```
NaN
    5
ans =
    3
    6
ans =
    4
    7
```

Example 8 – Reading From a String

Read in a string (quoted from Albert Einstein) using `textscan`:

```
str = ...
    ['Do not worry about your difficulties in Mathematics. ' ...
    'I can assure you mine are still greater.'];

s = textscan(str, '%s', 'delimiter', '.');

s{:}
ans =
    'Do not worry about your difficulties in Mathematics'
    'I can assure you mine are still greater'
```

Example 9 – Handling Multiple Delimiters

This example takes a comma-separated list of names, the test pilots known as the Mercury Seven, and uses `textscan` to return a list of their names in a cell array. When some names are removed from the input list, leaving multiple sequential delimiters, `textscan`, by default, accounts for this. If you override that default by calling `textscan` with the `multipleDelimsAsOne` option, `textscan` ignores the missing names.

Here is the full list of the astronauts:

```
Mercury7 = ...
    'Shepard,Grissom,Glenn,Carpenter,Schirra,Cooper,Slayton';
```


Remove the names Grissom and Cooper from the input string, and `textscan`, by default, does not treat the multiple delimiters as one, and returns an empty string for each missing name:

```
Mercury7 = 'Shepard,,Glenn,Carpenter,Schirra,,Slayton';
names = textscan(Mercury7, '%s', 'delimiter', ',');
names{:}'
ans =
    'Shepard'    ''    'Glenn'    'Carpenter'    'Schirra'    ''    'Slayton'
```

Using the same input string, but this time setting the `multipleDelimsAsOne` switch, `textscan` ignores the multiple delimiters:

```
names = textscan(Mercury7, '%s', 'delimiter', ',', ...
                'multipleDelimsAsOne', 1);
names{:}'
ans =
    'Shepard'    'Glenn'    'Carpenter'    'Schirra'    'Slayton'
```

See Also

`dlmread`, `dlmwrite`, `xlswrite`, `fopen`, `fseek`, `importdata`

textwrap

Purpose Wrapped string matrix for given uicontrol

Syntax `outstring = textwrap(h,instring)`
`[outstring,position]=textwrap(h,instring)`

Description `outstring = textwrap(h,instring)` returns a wrapped string cell array, `outstring`, that fits inside the uicontrol with handle `h`. `instring` is a cell array, with each cell containing a single line of text. `outstring` is the wrapped string matrix in cell array format. Each cell of the input string is considered a paragraph.

`[outstring,position]=textwrap(h,instring)` returns the recommended position of the uicontrol in the units of the uicontrol. `position` considers the extent of the multiline text in the x and y directions.

Example Place a text-wrapped string in a uicontrol:

```
pos = [10 10 100 10];
h = uicontrol('Style','Text','Position',pos);
string = {'This is a string for the uicontrol.',
         'It should be correctly wrapped inside.'};
[outstring,newpos] = textwrap(h,string);
pos(4) = newpos(4);
set(h,'String',outstring,'Position',[pos(1),pos(2),pos(3)+10,po
s(4)])
```

See Also `uicontrol`

Purpose Measure performance using stopwatch timer

Syntax

```
tic
    any statements
toc
t = toc
```

Description

tic starts a stopwatch timer.

toc prints the elapsed time since tic was used.

t = toc returns the elapsed time in t.

Remarks

The tic and toc functions work together to measure elapsed time. tic saves the current time that toc uses later to measure the elapsed time. The sequence of commands

```
tic
operations
toc
```

measures the amount of time MATLAB takes to complete one or more operations, and displays the time in seconds.

Examples

This example measures how the time required to solve a linear system varies with the order of a matrix.

```
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic
    x = A\b;
    t(n) = toc;
end
plot(t)
```

See Also clock, cputime, etime, profile

timer

Purpose Construct timer object

Syntax
T = timer
T = timer('PropertyName1', PropertyValue1, 'PropertyName2',
 PropertyValue2,...)

Description T = timer constructs a timer object with default attributes.
T = timer('PropertyName1', PropertyValue1, 'PropertyName2',
 PropertyValue2,...) constructs a timer object in which the given
property name/value pairs are set on the object. See “Timer Object
Properties” on page 2-3124 for a list of all the properties supported
by the timer object.

Note that the property name/property value pairs can be in any
format supported by the set function, i.e., property/value string pairs,
structures, and property/value cell array pairs.

Examples This example constructs a timer object with a timer callback function
handle, mycallback, and a 10 second interval.

```
t = timer('TimerFcn',@mycallback, 'Period', 10.0);
```

See Also delete, disp, get, isvalid, set, start, startat, stop, timerfind,
timerfindall, wait

Timer Object Properties The timer object supports the following properties that control its
attributes. The table includes information about the data type of each
property and its default value.

To view the value of the properties of a particular timer object, use the
get function. To set the value of the properties of a timer object, use
the set function.

Property Name	Property Description	Data Types, Values, and Defaults	
AveragePeriod	<p>The average time between TimerFcn executions since the timer started.</p> <p>Note: Value is NaN until timer executes two timer callbacks.</p>	Data type: Default: Read only:	double NaN Always
BusyMode	<p>Action taken when a timer has to execute TimerFcn before the completion of previous execution of TimerFcn.</p> <ul style="list-style-type: none"> • 'drop' — Do not execute the function. • 'error' — Generate an error. • 'queue' — Execute function at next opportunity. 	Data type: Values: Default: Read only:	Enumerated string 'drop' 'error' 'queue' 'drop' While Running = 'on'
ErrorFcn	<p>Function that the timer executes when an error occurs. This function executes before the StopFcn. See “Creating Callback Functions” for more information.</p>	Data type: Default: Read only:	Text string, function handle, or cell array None Never

timer

Property Name	Property Description	Data Types, Values, and Defaults	
ExecutionMode	Determines how the timer object schedules timer events. See “Timer Object Execution Modes” for more information.	Data type: Values: Default: Read only:	Enumerated string 'singleShot' 'fixedDelay' 'fixedRate' 'fixedSpacing' 'singleShot' While Running = 'on'
InstantPeriod	The time between the last two executions of TimerFcn.	Data type: Default: Read only:	double NaN Always
Name	User-supplied name	Data type: Default: Read only:	Text string 'timer- <i>i</i> ', where <i>i</i> is a number indicating the <i>i</i> th timer object created this session. To reset <i>i</i> to 1, execute the clear classes command. Never

Property Name	Property Description	Data Types, Values, and Defaults	
ObjectVisibility	Provides a way for application developers to prevent end-user access to the timer objects created by their application. The <code>timerfind</code> function does not return an object whose <code>ObjectVisibility</code> property is set to 'off'. Objects that are not visible are still valid. If you have access to the object (for example, from within the M-file that created it), you can set its properties.	Data type: Values: Default: Read only:	Enumerated string 'off' 'on' 'on' Never
Period	Specifies the delay, in seconds, between executions of <code>TimerFcn</code> .	Data type: Value: Default: Read only:	double Any number > 0.001 1.0 While Running = 'on'
Running	Indicates whether the timer is currently executing.	Data type: Values: Default: Read only:	Enumerated string 'off' 'on' 'off' Always
StartDelay	Specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in <code>TimerFcn</code> .	Data type: Value: Default: Read only:	double Any number >= 0 0 While Running = 'on'

timer

Property Name	Property Description	Data Types, Values, and Defaults	
StartFcn	Function the timer calls when it starts. See “Creating Callback Functions” for more information.	Data type: Default: Read only:	Text string, function handle, or cell array None Never
StopFcn	Function the timer calls when it stops. The timer stops when <ul style="list-style-type: none">• You call the timer stop function• The timer finishes executing TimerFcn, i.e., the value of TasksExecuted reaches the limit set by TasksToExecute.• An error occurs (The ErrorFcn is called first, followed by the StopFcn.) See “Creating Callback Functions” for more information.	Data type: Default: Read only:	Text string, function handle, or cell array None Never
Tag	User supplied label	Data type: Default: Read only:	Text string ' ' (empty string) Never

Property Name	Property Description	Data Types, Values, and Defaults	
TasksToExecute	Specifies the number of times the timer should execute the function specified in the TimerFcn property.	Data type: Value: Default: Read only:	double Any number > 0 1 Never
TasksExecuted	The number of times the timer has executed TimerFcn since the timer was started.	Data type: Value: Default: Read only:	double Any number >= 0 0 Always
TimerFcn	Timer callback function. See “Creating Callback Functions” for more information.	Data type: Default: Read only:	Text string, function handle, or cell array None Never
Type	Identifies the object type	Data type: Value: Read only:	Text string 'timer' Always
UserData	User-supplied data	Data type: Default: Read only:	User-defined [] Never

timerfind

Purpose Find timer objects

Syntax

```
out = timerfind
out = timerfind('P1', V1, 'P2', V2,...)
out = timerfind(S)
out = timerfind(obj, 'P1', V1, 'P2', V2,...)
```

Description out = timerfind returns an array, out, of all the timer objects that exist in memory.

out = timerfind('P1', V1, 'P2', V2,...) returns an array, out, of timer objects whose property values match those passed as parameter/value pairs, P1, V1, P2, V2. Parameter/value pairs may be specified as a cell array.

out = timerfind(S) returns an array, out, of timer objects whose property values match those defined in the structure, S. The field names of S are timer object property names and the field values are the corresponding property values.

out = timerfind(obj, 'P1', V1, 'P2', V2,...) restricts the search for matching parameter/value pairs to the timer objects listed in obj. obj can be an array of timer objects.

Note When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to timerfind.

Note that, for most properties, timerfind performs case-sensitive searches of property values. For example, if the value of an object's Name property is 'MyObject', timerfind will not find a match if you specify 'myobject'. Use the get function to determine the exact format of a property value. However, properties that have an enumerated list of possible values are not case sensitive. For example, timerfind will find an object with an ExecutionMode property value of 'singleShot' or 'singleshot'.

Examples

These examples use `timerfind` to find timer objects with the specified property values.

```
t1 = timer('Tag', 'broadcastProgress', 'Period', 5);  
t2 = timer('Tag', 'displayProgress');  
out1 = timerfind('Tag', 'displayProgress')  
out2 = timerfind({'Period', 'Tag'}, {5, 'broadcastProgress'})
```

See Also

`get`, `timer`, `timerfindall`

timerfindall

Purpose Find timer objects, including invisible objects

Syntax

```
out = timerfindall
out = timerfindall('P1', V1, 'P2', V2,...)
out = timerfindall(S)
out = timerfindall(obj, 'P1', V1, 'P2', V2,...)
```

Description `out = timerfindall` returns an array, `out`, containing all the timer objects that exist in memory, regardless of the value of the object's `ObjectVisibility` property.

`out = timerfindall('P1', V1, 'P2', V2,...)` returns an array, `out`, of timer objects whose property values match those passed as parameter/value pairs, `P1`, `V1`, `P2`, `V2`. Parameter/value pairs may be specified as a cell array.

`out = timerfindall(S)` returns an array, `out`, of timer objects whose property values match those defined in the structure, `S`. The field names of `S` are timer object property names and the field values are the corresponding property values.

`out = timerfindall(obj, 'P1', V1, 'P2', V2,...)` restricts the search for matching parameter/value pairs to the timer objects listed in `obj`. `obj` can be an array of timer objects.

Note When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to `timerfindall`.

Note that, for most properties, `timerfindall` performs case-sensitive searches of property values. For example, if the value of an object's `Name` property is `'MyObject'`, `timerfindall` will not find a match if you specify `'myobject'`. Use the `get` function to determine the exact format of a property value. However, properties that have an enumerated list of possible values are not case sensitive. For example, `timerfindall` will find an object with an `ExecutionMode` property value of `'singleShot'` or `'singleShot'`.

Examples

Create several timer objects.

```
t1 = timer;
t2 = timer;
t3 = timer;
```

Set the `ObjectVisibility` property of one of the objects to 'off'.

```
t2.ObjectVisibility = 'off';
```

Use `timerfind` to get a listing of all the timer objects in memory. Note that the listing does not include the timer object (timer-2) whose `ObjectVisibility` property is set to 'off'.

```
timerfind
```

Timer Object Array

Index:	ExecutionMode:	Period:	TimerFcn:	Name:
1	singleShot	1	''	timer-1
2	singleShot	1	''	timer-3

Use `timerfindall` to get a listing of all the timer objects in memory. This listing includes the timer object whose `ObjectVisibility` property is set to 'off'.

```
timerfindall
```

Timer Object Array

Index:	ExecutionMode:	Period:	TimerFcn:	Name:
1	singleShot	1	''	timer-1
2	singleShot	1	''	timer-2
3	singleShot	1	''	timer-3

See Also

`get`, `timer`, `timerfind`

timeseries

Purpose Create timeseries object

Syntax

```
ts = timeseries
ts = timeseries(Data)
ts = timeseries(Name)
ts = timeseries(Data,Time)
ts = timeseries(Data,Time,Quality)
ts = timeseries(Data,...,'Parameter',Value,...)
```

Description `ts = timeseries` creates an empty time-series object.

`ts = timeseries(Data)` creates a time series with the specified `Data`. `ts` has a default time vector that ranges from 0 to `N-1` with a 1-second interval, where `N` is the number of samples. The default name of the timeseries object is 'unnamed'.

`ts = timeseries(Name)` creates an empty time series with the name specified by a string `Name`. This name can differ from the time-series variable name.

`ts = timeseries(Data,Time)` creates a time series with the specified `Data` array and `Time`. When time values are date strings, you must specify `Time` as a cell array of date strings.

`ts = timeseries(Data,Time,Quality)` creates a timeseries object. The `Quality` attribute is an integer vector with values -128 to 127 that specifies the quality in terms of codes defined by `QualityInfo.Code`.

`ts = timeseries(Data,...,'Parameter',Value,...)` creates a timeseries object with optional parameter-value pairs after the `Data`, `Time`, and `Quality` arguments. You can specify the following parameters:

- `Name` — Time-series name entered as a string
- `IsTimeFirst` — Logical value (`true` or `false`) specifying whether the first or last dimension of the data array is aligned with the time vector. You can set this property when the data array is square and, therefore, the dimension that is aligned with time is ambiguous.

- `IsDatetime` — Logical value (true or false) that when set to true specifies that Time values are dates in the format of MATLAB serial dates.

Remarks

Definition: timeseries

The time-series object, called `timeseries`, is a MATLAB variable that contains time-indexed data and properties in a single, coherent structure. For example, in addition to data and time values, you can also use the time-series object to store events, descriptive information about data and time, data quality, and the interpolation method.

Definition: Data Sample

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

For example, suppose that `ts.data` has the size 5-by-4-by-3 and the time vector has the length 5. Then, the number of sample is 5 and the total number of data values is $5 \times 4 \times 3 = 60$.

Notes About Quality

When `Quality` is a vector, it must have the same length as the time vector. In this case, each `Quality` value applies to the corresponding data sample. When `Quality` is an array, it must have the same size as the data array. In this case, each `Quality` value applies to the corresponding data value of the `ts.data` array.

Examples

Example 1 – Using Default Time Vector

Create a `timeseries` object called `'LaunchData'` that contains four data sets, each stored as a column of length 5 and using the default time vector:

```
b = timeseries(rand(5, 4), 'Name', 'LaunchData')
```

Example 2 – Using Uniform Time Vector

Create a `timeseries` object containing a single data set of length 5 and a time vector starting at 1 and ending at 5:

```
b = timeseries(rand(5,1),[1 2 3 4 5])
```

Example 3

Create a `timeseries` object called 'FinancialData' containing five data points at a single time point:

```
b = timeseries(rand(1,5),1,'Name','FinancialData')
```


See Also

`addsample`, `tscollection`, `tsdata.event`, `tsprops`

Purpose

Add title to current axes

**GUI
Alternative**

To create or modify a plot's title from a GUI, use **Insert** → **Title** from the figure menu. Use the Property Editor, one of the plotting tools , to modify the position, font and other properties of a legend. For details, see The Property Editor in the MATLAB Graphics documentation.

Syntax

```
title('string')
title(fname)
title(..., 'PropertyName', PropertyValue, ...)
title(axes_handle, ...)
h = title(...)
```

Description

Each axes graphics object can have one title. The title is located at the top and in the center of the axes.

`title('string')` outputs the string at the top and in the center of the current axes.

`title(fname)` evaluates the function that returns a string and displays the string at the top and in the center of the current axes.

`title(..., 'PropertyName', PropertyValue, ...)` specifies property name and property value pairs for the text graphics object that `title` creates. Do not use the 'String' text property to set the title string; the content of the title should be given by the first argument.

`title(axes_handle, ...)` adds the title to the specified axes.

`h = title(...)` returns the handle to the text object used as the title.

Examples

Display today's date in the current axes:

```
title(date)
```

Include a variable's value in a title:

```
f = 70;
c = (f-32)/1.8;
```

title

```
title(['Temperature is ', num2str(c), 'C'])
```

Include a variable's value in a title and set the color of the title to yellow:

```
n = 3;
title(['Case number #', int2str(n)], 'Color', 'y')
```

Include Greek symbols in a title:

```
title('\ite^{\omega\tau} = cos(\omega\tau) + isin(\omega\tau)')
```

Include a superscript character in a title:

```
title('\alpha^2')
```

Include a subscript character in a title:

```
title('X_1')
```

The text object `String` property lists the available symbols.

Create a multiline title using a multiline cell array.

```
title({'First line'; 'Second line'})
```

Remarks

`title` sets the `Title` property of the current axes graphics object to a new text graphics object. See the text `String` property for more information.

See Also

`gtext`, `int2str`, `num2str`, `text`, `xlabel`, `ylabel`, `zlabel`

“Annotating Plots” on page 1-83 for related functions

Adding Titles to Graphs for more information on ways to add titles

Purpose Convert CDF epoch object to MATLAB datenum

Syntax `n = todatenum(obj)`

Description `n = todatenum(obj)` converts the CDF epoch object `ep_obj` into a MATLAB serial date number. Note that a CDF epoch is the number of milliseconds since 01-Jan-0000 whereas a MATLAB datenum is the number of days since 00-Jan-0000.

Examples Construct a CDF epoch object from a date string, and then convert the object back into a MATLAB date string:

```
dstr = datestr(today)
dstr =
    08-Oct-2003

obj = cdfepoch(dstr)
obj =
    cdfepoch object:
    08-Oct-2003 00:00:00

dstr2 = datestr(todatenum(obj))
dstr2 =
    08-Oct-2003
```

See Also `cdfepoch`, `cdfinfo`, `cdfread`, `cdfwrite`, `datenum`

toeplitz

Purpose Toeplitz matrix

Syntax `T = toeplitz(c,r)`
`T = toeplitz(r)`

Description A *Toeplitz* matrix is defined by one row and one column. A *symmetric Toeplitz* matrix is defined by just one row. `toeplitz` generates Toeplitz matrices given just the row or row and column description.

`T = toeplitz(c,r)` returns a nonsymmetric Toeplitz matrix `T` having `c` as its first column and `r` as its first row. If the first elements of `c` and `r` are different, a message is printed and the column element is used.

`T = toeplitz(r)` returns the symmetric or Hermitian Toeplitz matrix formed from vector `r`, where `r` defines the first row of the matrix.

Examples A Toeplitz matrix with diagonal disagreement is

```
c = [1 2 3 4 5];
r = [1.5 2.5 3.5 4.5 5.5];
toeplitz(c,r)
Column wins diagonal conflict:
ans =
    1.000    2.500    3.500    4.500    5.500
    2.000    1.000    2.500    3.500    4.500
    3.000    2.000    1.000    2.500    3.500
    4.000    3.000    2.000    1.000    2.500
    5.000    4.000    3.000    2.000    1.000
```

See Also `hankel`, `kron`

Purpose	Root directory for specified toolbox
Syntax	<pre>toolboxdir('tbxdirname') s = toolboxdir('tbxdirname') s = toolboxdir tbxdirname</pre>
Description	<p><code>toolboxdir('tbxdirname')</code> returns a string that is the absolute path to the specified toolbox, <code>tbxdirname</code>, where <code>tbxdirname</code> is the directory name for the toolbox.</p> <p><code>s = toolboxdir('tbxdirname')</code> returns the absolute path to the specified toolbox to the output argument, <code>s</code>.</p> <p><code>s = toolboxdir tbxdirname</code> is the command form of the syntax.</p>
Remarks	<p><code>toolboxdir</code> is particularly useful for the MATLAB Compiler. The base directory of all toolboxes installed with MATLAB is</p> <pre>matlabroot/toolbox/tbxdirname</pre> <p>However, in deployed mode, the base directories of the toolboxes are different. <code>toolboxdir</code> returns the correct root directory, whether running from MATLAB or an application deployed with the MATLAB Compiler.</p>
Example	<p>To obtain the pathname for the Control System Toolbox, run</p> <pre>s = toolboxdir('control')</pre> <p>MATLAB returns</p> <pre>s = \\myhome\r2006b\matlab\toolbox\control</pre>
See Also	<pre>matlabroot</pre> <p><code>ctfroot</code> in the MATLAB Compiler</p>

trace

Purpose	Sum of diagonal elements
Syntax	<code>b = trace(A)</code>
Description	<code>b = trace(A)</code> is the sum of the diagonal elements of the matrix A.
Algorithm	trace is a single-statement M-file. <code>t = sum(diag(A));</code>
See Also	det, eig

Purpose Transpose timeseries object

Syntax `ts1 = transpose(ts)`

Description `ts1 = transpose(ts)` returns a new timeseries object `ts1` with `IsTimeFirst` value set to the opposite of what it is for `ts`. For example, if `ts` has the first data dimension aligned with the time vector, `ts1` has the last data dimension aligned with the time vector.

Remarks The `transpose` function that is overloaded for the timeseries objects does not transpose the data. Instead, this function changes whether the first or the last dimension of the data is aligned with the time vector.

Note To transpose the data, you must transpose the `Data` property of the time series. For example, you can use the syntax `transpose(ts.Data)` or `(ts.Data)'`. `Data` must be a 2-D array.

Consider a time series with 10 samples with the property `IsTimeFirst = True`. When you transpose this time series, the data size is changed from 10-by-1 to 1-by-1-by-10. Note that the first dimension of the `Data` property is shown explicitly.

The following table summarizes how MATLAB displays the size for time-series data (up to three dimensions) before and after transposing.

Data Size Before and After Transposing

Size of Original Data	Size of Transposed Data
N-by-1	1-by-1-by-N
N-by-M	M-by-1-by-N
N-by-M-by-L	M-by-L-by-N

transpose (timeseries)

Examples

Suppose that a `timeseries` object `ts` has `ts.Data` size 10-by-3-by-2 and its time vector has a length of 10. The `IsTimeFirst` property of `ts` is set to `true`, which means that the first dimension of the data is aligned with the time vector. `transpose(ts)` modifies the `timeseries` object such that the last dimension of the data is now aligned with the time vector. This permutes the data such that the size of `ts.Data` becomes 3-by-2-by-10.

See Also

`ctranspose (timeseries)`, `tsprops`

Purpose

Trapezoidal numerical integration

Syntax

```
Z = trapz(Y)
Z = trapz(X,Y)
Z = trapz(...,dim)
```

Description

`Z = trapz(Y)` computes an approximation of the integral of `Y` via the trapezoidal method (with unit spacing). To compute the integral for spacing other than one, multiply `Z` by the spacing increment. Input `Y` can be complex.

If `Y` is a vector, `trapz(Y)` is the integral of `Y`.

If `Y` is a matrix, `trapz(Y)` is a row vector with the integral over each column.

If `Y` is a multidimensional array, `trapz(Y)` works across the first nonsingleton dimension.

`Z = trapz(X,Y)` computes the integral of `Y` with respect to `X` using trapezoidal integration. Inputs `X` and `Y` can be complex.

If `X` is a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`, `trapz(X,Y)` operates across this dimension.

`Z = trapz(...,dim)` integrates across the dimension of `Y` specified by scalar `dim`. The length of `X`, if given, must be the same as `size(Y,dim)`.

Examples**Example 1**

The exact value of $\int_0^{\pi} \sin(x) dx$ is 2.

To approximate this numerically on a uniformly spaced grid, use

```
X = 0:pi/100:pi;
Y = sin(X);
```

Then both

```
Z = trapz(X,Y)
```

and

```
Z = pi/100*trapz(Y)
```

produce

```
Z =  
    1.9998
```

Example 2

A nonuniformly spaced example is generated by

```
X = sort(rand(1,101)*pi);  
Y = sin(X);  
Z = trapz(X,Y);
```

The result is not as accurate as the uniformly spaced grid. One random sample produced

```
Z =  
    1.9984
```

Example 3

This example uses two complex inputs:

```
z = exp(1i*pi*(0:100)/100);  
  
trapz(z, 1./z)  
ans =  
    0.0000 + 3.1411i
```

See Also

cumsum, cumtrapz

Purpose Lay out tree or forest

Syntax `[x,y] = treelayout(parent,post)`
`[x,y,h,s] = treelayout(parent,post)`

Description `[x,y] = treelayout(parent,post)` lays out a tree or a forest. `parent` is the vector of parent pointers, with 0 for a root. `post` is an optional postorder permutation on the tree nodes. If you omit `post`, `treelayout` computes it. `x` and `y` are vectors of coordinates in the unit square at which to lay out the nodes of the tree to make a nice picture.

`[x,y,h,s] = treelayout(parent,post)` also returns the height of the tree `h` and the number of vertices `s` in the top-level separator.

See Also `etree`, `treepplot`, `etreeplot`, `sympfact`

treeplot

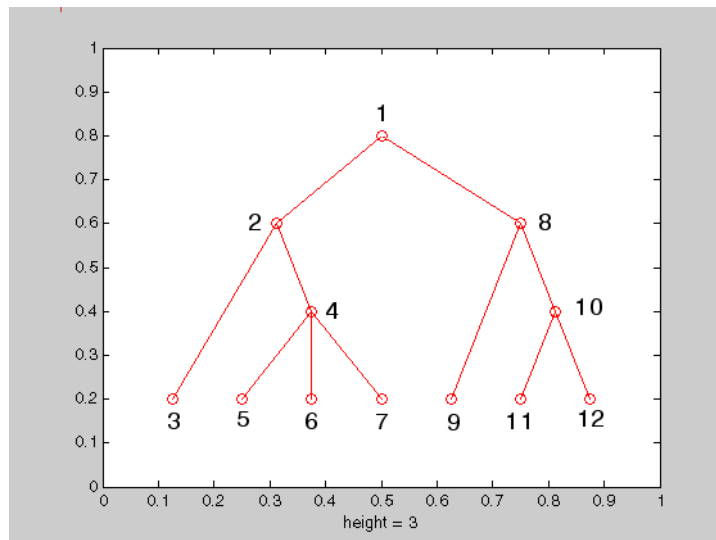
Purpose Plot picture of tree

Syntax `treeplot(p)`
`treeplot(p,nodeSpec,edgeSpec)`

Description `treeplot(p)` plots a picture of a tree given a vector of parent pointers, with $p(i) = 0$ for a root.

`treeplot(p,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `' '` to omit one or both.

Examples To plot a tree with 12 nodes, call `treeplot` with a 12-element input vector. The index of each element in the vector is shown adjacent to each node in the figure below. (These indices are shown only for the point of illustrating the example; they are not part of the `treeplot` output.)



To generate this plot, set the value of each element in the nodes vector to the index of its parent, (setting the parent of the root node to zero).

The node marked 1 in the figure is represented by nodes(1) in the input vector, and because this is the root node which has a parent of zero, you set its value to zero:

```
nodes(1) = 0;    % Root node
```

nodes(2) and nodes(8) are children of nodes(1), so set these elements of the input vector to 1:

```
nodes(2) = 1;    nodes(8) = 1;
```

nodes(5:7) are children of nodes(4), so set these elements to 4:

```
nodes(5) = 4;    nodes(6) = 4;    nodes(7) = 4;
```

Continue in this manner until each element of the vector identifies its parent. For the plot shown above, the nodes vector now looks like this:

```
nodes = [0 1 2 2 4 4 4 1 8 8 10 10];
```

Now call treeplot to generate the plot:

```
treeplot(nodes)
```

See Also

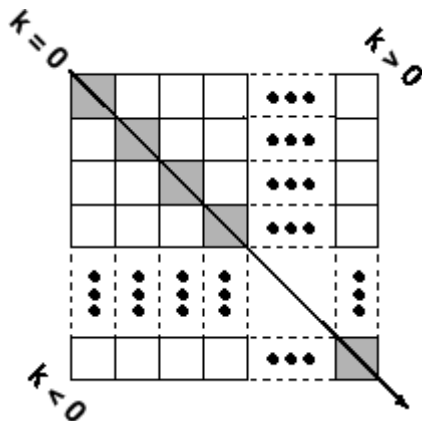
etree, etreeplot, treelayout

tril

Purpose Lower triangular part of matrix

Syntax
`L = tril(X)`
`L = tril(X,k)`

Description `L = tril(X)` returns the lower triangular part of `X`.
`L = tril(X,k)` returns the elements on and below the `k`th diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples `tril(ones(4,4), -1)`

`ans =`

```
0 0 0 0
1 0 0 0
1 1 0 0
1 1 1 0
```

See Also `diag`, `triu`

Purpose	Triangular mesh plot
Syntax	<pre>trimesh(Tri,X,Y,Z) trimesh(Tri,X,Y,Z,C) trimesh(...'PropertyName',PropertyValue...) h = trimesh(...)</pre>
Description	<p><code>trimesh(Tri,X,Y,Z)</code> displays triangles defined in the m-by-3 face matrix <code>Tri</code> as a mesh. Each row of <code>Tri</code> defines a single triangular face by indexing into the vectors or matrices that contain the X, Y, and Z vertices.</p> <p><code>trimesh(Tri,X,Y,Z,C)</code> specifies color defined by <code>C</code> in the same manner as the <code>surf</code> function. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.</p> <p><code>trimesh(...'PropertyName',PropertyValue...)</code> specifies additional patch property names and values for the patch graphics object created by the function.</p> <p><code>h = trimesh(...)</code> returns a handle to a patch graphics object.</p>
Example	<p>Create vertex vectors and a face matrix, then create a triangular mesh plot.</p> <pre>x = rand(1,50); y = rand(1,50); z = peaks(6*x 3,6*x 3); tri = delaunay(x,y); trimesh(tri,x,y,z)</pre>
See Also	<p><code>patch</code>, <code>tetramesh</code>, <code>triplot</code>, <code>trisurf</code>, <code>delaunay</code></p> <p>“Creating Surfaces and Meshes” on page 1-94 for related functions</p>

triplequad

Purpose Numerically evaluate triple integral

Syntax

```
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)
```

Description `triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)` evaluates the triple integral $\int_{x_{\min}}^{x_{\max}} \int_{y_{\min}}^{y_{\max}} \int_{z_{\min}}^{z_{\max}} \text{fun}(x,y,z) \, dz \, dy \, dx$ over the three dimensional rectangular region $x_{\min} \leq x \leq x_{\max}$, $y_{\min} \leq y \leq y_{\max}$, $z_{\min} \leq z \leq z_{\max}$. `fun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. `fun(x,y,z)` must accept a vector `x` and scalars `y` and `z`, and return a vector of values of the integrand.

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

`triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)` uses a tolerance `tol` instead of the default, which is $1.0e-6$.

`triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)` uses the quadrature function specified as `method`, instead of the default `quad`. Valid values for `method` are `@quadl` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quadl`.

Examples Pass M-file function handle `@integrnd` to `triplequad`:P

```
Q = triplequad(@integrnd,0,pi,0,1,-1,1);
```

where the M-file `integrnd.m` is

```
function f = integrnd(x,y,z)
f = y*sin(x)+z*cos(x);
```

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

```
F = @(x,y,z)y*sin(x)+z*cos(x);
```



```
Q = triplequad(F,0,pi,0,1,-1,1);
```

This example integrates $y*\sin(x)+z*\cos(x)$ over the region $0 \leq x \leq \pi, 0 \leq y \leq 1, -1 \leq z \leq 1$. Note that the integrand can be evaluated with a vector x and scalars y and z .

See Also

`dblquad`, `quad`, `quadl`, `function handle (@)`, “Anonymous Functions”

triplot

Purpose 2-D triangular plot

Syntax

```
triplot(TRI,x,y)
triplot(TRI,x,y,color)
h = triplot(...)
triplot(...,'param','value','param','value'...)
```

Description `triplot(TRI,x,y)` displays the triangles defined in the m -by-3 matrix `TRI`. A row of `TRI` contains indices into the vectors `x` and `y` that define a single triangle. The default line color is blue.

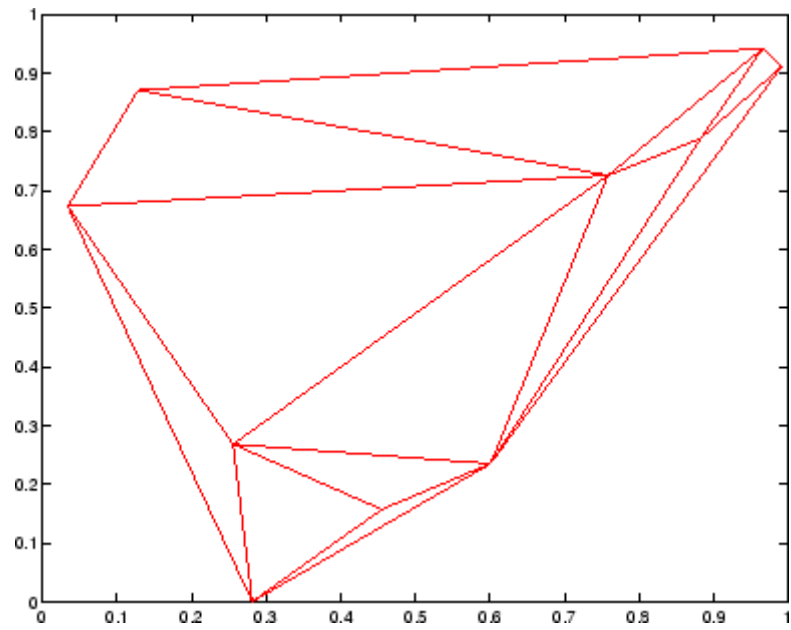
`triplot(TRI,x,y,color)` uses the string `color` as the line color. `color` can also be a line specification. See `ColorSpec` for a list of valid color strings. See `LineStyle` for information about line specifications.

`h = triplot(...)` returns a vector of handles to the displayed triangles.

`triplot(...,'param','value','param','value'...)` allows additional line property name/property value pairs to be used when creating the plot. See `Line Properties` for information about the available properties.

Examples This code plots the Delaunay triangulation for 10 randomly generated points.

```
rand('state',7);
x = rand(1,10);
y = rand(1,10);
TRI = delaunay(x,y);
triplot(TRI,x,y,'red')
```



See Also

ColorSpec, delaunay, line, Line Properties, LineSpec, plot, trimesh, trisurf

trisurf

Purpose Triangular surface plot

Syntax

```
trisurf(Tri,X,Y,Z)
trisurf(Tri,X,Y,Z,C)
trisurf(...'PropertyName',PropertyValue...)
h = trisurf(...)
```

Description

`trisurf(Tri,X,Y,Z)` displays triangles defined in the m -by-3 face matrix `Tri` as a surface. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the X , Y , and Z vertices.

`trisurf(Tri,X,Y,Z,C)` specifies color defined by `C` in the same manner as the `surf` function. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`trisurf(...'PropertyName',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

`h = trisurf(...)` returns a patch handle.

Example Create vertex vectors and a face matrix, then create a triangular surface plot.

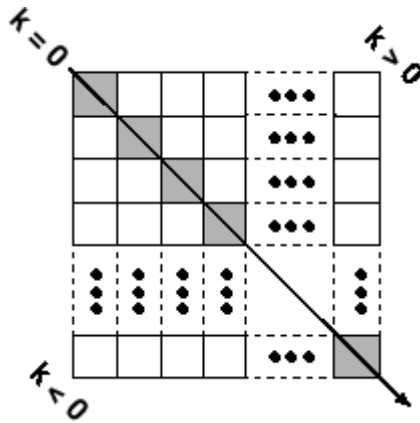
```
x = rand(1,50);
y = rand(1,50);
z = peaks(6*x 3,6*x 3);
tri = delaunay(x,y);
trisurf(tri,x,y,z)
```

See Also `patch`, `surf`, `tetramesh`, `trimesh`, `triplot`, `delaunay`
“Creating Surfaces and Meshes” on page 1-94 for related functions

Purpose Upper triangular part of matrix

Syntax
`U = triu(X)`
`U = triu(X,k)`

Description `U = triu(X)` returns the upper triangular part of `X`.
`U = triu(X,k)` returns the element on and above the `k`th diagonal of `X`.
`k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples `triu(ones(4,4), -1)`

ans =

```

1  1  1  1
1  1  1  1
0  1  1  1
0  0  1  1
```

See Also `diag`, `tril`

true

Purpose Logical 1 (true)

Syntax
true
true(n)
true(m, n)
true(m, n, p, ...)
true(size(A))

Description true is shorthand for logical 1.
true(n) is an n-by-n matrix of logical ones.
true(m, n) or true([m, n]) is an m-by-n matrix of logical ones.
true(m, n, p, ...) or true([m n p ...]) is an m-by-n-by-p-by-... array of logical ones.

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

true(size(A)) is an array of logical ones that is the same size as array A.

Remarks true(n) is much faster and more memory efficient than logical(ones(n)).

See Also false, logical

Purpose Attempt to execute block of code, and catch errors

Description The general form of a try statement is

```
try,  
    statement ,  
    . . . ,  
    statement ,  
catch,  
    statement ,  
    . . . ,  
    statement ,  
end
```

Normally, only the statements between the try and catch are executed. However, if an error occurs during execution of any of the statements, the error is captured into `lasterror`, and the statements between the catch and end are executed. If an error occurs within the catch statements, execution stops unless caught by another try...catch block. The error string produced by a failed try block can be obtained with `lasterror`.

See Also `catch`, `rethrow`, `end`, `lasterror`, `eval`, `evalin`

tscollection

Purpose Create tscollection object

Syntax

```
tsc = tscollection(TimeSeries)
tsc = tscollection(Time)
tsc = tscollection(Time,TimeSeries,'Parameter',Value,...)
```

Description `tsc = tscollection(TimeSeries)` creates a `tscollection` object `tsc` with one or more `timeseries` objects already in the MATLAB workspace. The argument `TimeSeries` can be a

- Single `timeseries` object
- Cell array of `timeseries` objects

`tsc = tscollection(Time)` creates an empty `tscollection` object with the time vector `Time`. When time values are date strings, you must specify `Time` as a cell array of date strings.

`tsc = tscollection(Time,TimeSeries,'Parameter',Value,...)` creates a `tscollection` object with optional parameter-value pairs you enter after the `Time` and `TimeSeries` arguments. You can specify the following parameters:

- **Name** — String that specifies the name of this `tscollection` object
- **IsDatetimeum** — Logical value (`true` or `false`) that when set to `true` specifies that the `Time` values are dates in the format of MATLAB serial dates.

Remarks **Definition: Time Series Collection**

A time series collection object is a MATLAB variable that groups several time series with a common time vector. The time series that you include in the collection are called members of this collection.

Properties of Time Series Collection Objects

This table lists the properties of the `tscollection` object. You can specify the `Time`, `TimeSeries`, and `Name` properties as input arguments in the constructor.

Property	Description
Name	<code>tscollection</code> name as a string. This can differ from the <code>tscollection</code> name in the MATLAB workspace.
Time	When <code>TimeInfo.StartDate</code> is empty, values are measured relative to 0. When <code>TimeInfo.StartDate</code> is defined, values represent date strings measured relative to the <code>StartDate</code> . The length of <code>Time</code> must be the same as the first or the last dimension of <code>Data</code> for each collection.
TimeInfo	Contains fields for contextual information about <code>Time</code> : <ul style="list-style-type: none"> • <code>Units</code> — Time units with any of the following values: 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', 'nanoseconds' • <code>Start</code> — Start time • <code>End</code> — End time (read only) • <code>Increment</code> — Interval between subsequent time values. NaN when times are not uniformly sampled. • <code>Length</code> — Length of the time vector (read only) • <code>Format</code> — String defining the date string display format. See <code>datestr</code>. • <code>StartDate</code> — Date string defining the reference date. See <code>setabstime</code>. • <code>UserData</code> — Any additional user-defined information

tscollection

Examples

The following example shows how to create a `tscollection` object.

- 1 Import the sample data.

```
load count.dat
```

- 2 Create three `timeseries` objects to store each set of data:

```
count1 = timeseries(count(:,1),1:24,'name', 'ts1');  
count2 = timeseries(count(:,2),1:24,'name', 'ts2');
```

- 3 Create a `tscollection` object named `tsc` and add to it two out of three time series already in the MATLAB workspace, by using the following syntax:

```
tsc = tscollection({count1 count2},'name','tsc')
```

See Also

`addts`, `datestr`, `setabstime (tscollection)`, `timeseries`, `tsprops`

Purpose Construct event object for timeseries object

Syntax
`e = tsdata.event(Name,Time)`
`e = tsdata.event(Name,Time,'Datenum')`

Description `e = tsdata.event(Name,Time)` creates an event object with the specified Name that occurs at the time Time. Time can either be a real value or a date string.

`e = tsdata.event(Name,Time,'Datenum')` uses 'Datenum' to indicate that the Time value is a serial date number generated by the datenum function. The Time value is converted to a date string after the event is created.

Remarks You add events by using the addevent method.

Fields of the `tsdata.event` object include the following:

- **EventData** — MATLAB array that stores any user-defined information about the event
- **Name** — String that specifies the name of the event
- **Time** — Time value when this event occurs, specified as a real number or a date string relative to **StartDate**
- **Units** — Time units
- **StartDate** — A reference date, specified in MATLAB `datestr` format. **StartDate** is empty when you have a numerical (non-date-string) time vector.

tsearch

Purpose Search for enclosing Delaunay triangle

Syntax `T = tsearch(x,y,TRI,xi,yi)`

Description `T = tsearch(x,y,TRI,xi,yi)` returns an index into the rows of TRI for each point in xi, yi. The tsearch command returns NaN for all points outside the convex hull. Requires a triangulation TRI of the points x,y obtained from delaunay.

See Also delaunay, delaunayn, dsearch, tsearchn

Purpose	N-D closest simplex search
Syntax	$t = \text{tsearchn}(X, \text{TES}, XI)$ $[t, P] = \text{tsearchn}(X, \text{TES}, XI)$
Description	<p>$t = \text{tsearchn}(X, \text{TES}, XI)$ returns the indices t of the enclosing simplex of the Delaunay tessellation TES for each point in XI. X is an m-by-n matrix, representing m points in N-dimensional space. XI is a p-by-n matrix, representing p points in N-dimensional space. tsearchn returns NaN for all points outside the convex hull of X. tsearchn requires a tessellation TES of the points X obtained from <code>delaunayn</code>.</p> <p>$[t, P] = \text{tsearchn}(X, \text{TES}, XI)$ also returns the barycentric coordinate P of XI in the simplex TES. P is a p-by-$n+1$ matrix. Each row of P is the Barycentric coordinate of the corresponding point in XI. It is useful for interpolation.</p>
Algorithm	tsearchn is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/ . For copyright information, see http://www.qhull.org/COPYING.txt .
See Also	<code>delaunayn</code> , <code>griddatan</code> , <code>tsearch</code>
Reference	[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," <i>ACM Transactions on Mathematical Software</i> , Vol. 22, No. 4, Dec. 1996, p. 469-483.

tsprops

Purpose Help on timeseries object properties

Syntax help timeseries/tsprops

Description help timeseries/tsprops lists the properties of the timeseries object and briefly describes each property.

Time Series Object Properties

Property	Description
Data	<p>Time-series data, where each data sample corresponds to a specific time.</p> <p>The data can be a scalar, a vector, or a multidimensional array. Either the first or last dimension of the data must be aligned with Time.</p> <p>By default, NaNs are used to represent missing or unspecified data. Set the <code>TreatNaNasMissing</code> property to determine how missing data is treated in calculations.</p>

Property	Description
DataInfo	<p>Contains fields for storing contextual information about Data:</p> <ul style="list-style-type: none">• Unit — String that specifies data units• Interpolation — A <code>tsdata.interpolation</code> object that specifies the interpolation method for this time series. For more information, type <code>help tsdata.interpolation</code> at the MATLAB prompt. Fields of the <code>tsdata.interpolation</code> object include:<ul style="list-style-type: none">▪ Fhandle — Function handle to a user-defined interpolation function▪ Name — String that specifies the name of the interpolation method. Predefined methods include 'linear' and 'zoh' (zero-order hold). 'linear' is the default.• UserData — Any user-defined information entered as a string

Property	Description
Events	<p>An array of <code>tsdata.event</code> objects that stores event information for this time series. You add events by using the <code>addevent</code> method. For more information, type <code>help tsdata.event</code> at the command line.</p> <p>Fields of the <code>tsdata.event</code> object include the following:</p> <ul style="list-style-type: none">• EventData — Any user-defined information about the event• Name — String that specifies the name of the event• Time — Time value when this event occurs, specified as a real number or a date string• Units — Time units• StartDate — A reference date specified in MATLAB date-string format. <code>StartDate</code> is empty when you have a numerical (non-date-string) time vector.

Property	Description
IsTimeFirst	<p>Logical value (true or false) specifies whether the first or last dimension of the Data array is aligned with the time vector.</p> <p>You can set this property when the Data array is square and it is ambiguous which dimension is aligned with time. By default, the first Data dimension that matches the length of the time vector is aligned with the time vector.</p> <p>When you set this property to:</p> <ul style="list-style-type: none"> • true — The first dimension of the data array is aligned with the time vector. For example: <code>ts=timeseries(rand(3,3),1:3, 'IsTimeFirst',true);</code> • false — The last dimension of the data array is aligned with the time vector. For example: <code>ts=timeseries(rand(3,3),1:3, 'IsTimeFirst',false);</code> <p>After a time series is created, this property is read only.</p>
Name	<p>Time-series name entered as a string. This name can differ from the name of the time-series variable in the MATLAB workspace.</p>
Quality	<p>An integer vector or array containing values -128 to 127 that specifies the quality in terms of codes defined by <code>QualityInfo.Code</code>.</p> <p>When Quality is a vector, it must have the same length as the time vector. In this case, each Quality value applies to a corresponding data sample.</p> <p>When Quality is an array, it must have the same size as the data array. In this case, each Quality value applies to the corresponding value of the data array.</p>

Property	Description
QualityInfo	<p>Provides a lookup table that converts numerical Quality codes to readable descriptions. QualityInfo fields include the following:</p> <ul style="list-style-type: none">• Code — Integer vector containing values -128 to 127 that define the “dictionary” of quality codes, which you can assign to each Data value by using the Quality property• Description — Cell vector of strings, where each element provides a readable description of the associated quality Code• UserData — Stores any additional user-defined information <p>Lengths of Code and Description must match.</p>
Time	<p>Array of time values.</p> <p>When TimeInfo.StartDate is empty, the numerical Time values are measured relative to 0 in specified units. When TimeInfo.StartDate is defined, the time values are date strings measured relative to the StartDate in specified units.</p> <p>The length of Time must be the same as either the first or the last dimension of Data.</p>

Property	Description
TimeInfo	<p>Uses the following fields for storing contextual information about Time:</p> <ul style="list-style-type: none">• Units — Time units can have any of following values: 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', or 'nanoseconds'• Start — Start time• End — End time (read only)• Increment — Interval between two subsequent time values• Length — Length of the time vector (read only)• Format — String defining the date string display format. See the MATLAB <code>datestr</code> function reference page for more information.• StartDate — Date string defining the reference date. See the MATLAB <code>setabstime (timeseries)</code> function reference page for more information.• UserData — Stores any additional user-defined information
TreatNaNasMissing	<p>Logical value that specifies how to treat NaN values in Data:</p> <ul style="list-style-type: none">• true — (Default) Treat all NaN values as missing data except during statistical calculations.• false — Include NaN values in statistical calculations, in which case NaN values are propagated to the result.

See Also

`datestr`, `get (timeseries)`, `set (timeseries)`, `setabstime (timeseries)`

tstool

Purpose Open Time Series Tools GUI

Syntax

```
tstool
tstool(ts)
tstool(tsc)
tstool(sldata)
tstool(ModelDataLogs, 'replace')
```

Description

`tstool` starts the Time Series Tools GUI without loading any data.

`tstool(ts)` starts the Time Series Tools GUI and loads the time-series object `ts` from the MATLAB workspace.

`tstool(tsc)` starts the Time Series Tools GUI and loads the time-series collection object `tsc` from the MATLAB workspace.

`tstool(sldata)` starts the Time Series Tools GUI and loads the logged-signal data `sldata` from a Simulink model. If a Simulink logged signal `Name` property contains a `/`, the entire logged signal, including all levels of the signal hierarchy, is not imported into Time Series Tools.

`tstool(ModelDataLogs, 'replace')` replaces the logged-signal data object `ModelDataLogs` in the Time Series Tools GUI with an updated logged signal after you rerun the Simulink model. Use this command to update the `ModelDataLogs` object in the Time Series Tools GUI if you change the model or the logged-signal data settings.

See Also `timeseries`, `tscollection`

Purpose

Display contents of file

Syntax

```
type('filename')  
type filename
```

Description

`type('filename')` displays the contents of the specified file in the MATLAB Command Window. Use the full path for `filename`, or use a MATLAB relative partial pathname.

If you do not specify a filename extension and there is no filename file without an extension, the `type` function adds the `.m` extension by default. The `type` function checks the directories specified in the MATLAB search path, which makes it convenient for listing the contents of M-files on the screen. Use `type` with `more` on to see the listing one screen at a time.

`type filename` is the command form of the syntax.

Examples

`type('foo.bar')` lists the contents of the file `foo.bar`.

`type foo` lists the contents of the file `foo`. If `foo` does not exist, `type foo` lists the contents of the file `foo.m`.

See Also

`cd`, `dbtype`, `delete`, `dir`, `more`, `partialpath`, `path`, `what`, `who`

typecast

Purpose Convert data types without changing underlying data

Syntax `Y = typecast(X, type)`

Description `Y = typecast(X, type)` converts a numeric value in `X` to the data type specified by `type`. Input `X` must be a full, noncomplex, numeric scalar or vector. The `type` input is a string set to one of the following: 'uint8', 'int8', 'uint16', 'int16', 'uint32', 'int32', 'uint64', 'int64', 'single', or 'double'.

`typecast` is different from the MATLAB `cast` function in that it does not alter the input data. `typecast` always returns the same number of bytes in the output `Y` as were in the input `X`. For example, casting the 16-bit integer 1000 to `uint8` with `typecast` returns the full 16 bits in two 8-bit segments (3 and 232) thus keeping its original value ($3 \times 256 + 232 = 1000$). The `cast` function, on the other hand, truncates the input value to 255.

The output of `typecast` can be formatted differently depending on what system you use it on. Some computer systems store data starting with its most significant byte (an ordering called *big-endian*), while others start with the least significant byte (called *little-endian*).

Note MATLAB issues an error if `X` contains fewer values than are needed to make an output value.

Examples

Example 1

This example converts between data types of the same size:

```
typecast(uint8(255), 'int8')
ans =
    -1

typecast(int16(-1), 'uint16')
ans =
```

65535

Example 2

Set X to a 1-by-3 vector of 32-bit integers, then cast it to an 8-bit integer type:

```
X = uint32([1 255 256])
X =
           1           255           256
```

Running this on a little-endian system produces the following results. Each 32-bit value is divided up into four 8-bit segments:

```
Y = typecast(X, 'uint8')
Y =
      1      0      0      0 255      0      0      0      0      1      0      0
```

The third element of X, 256, exceeds the 8 bits that it is being converted to in Y(9) and thus overflows to Y(10):

```
Y(9:12)
ans =
      0      1      0      0
```

Note that length(Y) is equal to 4.*length(X). Also note the difference between the output of typecast versus that of cast:

```
Z = cast(X, 'uint8')
Z =
      1 255 255
```

Example 3

This example casts a smaller data type (uint8) into a larger one (uint16). Displaying the numbers in hexadecimal format makes it easier to see just how the data is being rearranged:

```
format hex
X = uint8([44 55 66 77])
```

typecast

```
X =  
    2c    37    42    4d
```

The first typecast is done on a big-endian system. The four 8-bit segments of the input data are combined to produce two 16-bit segments:

```
Y = typecast(X, 'uint16')  
Y =  
    2c37    424d
```

The second is done on a little-endian system. Note the difference in byte ordering:

```
Y = typecast(X, 'uint16')  
Y =  
    372c    4d42
```

You can format the little-endian output into big-endian (and vice versa) using the `swapbytes` function:

```
Y = swapbytes(typecast(X, 'uint16'))  
Y =  
    2c37    424d
```

Example 4

This example attempts to make a 32-bit value from a vector of three 8-bit values. MATLAB issues an error because there are an insufficient number of bytes in the input:

```
format hex  
  
typecast(uint8([120 86 52]), 'uint32')  
??? Too few input values to make output type.  
  
Error in ==> typecast at 29  
out = typecastc(in, datatype);
```


Repeat the example, but with a vector of four 8-bit values, and it returns the expected answer:

```
typecast(uint8([120 86 52 18]), 'uint32')
ans =
    12345678
```

See Also

[cast](#), [class](#), [swapbytes](#)

uibuttongroup

Purpose Create container object to exclusively manage radio buttons and toggle buttons

Syntax `uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2, ...)`
`handle = uibuttongroup(...)`

Description A `uibuttongroup` groups components and manages exclusive selection behavior for radio buttons and toggle buttons that it contains. It can also contain other user interface controls, axes, `uipanel`s, and `uibuttongroups`. It cannot contain ActiveX controls.

`uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2,...)` creates a visible container component in the current figure window. This component manages exclusive selection behavior for `uicontrol`s of style `radiobutton` and `togglebutton`.

Use the `Parent` property to specify the parent as a figure, `uipanel`, or `uibuttongroup`. If you do not specify a parent, `uibuttongroup` adds the button group to the current figure. If no figure exists, one is created.

See the `Uibuttongroup Properties` reference page for more information.

A `uibuttongroup` object can have axes, `uicontrol`, `uipanel`, and `uibuttongroup` objects as children. However, only `uicontrol`s of style `radiobutton` and `togglebutton` are managed by the component.

For the children of a `uibuttongroup` object, the `Position` property is interpreted relative to the button group. If you move the button group, the children automatically move with it and maintain their positions in the button group.

If you have a button group that contains a set of radio buttons and toggle buttons and you want:

- An immediate action to occur when a radio button or toggle button is selected, you must include the code to control the radio and toggle buttons in the button group's `SelectionChangeFcn` callback function, not in the individual toggle button `Callback` functions. See the

SelectionChangeFcn property and the example on this reference page for more information.

- Another component such as a push button to base its action on the selection, then that component's Callback callback can get the handle of the selected radio button or toggle button from the button group's SelectedObject property.

`handle = uibuttongroup(...)` creates a uibuttongroup object and returns a handle to it in `handle`.

After creating a uibuttongroup, you can set and query its property values using `set` and `get`. Run `get(handle)` to see a list of properties and their current values. Run `set(handle)` to see a list of object properties you can set and their legal values.

Examples

This example creates a uibuttongroup with three radiobuttons. It manages the radiobuttons with the SelectionChangeFcn callback, `selcbk`.

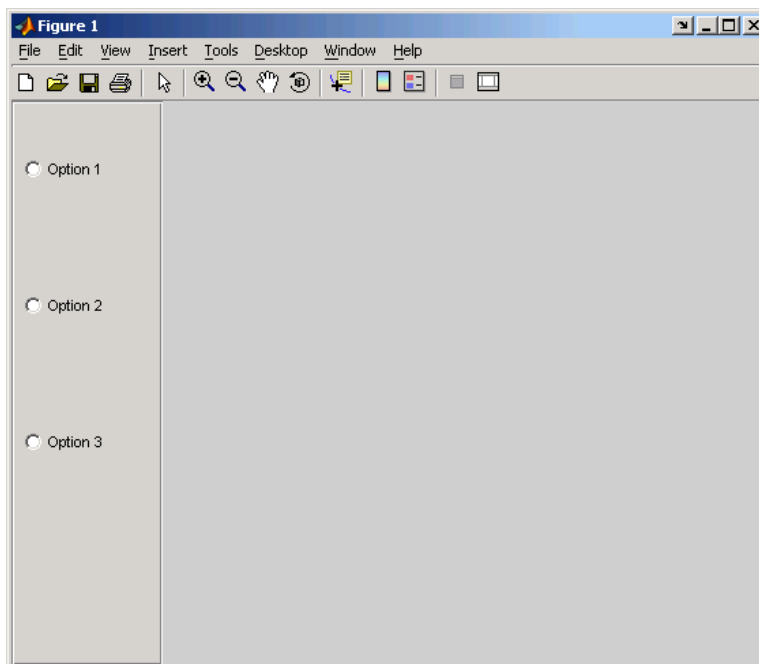
When you select a new radio button, `selcbk` displays the uibuttongroup handle on one line, the EventName, OldValue, and NewValue fields of the event data structure on a second line, and the value of the SelectedObject property on a third line.

```
% Create the button group.
h = uibuttongroup('visible','off','Position',[0 0 .2 1]);
% Create three radio buttons in the button group.
u0 = uicontrol('Style','Radio','String','Option 1',...
    'pos',[10 350 100 30],'parent',h,'HandleVisibility','off');
u1 = uicontrol('Style','Radio','String','Option 2',...
    'pos',[10 250 100 30],'parent',h,'HandleVisibility','off');
u2 = uicontrol('Style','Radio','String','Option 3',...
    'pos',[10 150 100 30],'parent',h,'HandleVisibility','off');
% Initialize some button group properties.
set(h,'SelectionChangeFcn',@selcbk);
set(h,'SelectedObject',[]); % No selection
set(h,'Visible','on');
```

uibbuttongroup

For the `SelectionChangeFcn` callback, `selcbk`, the source and event data structure arguments are available only if `selcbk` is called using a function handle. See `SelectionChangeFcn` for more information.

```
function selcbk(source,eventdata)
disp(source);
disp([eventdata.EventName,' ',...
      get(eventdata.OldValue,'String'),' ', ...
      get(eventdata.NewValue,'String')]);
disp(get(get(source,'SelectedObject'),'String'));
```



If you click Option 2 with no option selected, the `SelectionChangeFcn` callback, `selcbk`, displays:

```
3.0011
```

```
SelectionChanged Option 2  
Option 2
```

If you then click Option 1, the SelectionChangeFcn callback, selcbk, displays:

```
3.0011
```

```
SelectionChanged Option 2 Option 1  
Option 1
```

See Also

uicontrol, uipanel

Uibuttongroup Properties

Purpose

Describes the container objects used to manage buttons and toggle buttons.

Modifying Properties

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

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

Uibuttongroup takes its default property values from `uipanel`. To set a uibuttongroup default property value, set the default for the corresponding `uipanel` property. Note that you can set no default values for the uibuttongroup `SelectedObject` and `SelectionChangeFcn` properties.

For more information about changing the default value of a property see “Setting Default Property Values”. For an example, see the `CreateFcn` property.

Uibuttongroup Properties

This section describes all properties useful to uibuttongroup objects and lists valid values. Curly braces { } enclose default values.

Property Name	Description
<code>BackgroundColor</code>	Color of the uibuttongroup background
<code>BorderType</code>	Type of border around the uibuttongroup area.
<code>BorderWidth</code>	Width of the button group border in pixels.
<code>BusyAction</code>	Interruption of other callback routines
<code>ButtonDownFcn</code>	Button-press callback routine
<code>Children</code>	All children of the uibuttongroup

Uibuttongroup Properties

Property Name	Description
Clipping	Clipping of child axes, uipanel, and uibuttongroups to the uibuttongroup. Does not affect child uicontrols.
CreateFcn	Callback routine executed during object creation
DeleteFcn	Callback routine executed during object deletion
FontAngle	Title font angle
FontName	Title font name
FontSize	Title font size
FontUnits	Title font units
FontWeight	Title font weight
ForegroundColor	Title font color and color of 2-D border line
HandleVisibility	Handle accessibility from command line and GUIs
HighlightColor	3-D frame highlight color
HitTest	Selectable by mouse click
Interruptible	Callback routine interruption mode
Parent	uibuttongroup object's parent
Position	Uibuttongroup position relative to parent figure, uipanel, or uibuttongroup
ResizeFcn	User-specified resize routine
Selected	Whether object is selected
SelectedObject	Currently selected uicontrol of style radiobutton or togglebutton
SelectionChangeFcn	Callback routine executed when the selected radio button or toggle button changes.

Uibuttongroup Properties

Property Name	Description
SelectionHighlight	Object highlighted when selected
ShadowColor	3-D frame shadow color
Tag	User-specified object identifier
Title	Title string
TitlePosition	Location of title string in relation to the button group
UIContextMenu	Associates uicontext menu with the uibuttongroup
Units	Units used to interpret the position vector
UserData	User-specified data
Visible	Uibuttongroup visibility. Note Controls the Visible property of child axes, uipanel, and uibuttongroups. Does not affect child uicontrols.

BackgroundColor
ColorSpec

Color of the uibuttongroup background. A three-element RGB vector or one of the MATLAB predefined names, specifying the background color. See the ColorSpec reference page for more information on specifying color.

BorderType
none | {etchedin} | etchedout |
beveledin | beveledout | line

Border of the uibuttongroup area. Used to define the button group area graphically. Etched and beveled borders provide a 3-D look.

Uibuttongroup Properties

Use the `HighlightColor` and `ShadowColor` properties to specify the border color of etched and beveled borders. A line border is 2-D. Use the `ForegroundColor` property to specify its color.

`BorderWidth`
integer

Width of the button group border. The width of the button group borders in pixels. The default border width is 1 pixel. 3-D borders wider than 3 may not appear correctly at the corners.

`BusyAction`
cancel | {queue}

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`ButtonDownFcn`
string or function handle

Uibuttongroup Properties

Button-press callback routine. A callback routine that executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uibuttongroup. This is useful for implementing actions to interactively modify object properties, such as size and position, when they are clicked on (using the selectmoveresize function, for example).

If you define this routine as a string, the string can be a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

Children

vector of handles

Children of the uibuttongroup. A vector containing the handles of all children of the uibuttongroup. Although a uibuttongroup manages only uicontrols of style radiobutton and togglebutton, its children can be axes, uipanel, uibuttongroups, and other uicontrols. You can use this property to reorder the children.

Clipping

{on} | off

Clipping mode. By default, MATLAB clips a uibuttongroup's child axes, uipanel, and uibuttongroups to the uibuttongroup rectangle. If you set Clipping to off, the axis, uipanel, or uibuttongroup is displayed outside the button group rectangle. This property does not affect child uicontrols which, by default, can display outside the button group rectangle.

CreateFcn

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uibuttongroup object. MATLAB sets all property values for the uibuttongroup before executing the CreateFcn callback so these values are available to

the callback. Within the function, use `gcbo` to get the handle of the `uibuttongroup` being created.

Setting this property on an existing `uibuttongroup` object has no effect.

To define a default `CreateFcn` callback for all new `uibuttongroups` you must define the same default for all `uipanel`s. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uibuttongroup`. For example, the code

```
set(0,'DefaultUipanelCreateFcn','set(gcbo,...  
    'FontName','arial','FontSize',12)')
```

creates a default `CreateFcn` callback that runs whenever you create a new panel or button group. It sets the default font name and font size of the `uipanel` or `uibuttongroup` title.

To override this default and create a button group whose `FontName` and `FontSize` properties are set to different values, call `uibuttongroup` with code similar to

```
hpt = uibuttongroup(...,'CreateFcn','set(gcbo,...  
    'FontName','times','FontSize',14)')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uibuttongroup` call. In the example above, if instead of redefining the `CreateFcn` property for this `uibuttongroup`, you had explicitly set `FontSize` to 14, the default `CreateFcn` callback would have set `FontSize` back to the system dependent default.

Uibuttongroup Properties

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

DeleteFcn
string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the uibuttongroup object (e.g., when you issue a delete command or clear the figure containing the uibuttongroup). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine. The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.

FontAngle
{normal} | italic | oblique

Character slant used in the Title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to italic or oblique selects a slanted version of the font, when it is available on your system.

FontName
string

Font family used in the Title. The name of the font in which to display the Title. To display and print properly, this must be a font that your system supports. The default font is system dependent. To eliminate the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan), set FontName to the string FixedWidth. This string value is case insensitive.

```
set(uicontrol_handle, 'FontName', 'FixedWidth')
```

This then uses the value of the root FixedWidthFontName property, which can be set to the appropriate value for a locale

from `startup.m` in the end user's environment. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize`
integer

Title font size. A number specifying the size of the font in which to display the `Title`, in units determined by the `FontUnits` property. The default size is system dependent.

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

Title font size units. Normalized units interpret `FontSize` as a fraction of the height of the `uibuttongroup`. When you resize the `uibuttongroup`, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

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

Weight of characters in the title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `bold` causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor`
`ColorSpec`

Color used for title font and 2-D border line. A three-element RGB vector or one of the MATLAB predefined names, specifying the font or line color. See the `ColorSpec` reference page for more information on specifying color.

`HandleVisibility`
{on} | callback | off

Uibuttongroup Properties

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is on.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

Note `Uicontrols` of style `radiobutton` and `togglebutton` that are managed by a `uibuttongroup` should not be accessed outside the button group. Set the `HandleVisibility` of such radio buttons and toggle buttons to `off` or `callback` to prevent inadvertent access.

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

HighlightColor
ColorSpec

3-D frame highlight color. A three-element RGB vector or one of the MATLAB predefined names, specifying the highlight color. See the ColorSpec reference page for more information on specifying color.

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 property to the empty matrix.

Interruptible
{on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The Interruptible property of the object whose callback is executing
- Whether the executing callback contains drawnow, figure, getframe, pause, or waitfor statements
- The BusyAction property of the object whose callback is waiting to execute

If the Interruptible property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the drawnow, figure, getframe, pause, or waitfor functions, the function processes

Uibuttongroup Properties

any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the waiting callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine is processed according to the rules described above.

Parent
handle

Uibuttongroup parent. The handle of the `uibuttongroup`'s parent figure, `uipanel`, or `uibuttongroup`. You can move a `uibuttongroup` object to another figure, `uipanel`, or `uibuttongroup` by setting this property to the handle of the new parent.

Position
position rectangle

Size and location of uibuttongroup relative to parent. The rectangle defined by this property specifies the size and location of the button group within the parent figure window, `uipanel`, or `uibuttongroup`. Specify `Position` as

[left bottom width height]

left and bottom are the distance from the lower-left corner of the parent object to the lower-left corner of the uibuttongroup object. width and height are the dimensions of the uibuttongroup rectangle, including the title. All measurements are in units specified by the Units property.

ResizeFcn
string or function handle

Resize callback routine. MATLAB executes this callback routine whenever a user resizes the uibuttongroup and the figure **Resize** property is set to on, or in GUIDE, the **Resize behavior** option is set to Other. You can query the uibuttongroup **Position** property to determine its new size and position. During execution of the callback routine, the handle to the figure being resized is accessible only through the root **CallbackObject** property, which you can query using `gcbo`.

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

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

```
u = findobj('Tag','StatusBar');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
```

Uibuttongroup Properties

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

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

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

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

Selected

on | off (read only)

Is object selected? This property indicates whether the button group is selected. When this property is on, MATLAB displays selection handles if the `SelectionHighlight` property is also on. You can, for example, define the `ButtonDownFcn` function to set this property, allowing users to select the object with the mouse.

SelectedObject

scalar handle

Currently selected radio button or toggle button uicontrol in the managed group of components. Use this property to determine the currently selected component or to initialize selection of one of the radio buttons or toggle buttons. By default, `SelectedObject` is set to the first `uicontrol` radio button or toggle button that is added. Set it to `[]` if you want no selection. Note that `SelectionChangeFcn` does not execute when this property is set by the user.

SelectionChangeFcn

string or function handle

Uibuttongroup Properties

Callback routine executed when the selected radio button or toggle button changes. If this routine is called as a function handle, `uibuttongroup` passes it two arguments. The first argument, `source`, is the handle of the `uibuttongroup`. The second argument, `eventdata`, is an event data structure that contains the fields shown in the following table.

Event Data Structure Field	Description
EventName	'SelectionChanged'
OldValue	Handle of the object selected before this event. [] if none was selected.
NewValue	Handle of the currently selected object.

If you have a button group that contains a set of radio buttons and/or toggle buttons and you want an immediate action to occur when a radio button or toggle button is selected, you must include the code to control the radio and toggle buttons in the button group's `SelectionChangeFcn` callback function, not in the individual toggle button `Callback` functions.

If you want another component such as a push button to base its action on the selection, then that component's `Callback` callback can get the handle of the selected radio button or toggle button from the button group's `SelectedObject` property.

`SelectionHighlight`
{on} | off

Object highlighted when selected. When the `Selected` property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is off, MATLAB does not draw the handles.

`ShadowColor`
ColorSpec

Uibuttongroup Properties

3-D frame shadow color. ShadowColor is a three-element RGB vector or one of the MATLAB predefined names, specifying the shadow color. See the ColorSpec reference page for more information on specifying color.

Tag

string

User-specified object identifier. The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the findobj function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the Tag value 'FormatTb'.

```
h = findobj(figurehandles,'Tag','FormatTb')
```

Title

string

Title string. The text displayed in the button group title. You can position the title using the TitlePosition property.

If the string value is specified as a cell array of strings or padded string matrix, only the first string in the cell array or padded string matrix is displayed; the rest are ignored. Vertical slash ('|') characters are not interpreted as line breaks and instead show up in the text displayed in the uibuttongroup title.

Setting a property value to default, remove, or factory produces the effect described in “Defining Default Values”. To set Title to one of these words, you must precede the word with the backslash character. For example,

```
hp = uibuttongroup(...,'Title','\Default');
```

TitlePosition

{lefttop} | centertop | righttop |
leftbottom | centerbottom | rightbottom

Location of the title. This property determines the location of the title string, in relation to the uibuttongroup.

UIContextMenu

handle

Associate a context menu with a uibuttongroup. Assign this property the handle of a Uicontextmenu object. MATLAB displays the context menu whenever you right-click the uibuttongroup. Use the uicontextmenu function to create the context menu.

Units

inches | centimeters | {normalized} |
points | pixels | characters

Units of measurement. MATLAB uses these units to interpret the Position property. For the button group itself, units are measured from the lower-left corner of its parent figure window, panel, or button group. For children of the button group, they are measured from the lower-left corner of the button group.

- Normalized units map the lower-left corner of the button group or figure window to (0,0) and the upper-right corner to (1.0,1.0).
- pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).
- Character units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of Units, it is good practice to return it to its default value after completing your computation so as not

Uibuttongroup Properties

to affect other functions that assume `Units` is set to the default value.

`UserData`
matrix

User-specified data. Any data you want to associate with the `uibuttongroup` object. MATLAB does not use this data, but you can access it using `set` and `get`.

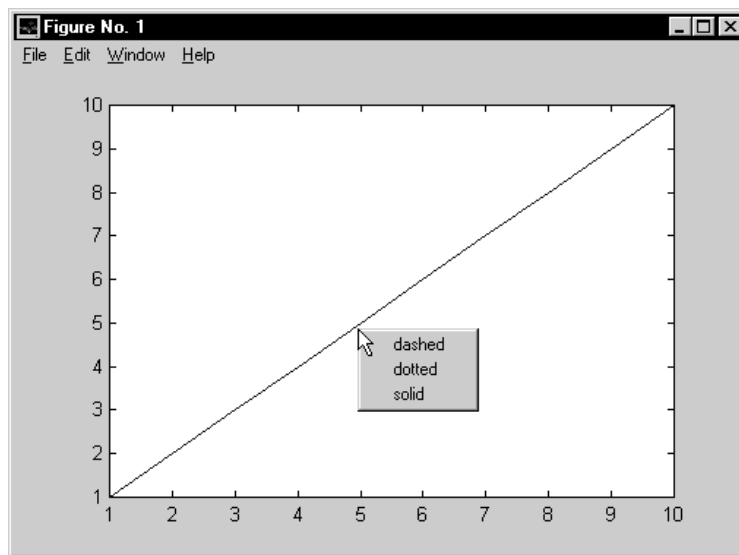
`Visible`
{on} | off

Uibuttongroup visibility. By default, a `uibuttongroup` object is visible. When set to `off`, the `uibuttongroup` is not visible, but still exists and you can query and set its properties.

Note The value of a `uibuttongroup`'s `Visible` property also controls the `Visible` property of child axes, `uipanel`s, and `uibuttongroup`s. This property does not affect the `Visible` property of child `uicontrol`s.

Purpose	Create context menu
Syntax	<code>handle = uicontextmenu('PropertyName',PropertyValue,...)</code>
Description	<p><code>handle = uicontextmenu('PropertyName',PropertyValue,...)</code> creates a context menu, which is a menu that appears when the user right-clicks on a graphics object. See the Uicontextmenu Properties reference page for more information.</p> <p>You create context menu items using the <code>uimenu</code> function. Menu items appear in the order the <code>uimenu</code> statements appear. You associate a context menu with an object using the <code>UIContextMenu</code> property for the object and specifying the context menu's handle as the property value.</p>
Example	<p>These statements define a context menu associated with a line. When the user right clicks or presses Alt+click anywhere on the line, the menu appears. Menu items enable the user to change the line style.</p> <pre>% Define the context menu cmenu = uicontextmenu; % Define the line and associate it with the context menu hline = plot(1:10, 'UIContextMenu', cmenu); % Define callbacks for context menu items cb1 = ['set(hline, 'LineStyle', '--)']; cb2 = ['set(hline, 'LineStyle', ':)']; cb3 = ['set(hline, 'LineStyle', '-')']; % Define the context menu items item1 = uimenu(cmenu, 'Label', 'dashed', 'Callback', cb1); item2 = uimenu(cmenu, 'Label', 'dotted', 'Callback', cb2); item3 = uimenu(cmenu, 'Label', 'solid', 'Callback', cb3);</pre> <p>When the user right clicks or presses Alt+click on the line, the context menu appears, as shown in this figure:</p>

uicontextmenu



See Also

[uibuttongroup](#), [uicontrol](#), [uimenu](#), [uipanel](#)

Purpose

Defines properties for context menus.

Modifying Properties

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

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

For more information about changing the default value of a property see “Setting Default Property Values”. For an example, see the `CreateFcn` property.

Uicontextmenu Properties

This section lists all properties useful to `uicontextmenu` objects along with valid values and descriptions of their use. Curly braces `{}` enclose default values.

Property	Purpose
<code>BusyAction</code>	Callback routine interruption
<code>Callback</code>	Control action
<code>Children</code>	The <code>uimenu</code> s defined for the <code>uicontextmenu</code>
<code>CreateFcn</code>	Callback routine executed during object creation
<code>DeleteFcn</code>	Callback routine executed during object deletion
<code>HandleVisibility</code>	Whether handle is accessible from command line and GUIs
<code>Interruptible</code>	Callback routine interruption mode
<code>Parent</code>	<code>Uicontextmenu</code> object’s parent

Uicontextmenu Properties

Property	Purpose
Position	Location of uicontextmenu when Visible is set to on
Tag	User-specified object identifier
Type	Class of graphics object
UserData	User-specified data
Visible	Uicontextmenu visibility

BusyAction

cancel | {queue}

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of BusyAction to decide whether or not to attempt to interrupt the executing callback.

- If the value is cancel, the event is discarded and the second callback does not execute.
- If the value is queue, and the Interruptible property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object's Interruptible property. See the Interruptible property for information about controlling a callback's interruptibility.

Callback
string

Control action. A routine that executes whenever you right-click an object for which a context menu is defined. The routine executes immediately before the context menu is posted. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

Children
matrix

The uimenu items defined for the uicontextmenu.

CreateFcn
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uicontextmenu object. MATLAB sets all property values for the uicontextmenu before executing the CreateFcn callback so these values are available to the callback. Within the function, use `gcb0` to get the handle of the uicontextmenu being created.

Setting this property on an existing uicontextmenu object has no effect.

You can define a default CreateFcn callback for all new uicontextmenus. This default applies unless you override it by specifying a different CreateFcn callback when you call `uicontextmenu`. For example, the code

```
set(0, 'DefaultUicontextmenuCreateFcn', 'set(gcb0, ...  
    'Visible', 'on')')
```

Uicontextmenu Properties

creates a default `CreateFcn` callback that runs whenever you create a new context menu. It sets the default `Visible` property of a context menu.

To override this default and create a context menu whose `Visible` property is set to a different value, call `uicontextmenu` with code similar to

```
hpt = uicontextmenu(...,'CreateFcn','set(gcbo,...  
    'Visible','off'))
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uicontextmenu` call. In the example above, if instead of redefining the `CreateFcn` property for this `uicontextmenu`, you had explicitly set `Visible` to `off`, the default `CreateFcn` callback would have set `Visible` back to the default, i.e., `on`.

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

`DeleteFcn`
string or function handle

Delete uicontextmenu callback routine. A callback routine that executes when you delete the `uicontextmenu` object (e.g., when you issue a `delete` command or clear the figure containing the `uicontextmenu`). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

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

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

`HandleVisibility`

`{on} | callback | off`

Control access to object’s handle. This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure’s `CurrentObject` property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility`

Uicontextmenu Properties

settings. This does not affect the values of the `HandleVisibility` properties.

`Interruptible`
{on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object's Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure's WindowButtonDownFcn callback routine, or an object's ButtonDownFcn or Callback routine are processed according to the rules described above.

Parent

handle

Uicontextmenu's parent. The handle of the uicontextmenu's parent object. You can move a uicontextmenu object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

Position

vector

Uicontextmenu's position. A two-element vector that defines the location of a context menu posted by setting the Visible property value to on. Specify Position as

[x y]

where vector elements represent the horizontal and vertical distances in pixels from the bottom left corner of the figure window, panel, or button group to the top left corner of the context menu.

Tag

string

User-specified object label. The Tag property provides a means to identify graphics objects with a user-specified label. This

Uicontextmenu Properties

is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define Tag as any string.

Type

string

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

UserData

matrix

User-specified data. Any data you want to associate with the uicontextmenu object. MATLAB does not use this data, but you can access it using set and get.

Visible

on | {off}

Uicontextmenu visibility. The Visible property can be used in two ways:

- Its value indicates whether the context menu is currently posted. While the context menu is posted, the property value is on; when the context menu is not posted, its value is off.
- Its value can be set to on to force the posting of the context menu. Similarly, setting the value to off forces the context menu to be removed. When used in this way, the Position property determines the location of the posted context menu.

Purpose	Create user interface control object
Syntax	<pre>handle = uicontrol('PropertyName',PropertyValue,...) handle = uicontrol(parent,'PropertyName',PropertyValue,...) handle = uicontrol uicontrol(uich)</pre>
Description	<p><code>uicontrol</code> creates a <code>uicontrol</code> graphics objects (user interface controls), which you use to implement graphical user interfaces.</p> <p><code>handle = uicontrol('PropertyName',PropertyValue,...)</code> creates a <code>uicontrol</code> and assigns the specified properties and values to it. It assigns the default values to any properties you do not specify. The default <code>uicontrol</code> style is a pushbutton. The default parent is the current figure. See the <code>Uicontrol Properties</code> reference page for more information.</p> <p><code>handle = uicontrol(parent,'PropertyName',PropertyValue,...)</code> creates a <code>uicontrol</code> in the object specified by the handle, <code>parent</code>. If you also specify a different value for the <code>Parent</code> property, the value of the <code>Parent</code> property takes precedence. <code>parent</code> can be the handle of a figure, <code>uipanel</code>, or <code>uibbuttongroup</code>.</p> <p><code>handle = uicontrol</code> creates a pushbutton in the current figure. The <code>uicontrol</code> function assigns all properties their default values.</p> <p><code>uicontrol(uich)</code> gives focus to the <code>uicontrol</code> specified by the handle, <code>uich</code>.</p> <p>When selected, most <code>uicontrol</code> objects perform a predefined action. MATLAB supports numerous styles of <code>uicontrols</code>, each suited for a different purpose:</p> <ul style="list-style-type: none">• Check boxes• Editable text fields• Frames• List boxes• Pop-up menus

- Push buttons
- Radio buttons
- Sliders
- Static text labels
- Toggle buttons

For information on using these uicontrols within GUIDE, the MATLAB GUI development environment, see [Examples: Programming GUI Components](#) in the [MATLAB Creating GUIs](#) documentation

Specifying the Uicontrol Style

To create a specific type of uicontrol, set the `Style` property as one of the following strings:

- `'checkbox'` – Check boxes generate an action when selected. These devices are useful when providing the user with a number of independent choices. To activate a check box, click the mouse button on the object. The state of the device is indicated on the display.
- `'edit'` – Editable text fields enable users to enter or modify text values. Use editable text when you want text as input. If `Max-Min>1`, then multiple lines are allowed. For multi-line edit boxes, a vertical scrollbar enables scrolling, as do the arrow keys.
- `'frame'` – Frames are rectangles that provide a visual enclosure for regions of a figure window. Frames can make a user interface easier to understand by grouping related controls. Frames have no callback routines associated with them. Only other uicontrols can appear within frames.

Frames are opaque, not transparent, so the order in which you define uicontrols is important in determining whether uicontrols within a frame are covered by the frame or are visible. *Stacking order* determines the order objects are drawn: objects defined first are drawn first; objects defined later are drawn over existing objects. If

you use a frame to enclose objects, you must define the frame before you define the objects.

Note Most frames in existing GUIs can now be replaced with panels (`uipanel`) or button groups (`uibuttongroup`). GUIDE continues to support frames in those GUIs that contain them, but the frame component does not appear in the GUIDE Layout Editor component palette.

- 'listbox' – List boxes display a list of items (defined using the `String` property) and enable users to select one or more items. The `Min` and `Max` properties control the selection mode:

If $\text{Max} - \text{Min} > 1$, then multiple selection is allowed.

If $\text{Max} - \text{Min} \leq 1$, then only single selection is allowed.

The `Value` property indicates selected entries and contains the indices into the list of strings; a vector value indicates multiple selections. MATLAB evaluates the list box's callback routine after any mouse button up event that changes the `Value` property. Therefore, you may need to add a Done button to delay action caused by multiple clicks on list items. List boxes differentiate between single and double clicks and set the figure `SelectionType` property to `normal` or `open` accordingly before evaluating the list box's `Callback` property.

- 'popupmenu' – Pop-up menus open to display a list of choices (defined using the `String` property) when pressed. When not open, a pop-up menu indicates the current choice. Pop-up menus are useful when you want to provide users with a number of mutually exclusive choices, but do not want to take up the amount of space that a series of radio buttons requires. You must specify a value for the `String` property.
- 'pushbutton' – Push buttons generate an action when pressed. To activate a push button, click the mouse button on the push button.
- 'radiobutton' – Radio buttons are similar to check boxes, but are intended to be mutually exclusive within a group of related radio

buttons (i.e., only one is in a pressed state at any given time). To activate a radio button, click the mouse button on the object. The state of the device is indicated on the display. Note that your code can implement the mutually exclusive behavior of radio buttons.

- 'slider' – Sliders accept numeric input within a specific range by enabling the user to move a sliding bar. Users move the bar by pressing the mouse button and dragging the pointer over the bar, or by clicking in the trough or on an arrow. The location of the bar indicates a numeric value, which is selected by releasing the mouse button. You can set the minimum, maximum, and current values of the slider.
- 'text' – Static text boxes display lines of text. Static text is typically used to label other controls, provide directions to the user, or indicate values associated with a slider. Users cannot change static text interactively and there is no way to invoke the callback routine associated with it.
- 'togglebutton' – Toggle buttons are controls that execute callbacks when clicked on and indicate their state, either on or off. Toggle buttons are useful for building toolbars.

Remarks

- The `uicontrol` function accepts property name/property value pairs, structures, and cell arrays as input arguments and optionally returns the handle of the created object. You can also set and query property values after creating the object using the `set` and `get` functions.
- A `uicontrol` object is a child of a `figure`, `uipanel`, or `uibuttongroup` and therefore does not require an axes to exist when placed in a figure window, `uipanel`, or `uibuttongroup`.
- When MATLAB is paused and a `uicontrol` has focus, pressing a keyboard key does not cause MATLAB to resume. Click anywhere outside a `uicontrol` and then press any key. See the `pause` function for more information.

Examples

Example 1

The following statement creates a push button that clears the current axes when pressed.

```
h = uicontrol('Style', 'pushbutton', 'String', 'Clear',...
             'Position', [20 150 100 70], 'Callback', 'cla');
```

This statement gives focus to the pushbutton.

```
uicontrol(h)
```

Example 2

You can create a `uicontrol` object that changes figure colormaps by specifying a pop-up menu and supplying an M-file name as the object's `Callback`:

```
hpop = uicontrol('Style', 'popup',...
                'String', 'hsv|hot|cool|gray',...
                'Position', [20 320 100 50],...
                'Callback', 'setmap');
```

The above call to `uicontrol` defines four individual choices in the menu: `hsv`, `hot`, `cool`, and `gray`. You specify these choices with the `String` property, separating the choices with the `|` character.

The `Callback`, in this case `setmap`, is the name of an M-file that defines a more complicated set of instructions than a single MATLAB command. `setmap` contains these statements:

```
val = get(hpop, 'Value');
if val == 1
    colormap(hsv)
elseif val == 2
    colormap(hot)
elseif val == 3
    colormap(cool)
elseif val == 4
    colormap(gray)
```

uicontrol

end

The Value property contains a number that indicates the selected choice. The choices are numbered sequentially from one to four. The setmap M-file can get and then test the contents of the Value property to determine what action to take.

See Also

textwrap, uibuttongroup, uimenu, uipanel

Purpose

Describes the uicontrol properties.

Modifying Properties

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

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see “Setting Default Property Values”. You can also set default uicontrol properties on the root and figure levels:

```
set(0, 'DefaultUicontrolProperty', PropertyValue...)  
set(gcf, 'DefaultUicontrolProperty', PropertyValue...)
```

where *Property* is the name of the uicontrol property whose default value you want to set and *PropertyValue* is the value you are specifying as the default. Use `set` and `get` to access uicontrol properties.

For information on using these uicontrols within GUIDE, the MATLAB GUI development environment, see Programming GUI Components in the MATLAB Creating GUIs documentation.

Uicontrol Properties

This section lists all properties useful to `uicontrol` objects along with valid values and descriptions of their use. Curly braces `{}` enclose default values.

Property	Purpose
<code>BackgroundColor</code>	Object background color
<code>BusyAction</code>	Callback routine interruption
<code>ButtonDownFcn</code>	Button-press callback routine
<code>Callback</code>	Control action

Uicontrol Properties

Property	Purpose
CData	Truecolor image displayed on the control
Children	Uicontrol objects have no children
CreateFcn	Callback routine executed during object creation
DeleteFcn	Callback routine executed during object deletion
Enable	Enable or disable the uicontrol
FontAngle	Character slant
FontName	Font family
FontSize	Font size
FontUnits	Font size units
FontWeight	Weight of text characters
ForegroundColor	Color of text
HandleVisibility	Whether handle is accessible from command line and GUIs
HitTest	Whether selectable by mouse click
HorizontalAlignment	Alignment of label string
Interruptible	Callback routine interruption mode
KeyPressFcn	Key press callback routine
ListboxTop	Index of top-most string displayed in list box
Max	Maximum value (depends on uicontrol object)
Min	Minimum value (depends on uicontrol object)
Parent	Uicontrol object's parent
Position	Size and location of uicontrol object

Property	Purpose
Selected	Whether object is selected
SelectionHighlight	Object highlighted when selected
SliderStep	Slider step size
String	Uicontrol object label, also list box and pop-up menu items
Style	Type of uicontrol object
Tag	User-specified object identifier
TooltipString	Content of object's tooltip
Type	Class of graphics object
UIContextMenu	Uicontextmenu object associated with the uicontrol
Units	Units to interpret position vector
UserData	User-specified data
Value	Current value of uicontrol object
Visible	Uicontrol visibility

BackgroundColor
ColorSpec

Object background color. The color used to fill the uicontrol rectangle. Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default color is determined by system settings. See ColorSpec for more information on specifying color.

BusyAction
cancel | {queue}

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for

Uicontrol Properties

which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`ButtonDownFcn`

string or function handle (GUIDE sets this property)

Button-press callback routine. A callback routine that can execute when you press a mouse button while the pointer is on or near a uicontrol. Specifically:

- If the uicontrol's `Enable` property is set to on, the `ButtonDownFcn` callback executes when you click the right or left mouse button in a 5-pixel border around the uicontrol or when you click the right mouse button on the control itself.
- If the uicontrol's `Enable` property is set to `inactive` or `off`, the `ButtonDownFcn` executes when you click the right or left mouse button in the 5-pixel border or on the control itself.

This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using `selectmoveresize`, for example).

Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

To add a `ButtonDownFcn` callback in GUIDE, select **View Callbacks** from the Layout Editor **View** menu, then select `ButtonDownFcn`. GUIDE sets this property to the appropriate string and adds the callback to the M-file the next time you save the GUI. Alternatively, you can set this property to the string `%automatic`. The next time you save the GUI, GUIDE sets this property to the appropriate string and adds the callback to the M-file.

Use the `Callback` property to specify the callback routine that executes when you activate the enabled uicontrol (e.g., click a push button).

Callback

string or function handle (GUIDE sets this property)

Control action. A routine that executes whenever you activate the uicontrol object (e.g., when you click on a push button or move a slider). Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

To execute the callback routine for an edit text control, type in the desired text and then do one of the following:

- Click another component, the menu bar, or the background of the GUI.
- For a single line editable text box, press `Enter`, or
- For a multiline editable text box, press **Ctrl+Enter**.

Callback routines defined for static text do not execute because no action is associated with these objects.

Uicontrol Properties

CData
matrix

Truecolor image displayed on control. A three-dimensional matrix of RGB values that defines a truecolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0.

Children
matrix

The empty matrix; `uicontrol` objects have no children.

Clipping
{on} | off

This property has no effect on `uicontrol` objects.

CreateFcn
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uicontrol` object. MATLAB sets all property values for the `uicontrol` before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the `uicontrol` being created.

Setting this property on an existing `uicontrol` object has no effect.

You can define a default `CreateFcn` callback for all new `uicontrols`. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uicontrol`. For example, the code

```
set(0, 'DefaultUicontrolCreateFcn', 'set(gcbo, ...  
    'BackgroundColor', 'white')')
```

creates a default `CreateFcn` callback that runs whenever you create a new uicontrol. It sets the default background color of all new uicontrols.

To override this default and create a uicontrol whose `BackgroundColor` is set to a different value, call `uicontrol` with code similar to

```
hpt = uicontrol(...,'CreateFcn','set(gcbo,...  
    'BackgroundColor','blue')')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uicontrol` call. In the example above, if instead of redefining the `CreateFcn` property for this uicontrol, you had explicitly set `BackgroundColor` to `blue`, the default `CreateFcn` callback would have set `BackgroundColor` back to the default, i.e., `white`.

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

`DeleteFcn`
string or function handle

Delete uicontrol callback routine. A callback routine that executes when you delete the uicontrol object (e.g., when you issue a `delete` command or `clear` the figure containing the uicontrol). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

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

Uicontrol Properties

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

Enable

{on} | inactive | off

Enable or disable the uicontrol. This property controls how uicontrols respond to mouse button clicks, including which callback routines execute.

- on – The uicontrol is operational (the default).
- inactive – The uicontrol is not operational, but looks the same as when Enable is on.
- off – The uicontrol is not operational and its image (set by the Cdata property) is grayed out.

When you left-click on a uicontrol whose Enable property is on, MATLAB performs these actions in this order:

- 1** Sets the figure’s SelectionType property.
- 2** Executes the uicontrol’s ClickedCallback routine.
- 3** Does not set the figure’s CurrentPoint property and does not execute either the control’s ButtonDownFcn or the figure’s WindowButtonDownFcn callback.

When you left-click on a uicontrol whose Enable property is off, or when you right-click a uicontrol whose Enable property has any value, MATLAB performs these actions in this order:

- 4** Sets the figure’s SelectionType property.
- 5** Sets the figure’s CurrentPoint property.
- 6** Executes the figure’s WindowButtonDownFcn callback.

Extent

position rectangle (read only)

Size of uicontrol character string. A four-element vector that defines the size and position of the character string used to label the uicontrol. It has the form:

```
[0,0,width,height]
```

The first two elements are always zero. `width` and `height` are the dimensions of the rectangle. All measurements are in units specified by the `Units` property.

Since the `Extent` property is defined in the same units as the uicontrol itself, you can use this property to determine proper sizing for the uicontrol with regard to its label. Do this by

- Defining the `String` property and selecting the font using the relevant properties.
- Getting the value of the `Extent` property.
- Defining the width and height of the `Position` property to be somewhat larger than the width and height of the `Extent`.

For multiline strings, the `Extent` rectangle encompasses all the lines of text. For single line strings, the `Extent` is returned as a single line, even if the string wraps when displayed on the control.

`FontAngle`
{normal} | italic | oblique

Character slant. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

`FontName`
string

Font family. The name of the font in which to display the `String`. To display and print properly, this must be a font that your system supports. The default font is system dependent.

Uicontrol Properties

To use a fixed-width font that looks good in any locale (and displays properly in Japan, where multibyte character sets are used), set `FontName` to the string `FixedWidth` (this string value is case sensitive):

```
set(uicontrol_handle, 'FontName', 'FixedWidth')
```

This parameter value eliminates the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` and rely on the root `FixedWidthFontName` property to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root `FixedWidthFontName` property to the appropriate value for that locale from `startup.m`. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

FontSize
size in `FontUnits`

Font size. A number specifying the size of the font in which to display the String, in units determined by the `FontUnits` property. The default point size is system dependent.

FontUnits
{points} | normalized | inches |
centimeters | pixels

Font size units. This property determines the units used by the `FontSize` property. Normalized units interpret `FontSize` as a fraction of the height of the uicontrol. When you resize the uicontrol, MATLAB modifies the screen `FontSize` accordingly. pixels, inches, centimeters, and points are absolute units (1 point = $\frac{1}{72}$ inch).

FontWeight

light | {normal} | demi | bold

Weight of text characters. MATLAB uses this property to select a font from those available on your particular system. Setting this property to bold causes MATLAB to use a bold version of the font, when it is available on your system.

ForegroundColor

ColorSpec

Color of text. This property determines the color of the text defined for the String property (the uicontrol label). Specify a color using a three-element RGB vector or one of MATLAB's predefined names. The default text color is black. See ColorSpec for more information on specifying color.

HandleVisibility

{on} | callback | off

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca,(gcf, gco, newplot, cla, clf, and close. Neither is the handle visible in the parent figure's CurrentObject property. Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when HandleVisibility is on.
- Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Uicontrol Properties

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

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

Note Radio buttons and toggle buttons that are managed by a `uibuttongroup` should not be accessed outside the button group. Set the `HandleVisibility` of such radio buttons and toggle buttons to `off` to prevent inadvertent access.

`HitTest`
`{on} | off`

Selectable by mouse click. This property has no effect on uicontrol objects.

`HorizontalAlignment`
`left | {center} | right`

Horizontal alignment of label string. This property determines the justification of the text defined for the `String` property (the uicontrol label):

- `left` — Text is left justified with respect to the uicontrol.
- `center` — Text is centered with respect to the uicontrol.
- `right` — Text is right justified with respect to the uicontrol.

On Microsoft Windows systems, this property affects only edit and text uicontrols.

Interruptible
{on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The Interruptible property of the object whose callback is executing
- Whether the executing callback contains drawnow, figure, getframe, pause, or waitfor statements
- The BusyAction property of the object whose callback is waiting to execute

If the Interruptible property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the drawnow, figure, getframe, pause, or waitfor functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.

Uicontrol Properties

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

`KeyPressFcn`

string or function handle

Key press callback function. A callback routine invoked by a key press when the callback's uicontrol object has focus. Focus is denoted by a border or a dotted border, respectively, in UNIX and Microsoft Windows. If no uicontrol has focus, the figure's key press callback function, if any, is invoked. `KeyPressFcn` can be a function handle, the name of an M-file, or any legal MATLAB expression.

If the specified value is the name of an M-file, the callback routine can query the figure's `CurrentCharacter` property to determine what particular key was pressed and thereby limit the callback execution to specific keys.

If the specified value is a function handle, the callback routine can retrieve information about the key that was pressed from its event data structure argument.

Event Data Structure Field	Description	Examples:			
		a	=	Shift	Shift/a
Character	Character interpretation of the key that was pressed.	'a'	'='	' '	'A'
Modifier	Current modifier, such as 'control', or an empty cell array if there is no modifier	{1x0 cell}	{1x0 cell}	{'shift'}	{'shift'}
Key	Name of the key that was pressed.	'a'	'equal'	'shift'	'a'

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

ListboxTop
scalar

Index of top-most string displayed in list box. This property applies only to the listbox style of uicontrol. It specifies which string appears in the top-most position in a list box that is not large enough to display all list entries. ListboxTop is an index into the array of strings defined by the String property and must have a value between 1 and the number of strings. Noninteger values are fixed to the next lowest integer.

Max
scalar

Maximum value. This property specifies the largest value allowed for the Value property. Different styles of uicontrols interpret Max differently:

- Check boxes – Max is the setting of the Value property while the check box is selected.

Uicontrol Properties

- Editable text – If $\text{Max} - \text{Min} > 1$, then editable text boxes accept multiline input. If $\text{Max} - \text{Min} \leq 1$, then editable text boxes accept only single line input.
- List boxes – If $\text{Max} - \text{Min} > 1$, then list boxes allow multiple item selection. If $\text{Max} - \text{Min} \leq 1$, then list boxes do not allow multiple item selection.
- Radio buttons – Max is the setting of the `Value` property when the radio button is selected.
- Sliders – Max is the maximum slider value and must be greater than the Min property. The default is 1.
- Toggle buttons – Max is the value of the `Value` property when the toggle button is selected. The default is 1.
- Pop-up menus, push buttons, and static text do not use the Max property.

`Min`

scalar

Minimum value. This property specifies the smallest value allowed for the `Value` property. Different styles of uicontrols interpret `Min` differently:

- Check boxes – Min is the setting of the `Value` property while the check box is not selected.
- Editable text – If $\text{Max} - \text{Min} > 1$, then editable text boxes accept multiline input. If $\text{Max} - \text{Min} \leq 1$, then editable text boxes accept only single line input.
- List boxes – If $\text{Max} - \text{Min} > 1$, then list boxes allow multiple item selection. If $\text{Max} - \text{Min} \leq 1$, then list boxes allow only single item selection.
- Radio buttons – Min is the setting of the `Value` property when the radio button is not selected.

- Sliders – Min is the minimum slider value and must be less than Max. The default is 0.
- Toggle buttons – Min is the value of the Value property when the toggle button is not selected. The default is 0.
- Pop-up menus, push buttons, and static text do not use the Min property.

Parent

handle

Uicontrol parent. The handle of the uicontrol's parent object. You can move a uicontrol object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

Position

position rectangle

Size and location of uicontrol. The rectangle defined by this property specifies the size and location of the control within the parent figure window, uipanel, or uibuttongroup. Specify Position as

```
[left bottom width height]
```

left and bottom are the distance from the lower-left corner of the parent object to the lower-left corner of the uicontrol object. width and height are the dimensions of the uicontrol rectangle. All measurements are in units specified by the Units property.

On Microsoft Windows systems, the height of pop-up menus is automatically determined by the size of the font. The value you specify for the height of the Position property has no effect.

The width and height values determine the orientation of sliders. If width is greater than height, then the slider is oriented

Uicontrol Properties

horizontally, If height is greater than width, then the slider is oriented vertically.

Selected

on | {off} (read only)

Is object selected. When this property is on, MATLAB displays selection handles if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

SelectionHighlight

{on} | off

Object highlight when selected. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles.

SliderStep

[min_step max_step]

Slider step size. This property controls the amount the slider Value changes when you click the mouse on the arrow button (min_step) or on the slider trough (max_step). Specify SliderStep as a two-element vector; each value must be in the range [0, 1]. The actual step size is a function of the specified SliderStep and the total slider range (Max - Min). The default, [0.01 0.10], provides a 1 percent change for clicks on the arrow button and a 10 percent change for clicks in the trough.

For example, if you create the following slider,

```
uicontrol('Style','slider','Min',1,'Max',7,...  
         'SliderStep',[0.1 0.6])
```

clicking on the arrow button moves the indicator by,

```
0.1*(7 - 1)
```



```
ans =  
    0.6000
```

and clicking in the trough moves the indicator by,

```
0.6*(7-1)  
ans =  
    3.6000
```

Note that if the specified step size moves the slider to a value outside the range, the indicator moves only to the Max or Min value.

See also the Max, Min, and Value properties.

String
string

Uicontrol label, list box items, pop-up menu choices. **For check boxes, editable text, push buttons, radio buttons, static text, and toggle buttons**, the text displayed on the object. For list boxes and pop-up menus, the set of entries or items displayed in the object.

Note If you specify a numerical value for String, MATLAB converts it to char but the result may not be what you expect. If you have numerical data, you should first convert it to a string, e.g., using num2string, before assigning it to the String property.

For uicontrol objects that display only one line of text, if the string value is specified as a cell array of strings or padded string matrix, only the first string of a cell array or of a padded string matrix is displayed; the rest are ignored. Vertical slash ('|') characters are not interpreted as line breaks and instead show up in the text displayed in the uicontrol.

Uicontrol Properties

For multiple line editable text or static text controls, line breaks occur between each row of the string matrix, and each cell of a cell array of strings. Vertical slash ('|') characters and \n characters are not interpreted as line breaks, and instead show up in the text displayed in the uicontrol.

For multiple items on a list box or pop-up menu, you can specify items as a cell array of strings, a padded string matrix, or within a string vector separated by vertical slash ('|') characters. Use the Value property to set the index of the initial item selected.

For **check boxes, push buttons, radio buttons, toggle buttons**, and the selected item in **popup menus**, when the specified text is clipped because it is too long for the uicontrol, an ellipsis (...) is appended to the text in the active GUI to indicate that it has been clipped.

For **editable text**, the String property value is set to the string entered by the user.

Reserved Words There are three reserved words: default, remove, factory (case sensitive). If you want to use one of these reserved words in the String property, you must precede it with a backslash ('\') character. For example,

```
h = uicontrol('Style','edit','String','\default');
```

Style

{pushbutton} | togglebutton | radiobutton | checkbox | edit | text | slider | frame | listbox | popupmenu

Style of uicontrol object to create. The Style property specifies the kind of uicontrol to create. See the uicontrol Description section for information on each type.

Tag

string (GUIDE sets this property)

User-specified object label. The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define Tag as any string.

TooltipString

string

Content of tooltip for object. The TooltipString property specifies the text of the tooltip associated with the uicontrol. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

Type

string (read only)

Class of graphics object. For uicontrol objects, Type is always the string 'uicontrol'.

UIContextMenu

handle

Associate a context menu with uicontrol. Assign this property the handle of a uicontextmenu object. MATLAB displays the context menu whenever you right-click over the uicontrol. Use the uicontextmenu function to create the context menu.

Units

{pixels} | normalized | inches | centimeters | points | characters (GUIDE default: normalized)

Units of measurement. MATLAB uses these units to interpret the Extent and Position properties. All units are measured from the lower-left corner of the parent object.

Uicontrol Properties

- Normalized units map the lower-left corner of the parent object to (0,0) and the upper-right corner to (1.0,1.0).
- pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).
- Character units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

`UserData`
matrix

User-specified data. Any data you want to associate with the uicontrol object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Value`
scalar or vector

Current value of uicontrol. The uicontrol style determines the possible values this property can have:

- Check boxes set `Value` to `Max` when they are on (when selected) and `Min` when off (not selected).
- List boxes set `Value` to a vector of indices corresponding to the selected list entries, where 1 corresponds to the first item in the list.
- Pop-up menus set `Value` to the index of the item selected, where 1 corresponds to the first item in the menu. The `Examples` section shows how to use the `Value` property to determine which item has been selected.

- Radio buttons set Value to Max when they are on (when selected) and Min when off (not selected).
- Sliders set Value to the number indicated by the slider bar.
- Toggle buttons set Value to Max when they are down (selected) and Min when up (not selected).
- Editable text, push buttons, and static text do not set this property.

Set the Value property either interactively with the mouse or through a call to the set function. The display reflects changes made to Value.

Visible
{on} | off

Uicontrol visibility. By default, all uicontrols are visible. When set to off, the uicontrol is not visible, but still exists and you can query and set its properties.

Note Setting Visible to off for uicontrols that are not displayed initially in the GUI, can result in faster startup time for the GUI.

uigetdir

Purpose Open standard dialog box for selecting a directory

Syntax

```
directory_name = uigetdir
directory_name = uigetdir('start_path')
directory_name = uigetdir('start_path','dialog_title')
```

Description `uigetdir` displays a dialog box enabling the user to browse through the directory structure and select a directory.

`directory_name = uigetdir` opens a dialog box in the current directory displaying the default title.

`directory_name = uigetdir('start_path')` opens a dialog box in the directory specified by `start_path`.

`directory_name = uigetdir('start_path','dialog_title')` opens a dialog box with the specified title.

Remarks

Returned `directory_name`

`directory_name` is a string containing the path to the directory selected in the dialog box. If the user presses the **Cancel** button or closes the dialog window, `directory_name` is returned as the number 0.

Specifying `start_path`

`start_path` specifies the directory to display when the dialog is first opened. If `start_path` is a string representing a valid directory path, the dialog box opens in the specified directory.

If `start_path` is an empty string (`''`), the dialog box opens in the current working directory.

If `start_path` is not a valid directory path, the dialog box opens in the base directory:

- On Windows systems, the base directory is the Windows desktop directory.
- On UNIX systems, the base directory is the directory from which MATLAB is started.

Specifying dialog_title

The placement of `dialog_title` in the dialog box depends on the computer system:

- On Windows systems, the string replaces the default caption inside the dialog box for specifying instructions to the user.
- On UNIX systems, the string replaces the default title of the dialog box.

If you do not specify the `dialog_title` argument, MATLAB uses the default string `Select Directory to Open`.

Adding and Moving Directories

On Windows systems, users can click the **New Folder** button to add a new directory to the directory structure displayed. Users can also drag and drop existing directories.

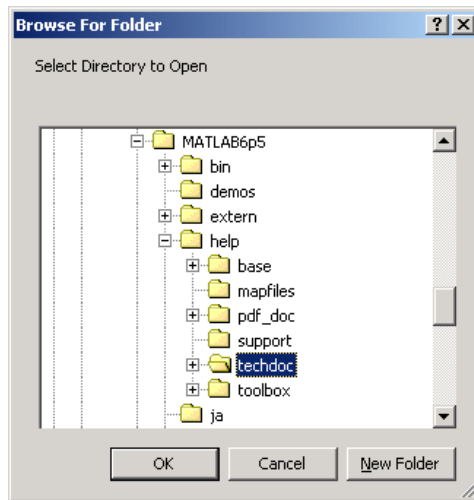
Examples

Example 1

The following statement displays the directories on the C: drive.

```
dname = uigetdir('C:\');
```

The dialog box is shown in the following figure.



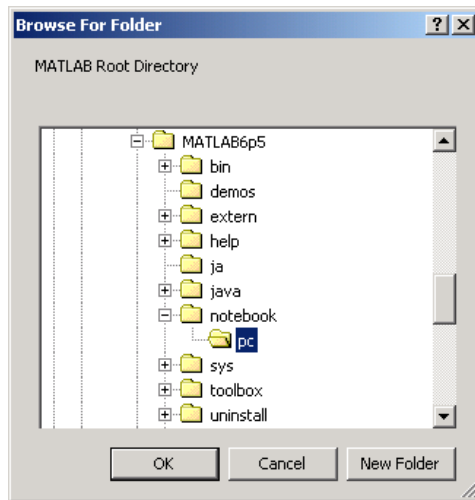
If the user selects the directory MATLAB6.5\help\techdoc, as shown in the figure, and clicks **OK**, uigetdir returns

```
dbname =  
C:\MATLAB6.5\help\techdoc
```

Example 2

The following statement uses the matlabroot command to display the MATLAB root directory in the dialog box:

```
uigetdir(matlabroot,'MATLAB Root Directory')
```

If the user selects the directory MATLAB6.5/notebook/pc, as shown in the figure, `uigetdir` returns a string like

```
C:\MATLAB6.5\notebook\pc
```

assuming that MATLAB is installed on drive C:\.

See Also

`uigetfile`, `uiputfile`

uigetfile

Purpose Open standard dialog box for retrieving files

Syntax

```
uigetfile
uigetfile('FilterSpec')
uigetfile('FilterSpec','DialogTitle')
uigetfile('FilterSpec','DialogTitle','DefaultName')
uigetfile(...,'Location',[x y])
uigetfile(...,'MultiSelect',selectmode)
[FileName,PathName] = uigetfile(...)
[FileName,PathName,FilterIndex] = uigetfile(...)
```

Description `uigetfile` displays a dialog box used to retrieve one or more files. The dialog box lists the files and directories in the current directory.

`uigetfile('FilterSpec')` displays a dialog box that lists files in the current directory. `FilterSpec` determines the initial display of files and can include the * wildcard. For example, `'*.m'` lists all the MATLAB M-files.

If `FilterSpec` is a string, `uigetfile` appends 'All Files' to the list of file types. If `FilterSpec` is a cell array, the first column contains the list of extensions, and the second column contains the list of descriptions. `FilterSpec` can also be a filename. In this case the filename becomes the default filename and the file's extension is used as the default filter. If `FilterSpec` is not specified, `uigetfile` uses the default list of file types (i.e., all MATLAB files).

`uigetfile('FilterSpec','DialogTitle')` displays a dialog box that has the title `DialogTitle`.

`uigetfile('FilterSpec','DialogTitle','DefaultName')` displays a dialog box in which a specified string, in this case `'DefaultName'`, appears in the **File name** field. `'Default Name'` can be a filename or the name of a directory. If it is the name of a directory, you must follow it with a slash (/) or backslash (\) separator.

`uigetfile(...,'Location',[x y])` positions the dialog box at position `[x,y]`, where `x` and `y` are the distances in pixel units from

the left and top edges of the screen. This feature is supported only on UNIX platforms.

`uigetfile(...,'MultiSelect',selectmode)` specifies if multiple file selection is enabled for the `uigetfile` dialog. Valid values for `selectmode` are 'on' and 'off'(default). If the value of 'MultiSelect' is 'on' and the user selects more than one file in the dialog box, then `FileName` is a cell array of strings, each of which represents the name of a selected file. Otherwise, `FileName` is a string representing the selected filename. Because multiple selections are always in the same directory, `PathName` is always a string that represents a single directory.

`[FileName,PathName] = uigetfile(...)` returns the name and path of the file selected in the dialog box. After the user clicks the **Done** button, `FileName` contains the name of the file selected and `PathName` contains the name of the path selected. If the user clicks the **Cancel** button or closes the dialog window, `FileName` and `PathName` are set to 0.

`[FileName,PathName,FilterIndex] = uigetfile(...)` returns the index of the filter selected in the dialog box. The indexing starts at 1. If the user clicks the **Cancel** button or closes the dialog window, `FilterIndex` is set to 0.

Remarks

A successful return occurs only if all the selected files exist. If the user selects a file that does not exist, an error message is displayed and control returns to the dialog box.

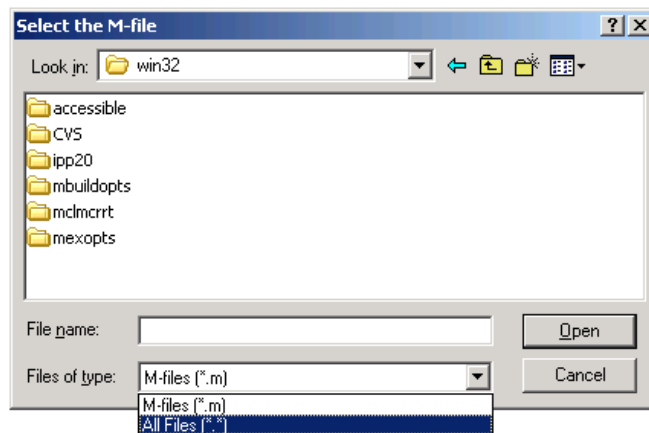
Examples

Example 1

The following statement displays a dialog box that enables the user to retrieve a file. The statement lists all MATLAB M-files within a selected directory. The name and path of the selected file are returned in `FileName` and `PathName`. Note that `uigetfile` appends `All Files (*.*)` to the file types when `FilterSpec` is a string.

```
[FileName,PathName] = uigetfile('*.m','Select the M-file');
```

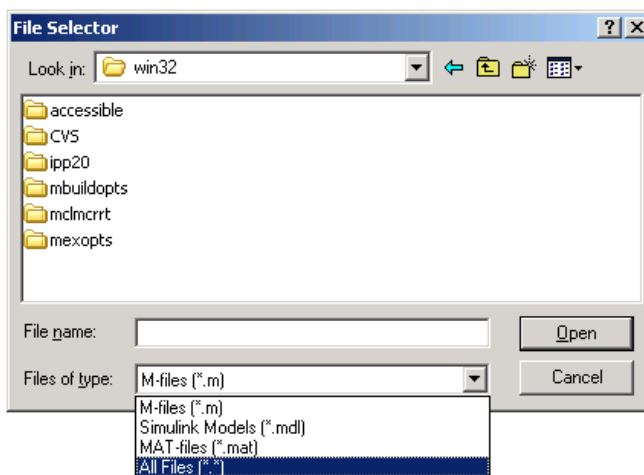
The dialog box is shown in the following figure.



Example 2

To create a list of file types that appears in the **Files of type** list box, separate the file extensions with semicolons, as in the following code. Note that `uigetfile` displays a default description for each known file type, such as Simulink Models for `.mdl` files.

```
[filename, pathname] = ...  
    uigetfile({'*.m'; '*.mdl'; '*.mat'; '*.*'}, 'File Selector');
```

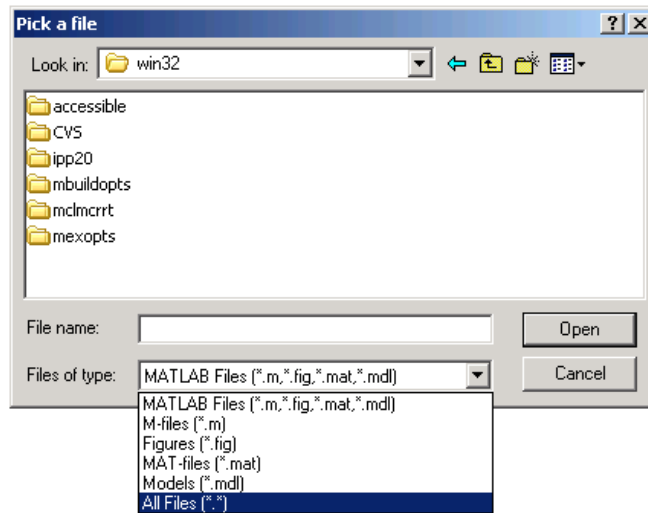


Example 3

If you want to create a list of file types and give them descriptions that are different from the defaults, use a cell array, as in the following code. This example also associates multiple file types with the 'MATLAB Files' description.

```
[filename, pathname] = uigetfile( ...
{'*.m;*.fig;*.mat;*.mdl', 'MATLAB Files (*.m,*.fig,*.mat,*.mdl)';
 '*.m', 'M-files (*.m)'; ...
 '*.fig', 'Figures (*.fig)'; ...
 '*.mat', 'MAT-files (*.mat)'; ...
 '*.mdl', 'Models (*.mdl)'; ...
 '*.*', 'All Files (*.*)'}, ...
'Pick a file');
```

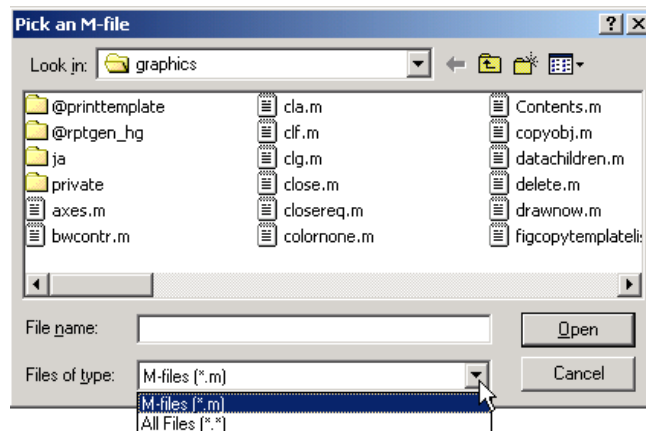
The first column of the cell array contains the file extensions, while the second contains the descriptions you want to provide for the file types. Note that the first entry of column one contains several extensions, separated by semicolons, all of which are associated with the description 'MATLAB Files (*.m,*.fig,*.mat,*.mdl)'. The code produces the dialog box shown in the following figure.



Example 4

The following code checks for the existence of the file and displays a message about the result of the open operation.

```
[filename, pathname] = uigetfile('*.*', 'Pick an M-file');  
if isequal(filename,0)  
    disp('User selected Cancel')  
else  
    disp(['User selected', fullfile(pathname, filename)])  
end
```

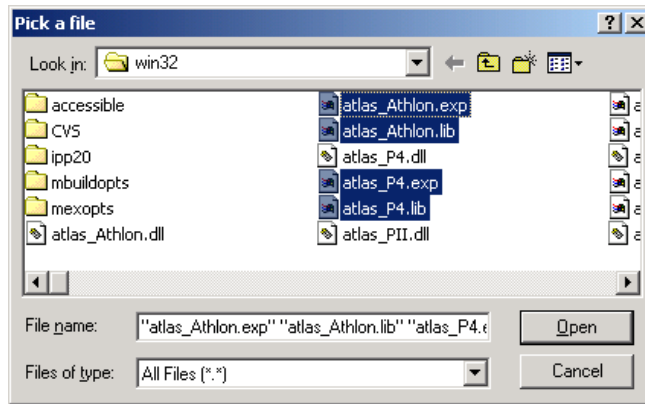


Example 5

This example creates a list of file types and gives them descriptions that are different from the defaults, then enables multiple file selection. The user can select multiple files by holding down the **Shift** or **Ctrl** key and clicking on a file.

```
[filename, pathname, filterindex] = uigetfile( ...
    { '*.mat', 'MAT-files (*.mat)'; ...
      '*.mdl', 'Models (*.mdl)'; ...
      '*.*', 'All Files (*.*)'}, ...
    'Pick a file', ...
    'MultiSelect', 'on');
```

uigetfile



See Also

`uigetdir`, `uiputfile`

Purpose

Open dialog box for retrieving preferences

Syntax

```
value = uigetpref(group,pref,title,question,pref_choices)
[val,dlgshown] = uigetpref(...)
```

Description

`value = uigetpref(group,pref,title,question,pref_choices)` returns one of the strings in `pref_choices`, by doing one of the following:

- Prompting the user with a multiple-choice question dialog box
- Returning a previous answer stored in the preferences database

By default, the dialog box is shown, with each choice on a different pushbutton, and with a checkbox controlling whether the returned value should be stored in preferences and automatically reused in subsequent invocations.

If the user checks the checkbox before choosing one of the push buttons, the push button choice is stored in preferences and returned in `value`. Subsequent calls to `uigetpref` detect that the last choice was stored in preferences, and return that choice immediately without displaying the dialog.

If the user does not check the checkbox before choosing a pushbutton, the selected preference is not stored in preferences. Rather, a special value, 'ask', is stored, indicating that subsequent calls to `uigetpref` should display the dialog box.

Note `uigetpref` uses the same preference database as `addpref`, `getpref`, `ispref`, `rmpref`, and `setpref`. However, it registers the preferences it sets in a separate list so that it, and `uisetpref`, can distinguish those preferences that are being managed with `uigetpref`.

For preferences registered with `uigetpref`, you can use `setpref` and `uisetpref` to explicitly change preference values to 'ask'.

group and pref define the preference. If the preference does not already exist, uigetpref creates it.

title defines the string displayed in the dialog box titlebar.

question is a descriptive paragraph displayed in the dialog, specified as a string array or cell array of strings. This should contain the question the user is being asked, and should be detailed enough to give the user a clear understanding of their choice and its impact. uigetpref inserts line breaks between rows of the string array, between elements of the cell array of strings, or between ' | ' or newline characters in the string vector.

pref_choices is either a string, cell array of strings, or '|'-separated strings specifying the strings to be displayed on the push buttons. Each string element is displayed in a separate push button. The string on the selected pushbutton is returned.

Make pref_choices a 2-by-n cell array of strings if the internal preference values are different from the strings displayed on the pushbuttons. The first row contains the preference strings, and the second row contains the related pushbutton strings. Note that the preference values are returned in value, not the button labels.

[val,dlgshown] = uigetpref(...) returns whether or not the dialog was shown.

Additional arguments can be passed in as parameter-value pairs:

(... 'CheckboxState', state) sets the initial state of the checkbox, either checked or unchecked. state can be either 0 (unchecked) or 1 (checked). By default it is 0.

(... 'CheckboxString', cbstr) sets the string cbstr on the checkbox. By default it is 'Never show this dialog again'.

(... 'HelpString', hstr) sets the string hstr on the help button. By default the string is empty and there is no help button.

(... 'HelpFcn', hfcn) sets the callback that is executed when the help button is pressed. By default it is doc('uigetpref'). Note that if there is no 'HelpString' option, a button is not created.

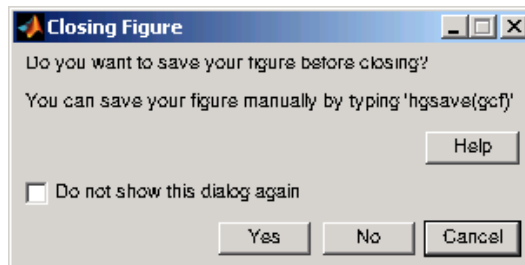
(... 'ExtraOptions', eo) creates extra buttons which are not mapped to any preference settings. eo can be a string or a cell array of strings. By default it is {} and no extra buttons are created. If the user chooses one of these buttons, the dialog is closed and the string is returned in value.

(... 'DefaultButton', dbstr) sets the string value dbstr that is returned if the dialog is closed. By default, it is the first button. Note that dbstr does not have to correspond to a preference or ExtraOption.

Note If the preference does not already exist in the preference database, uigetpref creates it. Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

This example creates the following preference dialog for the 'savefigurebeforeclosing' preference in the 'mygraphics' group.



It uses the cell array {'always', 'never'; 'Yes', 'No'} to define the preference values as 'always' and 'never', and their corresponding button labels as 'Yes' and 'No'.

```
[selectedButton,dlgShown]=uigetpref('mygraphics',... % Group
    'savefigurebeforeclosing',...           % Preference
    'Closing Figure',...                   % Window title
    {'Do you want to save your figure before closing?'
```

uigetpref

```
''
    'You can save your figure manually by typing ''hgsave(gcf)''},...
{'always','never';'Yes','No'},...      % Values and button strings
'ExtraOptions','Cancel',...           % Additional button
'DefaultButton','Cancel',...         % Default choice
'HelpString','Help',...              % String for Help button
'HelpFcn','doc(''closereq'');')      % Callback for Help button
```

See Also

addpref, getpref, ispref, rmpref, setpref, uisetpref

Purpose Open Import Wizard to import data

Syntax

```
uiimport
uiimport(filename)
uiimport('-file')
uiimport('-pastespecial')
S = uiimport(...)
```

Description `uiimport` starts the Import Wizard in the current directory, presenting options to load data from a file or the clipboard.

`uiimport(filename)` starts the Import Wizard, opening the file specified in `filename`. The Import Wizard displays a preview of the data in the file.

`uiimport('-file')` works as above but presents the file selection dialog first.

`uiimport('-pastespecial')` works as above but presents the clipboard contents first.

`S = uiimport(...)` works as above with resulting variables stored as fields in the struct `S`.

Note For ASCII data, you must verify that the Import Wizard correctly identified the column delimiter.

See Also `load`, `clipboard`

uimenu

Purpose Create menus on figure windows

Syntax
`handle = uimenu('PropertyName',PropertyValue,...)`
`handle = uimenu(parent,'PropertyName',PropertyValue,...)`

Description `uimenu` creates a hierarchy of menus and submenus that are displayed in the figure window's menu bar. You also use `uimenu` to create menu items for context menus.

`handle = uimenu('PropertyName',PropertyValue,...)` creates a menu in the current figure's menu bar using the values of the specified properties and assigns the menu handle to `handle`.

See the Uimenu Properties reference page for more information.

`handle = uimenu(parent,'PropertyName',PropertyValue,...)` creates a submenu of a parent menu or a menu item on a context menu specified by `parent` and assigns the menu handle to `handle`. If `parent` refers to a figure instead of another `uimenu` object or a `uicontextmenu`, MATLAB creates a new menu on the referenced figure's menu bar.

Remarks

MATLAB adds the new menu to the existing menu bar. If the figure does not have a menu bar, MATLAB creates one. Each menu choice can itself be a menu that displays its submenu when selected. `uimenu` accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.

The `uimenu` `Callback` property defines the action taken when you activate the created menu item.

Uimenu only appear in figures whose `Window Style` is `normal`. If a figure containing `uimenu` children is changed to `modal`, the `uimenu` children still exist and are contained in the `Children` list of the figure, but are not displayed until the `WindowStyle` is changed to `normal`.

The value of the figure `MenuBar` property affects the content of the figure menu bar. When `MenuBar` is `figure`, a set of built-in menus precedes any user-created `uimenu`s on the menu bar (MATLAB controls the built-in menus and their handles are not available to the user).

When `MenuBar` is `none`, `uimenu`s are the only items on the menu bar (that is, the built-in menus do not appear).

You can set and query property values after creating the menu using `set` and `get`.

Examples

This example creates a menu labeled **Workspace** whose choices allow users to create a new figure window, save workspace variables, and exit out of MATLAB. In addition, it defines an accelerator key for the Quit option.

```
f = uimenu('Label','Workspace');
uimenu(f,'Label','New Figure','Callback','figure');
uimenu(f,'Label','Save','Callback','save');
uimenu(f,'Label','Quit','Callback','exit',...
        'Separator','on','Accelerator','Q');
```

See Also

`uicontrol`, `uicontextmenu`, `gcb0`, `set`, `get`, `figure`

Uimenu Properties

Purpose

Defines the uimenu properties.

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The set and get commands enable you to set and query the values of properties

You can set default Uimenu properties on the root, figure and menu levels:

```
set(0, 'DefaultUimenuPropertyName', PropertyValue...)  
set(gcf, 'DefaultUimenuPropertyName', PropertyValue...)  
set(menu_handle, 'DefaultUimenuPropertyName', PropertyValue...)
```

Where *PropertyName* is the name of the Uimenu property and *PropertyValue* is the value you specify as the default for that property.

For more information about changing the default value of property see .

Uimenu Properties

This section lists all properties useful to uimenu objects along with valid values and instructions for their use. Curly braces { } enclose default values.

Property Name	Property Description
Accelerator	Keyboard equivalent
BusyAction	Callback routine interruption
Callback	Control action
Checked	Menu check indicator
Children	Handles of submenus

Property Name	Property Description
CreateFcn	Callback routine executed during object creation
DeleteFcn	Callback routine executed during object deletion
Enable	Enable or disable the uimenu
ForegroundColor	Color of text
HandleVisibility	Whether handle is accessible from command line and GUIs
Interruptible	Callback routine interruption mode
Label	Menu label
Parent	Uimenu object's parent
Position	Relative uimenu position
Separator	Separator line mode
Tag	User-specified object identifier
Type	Class of graphics object
UserData	User-specified data
Visible	Uimenu visibility

Accelerator
character

Keyboard equivalent. An alphabetic character specifying the keyboard equivalent for the menu item. This allows users to select a particular menu choice by pressing the specified character in conjunction with another key, instead of selecting the menu item with the mouse. The key sequence is platform specific:

Uimenu Properties

- For Microsoft Windows systems, the sequence is **Ctrl**+Accelerator. These keys are reserved for default menu items: c, v, and x.
- For UNIX systems, the sequence is **Ctrl**+Accelerator. These keys are reserved for default menu items: o, p, s, and w.

You can define an accelerator only for menu items that do not have children menus. Accelerators work only for menu items that directly execute a callback routine, not items that bring up other menus.

Note that the menu item does not have to be displayed (e.g., a submenu) for the accelerator key to work. However, the window focus must be in the figure when the key sequence is entered.

To remove an accelerator, set Accelerator to an empty string, ''.

BusyAction
cancel | {queue}

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of BusyAction to decide whether or not to attempt to interrupt the executing callback.

- If the value is cancel, the event is discarded and the second callback does not execute.
- If the value is queue, and the Interruptible property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

Callback

string or function handle

Menu action. A callback routine that executes whenever you select the menu. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

A menu with children (submenus) executes its callback routine before displaying the submenus. A menu without children executes its callback routine when you *release* the mouse button (i.e., on the button up event).

Checked

on | {off}

Menu check indicator. Setting this property to `on` places a check mark next to the corresponding menu item. Setting it to `off` removes the check mark. You can use this feature to create menus that indicate the state of a particular option. For example, suppose you have a menu item called **Show axes** that toggles the visibility of an axes between visible and invisible each time the user selects the menu item. If you want a check to appear next to the menu item when the axes are visible, add the following code to the callback for the **Show axes** menu item:

```
if strcmp(get(gcbo, 'Checked'), 'on')
    set(gcbo, 'Checked', 'off');
else
```

Uimenu Properties

```
        set(gcbo, 'Checked', 'on');  
    end
```

This changes the value of the Checked property of the menu item from on to off or vice versa each time a user selects the menu item.

Note that there is no formal mechanism for indicating that an unchecked menu item will become checked when selected.

Note This property is ignored for top level and parent menus.

Children
vector of handles

Handles of submenus. A vector containing the handles of all children of the uimenu object. The children objects of uimenu are other uimenu, which function as submenus. You can use this property to reorder the menus.

CreateFcn
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uimenu object. MATLAB sets all property values for the uimenu before executing the CreateFcn callback so these values are available to the callback. Within the function, use gcbo to get the handle of the uimenu being created.

Setting this property on an existing uimenu object has no effect.

You can define a default CreateFcn callback for all new uimenu. This default applies unless you override it by specifying a different CreateFcn callback when you call uimenu. For example, the code

```
set(0, 'DefaultUimenuCreateFcn', 'set(gcbo,...  
    'Visible','on'))
```

creates a default `CreateFcn` callback that runs whenever you create a new menu. It sets the default `Visible` property of a `uimenu` object.

To override this default and create a menu whose `Visible` property is set to a different value, call `uimenu` with code similar to

```
hpt = uimenu(..., 'CreateFcn', 'set(gcbo,...  
    'Visible','off'))
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uimenu` call. In the example above, if instead of redefining the `CreateFcn` property for this `uimenu`, you had explicitly set `Visible` to `off`, the default `CreateFcn` callback would have set `Visible` back to the default, i.e., `on`.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

`DeleteFcn`
string or function handle

Delete uimenu callback routine. A callback routine that executes when you delete the `uimenu` object (e.g., when you issue a `delete` command or cause the figure containing the `uimenu` to reset). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

Uimenu Properties

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which is more simply queried using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

Enable

{on} | off

Enable or disable the uimenu. This property controls whether a menu item can be selected. When not enabled (set to off), the menu `Label` appears dimmed, indicating the user cannot select it.

ForegroundColor

ColorSpec X-Windows only

Color of menu label string. This property determines color of the text defined for the `Label` property. Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default text color is black. See `ColorSpec` for more information on specifying color.

HandleVisibility

{on} | callback | off

Control access to object’s handle. This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure’s `CurrentObject` property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is on.

- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`Interruptible`
{on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`,

Uimenu Properties

getframe, pause, or waitfor functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object's Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure's WindowButtonDownFcn callback routine, or an object's ButtonDownFcn or Callback routine are processed according to the rules described above.

Label

string

Menu label. A string specifying the text label on the menu item. You can specify a mnemonic for the label using the '&' character. Except as noted below, the character that follows the '&' in the string appears underlined and selects the menu item when you type **Alt+** followed by that character while the menu is visible. The '&' character is not displayed. To display the '&' character in a label, use two '&' characters in the string:

'O&pen selection' yields **Open selection**

'Save && Go' yields **Save & Go**

'Save&&Go' yields **Save & Go**

'Save& Go' yields **Save& Go** (the space is not a mnemonic)

There are three reserved words: default, remove, factory (case sensitive). If you want to use one of these reserved words in the Label property, you must precede it with a backslash ('\ ') character. For example:

'\remove' yields **remove**

'\default' yields **default**

'\factory' yields **factory**

Parent

handle

Uimenu's parent. The handle of the uimenu's parent object. The parent of a uimenu object is the figure on whose menu bar it displays, or the uimenu of which it is a submenu. You can move a uimenu object to another figure by setting this property to the handle of the new parent.

Position

scalar

Relative menu position. The value of Position indicates placement on the menu bar or within a menu. Top-level menus are placed from left to right on the menu bar according to the value of their Position property, with 1 representing the left-most position. The individual items within a given menu are placed from top to bottom according to the value of their Position property, with 1 representing the top-most position.

Separator

on | {off}

Separator line mode. Setting this property to on draws a dividing line above the menu item.

Uimenu Properties

Tag

string

User-specified object label. The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define Tag as any string.

Type

string (read only)

Class of graphics object. For uimenu objects, Type is always the string 'uimenu'.

UserData

matrix

User-specified data. Any matrix you want to associate with the uimenu object. MATLAB does not use this data, but you can access it using the set and get commands.

Visible

{on} | off

Uimenu visibility. By default, all uimenu are visible. When set to off, the uimenu is not visible, but still exists and you can query and set its properties.

Purpose Convert to unsigned integer

Syntax

```
I = uint8(X)
I = uint16(X)
I = uint32(X)
I = uint64(X)
```

Description `I = uint*(X)` converts the elements of array `X` into unsigned integers. `X` can be any numeric object (such as a double). The results of a `uint*` operation are shown in the next table.

Operation	Output Range	Output Type	Bytes per Element	Output Class
uint8	0 to 255	Unsigned 8-bit integer	1	uint8
uint16	0 to 65,535	Unsigned 16-bit integer	2	uint16
uint32	0 to 4,294,967,295	Unsigned 32-bit integer	4	uint32
uint64	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	8	uint64

double and single values are rounded to the nearest `uint*` value on conversion. A value of `X` that is above or below the range for an integer class is mapped to one of the endpoints of the range. For example,

```
uint16(70000)
ans =
    65535
```

If `X` is already an unsigned integer of the same class, then `uint*` has no effect.

uint8, uint16, uint32, uint64

You can define or overload your own methods for `uint*` (as you can for any object) by placing the appropriately named method in an `@uint*` directory within a directory on your path. Type `help datatypes` for the names of the methods you can overload.

Remarks

Most operations that manipulate arrays without changing their elements are defined for integer values. Examples are `reshape`, `size`, the logical and relational operators, subscripted assignment, and subscripted reference.

Some arithmetic operations are defined for integer arrays on interaction with other integer arrays of the same class (e.g., where both operands are `uint16`). Examples of these operations are `+`, `-`, `.*`, `./`, `.\` and `.^`. If at least one operand is scalar, then `*`, `/`, `\`, and `^` are also defined. Integer arrays may also interact with scalar `double` variables, including constants, and the result of the operation is an integer array of the same class. Integer arrays saturate on overflow in arithmetic.

A particularly efficient way to initialize a large array is by specifying the data type (i.e., class name) for the array in the `zeros`, `ones`, or `eye` function. For example, to create a 100-by-100 `uint64` array initialized to zero, type

```
I = zeros(100, 100, 'uint64');
```

An easy way to find the range for any MATLAB integer type is to use the `intmin` and `intmax` functions as shown here for `uint32`:

```
intmin('uint32')           intmax('uint32')
ans =                      ans =
    0                      4294967295
```

See Also

`double`, `single`, `int8`, `int16`, `int32`, `int64`, `intmax`, `intmin`

Purpose Open file selection dialog box with appropriate file filters

Syntax

```
uiopen  
uiopen('MATLAB')  
uiopen('LOAD')  
uiopen('FIGURE')  
uiopen('SIMULINK')  
uiopen('EDITOR')
```

Description uiopen displays a file selection dialog from which a user can select a file to open. The dialog is the same as the one displayed when you select **Open** from the **File** menu in the MATLAB desktop.

Selecting a file in the dialog and clicking **Open** does the following:

- Gets the file using `uigetfile`
- Opens the file in the base workspace using the `open` command

Note uiopen cannot be compiled. If you want to create a file selection dialog that can be compiled, use `uigetfile`.

`uiopen` or `uiopen('MATLAB')` displays the dialog with the file filter set to all MATLAB files.

`uiopen('LOAD')` displays the dialog with the file filter set to MAT-files (*.mat).

`uiopen('FIGURE')` displays the dialog with the file filter set to figure files (*.fig).

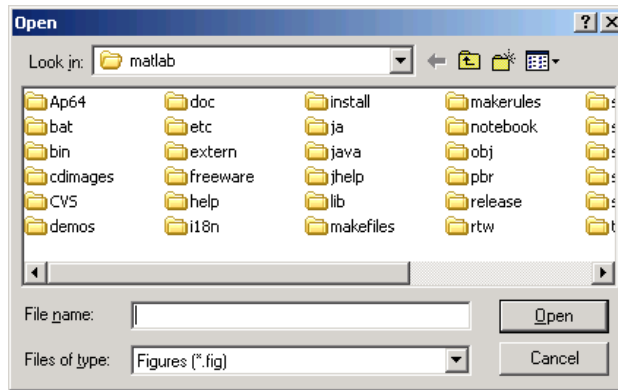
`uiopen('SIMULINK')` displays the dialog with the file filter set to model files (*.mdl).

`uiopen('EDITOR')` displays the dialog with the file filter set to all MATLAB files except for MAT-files and FIG-files. All files are opened in the MATLAB Editor.

uiopen

Examples

Typing `uiopen('figure')` sets the **Files of type** field to Figures (*.fig):



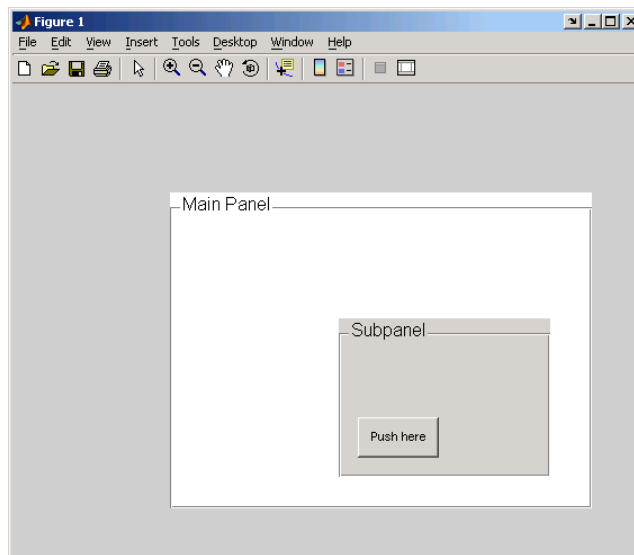
See Also

`uigetfile`, `uiputfile`, `uisave`

Purpose	Create panel container object
Syntax	<pre>h = uipanel('PropertyName1',value1,'PropertyName2',value2, ...) h = uipanel(parent,'PropertyName1',value1,'PropertyName2', value2,...)</pre>
Description	<p>A uipanel groups components. It can contain user interface controls with which the user interacts directly. It can also contain axes, other uipanels, and uibuttongroups. It cannot contain ActiveX controls.</p> <pre>h = uipanel('PropertyName1',value1,'PropertyName2',value2,...)</pre> creates a uipanel container object in a figure, uipanel, or uibuttongroup. Use the Parent property to specify the parent figure, uipanel, or uibuttongroup. If you do not specify a parent, uipanel adds the panel to the current figure. If no figure exists, one is created. See the Uipanel Properties reference page for more information.
	<pre>h = uipanel(parent,'PropertyName1',value1,'PropertyName2',value2,...)</pre> creates a uipanel in the object specified by the handle, parent. If you also specify a different value for the Parent property, the value of the Parent property takes precedence. parent must be a figure, uipanel, or uibuttongroup.
	<p>A uipanel object can have axes, uicontrol, uipanel, and uibuttongroup objects as children. For the children of a uipanel, the Position property is interpreted relative to the uipanel. If you move the panel, the children automatically move with it and maintain their positions relative to the panel.</p> <p>After creating a uipanel object, you can set and query its property values using set and get.</p>
Examples	<p>This example creates a uipanel in a figure, then creates a subpanel in the first panel. Finally, it adds a pushbutton to the subpanel. Both panels use the default Units property value, normalized. Note that default Units for the uicontrol pushbutton is pixels.</p>

uipanel

```
h = figure;  
hp = uipanel('Title','Main Panel','FontSize',12,...  
            'BackgroundColor','white',...  
            'Position',[.25 .1 .67 .67]);  
hsp = uipanel('Parent',hp,'Title','Subpanel','FontSize',12,...  
             'Position',[.4 .1 .5 .5]);  
hbsp = uicontrol('Parent',hsp,'String','Push here',...  
                'Position',[18 18 72 36]);
```



See Also

hgtransform, uibuttongroup, uicontrol

Purpose

Defines the uipanel properties.

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default uipanel properties by typing:

```
set(h, 'DefaultUipanelPropertyName', PropertyValue...)
```

Where `h` can be the root handle (0), a figure handle, or a uipanel handle. *PropertyName* is the name of the uipanel property and *PropertyValue* is the value you specify as the default for that property.

Note Default properties you set for uipanel also apply to `uibuttongroups`.

For more information about changing the default value of a property see “Setting Default Property Values”. For an example, see the `CreateFcn` property.

Uipanel Properties

This section lists all properties useful to `uipanel` objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property Name	Description
<code>BackgroundColor</code>	Color of the uipanel background
<code>BorderType</code>	Type of border around the uipanel area.

Uipanel Properties

Property Name	Description
BorderWidth	Width of the panel border.
BusyAction	Interruption of other callback routines
ButtonDownFcn	Button-press callback routine
Children	All children of the uipanel
Clipping	Clipping of child axes, uipanel, and uibuttongroups to the uipanel. Does not affect child uicontrols.
CreateFcn	Callback routine executed during object creation
DeleteFcn	Callback routine executed during object deletion
FontAngle	Title font angle
FontName	Title font name
FontSize	Title font size
FontUnits	Title font units
FontWeight	Title font weight
ForegroundColor	Title font color and/or color of 2-D border line
HandleVisibility	Handle accessibility from commandline and GUIs
HighlightColor	3-D frame highlight color
HitTest	Selectable by mouse click
Interruptible	Callback routine interruption mode
Parent	Uipanel object's parent
Position	Panel position relative to parent figure or uipanel
ResizeFcn	User-specified resize routine

Property Name	Description
Selected	Whether object is selected
SelectionHighlight	Object highlighted when selected
ShadowColor	3-D frame shadow color
Tag	User-specified object identifier
Title	Title string
TitlePosition	Location of title string in relation to the panel
UIContextMenu	Associates uicontext menu with the uipanel
Units	Units used to interpret the position vector
UserData	User-specified data
Visible	Uipanel visibility. <hr/> Note Controls the Visible property of child axes, uibuttongroups, and uipanels. Does not affect child uicontrols. <hr/>

BackgroundColor
ColorSpec

Color of the uipanel background. A three-element RGB vector or one of the MATLAB predefined names, specifying the background color. See the ColorSpec reference page for more information on specifying color.

BorderType
none | {etchedin} | etchedout | beveledin | beveledout
| line

Border of the uipanel area. Used to define the panel area graphically. Etched and beveled borders provide a 3-D look. Use

Uipanel Properties

the `HighlightColor` and `ShadowColor` properties to specify the border color of etched and beveled borders. A line border is 2-D. Use the `ForegroundColor` property to specify its color.

`BorderWidth`
integer

Width of the panel border. The width of the panel borders in pixels. The default border width is 1 pixel. 3-D borders wider than 3 may not appear correctly at the corners.

`BusyAction`
cancel | {queue}

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`ButtonDownFcn`
string or function handle

Button-press callback routine. A callback routine that executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uipanel. This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using the `selectmoveresize` function, for example).

If you define this routine as a string, the string can be a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

Children

vector of handles

Children of the uipanel. A vector containing the handles of all children of the uipanel. A `uipanel` object's children are axes, uipanels, `uibuttongroups`, and `uicontrols`. You can use this property to reorder the children.

Clipping

{on} | off

Clipping mode. By default, MATLAB clips a `uipanel`'s child axes, uipanels, and `uibuttongroups` to the `uipanel` rectangle. If you set `Clipping` to `off`, the axis, `uipanel`, or `uibuttongroup` is displayed outside the panel rectangle. This property does not affect child `uicontrols` which, by default, can display outside the panel rectangle.

CreateFcn

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uipanel` object. MATLAB sets all property values for the `uipanel` before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the `uipanel` being created.

Uipanel Properties

Setting this property on an existing uipanel object has no effect.

You can define a default `CreateFcn` callback for all new uipanel. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uipanel`. For example, the code

```
set(0, 'DefaultUipanelCreateFcn', 'set(gcbo,...  
    ' 'FontName','arial','FontSize',12)')
```

creates a default `CreateFcn` callback that runs whenever you create a new panel. It sets the default font name and font size of the uipanel title.

Note `Uibuttongroup` takes its default property values from `uipanel`. Defining a default property for all uipanel defines the same default property for all `uibuttongroups`.

To override this default and create a panel whose `FontName` and `FontSize` properties are set to different values, call `uipanel` with code similar to

```
hpt = uipanel(..., 'CreateFcn', 'set(gcbo,...  
    ' 'FontName','times','FontSize',14)')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uipushtool` call. In the example above, if instead of redefining the `CreateFcn` property for this uipanel, you had explicitly set `FontSize` to 14, the default `CreateFcn` callback would have set `FontSize` back to the system dependent default.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

DeleteFcn
string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the uipanel object (e.g., when you issue a delete command or clear the figure containing the uipanel). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine. The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.

FontAngle
{normal} | italic | oblique

Character slant used in the Title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to italic or oblique selects a slanted version of the font, when it is available on your system.

FontName
string

Font family used in the Title. The name of the font in which to display the Title. To display and print properly, this must be a font that your system supports. The default font is system dependent. To eliminate the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan), set FontName to the string FixedWidth (this string value is case insensitive).

```
set(uicontrol_handle, 'FontName', 'FixedWidth')
```

This then uses the value of the root FixedWidthFontName property which can be set to the appropriate value for a locale

Uipanel Properties

from `startup.m` in the end user's environment. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font

`FontSize`
integer

Title font size. A number specifying the size of the font in which to display the Title, in units determined by the `FontUnits` property. The default size is system dependent.

`FontUnits`
inches | centimeters | normalized | {points} | pixels

Title font size units. Normalized units interpret `FontSize` as a fraction of the height of the uipanel. When you resize the uipanel, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`FontWeight`
light | {normal} | demi | bold

Weight of characters in the title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `bold` causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor`
`ColorSpec`

Color used for title font and 2-D border line. A three-element RGB vector or one of the MATLAB predefined names, specifying the font or line color. See the `ColorSpec` reference page for more information on specifying color.

`HandleVisibility`
{on} | callback | off

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HighlightColor`
`ColorSpec`

3-D frame highlight color. A three-element RGB vector or one of the MATLAB predefined names, specifying the highlight color. See the `ColorSpec` reference page for more information on specifying color.

Uipanel 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.

Interruptible
{on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object's Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure's WindowButtonDownFcn callback routine, or an object's ButtonDownFcn or Callback routine are processed according to the rules described above.

Parent

handle

Uipanel parent. The handle of the uipanel's parent figure, uipanel, or uibuttongroup. You can move a uipanel object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

Position

position rectangle

Size and location of uipanel relative to parent. The rectangle defined by this property specifies the size and location of the panel within the parent figure window, uipanel, or uibuttongroup. Specify Position as

```
[left bottom width height]
```

left and bottom are the distance from the lower-left corner of the parent object to the lower-left corner of the uipanel object. width and height are the dimensions of the uipanel rectangle, including the title. All measurements are in units specified by the Units property.

ResizeFcn

string or function handle

Uipanel Properties

Resize callback routine. MATLAB executes this callback routine whenever a user resizes the uipanel and the figure `Resize` property is set to `on`, or in GUIDE, the `Resize` behavior option is set to `Other`. You can query the uipanel `Position` property to determine its new size and position. During execution of the callback routine, the handle to the figure being resized is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

You can use `ResizeFcn` to maintain a GUI layout that is not directly supported by the MATLAB `Position/Units` paradigm.

For example, consider a GUI layout that maintains an object at a constant height in pixels and attached to the top of the figure, but always matches the width of the figure. The following `ResizeFcn` accomplishes this; it keeps the uicontrol whose `Tag` is `'StatusBar'` 20 pixels high, as wide as the figure, and attached to the top of the figure. Note the use of the `Tag` property to retrieve the uicontrol handle, and the `gcbo` function to retrieve the figure handle. Also note the defensive programming regarding figure `Units`, which the callback requires to be in pixels in order to work correctly, but which the callback also restores to their previous value afterwards.

```
u = findobj('Tag','StatusBar');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

You can change the figure `Position` from within the `ResizeFcn` callback; however, the `ResizeFcn` is not called again as a result.

Note that the `print` command can cause the `ResizeFcn` to be called if the `PaperPositionMode` property is set to `manual` and you have defined a resize function. If you do not want your resize function called by `print`, set the `PaperPositionMode` to `auto`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See `Resize Behavior` for information on creating resize functions using `GUIDE`.

Selected

on | off (read only)

Is object selected? This property indicates whether the panel is selected. When this property is on, MATLAB displays selection handles if the `SelectionHighlight` property is also on. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

SelectionHighlight

{on} | off

Object highlighted when selected. When the `Selected` property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is off, MATLAB does not draw the handles.

ShadowColor

ColorSpec

3-D frame shadow color. A three-element RGB vector or one of the MATLAB predefined names, specifying the shadow color. See the `ColorSpec` reference page for more information on specifying color.

Tag

string

Uipanel Properties

User-specified object identifier. The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the `findobj` function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the Tag value 'FormatTb'.

```
h = findobj(figurehandles,'Tag','FormatTb')
```

Title

string

Title string. The text displayed in the panel title. You can position the title using the `TitlePosition` property.

If the string value is specified as a cell array of strings or padded string matrix, only the first string of a cell array or of a padded string matrix is displayed; the rest are ignored. Vertical slash ('|') characters are not interpreted as line breaks and instead show up in the text displayed in the uipanel title.

Setting a property value to `default`, `remove`, or `factory` produces the effect described in “Defining Default Values”. To set `Title` to one of these words, you must precede the word with the backslash character. For example,

```
hp = uipanel(...,'Title','\Default');
```

TitlePosition

{lefttop} | centertop | righttop | leftbottom |
centerbottom | rightbottom

Location of the title. This property determines the location of the title string, in relation to the uipanel.

UIContextMenu
handle

Associate a context menu with a uipanel. Assign this property the handle of a Uicontextmenu object. MATLAB displays the context menu whenever you right-click the uipanel. Use the uicontextmenu function to create the context menu.

Units
inches | centimeters | {normalized} | points | pixels
| characters

Units of measurement. MATLAB uses these units to interpret the Position property. For the panel itself, units are measured from the lower-left corner of the figure window. For children of the panel, they are measured from the lower-left corner of the panel.

- Normalized units map the lower-left corner of the panel or figure window to (0,0) and the upper-right corner to (1.0,1.0).
- pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).
- Character units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of Units, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume Units is set to the default value.

UserData
matrix

User-specified data. Any data you want to associate with the uipanel object. MATLAB does not use this data, but you can access it using set and get.

Uipanel Properties

Visible
{on} | off

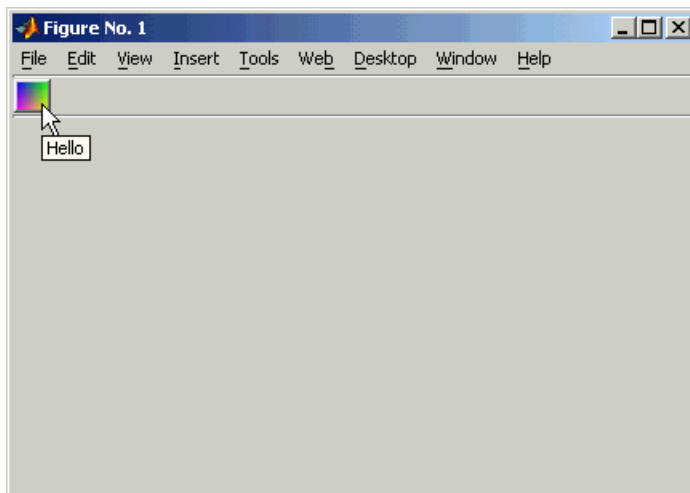
Uipanel visibility. By default, a `uipanel` object is visible. When set to `off`, the `uipanel` is not visible, but still exists and you can query and set its properties.

Note The value of a `uipanel`'s `Visible` property also controls the `Visible` property of child axes, `uipanel`s, and `uibuttongroups`. This property does not affect the `Visible` property of child `uicontrol`s.

Purpose	Create push button on toolbar
Syntax	<pre>hpt = uipushtool('PropertyName1',value1,'PropertyName2', value2,...) hpt = uipushtool(ht,...)</pre>
Description	<p><code>hpt = uipushtool('PropertyName1',value1,'PropertyName2',value2,...)</code> creates a push button on the <code>uitoolbar</code> at the top of the current figure window, and returns a handle to it. <code>uipushtool</code> assigns the specified property values, and assigns default values to the remaining properties. You can change the property values at a later time using the <code>set</code> function.</p> <p>Type <code>get(hpt)</code> to see a list of <code>uipushtool</code> object properties and their current values. Type <code>set(hpt)</code> to see a list of <code>uipushtool</code> object properties that you can set and their legal property values. See the Uipushtool Properties reference page for more information.</p> <p><code>hpt = uipushtool(ht,...)</code> creates a button with <code>ht</code> as a parent. <code>ht</code> must be a <code>uitoolbar</code> handle.</p>
Remarks	<p><code>uipushtool</code> accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.</p> <p><code>Uipushtools</code> appear in figures whose <code>Window Style</code> is <code>normal</code> or <code>docked</code>. They do not appear in figures whose <code>WindowStyle</code> is <code>modal</code>. If a figure containing a <code>uitoolbar</code> and its <code>uipushtool</code> children is changed to <code>WindowStyle modal</code>, the <code>uipushtools</code> still exist and are contained in the <code>Children</code> list of the <code>uitoolbar</code>, but are not displayed until the <code>WindowStyle</code> is changed to <code>normal</code> or <code>docked</code>.</p>
Examples	<p>This example creates a <code>uitoolbar</code> object and places a <code>uipushtool</code> object on it.</p> <pre>h = figure('ToolBar','none') ht = uitoolbar(h) a = [.20:.05:0.95];</pre>

uipushtool

```
b(:,:,1) = repmat(a,16,1)';  
b(:,:,2) = repmat(a,16,1);  
b(:,:,3) = repmat(flipdim(a,2),16,1);  
hpt = uipushtool(ht,'CData',b,'TooltipString','Hello')
```



See Also

[get](#), [set](#), [uicontrol](#), [uitoggletool](#), [uitoolbar](#)

Purpose

Defines the uipushtool properties.

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uipushtool properties by typing:

```
set(h, 'DefaultUipushtoolPropertyName', PropertyValue...)
```

Where `h` can be the root handle (0), a figure handle, a uicontrol handle, or a uipushtool handle. *PropertyName* is the name of the Uipushtool property and *PropertyValue* is the value you specify as the default for that property.

For more information about changing the default value of a property see [Setting Default Property Values](#).

Uipushtool Properties

This section lists all properties useful to uipushtool objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property	Purpose
BeingDeleted	This object is being deleted.
BusyAction	Callback routine interruption.
CData	Truecolor image displayed on the control.
ClickedCallback	Control action.
CreateFcn	Callback routine executed during object creation.
DeleteFcn	Delete uipushtool callback routine.

Uipushtool Properties

Property	Purpose
Enable	Enable or disable the uipushtool.
HandleVisibility	Control access to object's handle.
Interruptible	Callback routine interruption mode.
Parent	Handle of uipushtool's parent.
Separator	Separator line mode
Tag	User-specified object label.
TooltipString	Content of object's tooltip.
Type	Object class.
UserData	User specified data.
Visible	Uipushtool visibility.

BeingDeleted
on | {off} (read only)

This object is being deleted. The **BeingDeleted** property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the **BeingDeleted** property to on when the object's delete function callback is called (see the **DeleteFcn** property). It remains set to on while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's **BeingDeleted** property before acting.

BusyAction
cancel | {queue}

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new

event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`CData`

3-dimensional array

Tricolor image displayed on control. An n -by- m -by-3 array of RGB values that defines a tricolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0. If your `CData` array is larger than 16 in the first or second dimension, it may be clipped or cause other undesirable effects. If the array is clipped, only the center 16-by-16 part of the array is used.

`ClickedCallback`

string or function handle

Control action. A routine that executes when the uipushtool's `Enable` property is set to on, and you press a mouse button while the pointer is on the push tool itself or in a 5-pixel wide border around it.

Uipushtool Properties

CreateFcn

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uipushtool object. MATLAB sets all property values for the uipushtool before executing the CreateFcn callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the push tool being created.

Setting this property on an existing uipushtool object has no effect.

You can define a default CreateFcn callback for all new uipushtools. This default applies unless you override it by specifying a different CreateFcn callback when you call `uipushtool`. For example, the code

```
imga(:,:,1) = rand(20);  
imga(:,:,2) = rand(20);  
imga(:,:,3) = rand(20);  
set(0,'DefaultUipushtoolCreateFcn','set(gcbo,''Cdata'',imga)')
```

creates a default CreateFcn callback that runs whenever you create a new push tool. It sets the default image `imga` on the push tool.

To override this default and create a push tool whose `Cdata` property is set to a different image, call `uipushtool` with code similar to

```
a = [.05:.05:0.95];  
imgb(:,:,1) = repmat(a,19,1)';  
imgb(:,:,2) = repmat(a,19,1);  
imgb(:,:,3) = repmat(flipdim(a,2),19,1);  
hpt = uipushtool(...,'CreateFcn','set(gcbo,''CData'',imgb)',...)
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uipushtool` call. In the example above, if instead of redefining the `CreateFcn` property for this push tool, you had explicitly set `CData` to `imgb`, the default `CreateFcn` callback would have set `CData` back to `imga`.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

`DeleteFcn`
string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the `uipushtool` object (e.g., when you call the `delete` function or cause the figure containing the `uipushtool` to reset). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

`Enable`
{on} | off

Enable or disable the uipushtool. This property controls how `uipushtools` respond to mouse button clicks, including which callback routines execute.

Uipushtool Properties

- on – The uipushtool is operational (the default).
- off – The uipushtool is not operational and its image (set by the Cdata property) is grayed out.

When you left-click on a uipushtool whose Enable property is on, MATLAB performs these actions in this order:

- 1 Sets the figure's SelectionType property.
- 2 Executes the push tool's ClickedCallback routine.
- 3 Does not set the figure's CurrentPoint property and does not execute the figure's WindowButtonDownFcn callback.

When you left-click on a uipushtool whose Enable property is off, or when you right-click a uipushtool whose Enable property has any value, MATLAB performs these actions in this order:

- 4 Sets the figure's SelectionType property.
- 5 Sets the figure's CurrentPoint property.
- 6 Executes the figure's WindowButtonDownFcn callback.
- 7 Does not execute the push tool's ClickedCallback routine.

HandleVisibility

{on} | callback | off

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close. Neither is the handle visible in the parent figure's CurrentObject property. Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when HandleVisibility is on.

- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`Interruptible`
{on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`,

Uipushtool Properties

getframe, pause, or waitfor functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object's Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure's WindowButtonDownFcn callback routine, or an object's ButtonDownFcn or Callback routine are processed according to the rules described above.

Parent
handle

Uipushtool parent. The handle of the uipushtool's parent toolbar. You can move a uipushtool object to another toolbar by setting this property to the handle of the new parent.

Separator
on | {off}

Separator line mode. Setting this property to on draws a dividing line to the left of the uipushtool.

Tag
string

User-specified object identifier. The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the `findobj` function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified toolbars) that have the Tag value 'Copy'.

```
h = findobj(uitoolbarhandles,'Tag','Copy')
```

TooltipString
string

Content of tooltip for object. The TooltipString property specifies the text of the tooltip associated with the uipushtool. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

Type
string (read-only)

Object class. This property identifies the kind of graphics object. For uipushtool objects, Type is always the string 'uipushtool'.

UserData
array

User specified data. You can specify UserData as any array you want to associate with the uipushtool object. The object does not use this data, but you can access it using the set and get functions.

Visible
{on} | off

Uipushtool Properties

Uipushtool visibility. By default, all uipushtools are visible. When set to off, the uipushtool is not visible, but still exists and you can query and set its properties.

Purpose Open standard dialog box for saving files

Syntax

```
uiputfile
uiputfile('FilterSpec')
uiputfile('FilterSpec','DialogTitle')
uiputfile('FilterSpec','DialogTitle','DefaultName')
uiputfile(...,'Location',[x y])
[FileName,PathName] = uiputfile(...)
[FileName,PathName,FilterIndex] = uiputfile(...)
```

Description `uiputfile` displays a dialog box used to select a file for saving. The dialog box lists the files and directories in the current directory.

`uiputfile('FilterSpec')` displays a dialog box that lists files in the current directory. `FilterSpec` determines what files are displayed initially in the dialog box. For example `*.m` lists all MATLAB M-files.

If `FilterSpec` is a string, `uiputfile` appends 'All Files' to the list of file types. If `FilterSpec` is a cell array, the first column is used as the list of extensions, and the second column is used as the list of descriptions. `FilterSpec` can also be a filename. In this case the filename becomes the default filename and the file's extension is used as the default filter. If `FilterSpec` is not specified, `uiputfile` uses the default list of file types (i.e., all MATLAB files).

`uiputfile('FilterSpec','DialogTitle')` displays a dialog box that has the title `DialogTitle`. To use the default file types and specify a dialog title, enter

```
uiputfile('','DialogTitle')
```

`uiputfile('FilterSpec','DialogTitle','DefaultName')` displays a dialog box in which a specified string, in this case 'DefaultName', appears in the **File name** field. 'Default Name' can be a filename or the name of a directory. If it is the name of a directory, you must follow it with a slash (/) or backslash (\) separator.

`uiputfile(...,'Location',[x y])` positions the dialog box at screen position `[x,y]`, where `x` and `y` are the distances in pixel units from

uiputfile

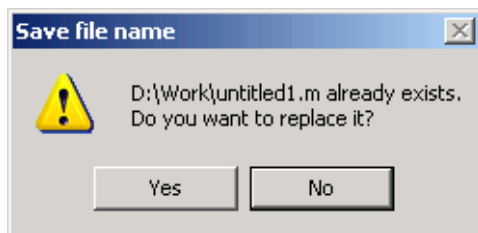
the left and top edges of the screen. This feature is supported only on UNIX platforms.

`[FileName,PathName] = uiputfile(...)` returns the name and path of the file selected in the dialog box. If the user clicks the **Cancel** button or closes the dialog window, `FileName` and `PathName` are set to 0.

`[FileName,PathName,FilterIndex] = uiputfile(...)` returns the index of the filter selected in the dialog box. The indexing starts at 1. If the user clicks the **Cancel** button or closes the dialog window, `FilterIndex` is set to 0.

Remarks

If the user specifies or selects an existing filename, `uiputfile` displays a warning message asking whether the user wants to overwrite the file.



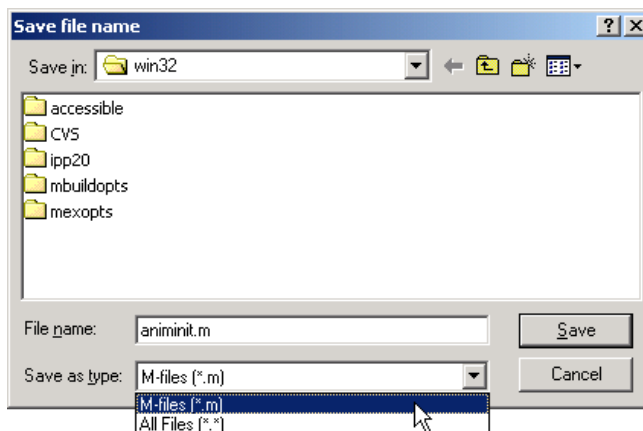
If the user selects **Yes**, `uiputfile` returns the name and path of the file. If the user selects **No**, `FileName` and `PathName` are set to 0.

Examples

Example 1

The following statement displays a dialog box titled 'Save file name' with the **Filename** field set to `animinit.m` and the filter set to M-files (`*.m`). Because `FilterSpec` is a string, the filter also includes All Files (`*.*`)

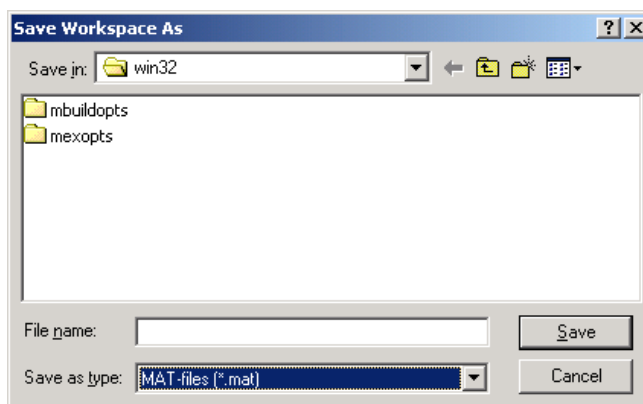
```
[file,path] = uiputfile('animinit.m','Save file name');
```



Example 2

The following statement displays a dialog box titled 'Save Workspace As' with the filter specifier set to MAT-files.

```
[file,path] = uiputfile('*.mat','Save Workspace As');
```

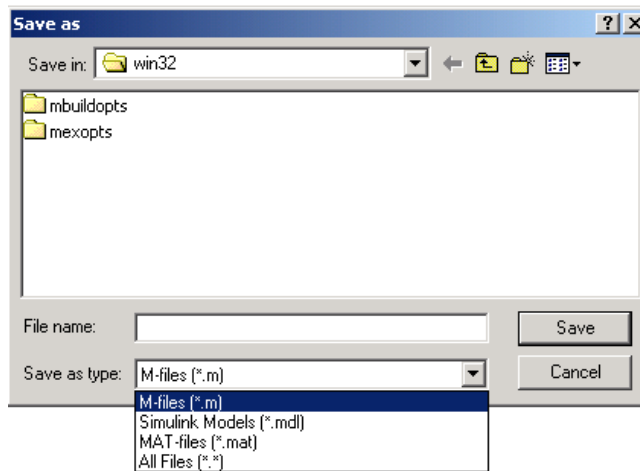


Example 3

To display several file types in the **Save as type** list box, separate each file extension with a semicolon, as in the following code. Note that

uiputfile displays a default description for each known file type, such as Simulink Models for .mdl files.

```
[filename, pathname] = uiputfile( ...  
    {'*.m'; '*.mdl'; '*.mat'; '*.*'}, ...  
    'Save as');
```

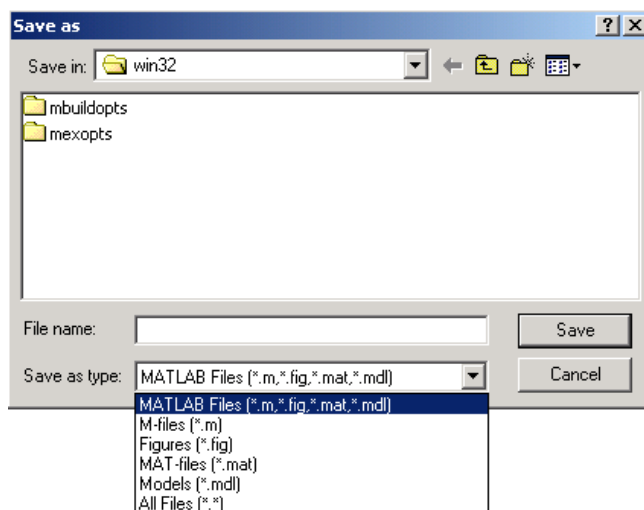


Example 4

If you want to create a list of file types and give them descriptions that are different from the defaults, use a cell array, as in the following code. This example also associates multiple file types with the 'MATLAB Files' description.

```
[filename, pathname, filterindex] = uiputfile( ...  
    {'*.m;*.fig;*.mat;*.mdl', 'MATLAB Files (*.m,*.fig,*.mat,*.mdl)';  
    '*.m', 'M-files (*.m)'; ...  
    '*.fig', 'Figures (*.fig)'; ...  
    '*.mat', 'MAT-files (*.mat)'; ...  
    '*.mdl', 'Models (*.mdl)'; ...  
    '*.*', 'All Files (*.*)'}, ...  
    'Save as');
```


The first column of the cell array contains the file extensions, while the second contains the descriptions you want to provide for the file types. Note that the first entry of column one contains several extensions, separated by semicolons, all of which are associated with the description 'MATLAB Files (*.m;*.fig;*.mat;*.mdl)'. The code produces the dialog box shown in the following figure.



Example 5

The following code checks for the existence of the file and displays a message about the result of the open operation.

```
[filename, pathname] = uiputfile('*.*', 'Pick an M-file');
if isequal(filename,0) | isequal(pathname,0)
    disp('User selected Cancel')
else
    disp(['User selected',fullfile(pathname,filename)])
end
```

See Also

uigetdir, uigetfile

uiresume, uiwait

Purpose Control program execution

Syntax
uiwait
uiwait(h)
uiwait(h,timeout)
uiresume(h)

Description The uiwait and uiresume functions block and resume MATLAB program execution.

uiwait blocks execution until uiresume is called or the current figure is deleted. This syntax is the same as uiwait(gcf).

uiwait(h) blocks execution until uiresume is called or the figure h is deleted.

uiwait(h,timeout) blocks execution until uiresume is called, the figure h is deleted, or timeout seconds elapse.

uiresume(h) resumes the M-file execution that uiwait suspended.

Remarks When creating a dialog, you should have a uicontrol component with a callback that calls uiresume or a callback that destroys the dialog box. These are the only methods that resume program execution after the uiwait function blocks execution.

uiwait is a convenient way to use the waitfor command. You typically use it in conjunction with a dialog box. It provides a way to block the execution of the M-file that created the dialog, until the user responds to the dialog box. When used in conjunction with a modal dialog, uiwait/uiresume can block the execution of the M-file *and* restrict user interaction to the dialog only.

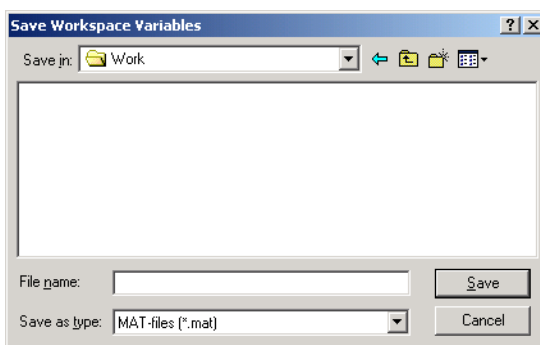
See Also uicontrol, uimenu, waitfor, figure, dialog

Purpose Open standard dialog box for saving workspace variables

Syntax

```
uisave  
uisave(variables)  
uisave(variables,filename)
```

Description `uisave` displays the Save Workspace Variables dialog box for saving workspace variables to a MAT-file, as shown in the figure below.



If you type a name in the **File name** field, such as `my_vars`, and click **Save**, the dialog saves all workspace variables in the `filemy_vars.mat`. The default filename is `matlab.mat`.

`uisave(variables)` saves only the variables listed in the string or cell array variables.

`uisave(variables,filename)` uses the specified filename as the default name in the Save Workspace Variables dialog box.

Note `uisave` cannot be compiled. If you want to create a dialog that can be compiled, use `uiputfile`.

See Also `uigetfile`, `uiputfile`, `uiopen`

uicolor

Purpose Open standard dialog box for setting object's ColorSpec

Syntax

```
c = uicolor
c = uicolor([r g b])
c = uicolor(h)
c = uicolor(..., 'dialogTitle')
```

Description

`c = uicolor` displays a color selection dialog appropriate to the platform, and returns the color selected by the user. The dialog is initialized to white.

`c = uicolor([r g b])` displays a dialog initialized to the specified color, and returns the color selected by the user. `r`, `g`, and `b` must be values between 0 and 1.

`c = uicolor(h)` displays a dialog initialized to the color of the object specified by handle `h`, returns the color selected by the user, and applies it to the object. `h` must be the handle to an object containing a color property.

`c = uicolor(..., 'dialogTitle')` displays a dialog with the specified title.

If the user presses **Cancel** from the dialog box, or if any error occurs, the output value is set to the input RGB triple, if provided; otherwise, it is set to 0.

See Also ColorSpec

Purpose	Open standard dialog box for setting object's font characteristics
Syntax	<pre>uifont uifont(h) uifont(S) uifont(..., 'DialogTitle') S = uifont(...)</pre>
Description	<p><code>uifont</code> enables you to change font properties (<code>FontName</code>, <code>FontUnits</code>, <code>FontSize</code>, <code>FontWeight</code>, and <code>FontAngle</code>) for a <code>text</code>, <code>axes</code>, or <code>uicontrol</code> object. The function returns a structure consisting of font properties and values. You can specify an alternate title for the dialog box.</p> <p><code>uifont</code> displays the dialog box and returns the selected font properties.</p> <p><code>uifont(h)</code> displays a dialog box, initializing the font property values with the values of those properties for the object whose handle is <code>h</code>. Selected font property values are applied to the current object. If a second argument is supplied, it specifies a name for the dialog box.</p> <p><code>uifont(S)</code> displays a dialog box, initializing the font property values with the values defined for the specified structure (<code>S</code>). <code>S</code> must define legal values for one or more of these properties: <code>FontName</code>, <code>FontUnits</code>, <code>FontSize</code>, <code>FontWeight</code>, and <code>FontAngle</code> and the field names must match the property names exactly. If other properties are defined, they are ignored. If a second argument is supplied, it specifies a name for the dialog box.</p> <p><code>uifont(..., 'DialogTitle')</code> displays a dialog box with the title <code>DialogTitle</code> and returns the values of the font properties selected in the dialog box.</p> <p><code>S = uifont(...)</code> returns the properties <code>FontName</code>, <code>FontUnits</code>, <code>FontSize</code>, <code>FontWeight</code>, and <code>FontAngle</code> as fields in a structure. If the user presses Cancel from the dialog box or if an error occurs, the output value is set to 0.</p>

uifont

Example

These statements create a text object, then display a dialog box (labeled Update Font) that enables you to change the font characteristics:

```
h = text(.5,.5,'Figure Annotation');  
uifont(h,'Update Font')
```

These statements create two push buttons, then set the font properties of one based on the values set for the other:

```
% Create push button with string ABC  
c1 = uicontrol('Style','pushbutton', ...  
    'Position',[10 10 100 20], 'String','ABC');  
% Create push button with string XYZ  
c2 = uicontrol('Style','pushbutton', ...  
    'Position',[10 50 100 20], 'String','XYZ');  
% Display set font dialog box for c1, make selections, save to d  
d = uifont(c1);  
% Apply those settings to c2  
set(c2, d)
```

See Also

axes, text, uicontrol

Purpose Manage preferences used in uigetpref

Syntax `uisetpref('clearall')`

Description `uisetpref('clearall')` resets the value of all preferences registered through `uigetpref` to 'ask'. This causes the dialog box to display when you call `uigetpref`.

Note Use `setpref` to set the value of a particular preference to 'ask'.

See Also `setpref`, `uigetpref`

uistack

Purpose Reorder visual stacking order of objects

Syntax

```
uistack(h)
uistack(h,stackopt)
uistack(h,stackopt,step)
```

Description `uistack(h)` raises the visual stacking order of the objects specified by the handles in `h` by one level (step of 1). All handles in `h` must have the same parent.

`uistack(h,stackopt)` moves the objects specified by `h` in the stacking order, where `stackopt` is one of the following:

- 'up' – moves `h` up one position in the stacking order
- 'down' – moves `h` down one position in the stacking order
- 'top' – moves `h` to the top of the current stack
- 'bottom' – moves `h` to the bottom of the current stack

`uistack(h,stackopt,step)` moves the objects specified by `h` up or down the number of levels specified by `step`.

Note In a GUI, axes objects are always at a lower level than `uicontrol` objects. You cannot stack an axes object on top of a `uicontrol` object.

See [Setting the Tab Order Programmatically](#) in the MATLAB documentation for information about changing the tab order.

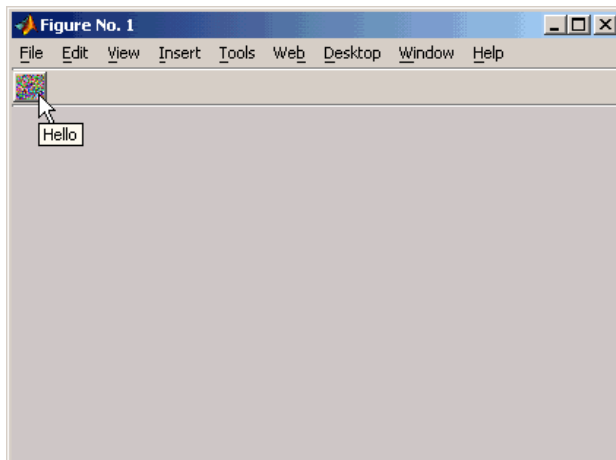
Example The following code moves the child that is third in the stacking order of the figure handle `hObject` down two positions.

```
v = allchild(hObject)
uistack(v(3), 'down', 2)
```


Purpose	Create toggle button on toolbar
Syntax	<pre>htt = uitoggletool('PropertyName1',value1,'PropertyName2', value2,...) htt = uitoggletool(ht,...)</pre>
Description	<p>htt = uitoggletool('PropertyName1',value1,'PropertyName2',value2,...) creates a toggle button on the uitoolbar at the top of the current figure window, and returns a handle to it. uitoggletool assigns the specified property values, and assigns default values to the remaining properties. You can change the property values at a later time using the set function.</p> <p>Type get(htt) to see a list of uitoggletool object properties and their current values. Type set(htt) to see a list of uitoggletool object properties you can set and legal property values. See the Uitoggletool Properties reference page for more information.</p> <p>htt = uitoggletool(ht,...) creates a button with ht as a parent. ht must be a uitoolbar handle.</p>
Remarks	<p>uitoggletool accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.</p> <p>Toggle tools appear in figures whose Window Style is normal or docked. They do not appear in figures whose WindowStyle is modal. If a figure containing a tool bar and its toggle tool children is changed to WindowStyle modal, the toggle tools still exist and are contained in the Children list of the tool bar, but are not displayed until the WindowStyle is changed to normal or docked.</p>
Examples	<p>This example creates a uitoolbar object and places a uitoggletool object on it.</p> <pre>h = figure('ToolBar','none'); ht = uitoolbar(h); a = rand(16,16,3);</pre>

uitoggletool

```
htt = uitoggletool(ht,'CData',a,'TooltipString','Hello');
```



See Also

`get`, `set`, `uicontrol`, `uipushtool`, `uitoolbar`

Purpose

Defines uitoggletool properties.

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uitoggletool properties by typing:

```
set(h, 'DefaultUitoggletoolPropertyName', PropertyValue...)
```

Where `h` can be the root handle (0), a figure handle, a uitoolbar handle, or a uitoggletool handle. *PropertyName* is the name of the Uitoggletool property and *PropertyValue* is the value you specify as the default for that property.

For more information about changing the default value of a property see “Setting Default Property Values” in the MATLAB Graphics documentation.

Properties

This section lists all properties useful to uitoggletool objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property	Purpose
BeingDeleted	This object is being deleted.
BusyAction	Callback routine interruption.
CData	Truecolor image displayed on the toggle tool.
ClickedCallback	Control action independent of the toggle tool position.

Uitoggletool Properties

Property	Purpose
CreateFcn	Callback routine executed during object creation.
DeleteFcn	Callback routine executed during object deletion.
Enable	Enable or disable the uitoggletool.
HandleVisibility	Control access to object's handle.
Interruptible	Callback routine interruption mode.
OffCallback	Control action when toggle tool is set to the off position.
OnCallback	Control action when toggle tool is set to the on position.
Parent	Handle of uitoggletool's parent toolbar.
Separator	Separator line mode.
State	Uitoggletool state.
Tag	User-specified object label.
TooltipString	Content of object's tooltip.
Type	Object class.
UserData	User specified data.
Visible	Uitoggletool visibility.

BeingDeleted
on | {off} (read only)

This object is being deleted. The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the BeingDeleted property to on when the object's delete function callback is called (see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`CData`
3-dimensional array

Tricolor image displayed on control. An n -by- m -by-3 array of RGB values that defines a tricolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0. If your `CData` array is larger than 16 in the first or second dimension, it may be clipped or cause other undesirable

Uitoggletool Properties

effects. If the array is clipped, only the center 16-by-16 part of the array is used.

ClickedCallback
string or function handle

Control action independent of the toggle tool position. A routine that executes after either the `OnCallback` routine or `OffCallback` routine runs to completion. The `uitoggletool`'s `Enable` property must be set to on.

CreateFcn
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uitoggletool` object. MATLAB sets all property values for the `uitoggletool` before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the toggle tool being created.

Setting this property on an existing `uitoggletool` object has no effect.

You can define a default `CreateFcn` callback for all new `uitoggletools`. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uitoggletool`. For example, the statement,

```
set(0, 'DefaultUitoggletoolCreateFcn', 'set(gcbo, 'Enable', ...  
    'off')')
```

creates a default `CreateFcn` callback that runs whenever you create a new toggle tool. It sets the toggle tool `Enable` property to off.

To override this default and create a toggle tool whose Enable property is set to on, you could call `uitoggletool` with code similar to

```
htt = uitoggletool(...,'CreateFcn','set(gcbo,'Enable',...  
                        'on')',...)
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uitoggletool` call. In the example above, if instead of redefining the `CreateFcn` property for this toggle tool, you had explicitly set `Enable` to on, the default `CreateFcn` callback would have set `CData` back to off.

See “Function Handle Callbacks” in the MATLAB Graphics documentation for information on how to use function handles to define a callback function.

`DeleteFcn`
string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the `uitoggletool` object (e.g., when you call the `delete` function or cause the figure containing the `uitoggletool` to reset). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

Uitoggletool Properties

See “Function Handle Callbacks” in the MATLAB Graphics documentation for information on how to use function handles to define a callback function.

Enable

{on} | off

Enable or disable the uitoggletool. This property controls how uitoggletools respond to mouse button clicks, including which callback routines execute.

- on – The uitoggletool is operational (the default).
- off – The uitoggletool is not operational and its image (set by the Cdata property) is grayed out.

When you left-click on a uitoggletool whose Enable property is on, MATLAB performs these actions in this order:

- 1** Sets the figure’s SelectionType property.
- 2** Executes the toggle tool’s ClickedCallback routine.
- 3** Does not set the figure’s CurrentPoint property and does not execute the figure’s WindowButtonDownFcn callback.

When you left-click on a uitoggletool whose Enable property is off, or when you right-click a uitoggletool whose Enable property has any value, MATLAB performs these actions in this order:

- 4** Sets the figure’s SelectionType property.
- 5** Sets the figure’s CurrentPoint property.
- 6** Executes the figure’s WindowButtonDownFcn callback.
- 7** Does not execute the toggle tool’s OnCallback, OffCallback, or ClickedCallback routines.

HandleVisibility

{on} | callback | off

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`Interruptible`
{`on`} | `off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

Uitoggletool Properties

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below).

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement.

`OffCallback`
string or function handle

Control action. A routine that executes if the `uitoggletool`'s `Enable` property is set to `on`, and either

- The toggle tool `State` is set to `off`.

- The toggle tool is set to the off position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The `ClickedCallback` routine, if there is one, runs after the `OffCallback` routine runs to completion.

OnCallback

string or function handle

Control action. A routine that executes if the uitoggletool's `Enable` property is set to on, and either

- The toggle tool `State` is set to on.
- The toggle tool is set to the on position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The `ClickedCallback` routine, if there is one, runs after the `OffCallback` routine runs to completion.

Parent

handle

Uitoggletool parent. The handle of the uitoggletool's parent toolbar. You can move a uitoggletool object to another toolbar by setting this property to the handle of the new parent.

Separator

on | {off}

Separator line mode. Setting this property to on draws a dividing line to left of the uitoggletool.

State

on | {off}

Uitoggletool state. When the state is on, the toggle tool appears in the down, or pressed, position. When the state is off, it appears

Uitoggletool Properties

in the up position. Changing the state causes the appropriate OnCallback or OffCallback routine to run.

Tag

string

User-specified object identifier. The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the findobj function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified toolbars) that have the Tag value 'Bold'.

```
h = findobj(uitoolbarhandles, 'Tag', 'Bold')
```

TooltipString

string

Content of tooltip for object. The TooltipString property specifies the text of the tooltip associated with the uitoggletool. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

Type

string (read-only)

Object class. This property identifies the kind of graphics object. For uitoggletool objects, Type is always the string 'uitoggletool'.

UserData

array

User specified data. You can specify UserData as any array you want to associate with the uitoggletool object. The object does

not use this data, but you can access it using the set and get functions.

Visible

{on} | off

Uitoggletool visibility. By default, all uitoggletools are visible. When set to `off`, the uitoggletool is not visible, but still exists and you can query and set its properties.

uitoolbar

Purpose Create toolbar on figure

Syntax `ht = uitoolbar('PropertyName1',value1,'PropertyName2',value2,
...)
ht = uitoolbar(h,...)`

Description `ht =
uitoolbar('PropertyName1',value1,'PropertyName2',value2,...)`
creates an empty toolbar at the top of the current figure window, and
returns a handle to it. `uitoolbar` assigns the specified property values,
and assigns default values to the remaining properties. You can change
the property values at a later time using the `set` function.

Type `get(ht)` to see a list of `uitoolbar` object properties and their
current values. Type `set(ht)` to see a list of `uitoolbar` object properties
that you can set and legal property values. See the [Uitoolbar Properties](#)
reference page for more information.

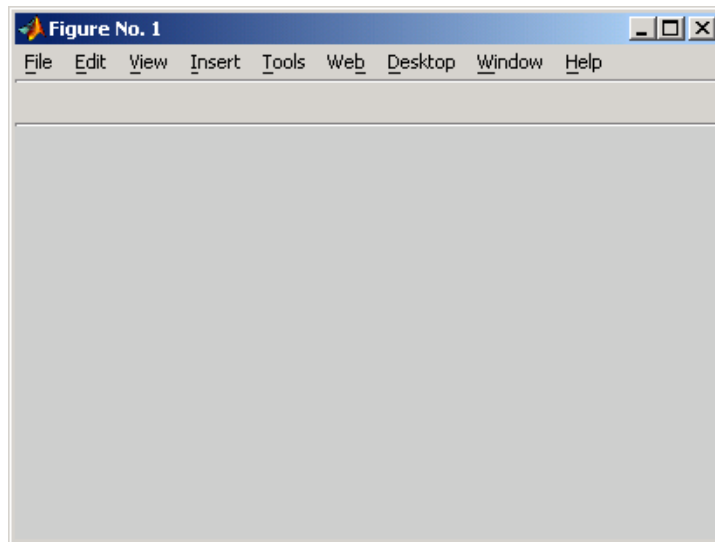
`ht = uitoolbar(h,...)` creates a toolbar with `h` as a parent. `h` must
be a figure handle.

Remarks `uitoolbar` accepts property name/property value pairs, as well as
structures and cell arrays of properties as input arguments.

Uitoolbars appear in figures whose `Window Style` is normal or docked.
They do not appear in figures whose `WindowStyle` is modal. If a figure
containing a `uitoolbar` is changed to `WindowStyle` modal, the `uitoolbar`
still exists and is contained in the `Children` list of the figure, but is not
displayed until the `WindowStyle` is changed to normal or docked.

Example This example creates a figure with no toolbar, then adds a toolbar to it.

```
h = figure('ToolBar','none')  
ht = uitoolbar(h)
```



For more information on using the menus and toolbar in a MATLAB figure window, see the online MATLAB Graphics documentation.

See Also

`set`, `get`, `uicontrol`, `uipushtool`, `uitoggletool`

Uitoolbar Properties

Purpose

Defines the uitoolbar properties.

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The set and get functions enable you to set and query the values of properties.

You can set default Uicontrol properties by typing:

```
set(h, 'DefaultUicontrolPropertyName', PropertyValue...)
```

Where `h` can be the root handle (0), a figure handle, or a uicontrol handle. *PropertyName* is the name of the Uicontrol property and *PropertyValue* is the value you specify as the default for that property.

For more information about changing the default value of a property see “Setting Default Property Values” in the MATLAB Graphics documentation.

Uicontrol Properties

This section lists all properties useful to uicontrol objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property	Purpose
BeingDeleted	This object is being deleted.
BusyAction	Callback routine interruption.
Children	Handles of uicontrol's children.
CreateFcn	Callback routine executed during object creation.

Property	Purpose
DeleteFcn	Callback routine executed during object deletion.
HandleVisibility	Control access to object's handle.
Interruptible	Callback routine interruption mode.
Parent	Handle of uitoolbar's parent.
Tag	User-specified object identifier.
Type	Object class.
UserData	User specified data.
Visible	Uitoolbar visibility.

BeingDeleted

on | {off} (read-only)

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's `BeingDeleted` property before acting.

BusyAction

cancel | {queue}

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

Uitoolbar Properties

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

Children

vector of handles

Handles of tools on the toolbar. A vector containing the handles of all children of the `uitoolbar` object, in the order in which they appear on the toolbar. The children objects of `uitoolbars` are `uipushtools` and `uitoggletools`. You can use this property to reorder the children.

CreateFcn

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uitoolbar` object. MATLAB sets all property values for the `uitoolbar` before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the toolbar being created.

Setting this property on an existing `uitoolbar` object has no effect.

You can define a default `CreateFcn` callback for all new uitoolbars. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uitoolbar`. For example, the statement,

```
set(0,'DefaultUitoolbarCreateFcn','set(gcbo,''Visibility'',...  
    ''off'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new toolbar. It sets the toolbar visibility to `off`.

To override this default and create a toolbar whose `Visibility` property is set to `on`, you could call `uitoolbar` with a call similar to

```
ht = uitoolbar(...,'CreateFcn','set(gcbo,''Visibility'',...  
    ''on'')',...)
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uitoolbar` call. In the example above, if instead of redefining the `CreateFcn` property for this toolbar, you had explicitly set `Visibility` to `on`, the default `CreateFcn` callback would have set `Visibility` back to `off`.

See “Function Handle Callbacks” in the MATLAB Graphics documentation for information on how to use function handles to define a callback function.

`DeleteFcn`
string or function handle

Callback routine executed during object deletion. A callback function that executes when the uitoolbar object is deleted (e.g.,

Uitoolbar Properties

when you call the `delete` function or cause the figure containing the uitoolbar to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

Within the function, use `gcbo` to get the handle of the toolbar being deleted.

`HandleVisibility`
{on} | callback | off

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility`

settings. This does not affect the values of the `HandleVisibility` properties.

`Interruptible`
{on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback.

Uitoolbar Properties

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

Parent

handle

Uitoolbar parent. The handle of the uitoolbar's parent figure. You can move a uitoolbar object to another figure by setting this property to the handle of the new parent.

Tag

string

User-specified object identifier. The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the `findobj` function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the Tag value 'FormatTb'.

```
h = findobj(figurehandles,'Tag','FormatTb')
```

Type

string (read-only)

Object class. This property identifies the kind of graphics object. For uitoolbar objects, Type is always the string 'uitoolbar'.

UserData
array

User specified data. You can specify UserData as any array you want to associate with the uitoolbar object. The object does not use this data, but you can access it using the set and get functions.

Visible
{on} | off

Uitoolbar visibility. By default, all uitoolbars are visible. When set to off, the uitoolbar is not visible, but still exists and you can query and set its properties.

undocheckout

Purpose	Undo previous checkout from source control system (UNIX)
GUI Alternatives	As an alternative to the undocheckout function, select Source Control > Undo Checkout in the File menu of the Editor/Debugger, Simulink, or Stateflow, or in the context menu of the Current Directory browser. For more information, see “Undoing the Checkout”.
Syntax	<pre>undocheckout('filename') undocheckout({'filename1','filename2', ..., 'filenamen'})</pre>
Description	<p><code>undocheckout('filename')</code> makes the file <code>filename</code> available for checkout, where <code>filename</code> does not reflect any of the changes you made after you last checked it out. Use the full pathname for <code>filename</code> and include the file extension.</p> <p><code>undocheckout({'filename1','filename2', ..., 'filenamen'})</code> makes <code>filename1</code> through <code>filenamen</code> available for checkout, where the files do not reflect any of the changes you made after you last checked them out. Use the full pathnames for <code>filenames</code> and include the file extensions.</p>
Examples	<p>Typing</p> <pre>undocheckout({'/myserver/mymfiles/clock.m', ... '/myserver/mymfiles/calendar.m'})</pre> <p>undoes the checkouts of <code>/myserver/mymfiles/clock.m</code> and <code>/myserver/mymfiles/calendar.m</code> from the source control system.</p>
See Also	<p><code>checkin</code>, <code>checkout</code></p> <p>For Windows platforms, use <code>verctrl</code>.</p>

Purpose	Convert Unicode characters to numeric bytes
Syntax	<pre>bytes = unicode2native(unicodestr) bytes = unicode2native(unicodestr, encoding)</pre>
Description	<p><code>bytes = unicode2native(unicodestr)</code> takes a char vector of Unicode characters, <code>unicodestr</code>, converts it to MATLAB's default character encoding scheme, and returns the bytes as a <code>uint8</code> vector, <code>bytes</code>. Output vector <code>bytes</code> has the same general array shape as the <code>unicodestr</code> input. You can save the output of <code>unicode2native</code> to a file using the <code>fwrite</code> function.</p> <p><code>bytes = unicode2native(unicodestr, encoding)</code> converts the Unicode characters to the character encoding scheme specified by the string <code>encoding</code>. <code>encoding</code> must be the empty string (<code>''</code>) or a name or alias for an encoding scheme. Some examples are <code>'UTF-8'</code>, <code>'latin1'</code>, <code>'US-ASCII'</code>, and <code>'Shift_JIS'</code>. For common names and aliases, see the Web site http://www.iana.org/assignments/character-sets. If <code>encoding</code> is unspecified or is the empty string (<code>''</code>), MATLAB's default encoding scheme is used.</p>
Examples	<p>This example begins with two strings containing Unicode characters. It assumes that string <code>str1</code> contains text in a Western European language and string <code>str2</code> contains Japanese text. The example writes both strings into the same file, using the ISO-8859-1 character encoding scheme for the first string and the Shift-JIS encoding scheme for the second string. The example uses <code>unicode2native</code> to convert the two strings to the appropriate encoding schemes.</p> <pre>fid = fopen('mixed.txt', 'w'); bytes1 = unicode2native(str1, 'ISO-8859-1'); fwrite(fid, bytes1, 'uint8'); bytes2 = unicode2native(str2, 'Shift_JIS'); fwrite(fid, bytes2, 'uint8'); fclose(fid);</pre>
See Also	<code>native2unicode</code>

union

Purpose Find set union of two vectors

Syntax

```
c = union(A, B)
c = union(A, B, 'rows')
[c, ia, ib] = union(...)
```

Description `c = union(A, B)` returns the combined values from A and B but with no repetitions. In set theoretic terms, $c = A \cup B$. Inputs A and B can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted in ascending order.

`c = union(A, B, 'rows')` when A and B are matrices with the same number of columns returns the combined rows from A and B with no repetitions.

`[c, ia, ib] = union(...)` also returns index vectors ia and ib such that $c = a(ia) \cup b(ib)$, or for row combinations, $c = a(ia,:) \cup b(ib,:)$. If a value appears in both a and b, union indexes its occurrence in b. If a value appears more than once in b or in a (but not in b), union indexes the last occurrence of the value.

Remarks Because NaN is considered to be not equal to itself, every occurrence of NaN in A or B is also included in the result c.

Examples

```
a = [-1 0 2 4 6];
b = [-1 0 1 3];
[c, ia, ib] = union(a, b);
c =
```

```
    -1     0     1     2     3     4     6
```

```
ia =
```

```
     3     4     5
```

```
ib =
```

1 2 3 4

See Also

`intersect`, `setdiff`, `setxor`, `unique`, `ismember`, `issorted`

unique

Purpose Find unique elements of vector

Syntax

```
b = unique(A)
b = unique(A, 'rows')
[b, m, n] = unique(...)
[b, m, n] = unique(..., occurrence)
```

Description `b = unique(A)` returns the same values as in `A` but with no repetitions. `A` can be a numeric or character array or a cell array of strings. If `A` is a vector or an array, `b` is a vector of unique values from `A`. If `A` is a cell array of strings, `b` is a cell vector of unique strings from `A`. The resulting vector `b` is sorted in ascending order and its elements are of the same class as `A`.

`b = unique(A, 'rows')` returns the unique rows of `A`.

`[b, m, n] = unique(...)` also returns index vectors `m` and `n` such that `b = A(m)` and `A = b(n)`. Each element of `m` is the greatest subscript such that `b = A(m)`. For row combinations, `b = A(m,:)` and `A = b(n,:)`.

`[b, m, n] = unique(..., occurrence)`, where `occurrence` is either `'first'` or `'last'`, returns index vectors `m` and `n` such that

- The elements of vector `m` are the lowest indices of unique elements in `A` when `occurrence` is the string `'first'` and the highest such indices when `occurrence` is `'last'`.
- The elements of vector `n` are the lowest indices of unique elements in `b` when `occurrence` is the string `'first'` and the highest such indices when `occurrence` is `'last'`.

If you do not specify `occurrence`, it defaults to `'last'`.

You can specify `'rows'` in the same command as `'first'` or `'last'`. The order of appearance in the argument list is not important.

Examples

```
A = [1 1 5 6 2 3 3 9 8 6 2 4]
A =
```

```

1 1 5 6 2 3 3 9 8 6 2 4

```

Get a sorted vector of unique elements of A. Also get indices of the first elements in A that make up vector b, and the first elements in b that make up vector A:

```

[b1, m1, n1] = unique(A, 'first')
b1 =
    1     2     3     4     5     6     8     9
m1 =
    1     5     6    12     3     4     9     8
n1 =
    1     1     5     6     2     3     3     8     7     6     2     4

```

Verify that $b1 = A(m1)$ and $A = b1(n1)$:

```

all(b1 == A(m1)) && all(A == b1(n1))
ans =
    1

```

Get a sorted vector of unique elements of A. Also get indices of the last elements in A that make up vector b, and the last elements in b that make up vector A:

```

[b2, m2, n2] = unique(A, 'last')
b2 =
    1     2     3     4     5     6     8     9
m2 =
    2    11     7    12     3    10     9     8
n2 =
    1     1     5     6     2     3     3     8     7     6     2     4

```

Verify that $b2 = A(m2)$ and $A = b2(n2)$:

```

all(b2 == A(m2)) && all(A == b2(n2))
ans =
    1

```

unique

Because NaNs are not equal to each other, `unique` treats them as unique elements.

```
unique([1 1 NaN NaN])  
ans =  
    1 NaN NaN
```

See Also

`intersect`, `ismember`, `issorted`, `setdiff`, `setxor`, `union`

Purpose Execute UNIX command and return result

Syntax

```

unix command
status = unix('command')
[status, result] = unix('command')
[status,result] = unix('command','-echo')
```

Description unix command calls upon the UNIX operating system to execute the given command.

status = unix('command') returns completion status to the status variable.

[status, result] = unix('command') returns the standard output to the result variable, in addition to completion status.

[status,result] = unix('command','-echo') displays the results in the Command Window as it executes, and assigns the results to w.

Note MATLAB uses a shell program to execute the given command. It determines which shell program to use by checking environment variables on your system. MATLAB first checks the MATLAB_SHELL variable, and if either empty or not defined, then checks SHELL. If SHELL is also empty or not defined, MATLAB uses /bin/sh.

Examples List all users that are currently logged in.

```
[s,w] = unix('who');
```

MATLAB returns 0 (success) in s and a string containing the list of users in w.

In this example

```

[s,w] = unix('why')
s =
    1
```

```
w =  
why: Command not found.
```

MATLAB returns a nonzero value in `s` to indicate failure, and returns an error message in `w` because `why` is not a UNIX command.

See Also

`dos`, `!` (exclamation point), `perl`, `system`

“Running External Programs” in the MATLAB Desktop Tools and Development Environment documentation

Purpose Unload external library from memory

Syntax `unloadlibrary('libname')`
`unloadlibrary libname`

Description `unloadlibrary('libname')` unloads the functions defined in shared library `shrlib` from memory. If you need to use these functions again, you must first load them back into memory using `loadlibrary`.

`unloadlibrary libname` is the command format for this function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

Examples Load the MATLAB sample shared library, `shrlibsample`. Call one of its functions, and then unload the library:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h
```

```
s.p1 = 476;    s.p2 = -299;    s.p3 = 1000;
calllib('shrlibsample', 'addStructFields', s)
ans =
    1177
```

```
unloadlibrary shrlibsample
```

See Also `loadlibrary`, `libisloaded`, `libfunctions`, `libfunctionsview`, `libpointer`, `libstruct`, `calllib`

unmkpp

Purpose Piecewise polynomial details

Syntax [breaks,coefs,l,k,d] = unmkpp(pp)

Description [breaks,coefs,l,k,d] = unmkpp(pp) extracts, from the piecewise polynomial pp, its breaks breaks, coefficients coefs, number of pieces l, order k, and dimension d of its target. Create pp using spline or the spline utility mkpp.

Examples This example creates a description of the quadratic polynomial

$$\frac{-x^2}{4} + x$$

as a piecewise polynomial pp, then extracts the details of that description.

```
pp = mkpp([-8 -4],[-1/4 1 0]);  
[breaks,coefs,l,k,d] = unmkpp(pp)
```

```
breaks =  
    -8    -4
```

```
coefs =  
 -0.2500    1.0000    0
```

```
l =  
    1
```

```
k =  
    3
```

```
d =  
    1
```

See Also mkpp, ppval, spline

Purpose Unregister all events for control

Syntax `h.unregisterallevents`
`unregisterallevents(h)`

Description `h.unregisterallevents` unregisters all events that have previously been registered with control, `h`. After calling `unregisterallevents`, the control will no longer respond to any events until you register them again using the `registerevent` function.

`unregisterallevents(h)` is an alternate syntax for the same operation.

Examples Create an `mwsamp` control, registering three events and their respective handler routines. Use the `eventlisteners` function to see the event handler used by each event:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f, ...
    {'Click' 'myclick'; 'DbtClick' 'my2click'; ...
    'MouseDown' 'mymoused'});
```

```
h.eventlisteners
ans =
    'click'          'myclick'
    'dblclick'      'my2click'
    'mousedown'    'mymoused'
```

Unregister all of these events at once with `unregisterallevents`. Now, calling `eventlisteners` returns an empty cell array, indicating that there are no longer any events registered with the control:

```
h.unregisterallevents;
h.eventlisteners
ans =
    {}
```

unregisterallevents

To unregister specific events, use the `unregisterevent` function. First, create the control and register three events:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f, ...
    {'Click' 'myclick'; 'Db1Click' 'my2click'; ...
    'MouseDown' 'mymoused'});
```

Next, unregister two of the three events. The `mousedown` event remains registered:

```
h.unregisterevent({'click' 'myclick'; 'dblclick' 'my2click'});
h.eventlisteners
ans =
    'mousedown'    'mymoused'
```

See Also

`events`, `eventlisteners`, `registerevent`, `unregisterevent`, `isevent`

Purpose Unregister event handler with control's event

Syntax `h.unregisterevent(event_handler)`
`unregisterevent(h, event_handler)`

Description `h.unregisterevent(event_handler)` unregisters certain event handler routines with their corresponding events. Once you unregister an event, the control no longer responds to any further occurrences of the event.

`unregisterevent(h, event_handler)` is an alternate syntax for the same operation.

You can unregister events at any time after a control has been created. Both events and event handlers are specified in the `event_handler` argument, which is a cell array. For example,

```
h.unregisterevent({'event_name', @event_handler});
```

See [Specifying Event Handlers in the External Interfaces documentation](#).

You must specify events in the `event_handler` argument using the names of the events. Unlike the `actxcontrol` and `registerevent` functions, `unregisterevent` does not accept numeric event identifiers.

Examples Create an `mwsamp` control and register all events with the same handler routine, `sampev`. Use the `eventlisteners` function to see the event handler used by each event. In this case, each event, when fired, will call `sampev.m`:

```
f = figure ('position', [100 200 200 200]);  
h = actxcontrol('mwsamp.mwsampctrl1.2', [0 0 200 200], f, ...  
    'sampev');  
  
h.eventlisteners  
ans =  
    'click'          'sampev'
```

unregisterevent

```
'dblclick'      'sampev'  
'mousedown'    'sampev'
```

Unregister just the `dblclick` event. Now, when you list the registered events using `eventlisteners`, you see that `dblclick` is no longer registered. The control will no longer respond when you double-click the mouse over it:

```
h.unregisterevent({'dblclick' 'sampev'});  
h.eventlisteners  
ans =  
  'click'        'sampev'  
  'mousedown'    'sampev'
```

This time, register the `click` and `dblclick` events with a different event handler for `myclick` and `my2click`, respectively:

```
h.unregisterallevents;  
h.registerevent({'click' 'myclick'; 'dblclick' 'my2click'});  
h.eventlisteners  
ans =  
  'click'        'myclick'  
  'dblclick'    'my2click'
```

You can unregister these same events by specifying event names and their handler routines in a cell array. Note that `eventlisteners` now returns an empty cell array, meaning that no events are registered for the `mwsamp` control:

```
h.unregisterevent({'click' 'myclick'; 'dblclick' 'my2click'});  
h.eventlisteners  
ans =  
  {}
```

In this last example, you could have used `unregisterallevents` instead:

```
h.unregisterallevents;
```

See Also

events, eventlisteners, registerevent, unregisterallevents, isevent

untar

Purpose Extract contents of tar file

Syntax

```
untar(tarfilename)
untar(tarfilename,outputdir)
untar(url, ...)
filenames = untar(...)
```

Description `untar(tarfilename)` extracts the archived contents of `tarfilename` into the current directory and sets the files' attributes. It overwrites any existing files with the same names as those in the archive if the existing files' attributes and ownerships permit it. For example, files from rerunning `untar` on the same tar filename do not overwrite any of those files that have a read-only attribute; instead, `untar` issues a warning for such files. On Windows platforms, the hidden, system, and archive attributes are not set.

`tarfilename` is a string specifying the name of the tar file. `tarfilename` is gunzipped to a temporary directory and deleted if its extension ends in `.tgz` or `.gz`. If an extension is omitted, `untar` searches for `tarfilename` appended with `.tgz`, `.tar.gz`, or `.tar` until a file exists. `tarfilename` can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path.

`untar(tarfilename,outputdir)` uncompresses the archive `tarfilename` into the directory `outputdir`. `outputdir` is created if it does not exist.

`untar(url, ...)` extracts the tar archive from an Internet URL. The URL must include the protocol type (e.g., `'http://'` or `'ftp://'`). The URL is downloaded to a temporary directory and deleted.

`filenames = untar(...)` extracts the tar archive and returns the relative pathnames of the extracted files into the string cell array `filenames`.

Examples Copy all `.m` files in the current directory to the directory `backup`:

```
tar('mymfiles.tar.gz','*.m');
untar('mymfiles','backup');
```


Run `untar` to list Cleve Moler's Numerical Computing with MATLAB examples to the output directory `ncm`:

```
url = 'http://www.mathworks.com/moler/ncm.tar.gz';  
ncmFiles = untar(url, 'ncm')
```

See Also

`gzip`, `gunzip`, `tar`, `unzip`, `zip`

unwrap

Purpose Correct phase angles to produce smoother phase plots

Syntax

```
Q = unwrap(P)
Q = unwrap(P,tol)
Q = unwrap(P,[],dim)
Q = unwrap(P,tol,dim)
```

Description `Q = unwrap(P)` corrects the radian phase angles in a vector `P` by adding multiples of $\pm 2\pi$ when absolute jumps between consecutive elements of `P` are greater than or equal to the default jump tolerance of π radians. If `P` is a matrix, `unwrap` operates columnwise. If `P` is a multidimensional array, `unwrap` operates on the first nonsingleton dimension.

`Q = unwrap(P,tol)` uses a jump tolerance `tol` instead of the default value, π .

`Q = unwrap(P,[],dim)` unwraps along `dim` using the default tolerance.

`Q = unwrap(P,tol,dim)` uses a jump tolerance of `tol`.

Note A jump tolerance less than π has the same effect as a tolerance of π . For a tolerance less than π , if a jump is greater than the tolerance but less than π , adding $\pm 2\pi$ would result in a jump larger than the existing one, so `unwrap` chooses the current point. If you want to eliminate jumps that are less than π , try using a finer grid in the domain.

Examples

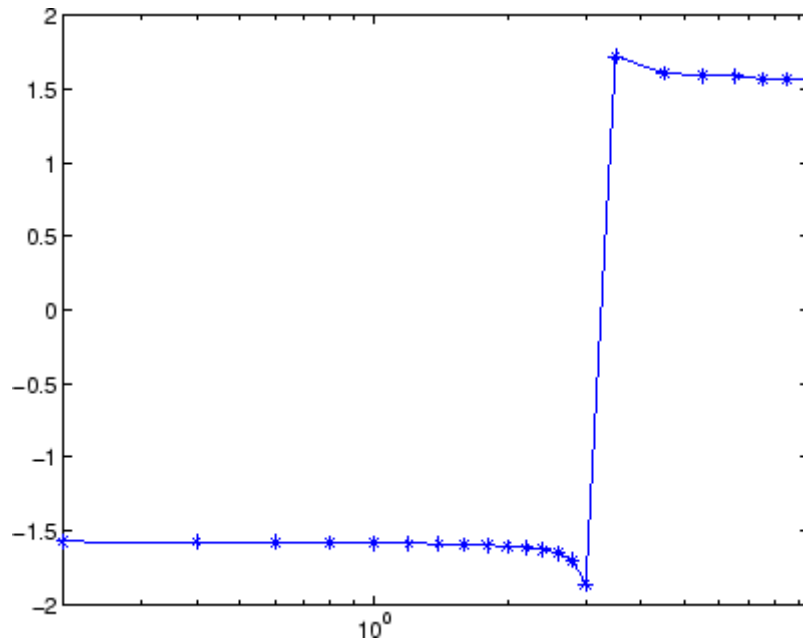
Example 1

The following phase data comes from the frequency response of a third-order transfer function. The phase curve jumps 3.5873 radians between `w = 3.0` and `w = 3.5`, from -1.8621 to 1.7252.

```
w = [0:.2:3,3.5:1:10];
p = [    0
     -1.5728
     -1.5747
     -1.5772
```

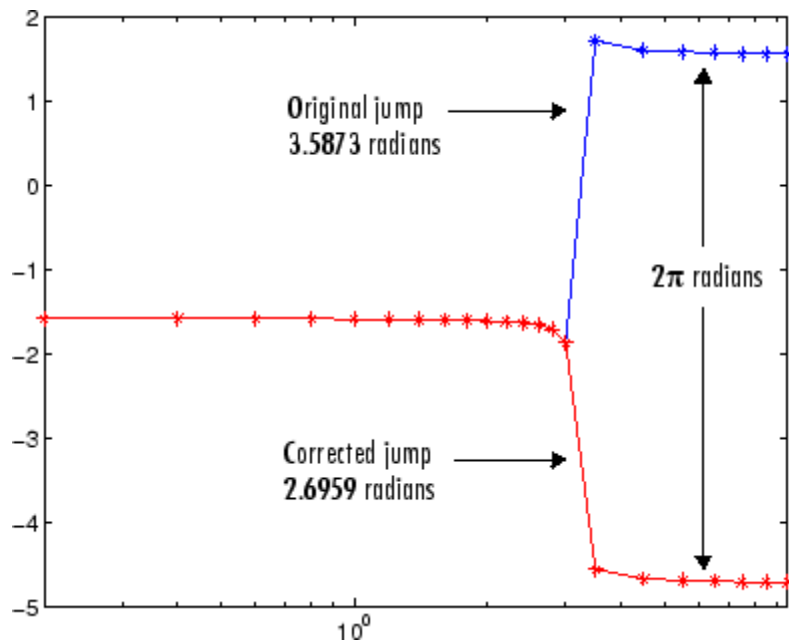
```
-1.5790  
-1.5816  
-1.5852  
-1.5877  
-1.5922  
-1.5976  
-1.6044  
-1.6129  
-1.6269  
-1.6512  
-1.6998  
-1.8621  
1.7252  
1.6124  
1.5930  
1.5916  
1.5708  
1.5708  
1.5708 ];  
semilogx(w,p,'b*-' ), hold
```

unwrap



Using `unwrap` to correct the phase angle, the resulting jump is 2.6959, which is less than the default jump tolerance π . This figure plots the new curve over the original curve.

```
semilogx(w,unwrap(p),'r*-')
```



Note If you have the Control System Toolbox, you can create the data for this example with the following code.

```
h = freqresp(tf(1,[1 .1 10 0]));
p = angle(h(:));
```

Example 2

Array P features smoothly increasing phase angles except for discontinuities at elements (3,1) and (1,2).

```
P = [      0      7.0686      1.5708      2.3562
      0.1963      0.9817      1.7671      2.5525
      6.6759      1.1781      1.9635      2.7489
```

unwrap

0.5890 1.3744 2.1598 2.9452]

The function $Q = \text{unwrap}(P)$ eliminates these discontinuities.

Q =

0	7.0686	1.5708	2.3562
0.1963	7.2649	1.7671	2.5525
0.3927	7.4613	1.9635	2.7489
0.5890	7.6576	2.1598	2.9452

See Also

abs, angle

Purpose Extract contents of zip file

Syntax

```
unzip(zipfilename)
unzip(zipfilename,outputdir)
unzip(url, ...)
filenames = unzip(...)
```

Description `unzip(zipfilename)` extracts the archived contents of `zipfilename` into the current directory and sets the files' attributes. It overwrites any existing files with the same names as those in the archive if the existing files' attributes and ownerships permit it. For example, files from rerunning `unzip` on the same zip filename do not overwrite any of those files that have a read-only attribute; instead, `unzip` issues a warning for such files.

`zipfilename` is a string specifying the name of the zip file. The `.zip` extension is appended to `zipfilename` if omitted. `zipfilename` can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path.

`unzip(zipfilename,outputdir)` extracts the contents of `zipfilename` into the directory `outputdir`.

`unzip(url, ...)` extracts the zipped contents from an Internet URL. The URL must include the protocol type (e.g., `http://`). The URL is downloaded to the temp directory and deleted.

`filenames = unzip(...)` extracts the zip archive and returns the relative pathnames of the extracted files into the string cell array `filenames`.

Examples

Example 1

Copy the demos HTML files to the directory archive:

```
% Zip the demos html files to demos.zip
zip('demos.zip','*.html',fullfile(matlabroot,'demos'))
% Unzip demos.zip to the 'directory' archive
unzip('demos','archive')
```

unzip

Example 2

Run `unzip` to list Cleve Moler's Numerical Computing with MATLAB examples to the output directory `ncm`.

```
url = 'http://www.mathworks.com/moler/ncm.zip';  
ncmFiles = unzip(url, 'ncm')
```

See Also

`gzip`, `gunzip`, `tar`, `untar`, `zip`

Purpose Convert string to uppercase

Syntax `t = upper('str')`
`B = upper(A)`

Description `t = upper('str')` converts any lowercase characters in the string `str` to the corresponding uppercase characters and leaves all other characters unchanged.

`B = upper(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `upper` to each string within `A`.

Examples `upper('attention!')` is ATTENTION!.

Remarks Character sets supported:

- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

See Also `lower`

urlread

Purpose Read content at URL

Syntax

```
s = urlread('url')
s = urlread('url','method','params')
[s,status] = urlread(...)
```

Description

`s = urlread('url')` reads the content at a URL into the string `s`. If the server returns binary data, `s` will be unreadable.

`s = urlread('url','method','params')` reads the content at a URL into the string `s`, passing information to the server as part of the request where *method* can be `get` or `post`, and *params* is a cell array of parameter name/parameter value pairs.

`[s,status] = urlread(...)` catches any errors and returns the error code.

Note If you need to specify a proxy server to connect to the Internet, select **File -> Preferences -> Web** and enter your proxy server address and port. Use this feature if you have a firewall.

Examples

Download Content from Web Page

Use `urlread` to download the contents of the Authors list at the MATLAB Central File Exchange:

```
urlstring = sprintf('%s%s', ...
    'http://www.mathworks.com/matlabcentral/', ...
    'fileexchange/loadAuthorIndex.do');

s = urlread(urlstring);
```

Download Content from File on FTP Server

```
s =
urlread('ftp://nic.merit.edu/documents/rfc/rfc0959.txt');
```

```
s(1:1730)
```

A portion of the file `rfc0959.txt` displays in the MATLAB Command Window.

Download Content from Local File

```
s = urlread('file:///c:/winnt/matlab.ini')
```

See Also

`urlwrite`

`tcpip` if the Instrument Control Toolbox is installed

urlwrite

Purpose Save contents of URL to file

Syntax

```
urlwrite('url','filename')  
f = urlwrite('url','filename')  
f = urlwrite('url','method','params')  
[f,status] = urlwrite(...)
```

Description `urlwrite('url','filename')` reads the contents of the specified URL, saving the contents to `filename`. If you do not specify the path for `filename`, the file is saved in the MATLAB current directory.

`f = urlwrite('url','filename')` reads the contents of the specified URL, saving the contents to `filename` and assigning `filename` to `f`.

`f = urlwrite('url','method','params')` saves the contents of the specified URL to `filename`, passing information to the server as part of the request where `method` can be `get` or `post`, and `params` is a cell array of parameter name/parameter value pairs.

`[f,status] = urlwrite(...)` catches any errors and returns the error code.

Note If you need to specify a proxy server to connect to the Internet, select **File -> Preferences -> Web** and enter your proxy server address and port. Use this feature if you have a firewall.

Examples Download the files submitted to the MATLAB Central File Exchange, saving the results to `samples.html` in the MATLAB current directory.

```
urlwrite('http://www.mathworks.com/matlabcentral/fileexchange  
/Category.jsp?type=category&id=1','samples.html');
```

View the file in the Help browser.

```
open('samples.html')
```

See Also

urlread

Purpose Determine whether Java feature is supported in MATLAB

Syntax `usejava(feature)`

Description `usejava(feature)` returns 1 if the specified feature is supported and 0 otherwise. Possible feature arguments are shown in the following table.

Feature	Description
'awt'	Abstract Window Toolkit components ¹ are available
'desktop'	The MATLAB interactive desktop is running
'jvm'	The Java Virtual Machine is running
'swing'	Swing components ² are available

1. Java's GUI components in the Abstract Window Toolkit
2. Java's lightweight GUI components in the Java Foundation Classes

Examples

The following conditional code ensures that the AWT's GUI components are available before the M-file attempts to display a Java Frame.

```
if usejava('awt')
    myFrame = java.awt.Frame;
else
    disp('Unable to open a Java Frame');
end
```

The next example is part of an M-file that includes Java code. It fails gracefully when run in a MATLAB session that does not have access to a JVM.

```
if ~usejava('jvm')
    error(['filename ' requires Java to run.']);
end
```

See Also

javachk

vander

Purpose Vandermonde matrix

Syntax `A = vander(v)`

Description `A = vander(v)` returns the Vandermonde matrix whose columns are powers of the vector `v`, that is, $A(i, j) = v(i)^{(n-j)}$, where $n = \text{length}(v)$.

Examples `vander(1:.5:3)`

`ans =`

1.0000	1.0000	1.0000	1.0000	1.0000
5.0625	3.3750	2.2500	1.5000	1.0000
16.0000	8.0000	4.0000	2.0000	1.0000
39.0625	15.6250	6.2500	2.5000	1.0000
81.0000	27.0000	9.0000	3.0000	1.0000

See Also `gallery`

Purpose

Variance

Syntax

```
V = var(X)
V = var(X,1)
V = var(X,w)
V = var(X,w,dim)
```

Description

`V = var(X)` returns the variance of `X` for vectors. For matrices, `var(X)` is a row vector containing the variance of each column of `X`. For `N`-dimensional arrays, `var` operates along the first nonsingleton dimension of `X`. The result `V` is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples.

`var` normalizes `V` by `N-1` if `N>1`, where `N` is the sample size. This is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples. For `N=1`, `V` is normalized by `N`.

`V = var(X,1)` normalizes by `N` and produces the second moment of the sample about its mean. `var(X,0)` is equivalent to `var(X)`.

`V = var(X,w)` computes the variance using the weight vector `w`. The length of `w` must equal the length of the dimension over which `var` operates, and its elements must be nonnegative. The elements of `w` must be positive. `var` normalizes `w` to sum of 1.

`V = var(X,w,dim)` takes the variance along the dimension `dim` of `X`. Pass in 0 for `w` to use the default normalization by `N-1`, or 1 to use `N`.

The variance is the square of the standard deviation (STD).

See Also

`corrcoef`, `cov`, `mean`, `median`, `std`

var (timeseries)

Purpose Variance of timeseries data

Syntax
`ts_var = var(ts)`
`ts_var = var(ts, 'PropertyName1', PropertyValue1, ...)`

Description `ts_var = var(ts)` returns the variance of `ts.data`. When `ts.Data` is a vector, `ts_var` is the variance of `ts.Data` values. When `ts.Data` is a matrix, `ts_var` is a row vector containing the variance 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, `var` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_var = var(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 an integer vector, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- 'Weighting' property has two possible values, 'none' (default) or 'time'. When you specify 'time', larger time values correspond to larger weights.

Examples The following example shows how to calculate the variance values of a multi-variate timeseries object.

1 Load a 24-by-3 data array.

```
load count.dat
```

2 Create a timeseries object with 24 time values.

```
count_ts = timeseries(count,[1:24], 'Name', 'CountPerSecond')
```

- 3** Calculate the variance of each data column for this timeseries object.

```
var(count_ts)
ans =
    1.0e+003 *
    0.6437    1.7144    4.6278
```

The variance is calculated independently for each data column in the timeseries object.

See Also

```
iqr (timeseries), mean (timeseries), median (timeseries), std
(timeseries), timeseries
```

varargin

Purpose Variable length input argument list

Syntax `function y = bar(varargin)`

Description `function y = bar(varargin)` accepts a variable number of arguments into function `bar.m`.

The `varargin` statement is used only inside a function M-file to contain optional input arguments passed to the function. The `varargin` argument must be declared as the last input argument to a function, collecting all the inputs from that point onwards. In the declaration, `varargin` must be lowercase.

Examples The function

```
function myplot(x,varargin)
    plot(x,varargin{:})
```

collects all the inputs starting with the second input into the variable `varargin`. `myplot` uses the comma-separated list syntax `varargin{:}` to pass the optional parameters to `plot`. The call

```
myplot(sin(0:.1:1),'color',[.5 .7 .3],'linestyle',':')
```

results in `varargin` being a 1-by-4 cell array containing the values `'color'`, `[.5 .7 .3]`, `'linestyle'`, and `'.'`.

See Also `varargout`, `nargin`, `nargout`, `nargchk`, `nargoutchk`, `inputname`

Purpose Variable length output argument list

Syntax `function varargout = foo(n)`

Description `function varargout = foo(n)` returns a variable number of arguments from function `foo.m`.

The `varargout` statement is used only inside a function M-file to contain the optional output arguments returned by the function. The `varargout` argument must be declared as the last output argument to a function, collecting all the outputs from that point onwards. In the declaration, `varargout` must be lowercase.

Examples The function

```
function [s,varargout] = mysize(x)
nout = max(nargout,1)-1;
s = size(x);
for k=1:nout, varargout(k) = {s(k)}; end
```

returns the size vector and, optionally, individual sizes. So

```
[s,rows,cols] = mysize(rand(4,5));
```

returns `s = [4 5]`, `rows = 4`, `cols = 5`.

See Also `varargin`, `nargin`, `nargout`, `nargchk`, `nargoutchk`, `inputname`

vectorize

Purpose	Vectorize expression
Syntax	<code>vectorize(s)</code> <code>vectorize(fun)</code>
Description	<p><code>vectorize(s)</code> where <code>s</code> is a string expression, inserts a <code>.</code> before any <code>^</code>, <code>*</code> or <code>/</code> in <code>s</code>. The result is a character string.</p> <p><code>vectorize(fun)</code> when <code>fun</code> is an inline function object, vectorizes the formula for <code>fun</code>. The result is the vectorized version of the inline function.</p>
See Also	<code>inline</code> , <code>cd</code> , <code>dbtype</code> , <code>delete</code> , <code>dir</code> , <code>partialpath</code> , <code>path</code> , <code>what</code> , <code>who</code>

Purpose	Version information for MathWorks products
Graphical Interface	As an alternative to the <code>ver</code> function, select About from the Help menu in any product that has a Help menu.
Syntax	<pre>ver ver product v = ver('product')</pre>
Description	<p><code>ver</code> displays a header containing the current version number, license number, operating system, and Java VM version for MATLAB, followed by the version numbers for Simulink, if installed, and all other MathWorks products installed.</p> <p><code>ver product</code> displays the MATLAB header information followed by the current version number for product. The name <code>product</code> corresponds to the directory name that holds the <code>Contents.m</code> file for that product. For example, <code>Contents.m</code> for the Control System Toolbox resides in the <code>control</code> directory. You therefore use <code>ver control</code> to obtain the version of this toolbox.</p> <p><code>v = ver('product')</code> returns the version information to structure array, <code>v</code>, having fields <code>Name</code>, <code>Version</code>, <code>Release</code>, and <code>Date</code>.</p>
Remarks	<p>To use <code>ver</code> with your own product, the first two lines of the <code>Contents.m</code> file for the product must be of the form</p> <pre> % Toolbox Description % Version xxx dd-mmm-yyyy</pre> <p>Do not include any spaces in the date and use a two-character day; that is, use <code>02-Sep-2002</code> instead of <code>2-Sep-2002</code>.</p>
Examples	<p>Return version information for the Control System Toolbox by typing</p> <pre>ver control</pre> <p>MATLAB returns</p>

```
-----  
-  
MATLAB Version 7.1.0.121 (R14) Service Pack 3  
MATLAB License Number: %$#)^(%)$^  
Operating System: Microsoft Windows XP Version 5.1 (Build 2600:  
Service Pack 2)  
Java VM Version: Java 1.5.0 with Sun Microsystems Inc. Java  
HotSpot(TM) Client VM mixed mode  
-----  
-  
Control System Toolbox                Version 6.2.1      (R14SP3 TC1)
```

Return version information for the Control System Toolbox in a structure array, `v`.

```
v = ver('control')  
v =  
    Name: 'Control System Toolbox'  
  Version: '6.2.1'  
  Release: 'R14SP3 TC1'  
    Date: '24-Sep-2005'
```

Display version information on MathWorks 'Real-Time' products:

```
v = ver;  
for k=1:length(v)  
    if strfind(v(k).Name, 'Real-Time')  
        disp(sprintf('%s, Version %s', ...  
                    v(k).Name, v(k).Version))  
    end  
end  
  
Real-Time Windows Target, Version 2.6.1  
Real-Time Workshop, Version 6.4  
Real-Time Workshop Embedded Coder, Version 4.4
```

See Also

`help`, `hostid`, `license`, `version`, `whatsnew`

Help > Check for Updates in the MATLAB desktop.

verctrl

Purpose Source control actions (Windows)

GUI Alternatives As an alternative to the `verctrl` function, use **Source Control** in the **File** menu of the Editor/Debugger, Simulink, or Stateflow, or in the context menu of the Current Directory browser.

Syntax

```
verctrl('action',{'filename1','filename2',...},0)
result=verctrl('action',{'filename1','filename2',...},0)
verctrl('action','filename',0)
result=verctrl('isdiff','filename',0)
list = verctrl('all_systems')
```

Description `verctrl('action',{'filename1','filename2',...},0)` performs the source control operation specified by `'action'` for a single file or multiple files. Enter one file as a string; specify multiple files using a cell array of strings. Use the full paths for each filename and include the extensions. Specify 0 as the last argument. Complete the resulting dialog box to execute the operation; for details about the dialog boxes, see the topic Source Control Interface on Windows Platforms in the MATLAB Desktop Tools and Development Environment documentation. Available values for `'action'` are as follows:

action Argument	Purpose
'add'	Adds files to the source control system. Files can be open in the Editor/Debugger or closed when added.
'checkin'	Checks files into the source control system, storing the changes and creating a new version.
'checkout'	Retrieves files for editing.
'get'	Retrieves files for viewing and compiling, but not editing. When you open the files, they are labeled as read-only.
'history'	Displays the history of files.

action Argument	Purpose
'remove'	Removes files from the source control system. It does not delete the files from disk, but only from the source control system.
'runsc'	Starts the source control system. The filename can be an empty string.
'uncheckout'	Cancels a previous checkout operation and restores the contents of the selected files to the precheckout version. All changes made to the files since the checkout are lost.

`result=verctrl('action',{ 'filename1', 'filename2', ... },0)` performs the source control operation specified by 'action' on a single file or multiple files. The action can be any one of: 'add', 'checkin', 'checkout', 'get', 'history', or 'undocheckout'. result is a logical 1 (true) when you complete the operation by clicking **OK** in the resulting dialog box, and is a logical 0 (false) when you abort the operation by clicking **Cancel** in the resulting dialog box.

`verctrl('action', 'filename', 0)` performs the source control operation specified by 'action' for a single file. Use the full pathname for 'filename'. Specify 0 as the last argument. Complete any resulting dialog boxes to execute the operation. Available values for 'action' are as follows:

action Argument	Purpose
'showdiff'	Displays the differences between a file and the latest checked in version of the file in the source control system.
'properties'	Displays the properties of a file.

`result=verctrl('isdiff','filename',0)` compares `filename` with the latest checked in version of the file in the source control system. `result` is a logical 1 (true) when the files are different, and is a logical 0 (false) when the files are identical. Use the full path for `'filename'`. Specify 0 as the last argument.

`list = verctrl('all_systems')` displays in the Command Window a list of all source control systems installed on your computer.

Examples

Check In a File

Check in `D:\file1.ext` to the source control system.

```
result = verctrl('checkin','D:\file1.ext', 0)
```

This opens the **Check in file(s)** dialog box. Click **OK** to complete the check in. MATLAB displays `result = 1`, indicating the checkin was successful.

Add Files to the Source Control System

Add `D:\file1.ext` and `D:\file2.ext` to the source control system.

```
verctrl('add',{'D:\file1.ext','D:\file2.ext'}, 0)
```

This opens the **Add to source control** dialog box. Click **OK** to complete the operation.

Display the Properties of a File

Display the properties of `D:\file1.ext`.

```
verctrl('properties','D:\file1.ext', 0)
```

This opens the source control properties dialog box for your source control system. The function is complete when you close the properties dialog box.

Show Differences for a File

To show the differences between the version of `file1.ext` that you just edited and saved, with the last version in source control, run

```
verctrl('showdiff','D:\file1.ext',0)
```

MATLAB displays differences dialog boxes and results specific to your source control system. After checking in the file, if you run this statement again, MATLAB displays

```
??? The file is identical to latest version under source control.
```

List All Installed Source Control Systems

To view all of the source control systems installed on your computer, type

```
list = verctrl ('all_systems')
```

MATLAB displays all the source control systems currently installed on your computer. For example:

```
list =  
'Microsoft Visual SourceSafe'  
'ComponentSoftware RCS'
```

See Also

`checkin`, `checkout`, `undocheckout`, `cmopts`

Source Control Interface on Windows Platforms topic in MATLAB Desktop Tools and Development Environment documentation

version

Purpose Version number for MATLAB

Graphical Interface As an alternative to the version function, select **About** from the **Help** menu in the MATLAB desktop.

Syntax

```
version
v = version
[v d] = version
version option
v = version('option')
```

Description `version` displays the MATLAB version number.

`v = version` returns the MATLAB version number in `v`.

`[v d] = version` also returns a string `d` containing the date of the version.

`version option` displays the following additional information about the version.

Option	Description
-date	Release date
-description	Release description. Mostly used for Service Pack releases.
-java	Java VM (JVM) version used by MATLAB
-release	Release number

`v = version('option')` returns additional information about the version. Valid string values for `option` are listed in the table above. You can only specify one output when using this syntax.

Remarks On Windows and UNIX platforms, MATLAB includes a JVM and uses that version. If you use the MATLAB Java interface and the Java classes you want to use require a different JVM than the version provided with MATLAB, it is possible to run MATLAB with a different

JVM. For details, see Solution 1-1812J on the MathWorks Support Web site.

On the Macintosh platform, MATLAB does not include a JVM, but uses whatever JVM is currently running on the machine.

Examples

```
[v d] = version
v =
    7.2.0.321 (R2006a)

d =
    February 14, 2005
```

Run the following command in MATLAB R14 Service Pack 3:

```
['Release R' version('-release') ', ' ...
    version('-description')]

ans =
    Release R14, Service Pack 3
```

See Also

ver, whatsnew

Help > Check for Updates in the MATLAB desktop.

vertcat

Purpose Concatenate arrays vertically

Syntax `C = vertcat(A1, A2, ...)`

Description `C = vertcat(A1, A2, ...)` vertically concatenates matrices A1, A2, and so on. All matrices in the argument list must have the same number of columns.

`vertcat` concatenates N-dimensional arrays along the first dimension. The remaining dimensions must match.

MATLAB calls `C = vertcat(A1, A2, ...)` for the syntax `C = [A1; A2; ...]` when any of A1, A2, etc. is an object.

Examples Create a 5-by-3 matrix, A, and a 3-by-3 matrix, B. Then vertically concatenate A and B.

```
A = magic(5);           % Create 5-by-3 matrix, A
A(:, 4:5) = []
```

```
A =
    17    24     1
    23     5     7
     4     6    13
    10    12    19
    11    18    25
```

```
B = magic(3)*100       % Create 3-by-3 matrix, B
```

```
B =
    800    100    600
    300    500    700
    400    900    200
```



```
C = vertcat(A,B)           % Vertically concatenate A and B
```

```
C =
```

```
    17    24     1  
    23     5     7  
     4     6    13  
    10    12    19  
    11    18    25  
   800   100   600  
   300   500   700  
   400   900   200
```

See Also

horzcat, cat

vertcat (timeseries)

Purpose Vertical concatenation of `timeseries` objects

Syntax `ts = vertcat(ts1,ts2,...)`

Description `ts = vertcat(ts1,ts2,...)` performs

```
ts = [ts1;ts2;...]
```

This operation appends `timeseries` objects. The time vectors must not overlap. The last time in `ts1` must be earlier than the first time in `ts2`. The data sample size of the `timeseries` objects must agree.

See Also `timeseries`

Purpose Vertical concatenation for tscollection objects

Syntax `tsc = vertcat(tsc1,tsc2,...)`

Description `tsc = vertcat(tsc1,tsc2,...)` performs
`tsc = [tsc1;tsc2;...]`

This operation appends tscollection objects. The time vectors must not overlap. The last time in tsc1 must be earlier than the first time in tsc2. All tscollection objects to be concatenated must have the same timeseries members.

See Also `horzcat (tscollection)`, `tscollection`

view

Purpose Viewpoint specification

Syntax

```
view(az,e1)
view([x,y,z])
view(2)
view(3)
view(T)
[az,e1] = view
T = view
```

Description The position of the viewer (the viewpoint) determines the orientation of the axes. You specify the viewpoint in terms of azimuth and elevation, or by a point in three-dimensional space.

`view(az,e1)` and `view([az,e1])` set the viewing angle for a three-dimensional plot. The azimuth, `az`, is the horizontal rotation about the z -axis as measured in degrees from the negative y -axis. Positive values indicate counterclockwise rotation of the viewpoint. `e1` is the vertical elevation of the viewpoint in degrees. Positive values of elevation correspond to moving above the object; negative values correspond to moving below the object.

`view([x,y,z])` sets the viewpoint to the Cartesian coordinates x , y , and z . The magnitude of (x,y,z) is ignored.

`view(2)` sets the default two-dimensional view, `az = 0`, `e1 = 90`.

`view(3)` sets the default three-dimensional view, `az = 37.5`, `e1 = 30`.

`view(T)` sets the view according to the transformation matrix T , which is a 4-by-4 matrix such as a perspective transformation generated by `viewmtx`.

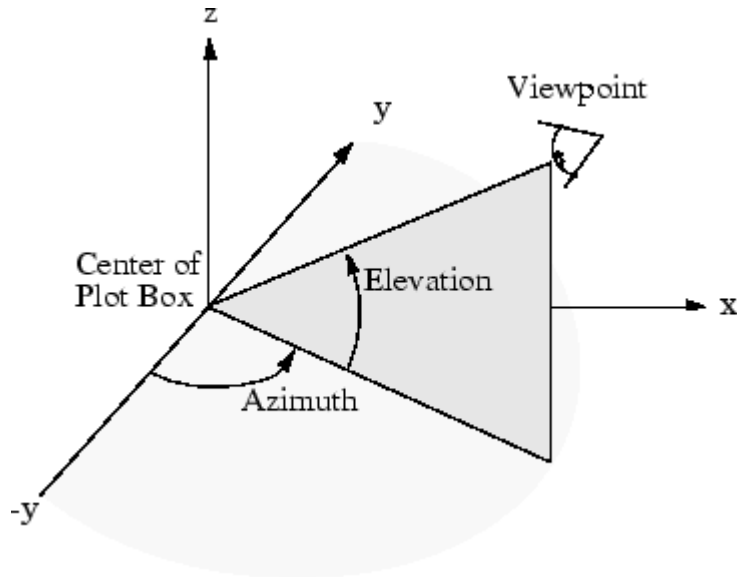
`[az,e1] = view` returns the current azimuth and elevation.

`T = view` returns the current 4-by-4 transformation matrix.

Remarks

Azimuth is a polar angle in the x - y plane, with positive angles indicating counterclockwise rotation of the viewpoint. Elevation is the angle above (positive angle) or below (negative angle) the x - y plane.

This diagram illustrates the coordinate system. The arrows indicate positive directions.

**Examples**

View the object from directly overhead.

```
az = 0;
el = 90;
view(az, el);
```

Set the view along the y -axis, with the x -axis extending horizontally and the z -axis extending vertically in the figure.

```
view([0 0]);
```

Rotate the view about the z -axis by 180° .

view

```
az = 180;  
el = 90;  
view(az, el);
```

See Also

[viewmtx](#), [hgtransform](#), [rotate3d](#)

“Controlling the Camera Viewpoint” on page 1-96 for related functions

Axes graphics object properties [CameraPosition](#), [CameraTarget](#),
[CameraViewAngle](#), [Projection](#)

[Defining the View](#) for more information on viewing concepts and techniques

[Transforming Objects](#) for information on moving and scaling objects in groups

Purpose View transformation matrices

Syntax

```
viewmtx
T = viewmtx(az,e1)
T = viewmtx(az,e1,phi)
T = viewmtx(az,e1,phi,xc)
```

Description `viewmtx` computes a 4-by-4 orthographic or perspective transformation matrix that projects four-dimensional homogeneous vectors onto a two-dimensional view surface (e.g., your computer screen).

`T = viewmtx(az,e1)` returns an *orthographic* transformation matrix corresponding to azimuth `az` and elevation `e1`. `az` is the azimuth (i.e., horizontal rotation) of the viewpoint in degrees. `e1` is the elevation of the viewpoint in degrees. This returns the same matrix as the commands

```
view(az,e1)
T = view
```

but does not change the current view.

`T = viewmtx(az,e1,phi)` returns a *perspective* transformation matrix. `phi` is the perspective viewing angle in degrees. `phi` is the subtended view angle of the normalized plot cube (in degrees) and controls the amount of perspective distortion.

Phi	Description
0 degrees	Orthographic projection
10 degrees	Similar to telephoto lens
25 degrees	Similar to normal lens
60 degrees	Similar to wide-angle lens

You can use the matrix returned to set the view transformation with `view(T)`. The 4-by-4 perspective transformation matrix transforms four-dimensional homogeneous vectors into unnormalized vectors of the

form (x,y,z,w) , where w is not equal to 1. The x - and y -components of the normalized vector $(x/w, y/w, z/w, 1)$ are the desired two-dimensional components (see example below).

`T = viewmtx(az,e1,phi,xc)` returns the perspective transformation matrix using `xc` as the target point within the normalized plot cube (i.e., the camera is looking at the point `xc`). `xc` is the target point that is the center of the view. You specify the point as a three-element vector, `xc = [xc,yc,zc]`, in the interval $[0,1]$. The default value is `xc = [0,0,0]`.

Remarks

A four-dimensional homogenous vector is formed by appending a 1 to the corresponding three-dimensional vector. For example, $[x, y, z, 1]$ is the four-dimensional vector corresponding to the three-dimensional point $[x, y, z]$.

Examples

Determine the projected two-dimensional vector corresponding to the three-dimensional point $(0.5,0.0,-3.0)$ using the default view direction. Note that the point is a column vector.

```
A = viewmtx(-37.5,30);
x4d = [.5 0 -3 1]';
x2d = A*x4d;
x2d = x2d(1:2)
x2d =
    0.3967
   -2.4459
```

Vectors that trace the edges of a unit cube are

```
x = [0 1 1 0 0 0 1 1 0 0 1 1 1 1 0 0];
y = [0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1];
z = [0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0];
```

Transform the points in these vectors to the screen, then plot the object.

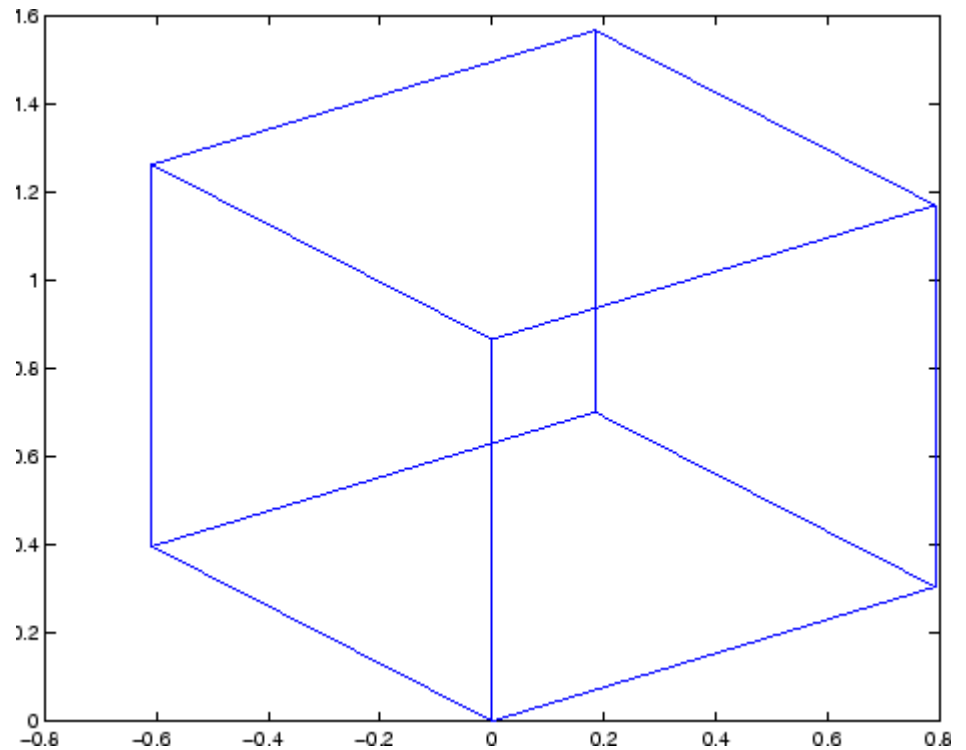
```
A = viewmtx(-37.5,30);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
```



```

x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:);
y2(:) = x2d(2,:);
plot(x2,y2)

```



Use a perspective transformation with a 25 degree viewing angle:

```

A = viewmtx(-37.5,30,25);
x4d = [.5 0 -3 1]';
x2d = A*x4d;
x2d = x2d(1:2)/x2d(4) % Normalize
x2d =

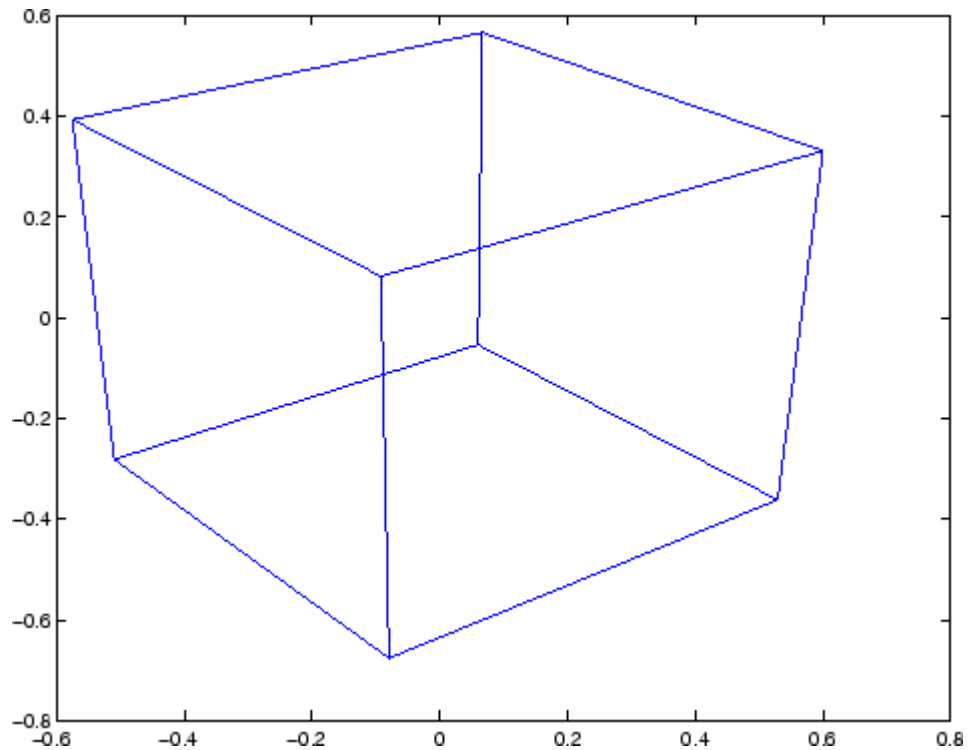
```

viewmtx

0.1777
-1.8858

Transform the cube vectors to the screen and plot the object:

```
A = viewmtx(-37.5,30,25);  
[m,n] = size(x);  
x4d = [x(:),y(:),z(:),ones(m*n,1)]';  
x2d = A*x4d;  
x2 = zeros(m,n); y2 = zeros(m,n);  
x2(:) = x2d(1,:)./x2d(4,:);  
y2(:) = x2d(2,:)./x2d(4,:);  
plot(x2,y2)
```



See Also

view, hgtransform

“Controlling the Camera Viewpoint” on page 1-96 for related functions

Defining the View for more information on viewing concepts and techniques

volumebounds

Purpose Coordinate and color limits for volume data

Syntax

```
lims = volumebounds(X,Y,Z,V)
lims = volumebounds(X,Y,Z,U,V,W)
lims = volumebounds(V), lims = volumebounds(U,V,W)
```

Description `lims = volumebounds(X,Y,Z,V)` returns the x, y, z, and color limits of the current axes for scalar data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax cmin cmax]
```

You can pass this vector to the `axis` command.

`lims = volumebounds(X,Y,Z,U,V,W)` returns the x, y, and z limits of the current axes for vector data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax]
```

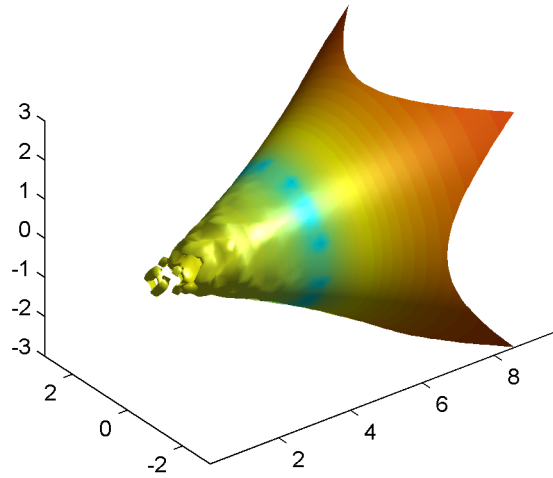
`lims = volumebounds(V)`, `lims = volumebounds(U,V,W)` assumes X, Y, and Z are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(V)`.

Examples This example uses `volumebounds` to set the axis and color limits for an isosurface generated by the `flow` function.

```
[x y z v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p)
daspect([1 1 1])
isocolors(x,y,z,flipdim(v,2),p)
shading interp
axis(volumebounds(x,y,z,v))
view(3)
camlight
lighting phong
```



See Also

`isosurface`, `streamslice`

“Volume Visualization” on page 1-98 for related functions

voronoi

Purpose Voronoi diagram

Syntax
voronoi(x,y)
voronoi(x,y,TRI)
voronoi(X,Y,options)
voronoi(AX,...)
voronoi(...,'LineStyleSpec')
h = voronoi(...)
[vx,vy] = voronoi(...)

Definition Consider a set of coplanar points P . For each point P_x in the set P , you can draw a boundary enclosing all the intermediate points lying closer to P_x than to other points in the set P . Such a boundary is called a *Voronoi polygon*, and the set of all Voronoi polygons for a given point set is called a *Voronoi diagram*.

Description voronoi(x,y) plots the bounded cells of the Voronoi diagram for the points x,y. Lines-to-infinity are approximated with an arbitrarily distant endpoint.

voronoi(x,y,TRI) uses the triangulation TRI instead of computing it via delaunay.

voronoi(X,Y,options) specifies a cell array of strings to be used as options in Qhull via delaunay.

If options is [], the default delaunay options are used. If options is {''}, no options are used, not even the default.

voronoi(AX,...) plots into AX instead of gca.

voronoi(...,'LineStyleSpec') plots the diagram with color and line style specified.

h = voronoi(...) returns, in h, handles to the line objects created.

[vx,vy] = voronoi(...) returns the finite vertices of the Voronoi edges in vx and vy so that plot(vx,vy,'-',x,y,'.') creates the Voronoi diagram. The lines-to-infinity are the last columns of vx and

`vy`. To ensure the lines-to-infinity do not affect the settings of the axis limits, use the commands:

```
h = plot(VX,VY,'-',X,Y,'. ');  
set(h(1:end-1),'xliminclude','off','yliminclude','off')
```

Note For the topology of the Voronoi diagram, i.e., the vertices for each Voronoi cell, use `voronoin`.

```
[v,c] = voronoin([x(:) y(:)])
```

Visualization

Use one of these methods to plot a Voronoi diagram:

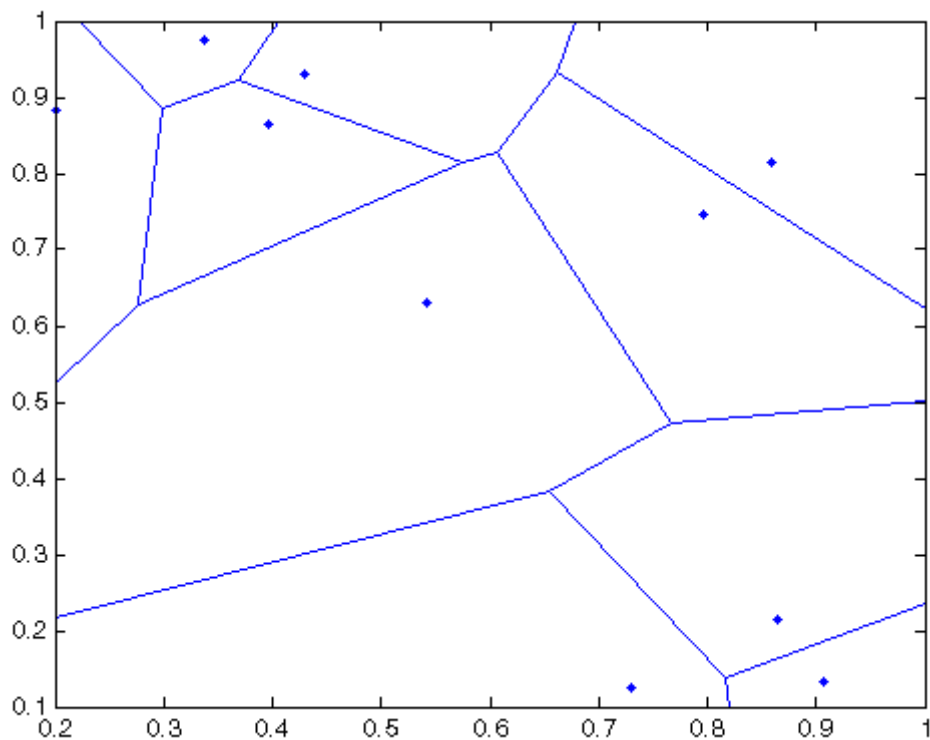
- If you provide no output argument, `voronoi` plots the diagram. See Example 1.
- To gain more control over color, line style, and other figure properties, use the syntax `[vx,vy] = voronoi(...)`. This syntax returns the vertices of the finite Voronoi edges, which you can then plot with the `plot` function. See Example 2.
- To fill the cells with color, use `voronoin` with `n = 2` to get the indices of each cell, and then use `patch` and other plot functions to generate the figure. Note that `patch` does not fill unbounded cells with color. See Example 3.

Examples

Example 1

This code uses the `voronoi` function to plot the Voronoi diagram for 10 randomly generated points.

```
rand('state',5);  
x = rand(1,10); y = rand(1,10);  
voronoi(x,y)
```



Example 2

This code uses the vertices of the finite Voronoi edges to plot the Voronoi diagram for the same 10 points.

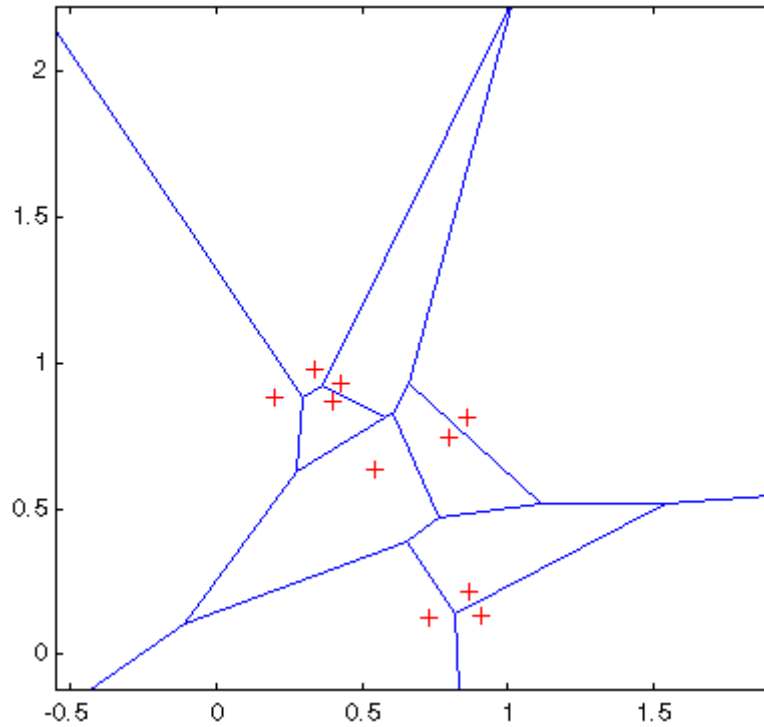
```
rand('state',5);  
x = rand(1,10); y = rand(1,10);  
[vx, vy] = voronoi(x,y);  
plot(x,y,'r+',vx,vy,'b-'); axis equal
```

Note that you can add this code to get the figure shown in Example 1.

```
xlim([min(x) max(x)])
```



```
ylim([min(y) max(y)])
```



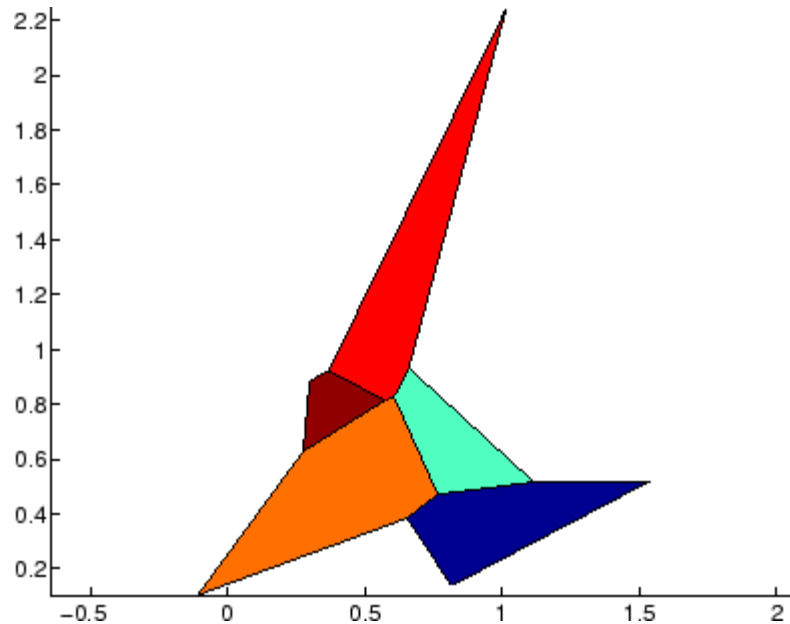
Example 3

This code uses `voronoin` and `patch` to fill the bounded cells of the same Voronoi diagram with color.

```
rand('state',5);
x=rand(10,2);
[v,c]=voronoin(x);
for i = 1:length(c)
    if all(c{i}~=1) % If at least one of the indices is 1,
                   % then it is an open region and we can't
                   % patch that.
```

voronoi

```
patch(v(c{i},1),v(c{i},2),i); % use color i.  
end  
end  
axis equal
```



Algorithm

If you supply no triangulation TRI, the voronoi function performs a Delaunay triangulation of the data that uses Qhull [1]. For information about Qhull, see <http://www.qhull.org/>. For copyright information, see <http://www.qhull.org/COPYING.txt>.

See Also

convhull, delaunay, LineSpec, plot, voronoin

Reference

[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/](http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/).

voronoin

Purpose N-D Voronoi diagram

Syntax `[V,C] = voronoin(X)`
`[V,C] = voronoin(X,options)`

Description `[V,C] = voronoin(X)` returns Voronoi vertices `V` and the Voronoi cells `C` of the Voronoi diagram of `X`. `V` is a `numv`-by-`n` array of the `numv` Voronoi vertices in `n`-dimensional space, each row corresponds to a Voronoi vertex. `C` is a vector cell array where each element contains the indices into `V` of the vertices of the corresponding Voronoi cell. `X` is an `m`-by-`n` array, representing `m` `n`-dimensional points, where `n > 1` and `m >= n+1`. The first row of `V` is a point at infinity. If any index in a cell of the cell array is 1, then the corresponding Voronoi cell contains the first point in `V`, a point at infinity. This means the Voronoi cell is unbounded.

`voronoin` uses `Qhull`.

`[V,C] = voronoin(X,options)` specifies a cell array of strings `options` to be used in `Qhull`. The default options are

- `{'Qbb'}` for 2- and 3-dimensional input
- `{'Qbb','Qx'}` for 4 and higher-dimensional input

If `options` is `[]`, the default options are used. If code is `{''}`, no options are used, not even the default. For more information on `Qhull` and its options, see <http://www.qhull.org>.

Visualization You can plot individual bounded cells of an `n`-dimensional Voronoi diagram. To do this, use `convhulln` to compute the vertices of the facets that make up the Voronoi cell. Then use `patch` and other plot functions to generate the figure. For an example, see “Tessellation and Interpolation of Scattered Data in Higher Dimensions” in the MATLAB Mathematics documentation.

Examples **Example 1**

Let

```
x = [ 0.5    0
      0      0.5
      -0.5  -0.5
      -0.2  -0.1
      -0.1   0.1
       0.1  -0.1
       0.1   0.1 ]
```

then

```
[V,C] = voronoin(x)
```

```
V =
      Inf      Inf
      0.3833   0.3833
      0.7000  -1.6500
      0.2875   0.0000
     -0.0000   0.2875
     -0.0000  -0.0000
     -0.0500  -0.5250
     -0.0500  -0.0500
     -1.7500   0.7500
     -1.4500   0.6500
```

C =

```
[1x4 double]
[1x5 double]
[1x4 double]
[1x4 double]
[1x4 double]
[1x5 double]
[1x4 double]
```

Use a for loop to see the contents of the cell array C.

```
for i=1:length(C), disp(C{i}), end
```

```
4    2    1    3
```

voronoin

```
10  5  2  1  9
 9  1  3  7
10  8  7  9
10  5  6  8
 8  6  4  3  7
 6  4  2  5
```

In particular, the fifth Voronoi cell consists of 4 points: $V(10, :)$, $V(5, :)$, $V(6, :)$, $V(8, :)$.

Example 2

The following example illustrates the options input to `voronoin`. The commands

```
X = [-1 -1; 1 -1; 1 1; -1 1];
[V,C] = voronoin(X)
```

return an error message.

```
? qhull input error: can not scale last coordinate. Input is
cocircular
or cospherical. Use option 'Qz' to add a point at infinity.
```

The error message indicates that you should add the option `'Qz'`. The following command passes the option `'Qz'`, along with the default `'Qbb'`, to `voronoin`.

```
[V,C] = voronoin(X,{'Qbb','Qz'})
V =
```

```
Inf  Inf
 0    0
```

```
C =
```

```
[1x2 double]
[1x2 double]
```

[1x2 double]
[1x2 double]

Algorithm

voronoin is based on Qhull [1]. For information about Qhull, see <http://www.qhull.org/>. For copyright information, see <http://www.qhull.org/COPYING.txt>.

See Also

convhull, convhulln, delaunay, delaunayn, voronoi

Reference

[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/>.

wait

Purpose Wait until timer stops running

Syntax `wait(obj)`

Description `wait(obj)` blocks the MATLAB command line and waits until the timer, represented by the timer object `obj`, stops running. When a timer stops running, the value of the timer object's `Running` property changes from 'on' to 'off'.

If `obj` is an array of timer objects, `wait` blocks the MATLAB command line until all the timers have stopped running.

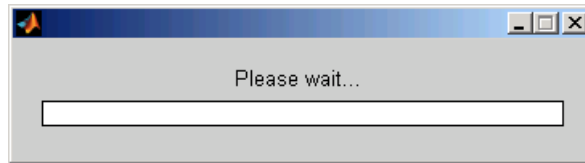
If the timer is not running, `wait` returns immediately.

See Also `timer`, `start`, `stop`

Purpose	Open waitbar
Syntax	<pre>h = waitbar(x,'title') waitbar(x,'title','CreateCancelBtn','button_callback') waitbar(...,property_name,property_value,...) waitbar(x) waitbar(x,h) waitbar(x,h,'updated title')</pre>
Description	<p>A waitbar shows what percentage of a calculation is complete, as the calculation proceeds.</p> <p><code>h = waitbar(x,'title')</code> displays a waitbar of fractional length <code>x</code>. The handle to the waitbar figure is returned in <code>h</code>. <code>x</code> must be between 0 and 1.</p> <p><code>waitbar(x,'title','CreateCancelBtn','button_callback')</code> specifying CreateCancelBtn adds a cancel button to the figure that executes the MATLAB commands specified in <code>button_callback</code> when the user clicks the cancel button or the close figure button. <code>waitbar</code> sets both the cancel button callback and the figure <code>CloseRequestFcn</code> to the string specified in <code>button_callback</code>.</p> <p><code>waitbar(...,property_name,property_value,...)</code> optional arguments <code>property_name</code> and <code>property_value</code> enable you to set figure properties for the waitbar.</p> <p><code>waitbar(x)</code> subsequent calls to <code>waitbar(x)</code> extend the length of the bar to the new position <code>x</code>.</p> <p><code>waitbar(x,h)</code> extends the length of the bar in the waitbar <code>h</code> to the new position <code>x</code>.</p> <p><code>waitbar(x,h,'updated title')</code> updates the title text in the waitbar figure, in addition to setting the fractional length to <code>x</code>.</p>
Example	<p><code>waitbar</code> is typically used inside a for loop that performs a lengthy computation. For example,</p> <pre>h = waitbar(0,'Please wait...');</pre>

waitbar

```
for i=1:100, % computation here %  
waitbar(i/100)  
end  
close(h)
```



See Also

“Predefined Dialog Boxes” on page 1-100 for related functions

Purpose	Wait for condition before resuming execution
Syntax	<pre>waitfor(h) waitfor(h, 'PropertyName') waitfor(h, 'PropertyName', PropertyValue)</pre>
Description	<p>The waitfor function blocks the caller's execution stream so that command-line expressions, callbacks, and statements in the blocked M-file do not execute until a specified condition is satisfied.</p> <p>waitfor(h) returns when the graphics object identified by h is deleted or when a Ctrl+C is typed in the Command Window. If h does not exist, waitfor returns immediately without processing any events.</p> <p>waitfor(h, 'PropertyName'), in addition to the conditions in the previous syntax, returns when the value of 'PropertyName' for the graphics object h changes. If 'PropertyName' is not a valid property for the object, waitfor returns immediately without processing any events.</p> <p>waitfor(h, 'PropertyName', PropertyValue), in addition to the conditions in the previous syntax, waitfor returns when the value of 'PropertyName' for the graphics object h changes to PropertyValue. waitfor returns immediately without processing any events if 'PropertyName' is set to PropertyValue.</p>
Remarks	<p>While waitfor blocks an execution stream, other execution streams in the form of callbacks may execute as a result of various events (e.g., pressing a mouse button).</p> <p>waitfor can block nested execution streams. For example, a callback invoked during a waitfor statement can itself invoke waitfor.</p>
See Also	<p>uiresume, uiwait</p> <p>“Predefined Dialog Boxes” on page 1-100 for related functions</p>

waitforbuttonpress

Purpose Wait for key press or mouse-button click

Syntax `k = waitforbuttonpress`

Description `k = waitforbuttonpress` blocks the caller's execution stream until the function detects that the user has clicked a mouse button or pressed a key while the figure window is active. The function returns

- 0 if it detects a mouse button click
- 1 if it detects a key press

Additional information about the event that causes execution to resume is available through the figure's `CurrentCharacter`, `SelectionType`, and `CurrentPoint` properties.

If a `WindowButtonDownFcn` is defined for the figure, its callback is executed before `waitforbuttonpress` returns a value.

Example These statements display text in the Command Window when the user either clicks a mouse button or types a key in the figure window:

```
w = waitforbuttonpress;
if w == 0
    disp('Button click')
else
    disp('Key press')
end
```

See Also `dragrect`, `ginput`, `rbbox`, `waitfor`
“Developing User Interfaces” on page 1-101 for related functions

Purpose Open warning dialog box

Syntax

```
warndlg
warndlg('warningstring')
warndlg('warningstring','dlgname')
h = warndlg('warnstring','dlgname',createmode)
h = warndlg(...)
```

Description warndlg displays a dialog box named 'Warning Dialog' containing the string 'This is the default warning string.' The warning dialog box disappears after you press the **OK** button.

warndlg('warningstring') displays a dialog box with the title 'Warning Dialog' containing the string specified by warningstring.

warndlg('warningstring','dlgname') displays a dialog box with the title dlgname that contains the string warningstring. The string warnstring can be any valid string format – cell arrays are preferred.

h = warndlg('warnstring','dlgname',createmode) specifies whether a warning dialog box is modal or non-modal. Valid values for createmode are 'modal', 'non-modal', and 'replace'. If createmode is 'modal' or 'replace', the first available warning dialog box with the specified title is updated to reflect the new properties of the warning dialog box. All other such warning dialog boxes are deleted. If createmode is 'non-modal', the warning dialog box is not replaced and a new handle is created. The default value for createmode is 'non-modal'.

createmode can also be a structure with fields WindowStyle and Interpreter. WindowStyle can be any of the values above. Interpreter can be 'tex' or 'none'. The default value for the Interpreter is 'none';

h = warndlg(...) returns the handle of the dialog box.

Warnings with Multiple Lines

To use multiple lines in your warning, define warningstring using either of the following:

warndlg

- sprintf with newline characters separating the lines

```
warndlg(sprintf('Message line 1 \n Message line 2'))
```

- Cell arrays of strings

```
warndlg({'Message line 1';'Message line 2'})
```

Examples

The statement

```
warndlg('Pressing OK will clear memory','!! Warning !!')
```

displays this dialog box:



See Also

dialog, errordlg, helpdlg, inputdlg, listdlg, msgbox, questdlg
figure, uiwait, uiresume, warning

“Predefined Dialog Boxes” on page 1-100 for related functions

Purpose

Warning message

Syntax

```
warning('message')
warning('message', a1, a2,...)
warning('message_id', 'message')
warning('message_id', 'message', a1, a2, ..., an)
s = warning(state, 'message_id')
s = warning(state, mode)
```

Description

`warning('message')` displays the text 'message' like the `disp` function, except that with `warning`, message display can be suppressed.

`warning('message', a1, a2,...)` displays a message string that contains formatting conversion characters, such as those used with the MATLAB `sprintf` function. Each conversion character in `message` is converted to one of the values `a1, a2, ...` in the argument list.

Note MATLAB converts special characters (like `\n` and `%d`) in the warning message string only when you specify more than one input argument with `warning`. See Example 4 below.

`warning('message_id', 'message')` attaches a unique identifier, or `message_id`, to the warning message. The identifier enables you to single out certain warnings during the execution of your program, controlling what happens when the warnings are encountered. See “Message Identifiers” and “Warning Control” in the MATLAB Programming documentation for more information on the `message_id` argument and how to use it.

`warning('message_id', 'message', a1, a2, ..., an)` includes formatting conversion characters in `message`, and the character translations in arguments `a1, a2, ..., an`.

`s = warning(state, 'message_id')` is a warning control statement that enables you to indicate how you want MATLAB to act on certain warnings. The `state` argument can be 'on', 'off', or 'query'. The

warning

`message_id` argument can be a message identifier string, 'all', or 'last'. See “Warning Control Statements” in the MATLAB Programming documentation for more information.

Output `s` is a structure array that indicates the previous state of the selected warnings. The structure has the fields `identifier` and `state`. See “Output from Control Statements” in the MATLAB Programming documentation for more.

`s = warning(state, mode)` is a warning control statement that enables you to display an M-stack trace or display more information with each warning. The `state` argument can be 'on', 'off', or 'query'. The `mode` argument can be 'backtrace' or 'verbose'. See “Debug, Backtrace, and Verbose Modes” in the MATLAB Programming documentation for more information.

Examples

Example 1

Generate a warning that displays a simple string:

```
if ~ischar(p1)
    warning('Input must be a string')
end
```

Example 2

Generate a warning string that is defined at run-time. The first argument defines a message identifier for this warning:

```
warning('MATLAB:paramAmbiguous', ...
        'Ambiguous parameter name, %s.', param)
```

Example 3

Using a message identifier, enable just the `actionNotTaken` warning from Simulink by first turning off all warnings and then setting just that warning to on:

```
warning off all
warning on Simulink:actionNotTaken
```


Use `query` to determine the current state of all warnings. It reports that you have set all warnings to `off` with the exception of `Simulink:actionNotTaken`:

```
warning query all
The default warning state is 'off'. Warnings not set to the
default are
```

```
State Warning Identifier
on Simulink:actionNotTaken
```

Example 4

MATLAB converts special characters (like `\n` and `%d`) in the warning message string only when you specify more than one input argument with `warning`. In the single argument case shown below, `\n` is taken to mean `backslash-n`. It is not converted to a newline character:

```
warning('In this case, the newline \n is not converted.')
Warning: In this case, the newline \n is not converted.
```

But, when more than one argument is specified, MATLAB does convert special characters. This is true regardless of whether the additional argument supplies conversion values or is a message identifier:

```
warning('WarnTests:convertTest', ...
        'In this case, the newline \n is converted.')
Warning: In this case, the newline
is converted.
```

Example 5

Turn on one particular warning, saving the previous state of this one warning in `s`. Remember that this nonquery syntax performs an implicit query prior to setting the new state:

```
s = warning('on', 'Control:parameterNotSymmetric');
```

warning

After doing some work that includes making changes to the state of some warnings, restore the original state of all warnings:

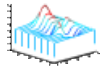
```
warning(s)
```


See Also

lastwarn, warndlg, error, lasterror, errordlg, dbstop, disp, sprintf

Purpose

Waterfall plot

**GUI Alternatives**

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see [Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation](#) and [Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation](#).

Syntax

```
waterfall(Z)
waterfall(X,Y,Z)
waterfall(...,C)
waterfall(axes_handles,...)
h = waterfall(...)
```

Description

The `waterfall` function draws a mesh similar to the `meshz` function, but it does not generate lines from the columns of the matrices. This produces a “waterfall” effect.

`waterfall(Z)` creates a waterfall plot using $x = 1:\text{size}(Z,1)$ and $y = 1:\text{size}(Z,1)$. Z determines the color, so color is proportional to surface height.

`waterfall(X,Y,Z)` creates a waterfall plot using the values specified in X , Y , and Z . Z also determines the color, so color is proportional to the surface height. If X and Y are vectors, X corresponds to the columns of Z , and Y corresponds to the rows, where $\text{length}(x) = n$, $\text{length}(y) = m$, and $[m,n] = \text{size}(Z)$. X and Y are vectors or matrices that define the x - and y -coordinates of the plot. Z is a matrix that defines the z -coordinates of the plot (i.e., height above a plane). If C is omitted, color is proportional to Z .

`waterfall(...,C)` uses scaled color values to obtain colors from the current colormap. Color scaling is determined by the range of C , which

waterfall

must be the same size as Z. MATLAB performs a linear transformation on C to obtain colors from the current colormap.

`waterfall(axes_handles, ...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = waterfall(...)` returns the handle of the patch graphics object used to draw the plot.

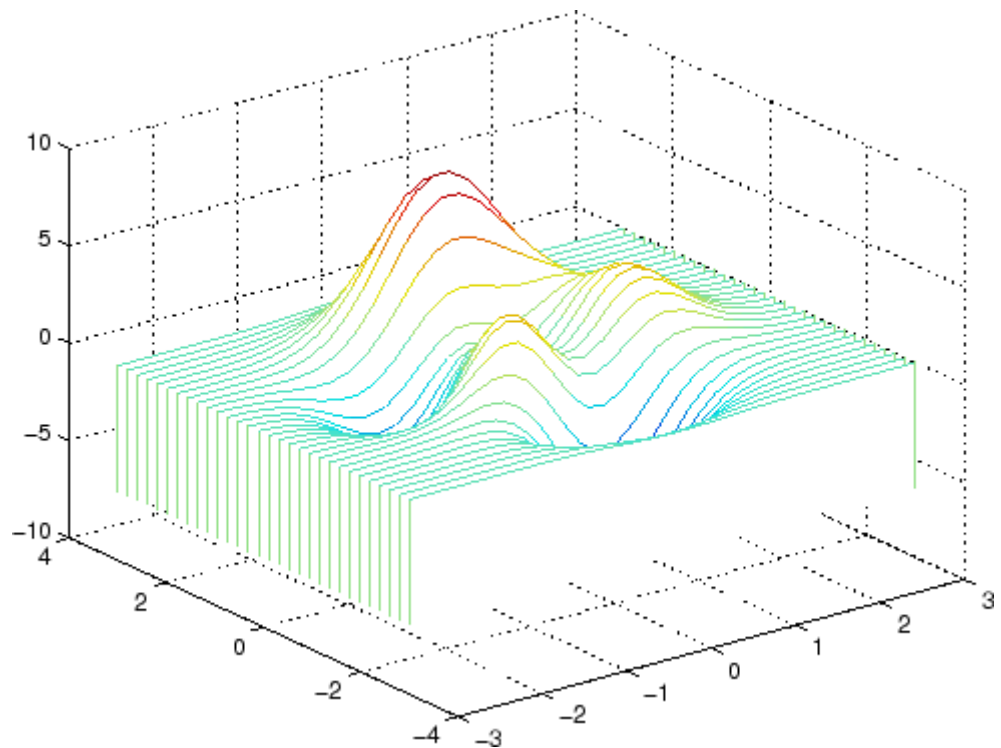
Remarks

For column-oriented data analysis, use `waterfall(Z')` or `waterfall(X',Y',Z')`.

Examples

Produce a waterfall plot of the peaks function.

```
[X,Y,Z] = peaks(30);  
waterfall(X,Y,Z)
```



Algorithm

The range of X, Y, and Z, or the current setting of the axes `Llim`, `YLim`, and `ZLim` properties, determines the range of the axes (also set by `axis`). The range of C, or the current setting of the axes `CLim` property, determines the color scaling (also set by `caxis`).

The `CData` property for the patch graphics objects specifies the color at every point along the edge of the patch, which determines the color of the lines.

The `waterfall` plot looks like a mesh surface; however, it is a patch graphics object. To create a surface plot similar to `waterfall`, use the `meshz` function and set the `MeshStyle` property of the surface to `'Row'`.

waterfall

For a discussion of parametric surfaces and related color properties, see surf.

See Also

axes, axis, caxis, meshz, ribbon, surf

Properties for patch graphics objects

Purpose Information about Microsoft WAVE (.wav) sound file

Syntax [m d] = wavinfo(filename)

Description [m d] = wavinfo(filename) returns information about the contents of the WAVE sound file specified by the string filename. Enclose the filename input in single quotes.

m is the string 'Sound (WAV) file', if filename is a WAVE file. Otherwise, it contains an empty string ('').

d is a string that reports the number of samples in the file and the number of channels of audio data. If filename is not a WAVE file, it contains the string 'Not a WAVE file'.

See Also wavread

wavplay

Purpose Play recorded sound on PC-based audio output device

Syntax `wavplay(y,Fs)`
`wavplay(...,'mode')`

Description `wavplay(y,Fs)` plays the audio signal stored in the vector `y` on a PC-based audio output device. You specify the audio signal sampling rate with the integer `Fs` in samples per second. The default value for `Fs` is 11025 Hz (samples per second). `wavplay` supports only 1- or 2-channel (mono or stereo) audio signals.

`wavplay(...,'mode')` specifies how `wavplay` interacts with the command line, according to the string `'mode'`. The string `'mode'` can be

- `'async'`: You have immediate access to the command line as soon as the sound begins to play on the audio output device (a nonblocking device call).
- `'sync'` (default value): You don't have access to the command line until the sound has finished playing (a blocking device call).

The audio signal `y` can be one of four data types. The number of bits used to quantize and play back each sample depends on the data type.

Data Types for wavplay

Data Type	Quantization
Double-precision (default value)	16 bits/sample
Single-precision	16 bits/sample
16-bit signed integer	16 bits/sample
8-bit unsigned integer	8 bits/sample

Remarks You can play your signal in stereo if `y` is a two-column matrix.

Examples

The MAT-files `gong.mat` and `chirp.mat` both contain an audio signal `y` and a sampling frequency `Fs`. Load and play the gong and the chirp audio signals. Change the names of these signals in between load commands and play them sequentially using the 'sync' option for `wavplay`.

```
load chirp;
y1 = y; Fs1 = Fs;
load gong;
wavplay(y1,Fs1,'sync') % The chirp signal finishes before the
wavplay(y,Fs)          % gong signal begins playing.
```

See Also

`wavrecord`

wavread

Purpose

Read Microsoft WAVE (.wav) sound file

Graphical Interface

As an alternative to `auread`, use the Import Wizard. To activate the Import Wizard, select **Import Data** from the **File** menu.

Syntax

```
y = wavread(filename)
[y,Fs,bits] = wavread(filename)
[...] = wavread(filename,N)
[...] = wavread(filename,[N1 N2])
siz = wavread(filename,'size')
```

Description

`wavread` supports multichannel data, with up to 32 bits per sample, and supports reading 24- and 32-bit .wav files.

`y = wavread(filename)` loads a WAVE file specified by `filename`, returning the sampled data in `y`. The `filename` input is a string enclosed in single quotes. The .wav extension is appended if no extension is given. Amplitude values are in the range [-1,+1].

`[y,Fs,bits] = wavread(filename)` returns the sample rate (`Fs`) in Hertz and the number of bits per sample (`bits`) used to encode the data in the file.

`[...] = wavread(filename,N)` returns only the first `N` samples from each channel in the file.

`[...] = wavread(filename,[N1 N2])` returns only samples `N1` through `N2` from each channel in the file.

`siz = wavread(filename,'size')` returns the size of the audio data contained in the file in place of the actual audio data, returning the vector `siz = [samples channels]`.

See Also

`auread`, `wavwrite`

Purpose	Record sound using PC-based audio input device
Syntax	<pre>y = wavrecord(n,Fs) y = wavrecord(...,ch) y = wavrecord(...,'dtype')</pre>
Description	<p><code>y = wavrecord(n,Fs)</code> records <code>n</code> samples of an audio signal, sampled at a rate of <code>Fs</code> Hz (samples per second). The default value for <code>Fs</code> is 11025 Hz.</p> <p><code>y = wavrecord(...,ch)</code> uses <code>ch</code> number of input channels from the audio device. <code>ch</code> can be either 1 or 2, for mono or stereo, respectively. The default value for <code>ch</code> is 1.</p> <p><code>y = wavrecord(...,'dtype')</code> uses the data type specified by the string <code>'dtype'</code> to record the sound. The string <code>'dtype'</code> can be one of the following:</p> <ul style="list-style-type: none">• <code>'double'</code> (default value), 16 bits/sample• <code>'single'</code>, 16 bits/sample• <code>'int16'</code>, 16 bits/sample• <code>'uint8'</code>, 8 bits/sample
Remarks	Standard sampling rates for PC-based audio hardware are 8000, 11025, 2250, and 44100 samples per second. Stereo signals are returned as two-column matrices. The first column of a stereo audio matrix corresponds to the left input channel, while the second column corresponds to the right input channel.
Examples	<p>Record 5 seconds of 16-bit audio sampled at 11025 Hz. Play back the recorded sound using <code>wavplay</code>. Speak into your audio device (or produce your audio signal) while the <code>wavrecord</code> command runs.</p> <pre>Fs = 11025; y = wavrecord(5*Fs,Fs,'int16'); wavplay(y,Fs);</pre>

wavrecord

See Also

wavplay

Purpose Write Microsoft WAVE (.wav) sound file

Syntax

```
wavwrite(y,filename)
wavwrite(y,Fs,filename)
wavwrite(y,Fs,N,filename)
```

Description wavwrite writes data to 8-, 16-, 24-, and 32-bit .wav files.

wavwrite(y,filename) writes the data stored in the variable y to a WAVE file called filename. The filename input is a string enclosed in single quotes. The data has a sample rate of 8000 Hz and is assumed to be 16-bit. Each column of the data represents a separate channel. Therefore, stereo data should be specified as a matrix with two columns. Amplitude values outside the range [-1,+1] are clipped prior to writing.

wavwrite(y,Fs,filename) writes the data stored in the variable y to a WAVE file called filename. The data has a sample rate of Fs Hz and is assumed to be 16-bit. Amplitude values outside the range [-1,+1] are clipped prior to writing.

wavwrite(y,Fs,N,filename) writes the data stored in the variable y to a WAVE file called filename. The data has a sample rate of Fs Hz and is N-bit, where N is 8, 16, 24, or 32. For N < 32, amplitude values outside the range [-1,+1] are clipped.

Note 8-, 16-, and 24-bit files are type 1 integer pulse code modulation (PCM). 32-bit files are written as type 3 normalized floating point.

See Also auwrite, wavread

Purpose Open Web site or file in Web browser or Help browser

Syntax

```
web
web url
web url -new
web url -notoolbar
web url -noaddressbox
web url -helpbrowser
web url -browser
web(...)
stat = web('url', '-browser')
[stat, h1] = web
[stat, h1, url] = web
```

Description

`web` opens an empty MATLAB Web browser. The MATLAB Web browser includes an address field where you can enter a URL, for example, to a Web site or file, a toolbar with common browser buttons, and a MATLAB desktop menu.

`web url` displays the specified URL, `url`, in the MATLAB Web browser. If any MATLAB Web browsers are already open, it displays the page in the browser that last had focus. Files up to 1.5MB in size display in the MATLAB Web browser, while larger files instead display in the default Web browser for your system. If `url` is located in the directory returned when you run `docroot`, the URL displays in the MATLAB Help browser instead of the MATLAB Web browser.

`web url -new` displays the specified URL, `url`, in a new MATLAB Web browser.

`web url -notoolbar` displays the specified URL, `url`, in a MATLAB Web browser that does not include the toolbar and address field. If any MATLAB Web browsers are already open, also use the **-new** option; otherwise `url` displays in the browser that last had focus, regardless of its toolbar status.

`web url -noaddressbox` displays the specified URL, `url`, in a MATLAB Web browser that does not include the address field. If any MATLAB Web browsers are already open, also use the **-new** option; otherwise `url`

displays in the browser that last had focus, regardless of its address field status.

`web url -helpbrowser` displays the specified URL, `url`, in the MATLAB Help browser.

`web url -browser` displays the default Web browser for your system and loads the file or Web site specified by the URL `url` in it. Generally, `url` specifies a local file or a Web site on the Internet. The URL can be in any form that the browser supports. On Windows and Macintosh, the default Web browser is determined by the operating system. On UNIX, the Web browser used is specified via `docopt` in the `doccmd` string.

`web(...)` is the functional form of `web`.

`stat = web('url', '-browser')` runs `web` and returns the status of `web` to the variable `stat`.

Value of <code>stat</code>	Description
0	Browser was found and launched.
1	Browser was not found.
2	Browser was found but could not be launched.

`[stat, h1] = web` returns the status of `web` to the variable `stat`, and returns a handle to the Java class, `h1`, for the last active browser.

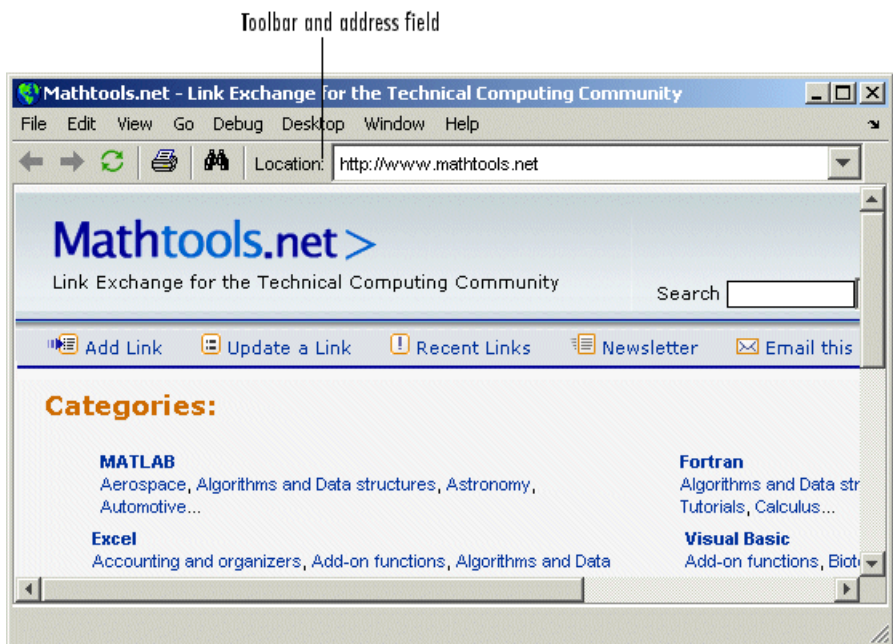
`[stat, h1, url] = web` returns the status of `web` to the variable `stat`, returns a handle to the Java class `h1`, for the last active browser, and returns its current URL to `url`.

Examples

Run

```
web http://www.mathtools.net
```

and MATLAB displays



`web http://www.mathworks.com` loads the MathWorks Web site home page into the MATLAB Web browser.

`web file:///disk/dir1/dir2/foo.html` opens the file `foo.html` in the MATLAB Web browser.

`web(['file:/// ' which('foo.html')])` opens `foo.html` if the file is on the MATLAB path or in the current directory.

`web('text://<html><h1>Hello World</h1></html>')` displays the HTML-formatted text `Hello World`.

`web ('http://www.mathworks.com', '-new', '-notoolbar')` loads the MathWorks Web site home page into a new MATLAB Web browser that does not include a toolbar or address field.

`web file:///disk/dir1/foo.html -helpbrowser` opens the file `foo.html` in the MATLAB Help browser.

`web file:///disk/dir1/foo.html` -browser opens the file `foo.html` in the system Web browser.

`web mailto:email_address` uses your system browser's default e-mail application to send a message to `email_address`.

`web http://www.mathtools.net` -browser opens a browser to `mathtools.net`. Then `[stat,h1,url]=web` returns

```
stat =  
    0  
  
h1 =  
com.mathworks.mde.webbrowser.WebBrowser[,0,0,591x140,  
layout=java.awt.BorderLayout,alignmentX=null,alignmentY=null,  
border=,flags=9,maximumSize=,minimumSize=,preferredSize=]  
  
url =  
http://www.mathtools.net/
```

Run `methods(h1)` to view allowable methods for the class. As an example, you can use the method `setCurrentLocation` to change the URL displayed in `h1`, as in

```
setCurrentLocation(h1,'http://www.mathworks.com')
```

See Also

`doc`, `docopt`, `helpbrowser`, `matlabcolon`

Web browser in the MATLAB Desktop Tools and Development Environment documentation

weekday

Purpose Day of week

Syntax
[N, S] = weekday(D)
[N, S] = weekday(D, form)
[N, S] = weekday(D, locale)
[N, S] = weekday(D, form, locale)

Description [N, S] = weekday(D) returns the day of the week in numeric (N) and string (S) form for a given serial date number or date string D. Input argument D can represent more than one date in an array of serial date numbers or a cell array of date strings.

[N, S] = weekday(D, form) returns the day of the week in numeric (N) and string (S) form, where the content of S depends on the form argument. If form is **'long'**, then S contains the full name of the weekday (e.g., Tuesday). If form is **'short'**, then S contains an abbreviated name (e.g., Tues) from this table.

The days of the week are assigned these numbers and abbreviations.

N	S (short)	S (long)
1	Sun	Sunday
2	Mon	Monday
3	Tue	Tuesday
4	Wed	Wednesday
5	Thu	Thursday
6	Fri	Friday
7	Sat	Saturday

[N, S] = weekday(D, locale) returns the day of the week in numeric (N) and string (S) form, where the format of the output depends on the locale argument. If locale is **'local'**, then weekday uses local format for its output. If locale is **'en_US'**, then weekday uses US English.

`[N, S] = weekday(D, form, locale)` returns the day of the week using the formats described above for `form` and `locale`.

Examples

Either

```
[n, s] = weekday(728647)
```

or

```
[n, s] = weekday('19-Dec-1994')
```

returns `n = 2` and `s = Mon`.

See Also

`datenum`, `datevec`, `eomday`

what

Purpose List MATLAB files in current directory

Graphical Interface As an alternative to the `what` function, use the “Current Directory Browser”. To open it, select **Current Directory** from the **Desktop** menu in the MATLAB desktop.

Syntax

```
what
what dirname
what class
s = what('dirname')
```

Description `what` lists the M, MAT, MEX, MDL, and P-files and the class directories that reside in the current working directory.

`what dirname` lists the files in directory `dirname` on the MATLAB search path. It is not necessary to enter the full pathname of the directory. The last component, or last two components, is sufficient.

`what class` lists the files in method directory, `@class`. For example, `what cfit` lists the MATLAB files in `toolbox/curvefit/curvefit/@cfit`.

`s = what('dirname')` returns the results in a structure array with these fields.

Field	Description
path	Path to directory
m	Cell array of M-file names
mat	Cell array of MAT-file names
mex	Cell array of MEX-file names
mdl	Cell array of MDL-file names
p	Cell array of P-file names
classes	Cell array of class names

Examples

List the files in toolbox/matlab/audiovideo:

```
what audiovideo
```

M-files in directory matlabroot\toolbox\matlab\audiovideo

Contents	auwrite	movie2avi
audiodevinfo	avifinfo	mu2lin
audioplayerreg	aviinfo	prefspanel
audiorecorderreg	aviread	render_basicaudiotoolbar
audiouniquename	lin2mu	render_fullaudiotoolbar
aufinfo	mmcompinfo	render_uimgraudiotoolbar
auread	mmfileinfo	sound

MAT-files in directory matlabroot\toolbox\matlab\audiovideo

audiotoolbaricons	gong	laughter	train
chirp	handel	splat	

MEX-files in directory matlabroot\toolbox\matlab\audiovideo

winaudioplayer	winaudiorecorder
----------------	------------------

Classes in directory matlabroot\toolbox\matlab\audiovideo

audioplayer	audiorecorder	avifile	mmreader
-------------	---------------	---------	----------

Obtain a structure array containing the MATLAB filenames in toolbox/matlab/general:

```
s = what('general')
s =
    path: 'matlabroot:\toolbox\matlab\general'
      m: {87x1 cell}
      mat: {0x1 cell}
      mex: {2x1 cell}
      mdl: {0x1 cell}
```

what

```
p: {'callgraphviz.p'}  
classes: {0x1 cell}
```

See Also

dir, exist, lookfor, mfilename, path, which, who

Purpose Release Notes for MathWorks products

Syntax `whatsnew`

Description `whatsnew` displays the Release Notes in the Help browser, presenting information about new features, problems from previous releases that have been fixed in the current release, and compatibility issues, all organized by product.

See Also `help`, `version`

which

Purpose

Locate functions and files

Graphical Interface

As an alternative to the `which` function, use the “Current Directory Browser”.

Syntax

```
which fun
which classname/fun
which private/fun
which classname/private/fun
which fun1 in fun2
which fun(a,b,c,...)
which file.ext
which fun -all
s = which('fun',...)
```

Description

`which fun` displays the full pathname for the argument `fun`. If `fun` is a

- MATLAB function or Simulink model in an M, P, or MDL file on the MATLAB path, then `which` displays the full pathname for the corresponding file
- Workspace variable, then `which` displays a message identifying `fun` as a variable
- Method in a loaded Java class, then `which` displays the package, class, and method name for that method

If `fun` is an overloaded function or method, then `which fun` returns only the pathname of the first function or method found.

`which classname/fun` displays the full pathname for the M-file defining the `fun` method in MATLAB class, `classname`. For example, `which serial/fopen` displays the path for `fopen.m` in the MATLAB class directory, `@serial`.

`which private/fun` limits the search to private functions. For example, `which private/orthog` displays the path for `orthog.m` in the `/private` subdirectory of `toolbox/matlab/elmata`.

`which classname/private/fun` limits the search to private methods defined by the MATLAB class, `classname`. For example, `which dfilt/private/todtf` displays the path for `todtf.m` in the private directory of the `dfilt` class.

`which fun1 in fun2` displays the pathname to function `fun1` in the context of the M-file `fun2`. You can use this form to determine whether a subfunction is being called instead of a function on the path. For example, `which get in editpath` tells you which `get` function is called by `editpath.m`.

During debugging of `fun2`, using `which fun1` gives the same result.

`which fun(a,b,c,...)` displays the path to the specified function with the given input arguments. For example, `which feval(g)`, when `g=inline('sin(x)')`, indicates that `inline/feval.m` would be invoked. `which toLowerCase(s)`, when `s=java.lang.String('my Java string')`, indicates that the `toLowerCase` method in class `java.lang.String` would be invoked.

`which file.ext` displays the full pathname of the specified file if that file is in the current working directory or on the MATLAB path. Use `exist` to check for the existence of files anywhere else.

`which fun -all d` displays the paths to all items on the MATLAB path with the name `fun`. You may use the `-all` qualifier with any of the above formats of the `which` function.

`s = which('fun',...)` returns the results of `which` in the string `s`. For workspace variables, `s` is the string 'variable'. You may specify an output variable in any of the above formats of the `which` function.

If `-all` is used with this form, the output `s` is always a cell array of strings, even if only one string is returned.

Examples

The statement below indicates that `pinv` is in the `matfun` directory of MATLAB.

```
which pinv
matlabroot\toolbox\matlab\matfun\pinv.m
```

which

To find the `fopen` function used on MATLAB serial class objects

```
which serial/fopen
matlabroot\toolbox\matlab\iofun\@serial\fopen.m % serial method
```

To find the `setTitle` method used on objects of the Java Frame class, the class must first be loaded into MATLAB. The class is loaded when you create an instance of the class:

```
frameObj = java.awt.Frame;

which setTitle
java.awt.Frame.setTitle % Frame method
```

When you specify an output variable, `which` returns a cell array of strings to the variable. You must use the *function* form of `which`, enclosing all arguments in parentheses and single quotes:

```
s = which('private/stradd', '-all');
whos s
  Name      Size      Bytes  Class
  s         3x1         562   cell array
Grand total is 146 elements using 562 bytes
```

See Also

`dir`, `doc`, `exist`, `lookfor`, `mfilename`, `path`, `type`, `what`, `who`

Purpose	Repeatedly execute statements while condition is true
Syntax	<pre>while<replaceable>expression</replaceable> <replaceable>statements</replaceable> end</pre>
Description	<p>while repeats statements an indefinite number of times. The statements are executed while the real part of <i>expression</i> has all nonzero elements. <i>expression</i> is usually of the form</p> <pre>expression <i>rel_op</i> expression</pre> <p>where <i>rel_op</i> is ==, <, >, <=, >=, or ~=.</p> <p>The scope of a while statement is always terminated with a matching end.</p>
Arguments	<p>expression</p> <p><i>expression</i> 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)).</p> <p>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.</p> <pre>(count < limit) && ((height - offset) >= 0)</pre> <p>statements</p> <p><i>statements</i> is one or more MATLAB statements to be executed only while the <i>expression</i> is true or nonzero.</p>
Remarks	<p>Nonscalar Expressions</p> <p>If the evaluated expression yields a nonscalar value, then every element of this value must be true or nonzero for the entire expression to be considered true. For example, the statement while (A < B) is</p>

true only if each element of matrix A is less than its corresponding element in matrix B. See “Example 2 – Nonscalar Expression” on page 2-3445, below.

Partial Evaluation of the Expression Argument

Within the context of an if or while expression, MATLAB does not necessarily evaluate all parts of a logical expression. In some cases it is possible, and often advantageous, to determine whether an expression is true or false through only partial evaluation.

For example, if A equals zero in statement 1 below, then the expression evaluates to false, regardless of the value of B. In this case, there is no need to evaluate B and MATLAB does not do so. In statement 2, if A is nonzero, then the expression is true, regardless of B. Again, MATLAB does not evaluate the latter part of the expression.

```
1) while (A && B)           2) while (A || B)
```

You can use this property to your advantage to cause MATLAB to evaluate a part of an expression only if a preceding part evaluates to the desired state. Here are some examples.

```
while (b ~= 0) && (a/b > 18.5)
    if exist('myfun.m') && (myfun(x) >= y)
        if iscell(A) && all(cellfun('isreal', A))
```

Empty Arrays

In most cases, using while on an empty array returns false. There are some conditions however under which while evaluates as true on an empty array. Two examples of this are

```
A = [];
while all(A), do_something, end
while 1|A, do_something, end
```

Short-Circuiting Behavior

When used in the context of a while or if expression, and only in this context, the element-wise | and & operators use short-circuiting in

evaluating their expressions. That is, $A|B$ and $A\&B$ ignore the second operand, B , if the first operand, A , is sufficient to determine the result.

See “Short-Circuiting in Elementwise Operators” for more information on this.

Examples

Example 1 – Simple while Statement

The variable `eps` is a tolerance used to determine such things as near singularity and rank. Its initial value is the *machine epsilon*, the distance from 1.0 to the next largest floating-point number on your machine. Its calculation demonstrates while loops.

```
eps = 1;
while (1+eps) > 1
    eps = eps/2;
end
eps = eps*2
```

This example is for the purposes of illustrating while loops only and should not be executed in your MATLAB session. Doing so will disable the `eps` function from working in that session.

Example 2 – Nonscalar Expression

Given matrices A and B ,

```
A =      B =
     1     0     1     1
     2     3     3     4
```

Expression	Evaluates As	Because
$A < B$	false	$A(1,1)$ is not less than $B(1,1)$.
$A < (B + 1)$	true	Every element of A is less than that same element of B with 1 added.

while

Expression	Evaluates As	Because
A & B	false	A(1,2) is false, and B is ignored due to short-circuiting.
B < 5	true	Every element of B is less than 5.

See Also

end, for, break, continue, return, all, any, if, switch

Purpose	Change axes background color
Syntax	<code>whitebg</code> <code>whitebg(fig)</code> <code>whitebg(ColorSpec)</code> <code>whitebg(fig, ColorSpec)</code> <code>whitebg(fig)</code>
Description	<p><code>whitebg</code> complements the colors in the current figure.</p> <p><code>whitebg(fig)</code> complements colors in all figures specified in the vector <code>fig</code>.</p> <p><code>whitebg(ColorSpec)</code> and <code>whitebg(fig, ColorSpec)</code> change the color of the axes, which are children of the figure, to the color specified by <code>ColorSpec</code>. Without a figure specification, <code>whitebg</code> or <code>whitebg(ColorSpec)</code> affects the current figure and the root's default properties so subsequent plots and new figures use the new colors.</p> <p><code>whitebg(fig, ColorSpec)</code> sets the default axes background color of the figures in the vector <code>fig</code> to the color specified by <code>ColorSpec</code>. Other axes properties and the figure background color can change as well so that graphs maintain adequate contrast. <code>ColorSpec</code> can be a 1-by-3 RGB color or a color string such as 'white' or 'w'.</p> <p><code>whitebg(fig)</code> complements the colors of the objects in the specified figures. This syntax is typically used to toggle between black and white axes background colors, and is where <code>whitebg</code> gets its name. Include the root window handle (0) in <code>fig</code> to affect the default properties for new windows or for <code>clf</code> reset.</p>
Remarks	<p><code>whitebg</code> works best in cases where all the axes in the figure have the same background color.</p> <p><code>whitebg</code> changes the colors of the figure's children, with the exception of shaded surfaces. This ensures that all objects are visible against the new background color. <code>whitebg</code> sets the default properties on the root such that all subsequent figures use the new background color.</p>

whitebg

Examples

Set the background color to blue-gray.

```
whitebg([0 .5 .6])
```

Set the background color to blue.

```
whitebg('blue')
```

See Also

ColorSpec, colordef

The figure graphics object property `InvertHardCopy`

“Color Operations” on page 1-95 for related functions

Purpose

List variables in workspace

Graphical Interface

As an alternative to whos, use the Workspace browser. Or use the Current Directory browser to view the contents of MAT-files without loading them.

Syntax

```
who
whos
who(variable_list)
whos(variable_list)
who(variable_list, qualifiers)
whos(variable_list, qualifiers)
s = who(variable_list, qualifiers)
s = whos(variable_list, qualifiers)
who variable_list qualifiers
whos variable_list qualifiers
```

Each of these syntaxes apply to both who and whos:

Description

who lists in alphabetical order all variables in the currently active workspace.

whos lists in alphabetical order all variables in the currently active workspace along with their sizes and types. It also reports the totals for sizes.

Note If who or whos is executed within a nested function, MATLAB lists the variables in the workspace of that function and in the workspaces of all functions containing that function. See the Remarks section, below.

who(variable_list) and whos(variable_list) list only those variables specified in variable_list, where variable_list is a comma-delimited list of quoted strings: 'var1', 'var2', ..., 'varN'. You can use the wildcard character * to display variables that

who, whos

match a pattern. For example, `who('A*')` finds all variables in the current workspace that start with A.

`who(variable_list, qualifiers)` and `whos(variable_list, qualifiers)` list those variables in `variable_list` that meet all qualifications specified in `qualifiers`. You can specify any or all of the following qualifiers, and in any order.

Qualifier Syntax	Description	Example
'global'	List variables in the global workspace.	<code>whos('global')</code>
'-file', filename	List variables in the specified MAT-file. Use the full path for filename.	<code>whos('-file', 'mydata')</code>
'-regexp', explist	List variables that match any of the regular expressions in <code>explist</code> .	<code>whos('-regexp', '[AB].', '\w\d')</code>

`s = who(variable_list, qualifiers)` returns cell array `s` containing the names of the variables specified in `variable_list` that meet the conditions specified in `qualifiers`.

`s = whos(variable_list, qualifiers)` returns structure `s` containing the following fields for the variables specified in `variable_list` that meet the conditions specified in `qualifiers`:

Field Name	Description
name	Name of the variable
size	Dimensions of the variable array
bytes	Number of bytes allocated for the variable array
class	Class of the variable. Set to the string '(unassigned)' if the variable has no value.

Field Name	Description
global	True if the variable is global; otherwise false
sparse	True if the variable is sparse; otherwise false
complex	True if the variable is complex; otherwise false
nesting	Structure having the following fields: <ul style="list-style-type: none"> function — Name of the nested or outer function that defines the variable level — Nesting level of that function
persistent	True if the variable is persistent; otherwise false

`who variable_list` qualifiers and `whos variable_list` qualifiers are the unquoted forms of the syntax. Both `variable_list` and `qualifiers` are space-delimited lists of unquoted strings.

Remarks

Nested Functions.

When you use `who` or `whos` inside of a nested function, MATLAB returns or displays all variables in the workspace of that function, and in the workspaces of all functions in which that function is nested. This applies whether you include calls to `who` or `whos` in your M-file code or if you call `who` or `whos` from the MATLAB debugger.

If your code assigns the output of `whos` to a variable, MATLAB returns the information in a structure array containing the fields described above. If you do not assign the output to a variable, MATLAB displays the information at the Command Window, grouped according to workspace.

If your code assigns the output of `who` to a variable, MATLAB returns the variable names in a cell array of strings. If you do not assign the output, MATLAB displays the variable names at the Command Window, but not grouped according to workspace.

Compressed Data.

Information returned by the command `whos -file` is independent of whether the data in that file is compressed or not. The byte counts returned by this command represent the number of bytes data occupies in the MATLAB workspace, and not in the file the data was saved to. See the function reference for `save` for more information on data compression.

MATLAB Objects.

`whos -file filename` does not return the sizes of any MATLAB objects that are stored in file *filename*.

Examples

Example 1

Show variable names starting with the letter a:

```
who a*
```

Show variables stored in MAT-file `mydata.mat`:

```
who -file mydata
```

Example 2

Return information on variables stored in file `mydata.mat` in structure array `s`:

```
s = whos('-file', 'mydata1')
s =
6x1 struct array with fields:
    name
    size
    bytes
    class
    global
    sparse
    complex
    nesting
```

```
persistent
```

Display the name, size, and class of each of the variables returned by whos:

```
for k=1:length(s)
disp([' ' s(k).name ' ' mat2str(s(k).size) ' ' s(k).class])
end
A [1 1] double
spArray [5 5] double
strArray [2 5] cell
x [3 2 2] double
y [4 5] cell
```

Example 3

Show variables that start with java and end with Array. Also show their dimensions and class name:

```
whos -file mydata2 -regexp \<java.*Array\>
Name           Size           Bytes  Class

javaCharArray  3x1             24     java.lang.String[][][]
javaDblArray   4x1             32     java.lang.Double[][]
javaIntArray   14x1            112     java.lang.Integer[][]
```

Example 4

The function shown here uses variables with persistent, global, sparse, and complex attributes:

```
function show_attributes
persistent p;
global g;
o = 1; g = 2;
s = sparse(eye(5));
c = [4+5i 9-3i 7+6i];
whos
```

who, whos

When the function is run, whos displays these attributes:

show_attributes					
Name	Size	Bytes	Class	Attributes	
c	1x3	48	double	complex	
g	1x1	8	double	global	
p	1x1	8	double	persistent	
s	5x5	84	double	sparse	

Example 5

Function whos_demo contains two nested functions. One of these functions calls whos; the other calls who:

```
function whos_demo
date_time = datestr(now);

[str pos] = textscan(date_time, '%s%s%s', ...
                    1, 'delimiter', '- :');
get_date(str);

str = textscan(date_time(pos+1:end), '%s%s%s', ...
              1, 'delimiter', '- :');
get_time(str);

function get_date(d)
    day = d{1};    mon = d{2};    year = d{3};
    whos
end
function get_time(t)
    hour = t{1};    min = t{2};    sec = t{3};
    who
end
end
```

When nested function get_date calls whos, MATLAB displays information on the variables in all workspaces that are in scope at the

time. This includes nested function `get_date` and also the function in which it is nested, `whos_demo`. The information is grouped by workspace:

```
whos_demo
  Name          Size          Bytes  Class
-----
---- get_date -----
d              1x3              378  cell
day            1x1              64  cell
mon            1x1              66  cell
year           1x1              68  cell

---- whos_demo -----
ans            0x0              0  (unassigned)
date_time      1x20             40  char
pos            1x1              8  double
str            1x3             378  cell
```

When nested function `get_time` calls `who`, MATLAB displays names of the variables in the workspaces that are in scope at the time. This includes nested function `get_time` and also the function in which it is nested, `whos_demo`. The information is not grouped by workspace in this case:

Your variables are:

```
hour      min      sec      t      ans      date_time
pos       str
```

See Also

`assignin`, `clear`, `computer`, `dir`, `evalin`, `exist`, `inmem`, `load`, `save`, `what`, `workspace`

wilkinson

Purpose Wilkinson's eigenvalue test matrix

Syntax `W = wilkinson(n)`

Description `W = wilkinson(n)` returns one of J. H. Wilkinson's eigenvalue test matrices. It is a symmetric, tridiagonal matrix with pairs of nearly, but not exactly, equal eigenvalues.

Examples `wilkinson(7)`

```
ans =  
  
     3     1     0     0     0     0     0  
     1     2     1     0     0     0     0  
     0     1     1     1     0     0     0  
     0     0     1     0     1     0     0  
     0     0     0     1     1     1     0  
     0     0     0     0     1     2     1  
     0     0     0     0     0     1     3
```

The most frequently used case is `wilkinson(21)`. Its two largest eigenvalues are both about 10.746; they agree to 14, but not to 15, decimal places.

See Also `eig`, `gallery`, `pascal`

Purpose Open file in appropriate application (Windows)

Syntax `winopen(filename)`

Description `winopen(filename)` opens `filename` in the appropriate Microsoft Windows application. The `filename` input is a string enclosed in single quotes. The `winopen` function uses the appropriate Windows shell command, and performs the same action as if you double-click the file in the Windows Explorer. If `filename` is not in the current directory, specify the absolute path for `filename`.

Examples Open the file `thesis.doc`, located in the current directory, in Microsoft Word:

```
winopen('thesis.doc')
```

Open `myresults.html` in your system's default Web browser:

```
winopen('D:/myfiles/myresults.html')
```

See Also `dos`, `open`, `web`

winqueryreg

Purpose Item from Microsoft Windows registry

Syntax

```
valnames = winqueryreg('name', 'rootkey', 'subkey')  
value = winqueryreg('rootkey', 'subkey', 'valname')  
value = winqueryreg('rootkey', 'subkey')
```

Description `valnames = winqueryreg('name', 'rootkey', 'subkey')` returns all value names in `rootkey\subkey` in a cell array of strings. The first argument is the literal quoted string, 'name'.

`value = winqueryreg('rootkey', 'subkey', 'valname')` returns the value for value name `valname` in `rootkey\subkey`.

If the value retrieved from the registry is a string, `winqueryreg` returns a string. If the value is a 32-bit integer, `winqueryreg` returns the value as an integer of MATLAB type `int32`.

`value = winqueryreg('rootkey', 'subkey')` returns a value in `rootkey\subkey` that has no value name property.

Note The literal **name** argument and the `rootkey` argument are case-sensitive. The `subkey` and `valname` arguments are not.

Remarks This function works only for the following registry value types:

- strings (`REG_SZ`)
- expanded strings (`REG_EXPAND_SZ`)
- 32-bit integer (`REG_DWORD`)

Examples **Example 1**

Get the value of CLSID for the MATLAB sample COM control `mwsampctrl.2`:

```
winqueryreg 'HKEY_CLASSES_ROOT' 'mwsamp.mwsampctrl.2\clsid'
```

```
ans =  
    {5771A80A-2294-4CAC-A75B-157DCDDD3653}
```

Example 2

Get a list in variable `mousechar` for registry subkey `Mouse`, which is under subkey `Control Panel`, which is under root key `HKEY_CURRENT_USER`.

```
mousechar = winqueryreg('name', 'HKEY_CURRENT_USER', ...  
    'control panel\mouse');
```

For each name in the `mousechar` list, get its value from the registry and then display the name and its value:

```
for k=1:length(mousechar)  
    setting = winqueryreg('HKEY_CURRENT_USER', ...  
        'control panel\mouse', mousechar{k});  
    str = sprintf('%s = %s', mousechar{k}, num2str(setting));  
    disp(str)  
end
```

```
ActiveWindowTracking = 0  
DoubleClickHeight = 4  
DoubleClickSpeed = 830  
DoubleClickWidth = 4  
MouseSpeed = 1  
MouseThreshold1 = 6  
MouseThreshold2 = 10  
SnapToDefaultButton = 0  
SwapMouseButton = 0
```

wk1finfo

Purpose Determine whether file contains 1-2-3 WK1 worksheet

Syntax `[extens, typ] = wk1finfo(filename)`

Description `[extens, typ] = wk1finfo(filename)` returns the string 'WK1' in `extens`, and ' 1-2-3 Spreadsheet' in `typ` if the file `filename` contains a readable worksheet. The `filename` input is a string enclosed in single quotes.

Examples This example returns information on spreadsheet file `matA.wk1`:

```
[extens, typ] = wk1finfo('matA.wk1')

extens =
    WK1
typ =
    123 Spreadsheet
```

See Also `wk1read`, `wk1write`, `csvread`, `csvwrite`

Purpose

Read Lotus1-2-3 WK1 spreadsheet file into matrix

Syntax

```
M = wk1read(filename)
M = wk1read(filename,r,c)
M = wk1read(filename,r,c,range)
```

Description

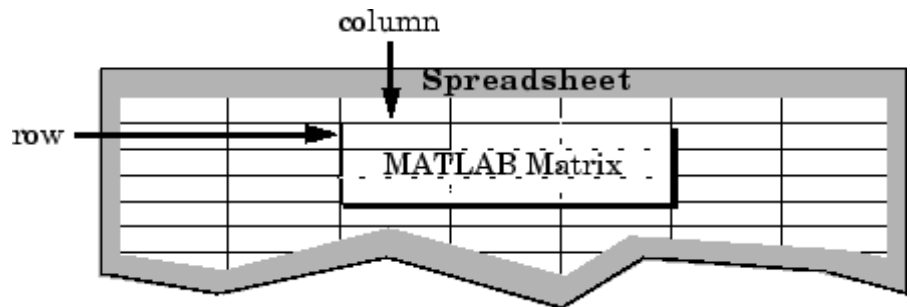
`M = wk1read(filename)` reads a Lotus1-2-3 WK1 spreadsheet file into the matrix `M`. The `filename` input is a string enclosed in single quotes.

`M = wk1read(filename,r,c)` starts reading at the row-column cell offset specified by (r,c) . r and c are zero based so that $r=0, c=0$ specifies the first value in the file.

`M = wk1read(filename,r,c,range)` reads the range of values specified by the parameter `range`, where `range` can be

- A four-element vector specifying the cell range in the format

```
[upper_left_row upper_left_col lower_right_row lower_right_col]
```



- A cell range specified as a string, for example, 'A1...C5'
- A named range specified as a string, for example, 'Sales'

Examples

Create a 8-by-8 matrix `A` and export it to Lotus spreadsheet `matA.wk1`:

```
A = [1:8; 11:18; 21:28; 31:38; 41:48; 51:58; 61:68; 71:78]
A =
```

wk1read

```
    1     2     3     4     5     6     7     8
  11    12    13    14    15    16    17    18
  21    22    23    24    25    26    27    28
  31    32    33    34    35    36    37    38
  41    42    43    44    45    46    47    48
  51    52    53    54    55    56    57    58
  61    62    63    64    65    66    67    68
  71    72    73    74    75    76    77    78
```

```
wk1write('matA.wk1', A);
```

To read in a limited block of the spreadsheet data, specify the upper left row and column of the block using zero-based indexing:

```
M = wk1read('matA.wk1', 3, 2)
M =
    33    34    35    36    37    38
    43    44    45    46    47    48
    53    54    55    56    57    58
    63    64    65    66    67    68
    73    74    75    76    77    78
```

To select a more restricted block of data, you can specify both the upper left and lower right corners of the block you want imported. Read in a range of values from row 4, column 3 (defining the upper left corner) to row 6, column 6 (defining the lower right corner). Note that, unlike the second and third arguments, the range argument [4 3 6 6] is one-based:

```
M = wk1read('matA.wk1', 3, 2, [4 3 6 6])
M =
    33    34    35    36
    43    44    45    46
    53    54    55    56
```

See Also

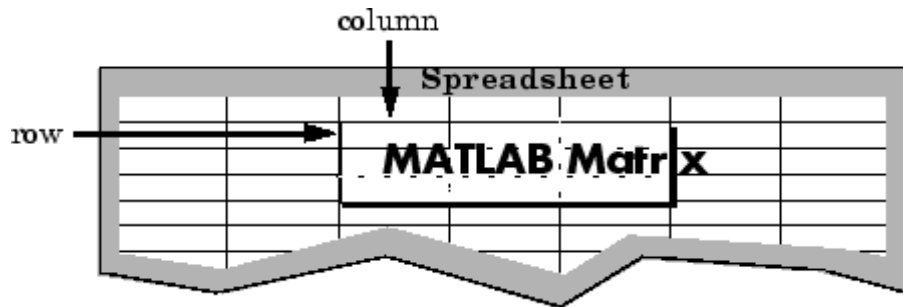
wk1write

Purpose Write matrix to Lotus 1-2-3 WK1 spreadsheet file

Syntax
`wk1write(filename,M)`
`wk1write(filename,M,r,c)`

Description `wk1write(filename,M)` writes the matrix `M` into a Lotus1-2-3 WK1 spreadsheet file named `filename`. The `filename` input is a string enclosed in single quotes.

`wk1write(filename,M,r,c)` writes the matrix starting at the spreadsheet location `(r,c)`. `r` and `c` are zero based so that `r=0, c=0` specifies the first cell in the spreadsheet.



Examples Write a 4-by-5 matrix `A` to spreadsheet file `matA.wk1`. Place the matrix with its upper left corner at row 2, column 3 using zero-based indexing:

```
A = [1:5; 11:15; 21:25; 31:35]
```

```
A =
     1     2     3     4     5
    11    12    13    14    15
    21    22    23    24    25
    31    32    33    34    35
```

```
wk1write('matA.wk1', A, 2, 3)
```

```
M = wk1read('matA.wk1')
```

```
M =
```

wk1write

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 2 3 4 5
0 0 0 11 12 13 14 15
0 0 0 21 22 23 24 25
0 0 0 31 32 33 34 35
```

See Also

wk1read, dlmwrite, dlmread, csvwrite, csvread

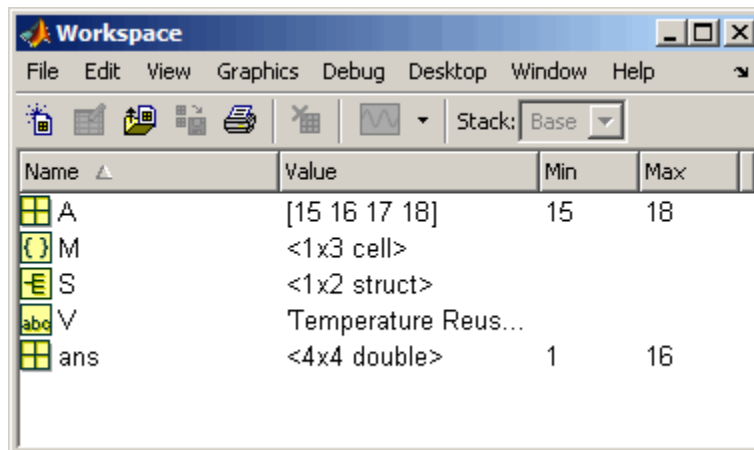
Purpose Open Workspace browser to manage workspace

GUI Alternatives As an alternative to the workspace function, select **Desktop > Workspace** in the MATLAB desktop.

Syntax workspace

Description workspace displays the Workspace browser, a graphical user interface that allows you to view and manage the contents of the MATLAB workspace. It provides a graphical representation of the whos display, and allows you to perform the equivalent of the clear, load, open, and save functions.

The Workspace browser also displays and automatically updates statistical calculations for each variable that you can choose to show or hide.



You can edit the value directly in the Workspace browser for small numeric and character arrays. To see and edit a graphical representation of larger variables and for other types, double-click the variable in the Workspace browser. The variable displays in the Array Editor, where you can view the full contents and edit it.

workspace


See Also

who

Purpose

Label x -, y -, and z -axis

GUI Alternative

To control the presence and appearance of axis labels on a graph, use the Property Editor, one of the plotting tools . For details, see The Property Editor in the MATLAB Graphics documentation.

Syntax

```
xlabel('string')  
xlabel(fname)  
xlabel(..., 'PropertyName', PropertyValue, ...)  
xlabel(axes_handle, ...)  
h = xlabel(...)
```

```
ylabel(...)  
ylabel(axes_handle, ...)  
h = ylabel(...)
```

```
zlabel(...)  
zlabel(axes_handle, ...)  
h = zlabel(...)
```

Description

Each axes graphics object can have one label for the x -, y -, and z -axis. The label appears beneath its respective axis in a two-dimensional plot and to the side or beneath the axis in a three-dimensional plot.

`xlabel('string')` labels the x -axis of the current axes.

`xlabel(fname)` evaluates the function `fname`, which must return a string, then displays the string beside the x -axis.

`xlabel(..., 'PropertyName', PropertyValue, ...)` specifies property name and property value pairs for the text graphics object created by `xlabel`.

xlabel, ylabel, zlabel

`xlabel(axes_handle,...)`, `ylabel(axes_handle,...)`, and `zlabel(axes_handle,...)` plot into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = xlabel(...)`, `h = ylabel(...)`, and `h = zlabel(...)` return the handle to the text object used as the label.

`ylabel(...)` and `zlabel(...)` label the *y*-axis and *z*-axis, respectively, of the current axes.

Remarks

Reissuing an `xlabel`, `ylabel`, or `zlabel` command causes the new label to replace the old label.

For three-dimensional graphics, MATLAB puts the label in the front or side, so that it is never hidden by the plot.

Examples

Create a multiline label for the *x*-axis using a multiline cell array:

```
xlabel({'first line';'second line'})
```

Create a bold label for the *y*-axis that contains a single quote:

```
ylabel('George' 's Popularity','fontsize',12,'fontweight','b')
```

See Also

`strings`, `text`, `title`


“Annotating Plots” on page 1-83 for related functions

Adding Axis Labels to Graphs for more information about labeling axes

Purpose

Set or query axis limits

GUI Alternative

To control the upper and lower axis limits on a graph, use the Property Editor, one of the plotting tools . For details, see The Property Editor in the MATLAB Graphics documentation.

Syntax

```
xlim
xlim([xmin xmax])
xlim('mode')
xlim('auto')
xlim('manual')
xlim(axes_handle,...)
```

Note that the syntax for each of these three functions is the same; only the `xlim` function is used for simplicity. Each operates on the respective x -, y -, or z -axis.

Description

`xlim` with no arguments returns the respective limits of the current axes.

`xlim([xmin xmax])` sets the axis limits in the current axes to the specified values.

`xlim('mode')` returns the current value of the axis limits mode, which can be either `auto` (the default) or `manual`.

`xlim('auto')` sets the axis limit mode to `auto`.

`xlim('manual')` sets the respective axis limit mode to `manual`.

`xlim(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, these functions operate on the current axes.

Remarks

`xlim`, `ylim`, and `zlim` set or query values of the axes object `XLim`, `YLim`, `ZLim`, and `XLimMode`, `YLimMode`, `ZLimMode` properties.

When the axis limit modes are `auto` (the default), MATLAB uses limits that span the range of the data being displayed and are round numbers.

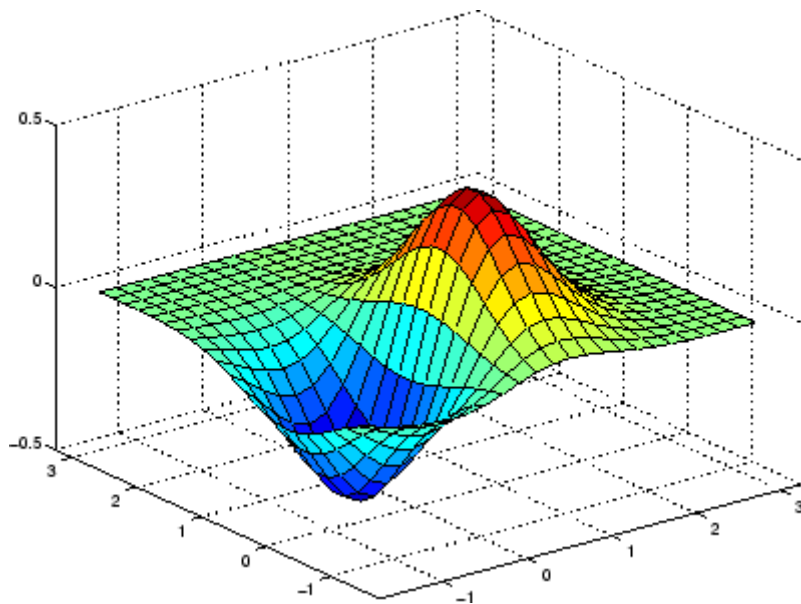
xlim, ylim, zlim

Setting a value for any of the limits also sets the corresponding mode to manual. Note that high-level plotting functions like `plot` and `surf` reset both the modes and the limits. If you set the limits on an existing graph and want to maintain these limits while adding more graphs, use the `hold` command.

Examples

This example illustrates how to set the x - and y -axis limits to match the actual range of the data, rather than the rounded values of $[-2\ 3]$ for the x -axis and $[-2\ 4]$ for the y -axis originally selected by MATLAB.

```
[x,y] = meshgrid([-1.75:.2:3.25]);  
z = x.*exp(-x.^2-y.^2);  
surf(x,y,z)  
xlim([-1.75 3.25])  
ylim([-1.75 3.25])
```



See Also

axis

The axes properties XLim, YLim, ZLim

“Setting the Aspect Ratio and Axis Limits” on page 1-97 for related functions

Understanding Axes Aspect Ratio for more information on how axis limits affect the axes

xlsinfo

Purpose Determine whether file contains Microsoft Excel (.xls) spreadsheet

Syntax

```
typ = xlsinfo(filename)
[typ, desc] = xlsinfo(filename)
[typ, desc, fmt] = xlsinfo(filename)
xlsinfo filename
```

Description

`typ = xlsinfo(filename)` returns the string 'Microsoft Excel Spreadsheet' if the file specified by `filename` is an XLS file that can be read by the MATLAB `xlsread` function. Otherwise, `typ` is the empty string, (''). The `filename` input is a string enclosed in single quotes.

`[typ, desc] = xlsinfo(filename)` returns in `desc` a cell array of strings containing the names of each spreadsheet in the file. If a spreadsheet is unreadable, the cell in `desc` that represents that spreadsheet contains an error message.

`[typ, desc, fmt] = xlsinfo(filename)` returns in the `fmt` output a string containing the actual format of the file as obtained from the Excel COM server. On UNIX systems, or on Windows when the COM server is not available, `fmt` is returned as an empty string, ('').

Note In the case where an Excel COM server cannot be started, functionality is limited in that some Excel files might not be readable.

`xlsinfo filename` is the command format for `xlsinfo`. It returns only the first output, `typ`, assigning it to the MATLAB default variable `ans`.

Examples Get information about an .xls file:

```
[typ, desc, fmt] = xlsinfo('myaccount.xls')

typ =
    Microsoft Excel Spreadsheet
```



```
desc =  
    'Sheet1'    'Income'    'Expenses'  
  
fmt =  
    xlWorkbookNormal
```

Export the .xls file to comma-separated value (CSV) format. Use `xlsinfo` to see the format of the exported file:

```
[typ, desc, fmt] = xlsinfo('myaccount.csv');  
fmt  
  
fmt =  
    xlCSV
```

Export the .xls file to HTML format. `xlsinfo` returns the following format string:

```
[typ, desc, fmt] = xlsinfo('myaccount.html');  
fmt  
  
fmt =  
    xlHtml
```

Export the .xls file to XML format. `xlsinfo` returns the following format string:

```
[typ, desc, fmt] = xlsinfo('myaccount.xml');  
fmt  
  
fmt =  
    xlXMLSpreadsheet
```

See Also`xlsread`, `xlswrite`

xlsread

Purpose Read Microsoft Excel spreadsheet file (.xls)

Syntax

```
num = xlsread(filename)
num = xlsread(filename, -1)
num = xlsread(filename, sheet)
num = xlsread(filename, 'range')
num = xlsread(filename, sheet, 'range')
num = xlsread(filename, sheet, 'range', 'basic')
num = xlsread(filename, ..., functionhandle)
[num, txt]= xlsread(filename, ...)
[num, txt, raw] = xlsread(filename, ...)
[num, txt, raw, X] = xlsread(filename, ..., functionhandle)
xlsread filename sheet range basic
```

Description num = xlsread(filename) returns numeric data in double array num from the first sheet in the Microsoft Excel spreadsheet file named filename. The filename input is a string enclosed in single quotes. The xlsread ignores leading rows or columns of text. However, if a cell not in a leading row or column is empty or contains text, xlsread puts a NaN in its place in the return array, num.

The full functionality of xlsread depends on the ability to start Excel as a COM server from MATLAB. If your system does not have this capability, the xlsread syntax that passes the 'basic' keyword is recommended. As long as the COM server is available, you can use xlsread on Excel files having formats other than XLS (for example, HTML).

Note xlsread on UNIX is being grandfathered. If the Excel COM server is not available, xlsread reads only strictly XLS files. It cannot read Excel files saved in HTML or other formats.

num = xlsread(filename, -1) opens the file filename in an Excel window, enabling you to interactively select the worksheet to be read and the range of data on that worksheet to import. To import an entire

worksheet, first select the sheet in the Excel window and then click the **OK** button in the Data Selection Dialog box. To import a certain range of data from the sheet, select the worksheet in the Excel window, drag and drop the mouse over the desired range, and then click **OK**. (See “COM Server Requirements” on page 2-3477 below.)

`num = xlsread(filename, sheet)` reads the specified worksheet, where `sheet` is either a positive, double scalar value or a quoted string containing the sheet name. To determine the names of the sheets in a spreadsheet file, use `xlsfinfo`.

`num = xlsread(filename, 'range')` reads data from a specific rectangular region of the default worksheet (Sheet1). Specify range using the syntax `'C1:C2'`, where `C1` and `C2` are two opposing corners that define the region to be read. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The range input is not case sensitive and uses Excel A1 notation. (See help in Excel for more information on this notation.) (Also, see “COM Server Requirements” on page 2-3477 below.)

`num = xlsread(filename, sheet, 'range')` reads data from a specific rectangular region (`range`) of the worksheet specified by `sheet`. See the previous two syntax formats for further explanation of the `sheet` and `range` inputs. (See “COM Server Requirements” on page 2-3477 below.)

`num = xlsread(filename, sheet, 'range', 'basic')` imports data from the spreadsheet in basic import mode. This is the mode used on UNIX platforms as well as on Windows when Excel is not available as a COM server. In this mode, `xlsread` does not use Excel as a COM server, and this limits import ability. Without Excel as a COM server, `range` is ignored and, consequently, the whole active range of a sheet is imported. (You can set `range` to the empty string (`''`)). Also, in basic mode, `sheet` is case-sensitive and must be a quoted string.

`num = xlsread(filename, ..., functionhandle)` calls the function associated with `functionhandle` just prior to obtaining spreadsheet values. This enables you to operate on the spreadsheet data (for

example, convert it to a numeric type) before reading it in. (See “COM Server Requirements” on page 2-3477 below.)

You can write your own custom function and pass a handle to this function to `xlsread`. When `xlsread` executes, it reads from the spreadsheet, executes your function on the data read from the spreadsheet, and returns the final results to you. When `xlsread` calls your function, it passes a range interface from Excel to provide access to the data read from the spreadsheet. Your function must include this interface both as an input and output argument. Example 5 below shows how you might use this syntax.

`[num, txt]= xlsread(filename, ...)` returns numeric data in array `num` and text data in cell array `txt`. All cells in `txt` that correspond to numeric data contain the empty string.

If `txt` includes data that was previously written to the file using `xlswrite`, and the range specified for that `xlswrite` operation caused undefined data ('#N/A') to be written to the worksheet, then cells containing that undefined data are represented in the `txt` output as 'ActiveX_VT_ERROR: '.

`[num, txt, raw] = xlsread(filename, ...)` returns numeric and text data in `num` and `txt`, and unprocessed cell content in cell array `raw`, which contains both numeric and text data. (See “COM Server Requirements” on page 2-3477 below.)

`[num, txt, raw, X] = xlsread(filename, ..., functionhandle)` calls the function associated with `functionhandle` just prior to reading from the spreadsheet file. This syntax returns one additional output `X` from the function mapped to by `functionhandle`. Example 6 below shows how you might use this syntax. (See “COM Server Requirements” on page 2-3477 below.)

`xlsread filename sheet range basic` is the command format for `xlsread`, showing its usage with all input arguments specified. When using this format, you must specify `sheet` as a string, (for example, `Income` or `Sheet4`) and not a numeric index. If the `sheet` name contains space characters, then quotation marks are required around the string, (for example, `'Income 2002'`).

Remarks

Handling Excel Date Values

MATLAB imports date fields from Excel files in the format in which they were stored in the Excel file. If stored in string or date format, `xlsread` returns the date as a string. If stored in a numeric format, `xlsread` returns a numeric date.

Both Excel and MATLAB represent numeric dates as a number of serial days elapsed from a specific reference date. However, Excel uses January 1, 1900 as the reference date while MATLAB uses January 0, 0000. Due to this difference in the way Excel and MATLAB compute numeric date values, any numeric date imported from Excel into MATLAB must first be converted before being used in the MATLAB application.

You can do this conversion after the `xlsread` completes, as shown below:

```
excelDates = xlsread(filename)
matlabDates = datenum('30-Dec-1899') + excelDates
datestr(matlabDates,2)
```

You can also do this as part of the `xlsread` operation by writing a conversion routine that acts directly on the Excel COM Range object, and then passing a function handle for your routine as an input to `xlsread`. The description above for the following syntax, along with Examples 5 and 6, explain how to do this:

```
[num, txt, raw, X] = xlsread(filename, ..., functionhandle)
```

COM Server Requirements

The following six syntax formats are supported only on computer systems capable of starting Excel as a COM server from MATLAB. They are not supported in basic mode.

```
num = xlsread(filename, -1)
num = xlsread(filename, 'range')
num = xlsread(filename, sheet, 'range')
[num, txt, raw] = xlsread(filename, ...)
num = xlsread(filename, ..., functionhandle)
```

```
[num, txt, raw, opt] = xlsread(filename, ...,  
functionhandle)
```

Examples

Example 1 – Reading Numeric Data

The Microsoft Excel spreadsheet file `testdata1.xls` contains this data:

```
1    6  
2    7  
3    8  
4    9  
5   10
```

To read this data into MATLAB, use this command:

```
A = xlsread('testdata1.xls')  
A =  
1    6  
2    7  
3    8  
4    9  
5   10
```

Example 2 – Handling Text Data

The Microsoft Excel spreadsheet file `testdata2.xls` contains a mix of numeric and text data:

```
1    6  
2    7  
3    8  
4    9  
5   text
```

`xlsread` puts a NaN in place of the text data in the result:

```
A = xlsread('testdata2.xls')  
A =  
1    6
```

```
2    7
3    8
4    9
5    NaN
```

Example 3 – Selecting a Range of Data

To import only rows 4 and 5 from worksheet 1, specify the range as 'A4:B5':

```
A = xlsread('testdata2.xls', 1, 'A4:B5')
```

```
A =
     4     9
     5    NaN
```

Example 4 – Handling Files with Row or Column Headers

A Microsoft Excel spreadsheet labeled Temperatures in file tempdata.xls contains two columns of numeric data with text headers for each column:

```
Time  Temp
12    98
13    99
14    97
```

If you want to import only the numeric data, use xlsread with a single return argument. Specify the filename and sheet name as inputs.

xlsread ignores any leading row or column of text in the numeric result.

```
ndata = xlsread('tempdata.xls', 'Temperatures')
```

```
ndata =
     12     98
     13     99
     14     97
```

To import both the numeric data and the text data, specify two return values for `xlsread`:

```
[ndata, headertext] = xlsread('tempdata.xls', 'Temperatures')

ndata =
    12    98
    13    99
    14    97

headertext =
    'Time'    'Temp'
```

Example 5 – Passing a Function Handle

This example calls `xlsread` twice, the first time as a simple read from a file, and the second time requesting that `xlsread` execute some user-defined modifications on the data prior to returning the results of the read. These modifications are performed by a user-written function, `setMinMax`, that you pass as a function handle in the call to `xlsread`. When `xlsread` executes, it reads from the spreadsheet, executes the function on the data read from the spreadsheet, and returns the final results to you.

Note The function passed to `xlsread` operates on the copy of the data read from the spreadsheet. It does not modify data in the spreadsheet itself.

Read a 10-by-3 numeric array from Excel spreadsheet `testsheet.xls`. with a simple `xlsread` statement that does not pass a function handle. Note that the values returned range from -587 to +4,149:

```
arr = xlsread('testsheet.xls')
arr =
    1.0e+003 *
    1.0020    4.1490    0.2300
```



```
1.0750    0.1220   -0.4550
-0.0301    3.0560    0.2471
0.4070    0.1420   -0.2472
2.1160   -0.0557   -0.5870
0.4040    2.9280    0.0265
0.1723    3.4440    0.1112
4.1180    0.1820    2.8630
0.9000    0.0573    1.9750
0.0163    0.2000   -0.0223
```

In preparation for the second part of this example, write a function `setMinMax` that restricts the values returned from the read to be in the range of 0 to 2000. You will need to pass this function in the call to `xlsread` which will then execute the function on the data it has read before returning it to you.

When `xlsread` calls your function, it passes a range interface from Excel to provide access to the data read from the spreadsheet. This is shown as `DataRange` in this example. Your function must include this interface both as an input and output argument. The output argument allows your function to pass modified data back to `xlsread`:

```
function [DataRange] = setMinMax(DataRange)
    maxval = 2000;    minval = 0;

    for k = 1:DataRange.Count
        v = DataRange.Value{k};
        if v > maxval || v < minval
            if v > maxval
                DataRange.Value{k} = maxval;
            else
                DataRange.Value{k} = minval;
            end
        end
    end
end
```

Now call `xlsread`, passing a function handle for the `setMinMax` function as the final argument. Note the changes from the values returned from the last call to `xlsread`:

```
arr = xlsread('testsheet.xls', '', '', '', @setMinMax)
arr =
    1.0e+003 *
    1.0020    2.0000    0.2300
    1.0750    0.1220         0
         0    2.0000    0.2471
    0.4070    0.1420         0
    2.0000         0         0
    0.4040    2.0000    0.0265
    0.1723    2.0000    0.1112
    2.0000    0.1820    2.0000
    0.9000    0.0573    1.9750
    0.0163    0.2000         0
```

Example 6 – Passing a Function Handle with Additional Output

This example adds onto the previous one by returning an additional output from the call to `setMinMax`. Modify the function so that it not only limits the range of values returned, but also reports which elements of the spreadsheet matrix have been altered. Return this information in a new output argument, `indices`:

```
function [DataRange, indices] = setMinMax(DataRange)
maxval = 2000; minval = 0;
indices = [];

for k = 1:DataRange.Count
    v = DataRange.Value{k};
    if v > maxval || v < minval
        if v > maxval
            DataRange.Value{k} = maxval;
        else
            DataRange.Value{k} = minval;
        end
    end
end
```

```

        indices = [indices k];
    end
end

```

When you call `xlsread` this time, account for the three initial outputs, and add a fourth called `idx` to accept the indices returned from `setMinMax`. Call `xlsread` again, and you will see just where the returned matrix has been modified:

```

[arr txt raw idx] = xlsread('testsheet.xls', ...
                            '', '', '', @setMinMax);

idx
idx =
    3    5    8   11   13   15   16   17   22   24   25   28   30
arr
arr =
    1.0e+003 *
    1.0020    2.0000    0.2300
    1.0750    0.1220     0
     0        2.0000    0.2471
    0.4070    0.1420     0
    2.0000     0         0
    0.4040    2.0000    0.0265
    0.1723    2.0000    0.1112
    2.0000    0.1820    2.0000
    0.9000    0.0573    1.9750
    0.0163    0.2000     0

```

See Also

`xlswrite`, `xlsinfo`, `wk1read`, `textread`, `function_handle`

xlswrite

Purpose Write Microsoft Excel spreadsheet file (.xls)

Syntax

```
xlswrite(filename, M)
xlswrite(filename, M, sheet)
xlswrite(filename, M, 'range')
xlswrite(filename, M, sheet, 'range')
status = xlswrite(filename, ...)
[status, message] = xlswrite(filename, ...)
xlswrite filename M sheet range
```

Description `xlswrite(filename, M)` writes matrix `M` to the Excel file `filename`. The `filename` input is a string enclosed in single quotes. The input matrix `M` is an `m`-by-`n` numeric, character, or cell array, where `m` < 65536 and `n` < 256. The matrix data is written to the first worksheet in the file, starting at cell A1.

`xlswrite(filename, M, sheet)` writes matrix `M` to the specified worksheet `sheet` in the file `filename`. The `sheet` argument can be either a positive, double scalar value representing the worksheet index, or a quoted string containing the sheet name.

If `sheet` does not exist, a new sheet is added at the end of the worksheet collection. If `sheet` is an index larger than the number of worksheets, empty sheets are appended until the number of worksheets in the workbook equals `sheet`. In either case, MATLAB generates a warning indicating that it has added a new worksheet.

`xlswrite(filename, M, 'range')` writes matrix `M` to a rectangular region specified by `range` in the first worksheet of the file `filename`. Specify `range` using one of the following quoted string formats:

- A cell designation, such as 'D2', to indicate the upper left corner of the region to receive the matrix data.
- Two cell designations separated by a colon, such as 'D2:H4', to indicate two opposing corners of the region to receive the matrix data. The range 'D2:H4' represents the 3-by-5 rectangular region between the two corners D2 and H4 on the worksheet.

The range input is not case sensitive and uses Excel A1 notation. (See help in Excel for more information on this notation.)

The size defined by range should fit the size of M or contain only the first cell, (e.g., 'A2'). If range is larger than the size of M, Excel fills the remainder of the region with #N/A. If range is smaller than the size of M, only the submatrix that fits into range is written to the file specified by filename.

`xlswrite(filename, M, sheet, 'range')` writes matrix M to a rectangular region specified by range in worksheet sheet of the file filename. See the previous two syntax formats for further explanation of the sheet and range inputs.

`status = xlswrite(filename, ...)` returns the completion status of the write operation in status. If the write completed successfully, status is equal to logical 1 (true). Otherwise, status is logical 0 (false). Unless you specify an output for xlswrite, no status is displayed in the Command Window.

`[status, message] = xlswrite(filename, ...)` returns any warning or error message generated by the write operation in the MATLAB structure message. The message structure has two fields:

- `message` — String containing the text of the warning or error message
- `identifier` — String containing the message identifier for the warning or error

`xlswrite filename M sheet range` is the command format for xlswrite, showing its usage with all input arguments specified. When using this format, you must specify sheet as a string (for example, Income or Sheet4). If the sheet name contains space characters, then quotation marks are required around the string (for example, 'Income 2002').

Note The above functionality depends upon having Microsoft Excel as a COM server. In absence of Excel, matrix M is written as a text file in Comma-Separated Value (CSV) format. In this mode, the sheet and range arguments are ignored.

Examples

Example 1 – Writing Numeric Data to the Default Worksheet

Write a 7-element vector to Microsoft Excel file `testdata.xls`. By default, the data is written to cells A1 through G1 in the first worksheet in the file:

```
xlswrite('testdata', [12.7 5.02 -98 63.9 0 -.2 56])
```

Example 2 – Writing Mixed Data to a Specific Worksheet

This example writes the following mixed text and numeric data to the file `tempdata.xls`:

```
d = {'Time', 'Temp'; 12 98; 13 99; 14 97};
```

Call `xlswrite`, specifying the worksheet labeled `Temperatures`, and the region within the worksheet to write the data to. The 4-by-2 matrix will be written to the rectangular region that starts at cell E1 in its upper left corner:

```
s = xlswrite('tempdata.xls', d, 'Temperatures', 'E1')
s =
    1
```

The output status `s` shows that the write operation succeeded. The data appears as shown here in the output file:

Time	Temp
12	98
13	99
14	97

Example 3 – Appending a New Worksheet to the File

Now write the same data to a worksheet that doesn't yet exist in `tempdata.xls`. In this case, MATLAB appends a new sheet to the workbook, calling it by the name you supplied in the `sheets` input argument, `'NewTemp'`. MATLAB displays a warning indicating that it has added a new worksheet to the file:

```
xlswrite('tempdata.xls', d, 'NewTemp', 'E1')  
Warning: Added specified worksheet.
```

If you don't want to see these warnings, you can turn them off using the command indicated in the message above:

```
warning off MATLAB:xlswrite:AddSheet
```

Now try the command again, this time creating another new worksheet, `NewTemp2`. Although the message is not displayed this time, you can still retrieve it and its identifier from the second output argument, `m`:

```
[stat msg] = xlswrite('tempdata.xls', d, 'NewTemp2', 'E1');  
  
msg  
msg =  
    message: 'Added specified worksheet.'  
    identifier: 'MATLAB:xlswrite:AddSheet'
```

See Also

`xlsread`, `xlsfinfo`, `wk1read`, `textread`

xmlread

Purpose Parse XML document and return Document Object Model node

Syntax DOMnode = xmlread(filename)

Description DOMnode = xmlread(filename) reads a URL or filename and returns a Document Object Model node representing the parsed document. The filename input is a string enclosed in single quotes. The node can be manipulated by using standard DOM functions.

A properly parsed document displays to the screen as

```
xDoc = xmlread(...)  
xDoc =  
    [#document: null]
```

Remarks Find out more about the Document Object Model at the World Wide Web Consortium (W3C) Web site, <http://www.w3.org/DOM/>. For specific information on using Java DOM objects, visit the Sun Web site, <http://www.java.sun.com/xml/docs/api>.

Examples

Example 1

All XML files have a single root element. Some XML files declare a preferred schema file as an attribute of this element. Use the `getAttribute` method of the DOM node to get the name of the preferred schema file:

```
xDoc = xmlread(fullfile(matlabroot, ...  
                        'toolbox/matlab/general/info.xml'));  
  
xRoot = xDoc.getDocumentElement;  
schemaURL = ...  
    char(xRoot.getAttribute('xsi:noNamespaceSchemaLocation'))  
  
schemaURL =  
    http://www.mathworks.com/namespace/info/v1/info.xsd
```


Example 2

Each `info.xml` file on the MATLAB path contains several `listitem` elements with a `label` and `callback` element. This script finds the `callback` that corresponds to the `label` 'Plot Tools':

```
infoLabel = 'Plot Tools';
infoCbk = '';
itemFound = false;

xDoc = xmlread(fullfile(matlabroot, ...
                        'toolbox/matlab/general/info.xml'));

% Find a deep list of all listitem elements.
allListItems = xDoc.getElementsByTagName('listitem');

% Note that the item list index is zero-based.
for k = 0:allListItems.getLength-1
    thisListItem = allListItems.item(k);
    childNode = thisListItem.getFirstChild;

    while ~isempty(childNode)
        %Filter out text, comments, and processing instructions.
        if childNode.getNodeType == childNode.ELEMENT_NODE
            % Assume that each element has a single
            % org.w3c.dom.Text child.
            childText = char(childNode.getFirstChild.getData);

            switch char(childNode.getTagName)
            case 'label';
                itemFound = strcmp(childText, infoLabel);
            case 'callback' ;
                infoCbk = childText;
            end
        end % End IF
        childNode = childNode.getNextSibling;
    end % End WHILE
```

```
        if itemFound
            break;
        else
            infoCbk = '';
        end
    end % End FOR

    disp(sprintf('Item %s has a callback of %s.', ...
                infoLabel, infoCbk))
```

Example 3

This function parses an XML file using methods of the DOM node returned by `xmlread`, and stores the data it reads in the `Name`, `Attributes`, `Data`, and `Children` fields of a MATLAB structure:

```
function theStruct = parseXML(filename)
% PARSEXML Convert XML file to a MATLAB structure.
try
    tree = xmlread(filename);
catch
    error('Failed to read XML file %s.',filename);
end

% Recurse over child nodes. This could run into problems
% with very deeply nested trees.
try
    theStruct = parseChildNodes(tree);
catch
    error('Unable to parse XML file %s.');
```

```
end

% ----- Subfunction PARSECHILDNODES -----
function children = parseChildNodes(theNode)
% Recurse over node children.
children = [];
if theNode.hasChildNodes
```

```
childNodes = theNode.getChildNodes;
numChildNodes = childNodes.getLength();
allocCell = cell(1, numChildNodes);

children = struct(
    'Name', allocCell, 'Attributes', allocCell, ...
    'Data', allocCell, 'Children', allocCell);

for count = 1:numChildNodes
    theChild = childNodes.item(count-1);
    children(count) = makeStructFromNode(theChild);
end
end

% ----- Subfunction MAKESTRUCTFROMNODE -----
function nodeStruct = makeStructFromNode(theNode)
% Create structure of node info.

nodeStruct = struct(
    'Name', char(theNode.getNodeName), ...
    'Attributes', parseAttributes(theNode), ...
    'Data', '', ...
    'Children', parseChildNodes(theNode));

if any(strcmp(methods(theNode), 'getData'))
    nodeStruct.Data = char(theNode.getData);
else
    nodeStruct.Data = '';
end

% ----- Subfunction PARSEATTRIBUTES -----
function attributes = parseAttributes(theNode)
% Create attributes structure.

attributes = [];
if theNode.hasAttributes
    theAttributes = theNode.getAttributes;
```

xmlread

```
numAttributes = theAttributes.getLength();
allocCell = cell(1, numAttributes);
attributes = struct('Name', allocCell, 'Value', allocCell);

for count = 1:numAttributes
    attrib = theAttributes.item(count-1);
    attributes(count).Name = char(attrib.getName);
    attributes(count).Value = char(attrib.getValue);
end
end
```

See Also

xmlwrite, xslt

Purpose	Serialize XML Document Object Model node
Syntax	<pre>xmlwrite(filename, DOMnode) str = xmlwrite(DOMnode)</pre>
Description	<p><code>xmlwrite(filename, DOMnode)</code> serializes the Document Object Model node <code>DOMnode</code> to the file specified by <code>filename</code>. The <code>filename</code> input is a string enclosed in single quotes.</p> <p><code>str = xmlwrite(DOMnode)</code> serializes the Document Object Model node <code>DOMnode</code> and returns the node tree as a string, <code>s</code>.</p>
Remarks	Find out more about the Document Object Model at the World Wide Web Consortium (W3C) Web site, http://www.w3.org/DOM/ . For specific information on using Java DOM objects, visit the Sun Web site, http://www.java.sun.com/xml/docs/api .
Example	<pre>% Create a sample XML document. docNode = com.mathworks.xml.XMLUtils.createDocument... ('root_element') docRootNode = docNode.getDocumentElement; for i=1:20 thisElement = docNode.createElement('child_node'); thisElement.appendChild... (docNode.createTextNode(sprintf('%i',i))); docRootNode.appendChild(thisElement); end docNode.appendChild(docNode.createComment('this is a comment')); % Save the sample XML document. xmlFileName = [tempname, '.xml']; xmlwrite(xmlFileName, docNode); edit(xmlFileName);</pre>
See Also	<code>xmlread</code> , <code>xslt</code>

xor

Purpose Logical exclusive-OR

Syntax `C = xor(A, B)`

Description `C = xor(A, B)` performs an exclusive OR operation on the corresponding elements of arrays A and B. The resulting element `C(i, j, ...)` is logical true (1) if `A(i, j, ...)` or `B(i, j, ...)`, but not both, is nonzero.

A	B	C
Zero	Zero	0
Zero	Nonzero	1
Nonzero	Zero	1
Nonzero	Nonzero	0

Examples Given `A = [0 0 pi eps]` and `B = [0 -2.4 0 1]`, then

```
C = xor(A,B)
C =
    0     1     1     0
```

To see where either A or B has a nonzero element and the other matrix does not,

```
spy(xor(A,B))
```

See Also `all`, `any`, `find`, Elementwise Logical Operators, Short-Circuit Logical Operators

Purpose	Transform XML document using XSLT engine
Syntax	<pre>result = xslt(source, style, dest) [result,style] = xslt(...) xslt(...,'-web')</pre>
Description	<p><code>result = xslt(source, style, dest)</code> transforms an XML document using a stylesheet and returns the resulting document's URL. The function uses these inputs, the first of which is required:</p> <ul style="list-style-type: none">• <code>source</code> is the filename or URL of the source XML file. <code>source</code> can also specify a DOM node.• <code>style</code> is the filename or URL of an XSL stylesheet.• <code>dest</code> is the filename or URL of the desired output document. If <code>dest</code> is absent or empty, the function uses a temporary filename. If <code>dest</code> is <code>'-tostring'</code>, the function returns the output document as a MATLAB string. <p><code>[result,style] = xslt(...)</code> returns a processed stylesheet appropriate for passing to subsequent XSLT calls as <code>style</code>. This prevents costly repeated processing of the stylesheet.</p> <p><code>xslt(...,'-web')</code> displays the resulting document in the Help Browser.</p>
Remarks	Find out more about XSL stylesheets and how to write them at the World Wide Web Consortium (W3C) web site, http://www.w3.org/Style/XSL/ .
Example	<p>This example converts the file <code>info.xml</code> using the stylesheet <code>info.xsl</code>, writing the output to the file <code>info.html</code>. It launches the resulting HTML file in the Help Browser. MATLAB has several <code>info.xml</code> files that are used by the Start menu.</p> <pre>xslt info.xml info.xsl info.html -web</pre>

xslt

See Also

`xmlread`, `xmlwrite`

Purpose Create array of all zeros

Syntax

```
B = zeros(n)
B = zeros(m,n)
B = zeros([m n])
B = zeros(m,n,p,...)
B = zeros([m n p ...])
B = zeros(size(A))
zeros(m, n,...,classname)
zeros([m,n,...],classname)
```

Description

`B = zeros(n)` returns an n -by- n matrix of zeros. An error message appears if n is not a scalar.

`B = zeros(m,n)` or `B = zeros([m n])` returns an m -by- n matrix of zeros.

`B = zeros(m,n,p,...)` or `B = zeros([m n p ...])` returns an m -by- n -by- p -by-... array of zeros.

Note The size inputs m , n , p , ... should be nonnegative integers. Negative integers are treated as 0.

`B = zeros(size(A))` returns an array the same size as A consisting of all zeros.

`zeros(m, n,...,classname)` or `zeros([m,n,...],classname)` is an m -by- n -by-... array of zeros of data type `classname`. `classname` is a string specifying the data type of the output. `classname` can have the following values: 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'.

Example

```
x = zeros(2,3,'int8');
```

Remarks The MATLAB language does not have a dimension statement; MATLAB automatically allocates storage for matrices. Nevertheless, for large

zeros

matrices, MATLAB programs may execute faster if the `zeros` function is used to set aside storage for a matrix whose elements are to be generated one at a time, or a row or column at a time. For example

```
x = zeros(1,n);  
for i = 1:n, x(i) = i; end
```

See Also

`eye`, `ones`, `rand`, `randn`, `complex`

Purpose Compress files into zip file

Syntax

```
zip(zipfile,files)
zip(zipfile,files,rootdir)
entrynames = zip(...)
```

Description `zip(zipfile,files)` creates a zip file with the name `zipfile` from the list of files and directories specified in `files`. Relative paths are stored in the zip file, but absolute paths are not. Directories recursively include all of their content.

`zipfile` is a string specifying the name of the zip file. The `.zip` extension is appended to `zipfile` if omitted.

`files` is a string or cell array of strings containing the list of files or directories included in `zipfile`. Individual files that are on the MATLAB path can be specified as partial pathnames. Otherwise an individual file can be specified relative to the current directory or with an absolute path. Directories must be specified relative to the current directory or with absolute paths. On UNIX systems, directories can also start with `~/` or `~username/`, which expands to the current user's home directory or the specified user's home directory, respectively. The wildcard character `*` can be used when specifying files or directories, except when relying on the MATLAB path to resolve a filename or partial pathname.

`zip(zipfile,files,rootdir)` allows the path for `files` to be specified relative to `rootdir` rather than the current directory.

`entrynames = zip(...)` returns a string cell array of the relative path entry names contained in `zipfile`.

Examples

Zip a File

Create a zip file of the file `guide.viewlet`, which is in the `demos` directory of MATLAB. It saves the zip file in `d:/mymfiles/viewlet.zip`.

```
file = fullfile(matlabroot,'demos','guide.viewlet');
zip('d:/mymfiles/viewlet.zip',file)
```

Run `zip` for the files `guide.viewlet` and `import.viewlet` and save the zip file in `viewlets.zip`. The source files and zipped file are in the current directory.

```
zip('viewlets.zip',{'guide.viewlet','import.viewlet'})
```

Zip Selected Files

Run `zip` for all `.m` and `.mat` files in the current directory to the file `backup.zip`:

```
zip('backup',{'*.m','*.mat'});
```

Zip a Directory

Run `zip` for the directory `D:/mymfiles` and its contents to the zip file `mymfiles` in the directory one level up from the current directory.

```
zip('../mymfiles','D:/mymfiles')
```

Run `zip` for the files `thesis.doc` and `defense.ppt`, which are located in `d:/PhD`, to the zip file `thesis.zip` in the current directory.

```
zip('thesis.zip',{'thesis.doc','defense.ppt'],'d:/PhD')
```


See Also

`gzip`, `gunzip`, `tar`, `untar`, `unzip`

Purpose

Turn zooming on or off or magnify by factor

GUI Alternatives

Use the Zoom tools  on the figure toolbar to zoom in or zoom out on a plot, or select **Zoom In** or **Zoom Out** from the figure's **Tools** menu. For details, see “Zooming in 2-D and 3-D” in the MATLAB Graphics documentation.

Syntax

```
zoom on
zoom off
zoom out
zoom reset
zoom
zoom
zoom(factor)
zoom(fig, option)
h = zoom(figure_handle)
```

Description

`zoom on` turns on interactive zooming. When interactive zooming is enabled in a figure, pressing a mouse button while your cursor is within an axes zooms into the point or out from the point beneath the mouse. Zooming changes the axes limits. When using zoom mode, you

- Zoom in by positioning the mouse cursor where you want the center of the plot to be and either
 - press the mouse button or
 - rotate the mouse scroll wheel away from you (upward).
- Zoom out by positioning the mouse cursor where you want the center of the plot to be and either
 - simultaneously press **Shift** and the mouse button, or
 - rotate the mouse scroll wheel toward you (downward).

Each mouse click or scroll wheel click zooms in or out by a factor of 2.

Clicking and dragging over an axes when zooming in is enabled draws a rubberband box. When you release the mouse button, the axes zoom in to the region enclosed by the rubberband box.

Double-clicking over an axes returns the axes to its initial zoom setting in both zoom-in and zoom-out modes.

`zoom off` turns interactive zooming off.

`zoom out` returns the plot to its initial zoom setting.

`zoom reset` remembers the current zoom setting as the initial zoom setting. Later calls to `zoom out`, or double-clicks when interactive zoom mode is enabled, will return to this zoom level.

`zoom toggles` the interactive zoom status between off and on (restoring the most recently used zoom tool).

`zoom xon` and `zoom yon` set `zoom on` for the x - and y -axis, respectively.

`zoom(factor)` zooms in or out by the specified zoom factor, without affecting the interactive zoom mode. Values greater than 1 zoom in by that amount, while numbers greater than 0 and less than 1 zoom out by $1/\text{factor}$.

`zoom(fig, option)` Any of the above options can be specified on a figure other than the current figure using this syntax.

`h = zoom(figure_handle)` returns a zoom *mode object* for figure `figure_handle` for you to customize the mode's behavior.

Using Zoom Mode Objects

You access the following properties of zoom mode objects via `get` and modify some of them using `set`:

Enable 'on' | 'off'

Specifies whether this figure mode is currently enabled on the figure.

FigureHandle <handle>

The associated figure handle. This read-only property cannot be set.

Motion 'horizontal' | 'vertical' | 'both'

The type of zooming enabled for the figure.

Direction 'in' | 'out'

The direction of the zoom operation.

RightClickAction 'InverseZoom' | 'InverseZoom'

The behavior of a right-click action. A value of 'InverseZoom' will cause a right-click to zoom out. A value of 'InverseZoom' will display a context menu. This setting will persist between MATLAB sessions.

ButtonDownFilter <function_handle>

The application can inhibit the zoom operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to handle callbacks), as follows:

```
function [res] = myfunction(obj,event_obj)
% OBJ          handle to the object that has been clicked on.
% EVENT_OBJ    handle to event object (empty in this release).
% RES          a logical flag to determine whether the zoom
%              operation should take place or the 'ButtonDownFcn'
%              property of the object should take precedence.
```

ActionPreCallback <function_handle>

Set this callback to listen to when a zoom operation will start. The input function handle should reference a function with two implicit arguments (similar to handle callbacks), as follows:

```
function myfunction(obj,event_obj)
% obj          handle to the figure that has been clicked on.
% event_obj    handle to event object.
```

The event object has the following read-only property:

Axes	The handle of the axes that is being panned.
------	--

ActionPostCallback <function_handle>

Set this callback to listen to when a zoom operation has finished. The input function handle should reference a function with two implicit arguments (similar to handle callbacks), as follows:

```
function myfunction(obj,event_obj)
% obj          handle to the figure that has been clicked on.
% event_obj    handle to event object. The object has the same
%              properties as the event_obj of the
%              'ActionPreCallback' callback.
```

UIContextMenu <handle>

Specifies a custom context menu to be displayed during a right-click action. This property is ignored if the 'RightClickZoomOut' property has been set to 'on'.

```
flags = isAllowAxesZoom(h,axes)
```

Calling the function `isAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `axes`, as input will return a logical array of the same dimension as the axes handle vector which indicate whether a zoom operation is permitted on the axes objects.

```
setAllowAxesZoom(h,axes,flag)
```

Calling the function `setAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, will either allow or disallow a zoom operation on the axes objects.

```
info = getAxesZoomMotion(h,axes)
```

Calling the function `getAxesZoomMotion` on the zoom object, `H`, with a vector of axes handles, `AXES`, as input will return a character cell array of the same dimension as the axes handle vector which indicates the type of zoom operation for each axes. Possible values for the type of operation are 'horizontal', 'vertical' or 'both'.

```
setAxesZoomMotion(h,axes,style)
```


Calling the function `setAxesZoomMotion` on the zoom object, `h`, with a vector of axes handles, `axes`, and a character array, `style`, will set the style of zooming on each axes.

Examples

Example 1

Simple zoom

```
plot(1:10);  
zoom on  
% zoom in on the plot
```

Example 2

Create zoom mode object and constrain to *x*-axis zooming:

```
plot(1:10);  
h = zoom;  
set(h,'Motion','horizontal','Enable','on');  
% zoom in on the plot in the horizontal direction.
```

Example 3

Create four axes as subplots and set zoom style differently for each by setting a different property for each axes handle:

```
ax1 = subplot(2,2,1);  
plot(1:10);  
h = zoom;  
ax2 = subplot(2,2,2);  
plot(rand(3));  
setAllowAxesZoom(h,ax2,false);  
ax3 = subplot(2,2,3);  
plot(peaks);  
setAxesZoomMotion(h,ax3,'horizontal');  
ax4 = subplot(2,2,4);  
contour(peaks);  
setAxesZoomMotion(h,ax4,'vertical');  
% Zoom in on the plots.
```

Example 4

Create a `ButtonDown` callback for zoom mode objects to trigger. Copy the following code to a new M-file, execute it, and observe zooming behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
h = zoom;
set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
    flag = true;
else
    flag = false;
end
```

Example 5

Create callbacks for pre- and post-`ButtonDown` events for zoom mode objects to trigger. Copy the following code to a new M-file, execute it, and observe zoom behavior:

```
function demo
% Listen to zoom events
plot(1:10);
h = zoom;
set(h,'ActionPreCallback',@myprecallback);
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');
```

```
%  
function myprecallback(obj, evd)  
disp('A zoom is about to occur.');
```

```
%  
function mypostcallback(obj, evd)  
newLim = get(evd.Axes, 'XLim');  
msgbox(sprintf('The new X-Limits are [%.2f %.2f].', newLim));
```

Remarks

zoom changes the axes limits by a factor of two (in or out) each time you press the mouse button while the cursor is within an axes. You can also click and drag the mouse to define a zoom area, or double-click to return to the initial zoom level.

You can create a zoom mode object once and use it to customize the behavior of different axes, as example 3 illustrates. You can also change its callback functions on the fly.

When you assign different zoom behaviors to different subplot axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse will carry over to the linked axes, regardless of the behavior you previously set for the other axes.

See Also

`linkaxes`, `pan`, `rotate3d`

“Object Manipulation” on page 1-97 for related functions

- & 2-14 2-16
- ' 2-2
- * 2-2
- + 2-2
- 2-2
- / 2-2
- : 2-23
- < 2-12
- > 2-12
- @ 2-1216
- \ 2-2
- ^ 2-2
- | 2-14 2-16
- ~ 2-14 2-16
- && 2-16
- == 2-12
-) 2-21
- || 2-16
- ~= 2-12
- 1-norm 2-2101 2-2489
- 2-norm (estimate of) 2-2103

- A**
- abs 2-25
- absolute accuracy
 - BVP 2-362
 - DDE 2-740
 - ODE 2-2148
- absolute value 2-25
- Accelerator
 - Uimenu property 2-3257
- accumarray 2-26
- accuracy
 - of linear equation solution 2-548
 - of matrix inversion 2-548
- acos 2-32
- acosd 2-34
- acosh 2-35
- acot 2-37
- acotd 2-39
- acoth 2-40
- acsc 2-42
- acscd 2-44
- acsch 2-45
- activelegend 1-83 2-2319
- actxcontrol 2-47
- actxcontrollist 2-54
- actxcontrolselect 2-55
- actxserver 2-58
- Adams-Bashforth-Moulton ODE solver 2-2135
- addevent 2-61
- addframe
 - AVI files 2-63
- addition (arithmetic operator) 2-2
- addpath 2-65
- addpref function 2-67
- addproperty 2-68
- addressing selected array elements 2-23
- addsample 2-70
- addsampletocollection 2-72
- addtodate 2-74
- addts 2-75
- adjacency graph 2-840
- airy 2-77
- Airy functions
 - relationship to modified Bessel functions 2-77
- align function 2-79
- aligning scattered data
 - multi-dimensional 2-2089
 - two-dimensional 2-1346
- ALim, Axes property 2-211
- all 2-83
- allchild function 2-85
- allocation of storage (automatic) 2-3497
- alpha channels
 - in PNG files 2-1539
- AlphaData
 - image property 2-1506

- surface property 2-2973
- surfaceplot property 2-2993
- AlphaDataMapping
 - image property 2-1507
 - patch property 2-2226
 - surface property 2-2973
 - surfaceplot property 2-2993
- AmbientLightColor, Axes property 2-212
- AmbientStrength
 - Patch property 2-2227
 - Surface property 2-2974
 - surfaceplot property 2-2994
- amd 2-92 2-1752
- analytical partial derivatives (BVP) 2-363
- and 2-97
- and (M-file function equivalent for &) 2-15
- AND, logical
 - bit-wise 2-327
- angle 2-99
- annotating graphs
 - deleting annotations 2-102
- annotating plots 2-2320
- ans 2-136
- anti-diagonal 2-1373
- any 2-137
- arccosecant 2-42
- arccosine 2-32
- arccotangent 2-37
- arcsecant 2-167
- arcsine 2-172
- arctangent 2-179
 - four-quadrant 2-181
- arguments, M-file
 - checking number of inputs 2-2080
 - checking number of outputs 2-2084
 - number of input 2-2082
 - number of output 2-2082
 - passing variable numbers of 2-3373
- arithmetic operations, matrix and array
 - distinguished 2-2
- arithmetic operators
 - reference 2-2
- array
 - addressing selected elements of 2-23
 - displaying 2-824
 - left division (arithmetic operator) 2-4
 - maximum elements of 2-1954
 - mean elements of 2-1959
 - median elements of 2-1962
 - minimum elements of 2-1995
 - multiplication (arithmetic operator) 2-3
 - of all ones 2-2167
 - of all zeros 2-3497
 - of random numbers 2-2472 2-2477
 - power (arithmetic operator) 2-4
 - product of elements 2-2387
 - removing first n singleton dimensions
 - of 2-2709
 - removing singleton dimensions of 2-2799
 - reshaping 2-2567
 - right division (arithmetic operator) 2-3
 - shift circularly 2-469
 - shifting dimensions of 2-2709
 - size of 2-2722
 - sorting elements of 2-2736
 - structure 2-1025 2-1300 2-2587 2-2699
 - sum of elements 2-2954
 - swapping dimensions of 2-1638 2-2295
 - transpose (arithmetic operator) 2-4
- arrayfun 2-160
- arrays
 - detecting empty 2-1651
 - editing 2-3465
 - maximum size of 2-546
 - opening 2-2168
- arrowhead matrix 2-533
- ASCII
 - delimited files
 - writing 2-836
- ASCII data

- converting sparse matrix after loading
 - from 2-2749
- reading 2-832
- reading from disk 2-1853
- saving to disk 2-2623
- ascii function 2-166
- asec 2-167
- asecd 2-169
- asech 2-170
- asin 2-172
- asind 2-174
- asinh 2-175
- aspect ratio of axes 2-663 2-2259
- assignin 2-177
- atan 2-179
- atan2 2-181
- atand 2-183
- atanh 2-184
- .au files
 - reading 2-197
 - writing 2-198
- audio
 - saving in AVI format 2-199
 - signal conversion 2-1803 2-2063
- audioplayer 1-78 2-186
- audiorecorder 1-78 2-191
- aufinfo 2-196
- auread 2-197
- AutoScale
 - quivergroup property 2-2450
- AutoScaleFactor
 - quivergroup property 2-2450
- autoselection of OpenGL 2-1058
- auwrite 2-198
- average of array elements 2-1959
- average,running 2-1094
- avi 2-199
- avifile 2-199
- aviinfo 2-202
- aviread 2-204

- axes 2-205
 - editing 2-2320
 - setting and querying data aspect ratio 2-663
 - setting and querying limits 2-3469
 - setting and querying plot box aspect ratio 2-2259
- Axes
 - creating 2-205
 - defining default properties 2-210
 - fixed-width font 2-228
 - property descriptions 2-211
- axis 2-249
- axis crossing, *see* zero of a function
- azimuth (spherical coordinates) 2-2765
- azimuth of viewpoint 2-3389

B

- BackFaceLighting
 - Surface property 2-2974
 - surfaceplot property 2-2994
- BackFaceLightingpatch property 2-2227
- background color chunk
 - PNG files 2-1539
- BackgroundColor
 - annotation textbox property 2-129
 - Text property 2-3072
- BackgroundColor
 - Uicontrol property 2-3217
- BackingStore, Figure property 2-1030
- badly conditioned 2-2489
- balance 2-255
- BarLayout
 - barseries property 2-270
- BarWidth
 - barseries property 2-270
- base to decimal conversion 2-285
- base two operations
 - conversion from decimal to binary 2-758

- logarithm 2-1872
- next power of two 2-2097
- base2dec 2-285
- BaseLine
 - barseries property 2-270
 - stem property 2-2844
- BaseValue
 - area property 2-146
 - barseries property 2-271
 - stem property 2-2844
- beep 2-286
- BeingDeleted
 - areaserries property 2-146
 - barseries property 2-271
 - contour property 2-574
 - errorbar property 2-905
 - group property 2-1031 2-1507 2-3073
 - hggroup property 2-1428
 - hgtransform property 2-1448
 - light property 2-1794
 - line property 2-1809
 - lineseries property 2-1820
 - quivergroup property 2-2450
 - rectangle property 2-2506
 - scatter property 2-2646
 - stairservices property 2-2812
 - stem property 2-2844
 - surface property 2-2975
 - surfaceplot property 2-2995
 - transform property 2-2227
 - Uipushtool property 2-3292
 - Uitoggletool property 2-3316
 - Uitoolbar property 2-3329
- Bessel functions
 - first kind 2-294
 - modified, first kind 2-291
 - modified, second kind 2-297
 - second kind 2-300
- Bessel functions, modified
 - relationship to Airy functions 2-77
- Bessel's equation
 - (defined) 2-294
 - modified (defined) 2-291
- besseli 2-291
- besselj 2-294
- besselk 2-297
- bessely 2-300
- beta 2-304
- beta function
 - (defined) 2-304
 - incomplete (defined) 2-306
 - natural logarithm 2-308
- betainc 2-306
- betaln 2-308
- bicg 2-309
- bicgstab 2-318
- BiConjugate Gradients method 2-309
- BiConjugate Gradients Stabilized
 - method 2-318
- big endian formats 2-1145
- bin2dec 2-324
- binary
 - data
 - writing to file 2-1228
 - files
 - reading 2-1179
 - mode for opened files 2-1144
- binary data
 - reading from disk 2-1853
 - saving to disk 2-2623
- binary function 2-325
- binary to decimal conversion 2-324
- bisection search 2-1238
- bit depth
 - querying 2-1524
- bit depths
 - See also* index entries for individual file formats 2-1535
 - supported 2-1535
- bit-wise operations

- AND 2-327
 - get 2-330
 - OR 2-333
 - set bit 2-334
 - shift 2-335
 - XOR 2-337
- bitand 2-327
- bitcmp 2-328
- bitget 2-330
- bitmaps
 - reading 2-1535
 - writing 2-1545
- bitmax 2-331
- bitor 2-333
- bitset 2-334
- bitshift 2-335
- bitxor 2-337
- blanks 2-338
 - removing trailing 2-754
- blkdiag 2-339
- BMP files
 - reading 2-1535
 - writing 2-1545
- bold font
 - TeX characters 2-3093
- boundary value problems 2-369
- box 2-340
- Box, Axes property 2-213
- braces, curly (special characters) 2-19
- brackets (special characters) 2-19
- break 2-341
- breakpoints
 - listing 2-704
 - removing 2-692
 - resuming execution from 2-695
 - setting in M-files 2-708 2-710
- brighten 2-342
- browser
 - for help 2-1413
- bubble plot (scatter function) 2-2641
- Buckminster Fuller 2-3044
- builtin 1-68 2-344
- BusyAction
 - areaserie property 2-146
 - Axes property 2-213
 - barseries property 2-271
 - contour property 2-574
 - errorbar property 2-905
 - Figure property 2-1031
 - hggroup property 2-1428
 - hgtransform property 2-1448
 - Image property 2-1508
 - Light property 2-1794
 - Line property 2-1809 2-1820
 - patch property 2-2228
 - quivergroup property 2-2451
 - rectangle property 2-2506
 - Root property 2-2591
 - scatter property 2-2646
 - stairs property 2-2812
 - stem property 2-2845
 - Surface property 2-2975
 - surfaceplot property 2-2995
 - Text property 2-3074
 - Uicontextmenu property 2-3202
 - Uicontrol property 2-3217
 - Uimenu property 2-3258
 - Uipushtool property 2-3292
 - Uitoggletool property 2-3317
 - Uitoolbar property 2-3329
- ButtonDownFcn
 - area series property 2-147
 - Axes property 2-214
 - barseries property 2-272
 - contour property 2-575
 - errorbar property 2-906
 - Figure property 2-1032
 - hggroup property 2-1429
 - hgtransform property 2-1449
 - Image property 2-1508

- Light property 2-1795
 - Line property 2-1810
 - lineseries property 2-1821
 - patch property 2-2228
 - quivergroup property 2-2451
 - rectangle property 2-2507
 - Root property 2-2591
 - scatter property 2-2647
 - stairseseries property 2-2813
 - stem property 2-2845
 - Surface property 2-2976
 - surfaceplot property 2-2996
 - Text property 2-3074
 - Uicontrol property 2-3218
 - BVP solver properties
 - analytical partial derivatives 2-363
 - error tolerance 2-361
 - Jacobian matrix 2-363
 - mesh 2-366
 - singular BVPs 2-366
 - solution statistics 2-367
 - vectorization 2-362
 - bvp4c 2-345
 - bvpget 2-356
 - bvpinit 2-357
 - bvpset 2-360
 - bvpxtend 2-369
- C**
- caching
 - MATLAB directory 2-2251
 - calendar 2-370
 - call history 2-2394
 - Callback
 - Uicontextmenu property 2-3203
 - Uicontrol property 2-3219
 - Uimenu property 2-3259
 - CallbackObject, Root property 2-2591
 - calllib 2-371
 - callSoapService 2-373
 - camdolly 2-374
 - camera
 - dollying position 2-374
 - moving camera and target postions 2-374
 - placing a light at 2-378
 - positioning to view objects 2-380
 - rotating around camera target 1-96 2-382 2-384
 - rotating around viewing axis 2-388
 - setting and querying position 2-385
 - setting and querying projection type 2-387
 - setting and querying target 2-389
 - setting and querying up vector 2-391
 - setting and querying view angle 2-393
 - CameraPosition, Axes property 2-215
 - CameraPositionMode, Axes property 2-215
 - CameraTarget, Axes property 2-216
 - CameraTargetMode, Axes property 2-216
 - CameraUpVector, Axes property 2-216
 - CameraUpVectorMode, Axes property 2-216
 - CameraViewAngle, Axes property 2-217
 - CameraViewAngleMode, Axes property 2-217
 - camlight 2-378
 - camlookat 2-380
 - camorbit 2-382
 - campan 2-384
 - campos 2-385
 - camproj 2-387
 - camroll 2-388
 - camtarget 2-389
 - camup 2-391
 - camva 2-393
 - camzoom 2-395
 - CaptureMatrix, Root property 2-2591
 - CaptureRect, Root property 2-2592
 - cart2pol 2-396
 - cart2sph 2-397
 - Cartesian coordinates 2-396 to 2-397 2-2330 2-2765

- case 2-398
 - in switch statement (defined) 2-3030
 - lower to upper 2-3361
 - upper to lower 2-1884
- cast 2-400
- cat 2-401
- catch 2-403
- caxis 2-404
- Cayley-Hamilton theorem 2-2350
- cd 2-409
- cd (ftp) function 2-411
- CData
 - Image property 2-1509
 - scatter property 2-2647
 - Surface property 2-2976
 - surfaceplot property 2-2996
 - Uicontrol property 2-3220
 - Uipushtool property 2-3293
 - Uitoggletool property 2-3317
- CDataMapping
 - Image property 2-1511
 - patch property 2-2231
 - Surface property 2-2977
 - surfaceplot property 2-2997
- CDataMode
 - surfaceplot property 2-2997
- CDatapatch property 2-2229
- CDataSource
 - scatter property 2-2648
 - surfaceplot property 2-2997
- cdf2rdf 2-412
- cdfepoch 2-414
- cdfinfo 2-415
- cdfread 2-419
- cdfwrite 2-421
- ceil 2-424
- cell 2-425
- cell array
 - conversion to from numeric array 2-2110
 - creating 2-425
 - structure of, displaying 2-438
- cell2mat 2-427
- cell2struct 2-429
- celldisp 2-431
- cellfun 2-432
- cellplot 2-438
- cgs 2-441
- char 1-50 1-57 1-62 2-447
- characters
 - conversion, in format specification
 - string 2-1166 2-2787
 - escape, in format specification
 - string 2-1167 2-2788
- check boxes 2-3210
- Checked, Uimenu property 2-3259
- checkerboard pattern (example) 2-2558
- checkin 2-448
 - examples 2-449
 - options 2-448
- checkout 2-451
 - examples 2-452
 - options 2-451
- child functions 2-2389
- Children
 - areaseries property 2-147
 - Axes property 2-218
 - barseries property 2-273
 - contour property 2-575
 - errorbar property 2-906
 - Figure property 2-1032
 - hggroup property 2-1429
 - hgtransform property 2-1449
 - Image property 2-1511
 - Light property 2-1795
 - Line property 2-1810
 - lineseries property 2-1821
 - patch property 2-2231
 - quivergroup property 2-2452
 - rectangle property 2-2507
 - Root property 2-2592

- scatter property 2-2648
- stairs series property 2-2813
- stem property 2-2846
- Surface property 2-2978
- surfaceplot property 2-2998
- Text property 2-3075
- Uicontextmenu property 2-3203
- Uicontrol property 2-3220
- Uimenu property 2-3260
- Uitoolbar property 2-3330
- chol 2-454
- Cholesky factorization 2-454
 - (as algorithm for solving linear equations) 2-2019
 - lower triangular factor 2-2217
 - minimum degree ordering and (sparse) 2-3043
 - preordering for 2-533
- cholinc 2-458
- cholupdate 2-466
- circle
 - rectangle function 2-2501
- circshift 2-469
- cla 2-470
- clabel 2-471
- class 2-477
- class, object, *see* object classes
- classes
 - field names 2-1025
 - loaded 2-1571
- clc 2-479 2-486
- clear 2-480
 - serial port I/O 2-485
- clearing
 - Command Window 2-479
 - items from workspace 2-480
 - Java import list 2-482
- clf 2-486
- ClickedCallback
 - Uipushtool property 2-3293
 - Uitoggletool property 2-3318
- CLim, Axes property 2-219
- CLimMode, Axes property 2-220
- clipboard 2-487
- Clipping
 - areaseries property 2-148
 - Axes property 2-220
 - barseries property 2-273
 - contour property 2-576
 - errrobar property 2-907
 - Figure property 2-1033
 - hggroup property 2-1430
 - hgtransform property 2-1450
 - Image property 2-1512
 - Light property 2-1795
 - Line property 2-1811
 - lineseries property 2-1821
 - quivergroup property 2-2452
 - rectangle property 2-2508
 - Root property 2-2592
 - scatter property 2-2649
 - stairs series property 2-2814
 - stem property 2-2846
 - Surface property 2-2978
 - surfaceplot property 2-2998
 - Text property 2-3075
 - Uicontrol property 2-3220
- Clippingpatch property 2-2231
- clock 2-488
- close 2-489
 - AVI files 2-491
- close (ftp) function 2-492
- CloseRequestFcn, Figure property 2-1033
- closest point search 2-856
- closest triangle search 2-3165
- closing
 - files 2-988
 - MATLAB 2-2441
- cmapeditor 2-513
- cmopts 2-494

- colamd 2-496
- colmmd 2-500
- colon operator 2-23
- Color
 - annotation arrow property 2-104
 - annotation doublearrow property 2-108
 - annotation line property 2-115
 - annotation textarrow property 2-121
 - annotation textbox property 2-129
 - Axes property 2-220
 - errorbar property 2-907
 - Figure property 2-1035
 - Light property 2-1795
 - Line property 2-1811
 - lineseries property 2-1821
 - quivergroup property 2-2452
 - stairs series property 2-2814
 - stem property 2-2846
 - Text property 2-3075
- color of fonts, see also FontColor
 - property 2-3093
- colorbar 2-502
- colormap 2-508
 - editor 2-513
- Colormap, Figure property 2-1035
- colormaps
 - converting from RGB to HSV 1-95 2-2577
 - plotting RGB components 1-95 2-2578
- ColorOrder, Axes property 2-220
- ColorSpec 2-531
- colperm 2-533
- COM
- object methods
 - activexcontrol 2-47
 - activexcontrollist 2-54
 - activexcontrolselect 2-55
 - activexserver 2-58
 - addproperty 2-68
 - delete 2-784
 - deleteproperty 2-790
 - eventlisteners 2-931
 - events 2-933
 - get 1-106 2-1283
 - inspect 2-1581
 - invoke 2-1635
 - iscom 2-1649
 - isevent 2-1659
 - isinterface 2-1670
 - ismethod 2-1678
 - isprop 2-1698
 - load 2-1858
 - move 2-2044
 - propedit 2-2397
 - registerevent 2-2548
 - release 2-2552
 - save 2-2630
 - send 2-2675
 - set 1-108 2-2685
 - unregisterallev events 2-3347
 - unregisterevent 2-3349
- server methods
 - Execute 2-935
 - Feval 2-997
- combinations of n elements 2-2088
- combs 2-2088
- comet 2-535
- comet3 2-537
- comma (special characters) 2-20
- command syntax 2-1409 2-3049
- Command Window
 - clearing 2-479
 - cursor position 1-4 2-1469

- get width 2-540
- commandhistory 2-539
- commands
 - help for 2-1408 2-1418
 - system 1-4 1-11 2-3052
 - UNIX 2-3343
- commandwindow 2-540
- comments
 - block of 2-21
- common elements, *see* set operations, intersection
- compan 2-541
- companion matrix 2-541
- compass 2-542
- complementary error function
 - (defined) 2-896
 - scaled (defined) 2-896
- complete elliptic integral
 - (defined) 2-881
 - modulus of 2-879 2-881
- complex 2-544 2-1498
 - exponential (defined) 2-943
 - logarithm 2-1869 to 2-1870
 - numbers 2-1478
 - numbers, sorting 2-2736 2-2740
 - phase angle 2-99
 - sine 2-2716
 - unitary matrix 2-2420
 - See also* imaginary
- complex conjugate 2-558
 - sorting pairs of 2-633
- complex data
 - creating 2-544
- complex numbers, magnitude 2-25
- complex Schur form 2-2662
- compression
 - lossy 2-1549
- computer 2-546
- computer MATLAB is running on 2-546
- concatenation
 - of arrays 2-401
- cond 2-548
- condeig 2-549
- condest 2-550
- condition number of matrix 2-548 2-2489
 - improving 2-255
- coneplot 2-552
- conj 2-558
- conjugate, complex 2-558
 - sorting pairs of 2-633
- connecting to FTP server 2-1208
- contents.m file 2-1409
- context menu 2-3199
- continuation (... , special characters) 2-20
- continue 2-559
- continued fraction expansion 2-2483
- contour
 - and mesh plot 2-963
 - filled plot 2-955
 - functions 2-951
 - of mathematical expression 2-952
 - with surface plot 2-981
- contour3 2-566
- contourc 2-569
- contourf 2-571
- ContourMatrix
 - contour property 2-576
- contours
 - in slice planes 2-593
- contourslice 2-593
- contrast 2-597
- conv 2-598
- conv2 2-600
- conversion
 - base to decimal 2-285
 - binary to decimal 2-324
 - Cartesian to cylindrical 2-396
 - Cartesian to polar 2-396
 - complex diagonal to real block diagonal 2-412

- cylindrical to Cartesian 2-2330
- decimal number to base 2-751 2-757
- decimal to binary 2-758
- decimal to hexadecimal 2-759
- full to sparse 2-2746
- hexadecimal to decimal 2-1422
- integer to string 2-1594
- lowercase to uppercase 2-3361
- matrix to string 2-1923
- numeric array to cell array 2-2110
- numeric array to logical array 2-1873
- numeric array to string 2-2112
- partial fraction expansion to
 - pole-residue 2-2569
- polar to Cartesian 2-2330
- pole-residue to partial fraction
 - expansion 2-2569
- real to complex Schur form 2-2620
- spherical to Cartesian 2-2765
- string matrix to cell array 2-440
- string to numeric array 2-2868
- uppercase to lowercase 2-1884
- vector to character string 2-447
- conversion characters in format specification
 - string 2-1166 2-2787
- convex hulls
 - multidimensional vizualization 2-609
 - two-dimensional visualization 2-606
- convhull 2-606
- convhulln 2-609
- convn 2-612
- convolution 2-598
 - inverse, *see* deconvolution
 - two-dimensional 2-600
- coordinate system and viewpoint 2-3389
- coordinates
 - Cartesian 2-396 to 2-397 2-2330 2-2765
 - cylindrical 2-396 to 2-397 2-2330
 - polar 2-396 to 2-397 2-2330
 - spherical 2-2765
- coordinates. 2-396
 - See also* conversion
- copyfile 2-613
- copyobj 2-616
- corrcoef 2-618
- cos 2-621
- cosd 2-623
- cosecant
 - hyperbolic 2-642
 - inverse 2-42
 - inverse hyperbolic 2-45
- cosh 2-624
- cosine 2-621
 - hyperbolic 2-624
 - inverse 2-32
 - inverse hyperbolic 2-35
- cot 2-626
- cotangent 2-626
 - hyperbolic 2-629
 - inverse 2-37
 - inverse hyperbolic 2-40
- cotd 2-628
- coth 2-629
- cov 2-631
- cplxpair 2-633
- cputime 2-634
- createClassFromWsd1 2-635
- CreateFcn
 - areaserie property 2-148
 - Axes property 2-221
 - barseries property 2-273
 - contour property 2-577
 - errorbar property 2-907
 - Figure property 2-1036
 - group property 2-1450
 - hggroup property 2-1430
 - Image property 2-1512
 - Light property 2-1795
 - Line property 2-1811
 - lineseries property 2-1822

- patch property 2-2231
 - quivergroup property 2-2453
 - rectangle property 2-2508
 - Root property 2-2592
 - scatter property 2-2649
 - stairs series property 2-2814
 - stemseries property 2-2847
 - Surface property 2-2978
 - surfaceplot property 2-2998
 - Text property 2-3075
 - Uicontextmenu property 2-3203
 - Uicontrol property 2-3220
 - Uimenu property 2-3260
 - Uipushtool property 2-3294
 - Uitoggletool property 2-3318
 - Uitoolbar property 2-3330
 - createSoapMessage 2-637
 - creating your own MATLAB functions 2-1214
 - cross 2-638
 - cross product 2-638
 - csc 2-639
 - cscd 2-641
 - csch 2-642
 - csvread 2-644
 - csvwrite 2-647
 - ctranspose (M-file function equivalent for \q) 2-7
 - ctranspose (timeseries) 2-649
 - cubic interpolation 2-1610 2-1613 2-1617 2-2269
 - piecewise Hermite 2-1600
 - cubic spline interpolation
 - one-dimensional 2-1600 2-1610 2-1613 2-1617
 - cumprod 2-651
 - cumsum 2-653
 - cumtrapz 2-654
 - cumulative
 - product 2-651
 - sum 2-653
 - CUR files
 - reading 2-1535
 - curl 2-656
 - curly braces (special characters) 2-19
 - current directory 2-2413
 - changing 2-409
 - CurrentAxes 2-1036
 - CurrentAxes, Figure property 2-1036
 - CurrentCharacter, Figure property 2-1037
 - CurrentFigure, Root property 2-2592
 - CurrentMenu, Figure property (obsolete) 2-1037
 - CurrentObject, Figure property 2-1037
 - CurrentPoint
 - Axes property 2-221
 - Figure property 2-1038
 - cursor images
 - reading 2-1537
 - cursor position 1-4 2-1469
 - Curvature, rectangle property 2-2508
 - curve fitting (polynomial) 2-2342
 - customverctrl 2-659
 - Cuthill-McKee ordering, reverse 2-3033 2-3044
 - cylinder 2-660
 - cylindrical coordinates 2-396 to 2-397 2-2330
- ## D
- daspect 2-663
 - data
 - ASCII
 - reading from disk 2-1853
 - ASCII, saving to disk 2-2623
 - binary
 - writing to file 2-1228
 - binary, saving to disk 2-2623
 - computing 2-D stream lines 1-99 2-2875
 - computing 3-D stream lines 1-99 2-2877

- formatted
 - reading from files 2-1195
 - writing to file 2-1165
 - formatting 2-1165 2-2786
 - isosurface from volume data 2-1692
 - reading binary from disk 2-1853
 - reading from files 2-3098
 - reducing number of elements in 1-99 2-2523
 - smoothing 3-D 1-99 2-2734
 - writing to strings 2-2786
- data aspect ratio of axes 2-663
- data types
 - complex 2-544
- data, aligning scattered
 - multi-dimensional 2-2089
 - two-dimensional 2-1346
- data, ASCII
 - converting sparse matrix after loading from 2-2749
- DataAspectRatio, Axes property 2-223
- DataAspectRatioMode, Axes property 2-226
- datatipinfo 2-671
- date 2-672
- date and time functions 2-891
- date string
 - format of 2-677
- date vector 2-689
- datenum 2-673
- datestr 2-677
- datevec 2-688
- dbc clear 2-692
- dbcont 2-695
- dbdown 2-696
- dblquad 2-697
- dbmex 2-699
- dbquit 2-700
- dbstack 2-702
- dbstatus 2-704
- dbstep 2-706
- dbstop 2-708
- dbtype 2-717
- dbup 2-718
- DDE solver properties
 - error tolerance 2-739
 - event location 2-745
 - solver output 2-741
 - step size 2-743
- dde23 2-719
- ddeadv 1-108 2-724
- ddeexec 2-726
- ddeget 2-727
- ddeinit 1-108 2-728
- ddephas2 output function 2-742
- ddephas3 output function 2-742
- ddeplot output function 2-742
- ddepoke 2-729
- ddeprint output function 2-742
- ddereq 2-731
- ddesd 2-733
- ddeset 2-738
- ddeterm 2-749
- ddeunadv 2-750
- deal 2-751
- deblank 2-754
- debugging
 - changing workspace context 2-696
 - changing workspace to calling M-file 2-718
 - displaying function call stack 2-702
 - M-files 2-1739 2-2389
 - MEX-files on UNIX 2-699
 - removing breakpoints 2-692
 - resuming execution from breakpoint 2-706
 - setting breakpoints in 2-708 2-710
 - stepping through lines 2-706
- dec2base 2-751 2-757
- dec2bin 2-758
- dec2hex 2-759
- decic function 2-760

- decimal number to base conversion 2-751
 - 2-757
- decimal point (.)
 - (special characters) 2-20
 - to distinguish matrix and array operations 2-2
- decomposition
 - Dulmage-Mendelsohn 2-840
 - economy-size 2-2420 2-3022
 - orthogonal-triangular (QR) 2-2420
 - Schur 2-2662
 - singular value 2-2482 2-3022
- deconv 2-762
- deconvolution 2-762
- definite integral 2-2432
- del operator 2-763
- del2 2-763
- delaunay 2-766
- Delaunay tessellation
 - 3-dimensional visualization 2-773
 - multidimensional visualization 2-777
- Delaunay triangulation
 - visualization 2-766
- delaunay3 2-773
- delaunayn 2-777
- delete 2-782 2-784
 - serial port I/O 2-787
 - timer object 2-789
- delete (ftp) function 2-786
- DeleteFcn
 - areaseries property 2-149
 - Axes property 2-226
 - barseries property 2-274
 - contour property 2-577
 - errorbar property 2-907
 - Figure property 2-1040
 - hggroup property 2-1430
 - hgtransform property 2-1451
 - Image property 2-1512
 - Light property 2-1796
 - lineseries property 2-1822
 - quivergroup property 2-2453
 - Root property 2-2593
 - scatter property 2-2649
 - stairs series property 2-2815
 - stem property 2-2847
 - Surface property 2-2978
 - surfaceplot property 2-2999
 - Text property 2-3076 to 2-3077
 - Uicontextmenu property 2-3204 2-3221
 - Uimenu property 2-3261
 - Uipushtool property 2-3295
 - Uitoggletool property 2-3319
 - Uitoolbar property 2-3331
- DeleteFcn, line property 2-1812
- DeleteFcn, rectangle property 2-2509
- DeleteFcnpatch property 2-2232
- deleteproperty 2-790
- deleting
 - files 2-782
 - items from workspace 2-480
- delevent 2-792
- delimiters in ASCII files 2-832 2-836
- delsample 2-793
- delsamplefromcollection 2-794
- demo 2-795
- demos
 - in Command Window 2-859
- density
 - of sparse matrix 2-2098
- depdir 2-800
- dependence, linear 2-2946
- dependent functions 2-2389
- defun 2-801
- derivative
 - approximate 2-816
 - polynomial 2-2339
- det 2-805
- detecting
 - alphabetic characters 2-1674

- empty arrays 2-1651
- global variables 2-1664
- logical arrays 2-1675
- members of a set 2-1676
- objects of a given class 2-1643
- positive, negative, and zero array
 - elements 2-2715
 - sparse matrix 2-1706
- determinant of a matrix 2-805
- detrend 2-806
- detrend (timeseries) 2-808
- deval 2-809
- diag 2-811
- diagonal 2-811
 - anti- 2-1373
 - k-th (illustration) 2-3150
 - main 2-811
 - sparse 2-2751
- dialog 2-813
- dialog box
 - error 2-921
 - help 2-1416
 - input 2-1576
 - list 2-1851
 - message 2-2057
 - print 1-88 1-100 2-2378
 - question 1-100 2-2439
 - warning 1-101 2-3413
- diary 2-814
- Diary, Root property 2-2593
- DiaryFile, Root property 2-2593
- diff 2-816
- differences
 - between adjacent array elements 2-816
 - between sets 2-2697
- differential equation solvers
 - defining an ODE problem 2-2138
 - ODE boundary value problems 2-345
 - adjusting parameters 2-360
 - extracting properties 2-356
 - extracting properties of 2-924 to 2-925
 - 2-3147 to 2-3148
 - forming initial guess 2-357
 - ODE initial value problems 2-2125
 - adjusting parameters of 2-2146
 - extracting properties of 2-2145
 - parabolic-elliptic PDE problems 2-2277
- diffuse 2-818
- DiffuseStrength
 - Surface property 2-2979
 - surfaceplot property 2-3000
- DiffuseStrengthpatch property 2-2233
- digamma function 2-2399
- dimension statement (lack of in MATLAB) 2-3497
- dimensions
 - size of 2-2722
- Diophantine equations 2-1267
- dir 2-819
- dir (ftp) function 2-822
- direct term of a partial fraction
 - expansion 2-2569
- directories 2-409
 - adding to search path 2-65
 - checking existence of 2-938
 - copying 2-613
 - creating 2-2006
 - listing contents of 2-819
 - listing MATLAB files in 2-3436
 - listing, on UNIX 2-1885
 - MATLAB
 - caching 2-2251
 - removing 2-2583
 - removing from search path 2-2588
 - See also* directory, search path
- directory 2-819
 - changing on FTP server 2-411

- listing for FTP server 2-822
- making on FTP server 2-2009
- MATLAB location 2-1934
- root 2-1934
- temporary system 2-3060
- See also* directories
- directory, changing 2-409
- directory, current 2-2413
- disconnect 2-492
- discontinuities, eliminating (in arrays of phase angles) 2-3357
- discontinuities, plotting functions with 2-979
- discontinuous problems 2-1142
- disp 2-824
 - memmapfile object 2-1965
 - serial port I/O 2-825
 - timer object 2-826
- display 2-828
- display format 2-1152
- displaying output in Command Window 2-2042
- DisplayName
 - areaserie property 2-149
 - barserie property 2-274
 - contour property 2-578
 - errorbar property 2-908
 - lineserie property 2-1823
 - quivergroup property 2-2454
 - scatter property 2-2650
 - stairserie property 2-2815
 - stem property 2-2848
- distribution
 - Gaussian 2-896
- Dithermap 2-1040
- DithermapMode, Figure property 2-1040
- division
 - array, left (arithmetic operator) 2-4
 - array, right (arithmetic operator) 2-3
 - by zero 2-1564
 - matrix, left (arithmetic operator) 2-3
 - matrix, right (arithmetic operator) 2-3
 - of polynomials 2-762
- divisor
 - greatest common 2-1267
- dll libraries
 - MATLAB functions
 - calllib 2-371
 - libfunctions 2-1777
 - libfunctionsview 2-1779
 - libisloaded 2-1781
 - libpointer 2-1783
 - libstruct 2-1785
 - loadlibrary 2-1861
 - unloadlibrary 2-3345
- dlmread 2-832
- dlmwrite 2-836
- dmperm 2-840
- Dockable, Figure property 2-1040
- docsearch 2-845
- documentation
 - displaying online 2-1413
- dolly camera 2-374
- dos 2-847
 - UNC pathname error 2-848
- dot 2-849
- dot product 2-638 2-849
- dot-parentheses (special characters) 2-20
- double 1-57 2-850
- double click, detecting 2-1061
- double integral
 - numerical evaluation 2-697
- DoubleBuffer, Figure property 2-1041
- downloading files from FTP server 2-1994
- dragrect 2-851
- drawing shapes
 - circles and rectangles 2-2501
- DrawMode, Axes property 2-227
- drawnow 2-853
- dsearch 2-855
- dsearchn 2-856

Dulmage-Mendelsohn decomposition 2-840
dynamic fields 2-20

E

echo 2-857
Echo, Root property 2-2593
echodemo 2-859
edge finding, Sobel technique 2-602
EdgeAlpha
 patch property 2-2233
 surface property 2-2979
 surfaceplot property 2-3000
EdgeColor
 annotation ellipse property 2-113
 annotation rectangle property 2-118
 annotation textbox property 2-129
 areaseries property 2-149
 barseries property 2-274
 patch property 2-2233
 Surface property 2-2980
 surfaceplot property 2-3000
 Text property 2-3076
EdgeColor, rectangle property 2-2509
EdgeLighting
 patch property 2-2234
 Surface property 2-2980
 surfaceplot property 2-3001
editable text 2-3210
editing
 M-files 2-861
eig 2-863
eigensystem
 transforming 2-412
eigenvalue
 accuracy of 2-863
 complex 2-412
 matrix logarithm and 2-1878
 modern approach to computation of 2-2335
 of companion matrix 2-541

 problem 2-864 2-2340
 problem, generalized 2-864 2-2340
 problem, polynomial 2-2340
 repeated 2-865
 Wilkinson test matrix and 2-3456
eigenvalues
 effect of roundoff error 2-255
 improving accuracy 2-255
eigenvector
 left 2-864
 matrix, generalized 2-2469
 right 2-864
eigs 2-869
elevation (spherical coordinates) 2-2765
elevation of viewpoint 2-3389
ellipj 2-879
ellipke 2-881
ellipsoid 1-86 2-883
elliptic functions, Jacobian
 (defined) 2-879
elliptic integral
 complete (defined) 2-881
 modulus of 2-879 2-881
else 2-885
elseif 2-886
Enable
 Uicontrol property 2-3222
 Uimenu property 2-3262
 Uipushtool property 2-3295
 Uitogglehtool property 2-3320
end 2-889
end caps for isosurfaces 2-1682
end of line, indicating 2-21
end-of-file indicator 2-993
eomday 2-891
eps 2-892
eq 2-894
equal arrays
 detecting 2-1654 2-1657
equal sign (special characters) 2-19

- equations, linear
 - accuracy of solution 2-548
 - EraseMode
 - areaserie property 2-150
 - barserie property 2-275
 - contour property 2-578
 - errorbar property 2-908
 - hggroup property 2-1431
 - hgtransform property 2-1451
 - Image property 2-1513
 - Line property 2-1812
 - lineserie property 2-1823
 - quivergroup property 2-2454
 - rectangle property 2-2510
 - scatter property 2-2650
 - stairserie property 2-2815
 - stem property 2-2848
 - Surface property 2-2981
 - surfaceplot property 2-3002
 - Text property 2-3078
 - EraseModepatch property 2-2235
 - error 2-898
 - roundoff, *see* roundoff error
 - error function
 - complementary 2-896
 - (defined) 2-896
 - scaled complementary 2-896
 - error message
 - displaying 2-898
 - Index into matrix is negative or zero 2-1874
 - retrieving last generated 2-1742 2-1749
 - error messages
 - Out of memory 2-2201
 - error tolerance
 - BVP problems 2-361
 - DDE problems 2-739
 - ODE problems 2-2147
 - errorbars 2-902
 - errordlg 2-921
 - ErrorMessage, Root property 2-2593
 - errors
 - in file input/output 2-994
 - ErrorType, Root property 2-2594
 - escape characters in format specification
 - string 2-1167 2-2788
 - etime 2-923
 - etree 2-924
 - etreeplot 2-925
 - eval 2-926
 - evalc 2-928
 - evalin 2-929
 - event location (DDE) 2-745
 - event location (ODE) 2-2154
 - eventlisteners 2-931
 - events 2-933
- examples
 - calculating isosurface normals 2-1689
 - contouring mathematical expressions 2-952
 - isosurface end caps 2-1682
 - isosurfaces 2-1693
 - mesh plot of mathematical function 2-961
 - mesh/contour plot 2-965
 - plotting filled contours 2-956
 - plotting function of two variables 2-969
 - plotting parametric curves 2-972
 - polar plot of function 2-975
 - reducing number of patch faces 2-2520
 - reducing volume data 2-2523
 - subsampling volume data 2-2951
 - surface plot of mathematical function 2-979
 - surface/contour plot 2-983
 - Excel spreadsheets
 - loading 2-3474
 - exclamation point (special characters) 2-21
 - Execute 2-935
 - executing statements repeatedly 2-1150 2-3443

execution
 improving speed of by setting aside
 storage 2-3497
 pausing M-file 2-2257
 resuming from breakpoint 2-695
 time for M-files 2-2389

exifread 2-937

exist 2-938

exit 2-942

exp 2-943

expint 2-944

expm 2-945

expm1 2-947

exponential 2-943
 complex (defined) 2-943
 integral 2-944
 matrix 2-945

exponentiation
 array (arithmetic operator) 2-4
 matrix (arithmetic operator) 2-4

export2wsdlg 2-948

extension, filename
 .m 2-1214
 .mat 2-2623

Extent
 Text property 2-3079
 Uicontrol property 2-3222

eye 2-950

ezcontour 2-951

ezcontourf 2-955

ezmesh 2-959

ezmeshc 2-963

ezplot 2-967

ezplot3 2-971

ezpolar 2-974

ezsurf 2-977

ezsurf c 2-981

F

F-norm 2-2101

FaceAlpha
 annotation textbox property 2-130

FaceAlpha patch property 2-2236

FaceAlphasurface property 2-2982

FaceAlphasurfaceplot property 2-3003

FaceColor
 annotation ellipse property 2-113
 annotation rectangle property 2-118
 areaserie s property 2-151
 barseries property 2-276
 Surface property 2-2983
 surfaceplot property 2-3004

FaceColor, rectangle property 2-2511

FaceColor patch property 2-2236

FaceLighting
 Surface property 2-2983
 surfaceplot property 2-3004

FaceLighting patch property 2-2237

faces, reducing number in patches 1-99 2-2519

Faces, patch property 2-2237

FaceVertexAlphaData, patch property 2-2238

FaceVertexCData, patch property 2-2239

factor 2-985

factorial 2-986

factorization 2-2420
 LU 2-1901
 QZ 2-2341 2-2469
 See also decomposition

factorization, Cholesky 2-454
 (as algorithm for solving linear
 equations) 2-2019
 minimum degree ordering and
 (sparse) 2-3043
 preordering for 2-533

factors, prime 2-985

false 2-987

fclose 2-988
 serial port I/O 2-989

- feather 2-991
- feof 2-993
- ferror 2-994
- feval 2-995
- Feval 2-997
- fft 2-1002
- FFT, *see* Fourier transform
- fft2 2-1007
- fftn 2-1008
- fftshift 2-1010
- fftw 2-1012
- FFTW 2-1005
- fgetl 2-1017
 - serial port I/O 2-1018
- fgets 2-1021
 - serial port I/O 2-1022
- field names of a structure, obtaining 2-1025
- fieldnames 2-1025
- fields, noncontiguous, inserting data into 2-1228
- fields, of structures
 - dynamic 2-20
- fig files
 - annotating for printing 2-1176
- figure 2-1027
- Figure
 - creating 2-1027
 - defining default properties 2-1029
 - properties 2-1030
 - redrawing 1-92 2-2526
- figure windows, displaying 2-1107
- figurepalette 1-83 2-1073
- figures
 - annotating 2-2320
 - opening 2-2168
 - saving 2-2633
- Figures
 - updating from M-file 2-853
- file
 - extension, getting 2-1085
 - modification date 2-819
 - position indicator
 - finding 2-1207
 - setting 2-1205
 - setting to start of file 2-1194
- file formats
 - getting list of supported formats 2-1526
 - reading 2-1534
 - writing 2-1544
- file size
 - querying 2-1523
- fileattrib 2-1075
- filebrowser 2-1081
- filemarker 2-1084
- filename
 - building from parts 2-1211
 - parts 2-1085
 - temporary 2-3061
- filename extension
 - .m 2-1214
 - .mat 2-2623
- fileparts 2-1085
- files 2-988
 - ASCII delimited
 - reading 2-832
 - writing 2-836
 - beginning of, rewinding to 2-1194 2-1531
 - checking existence of 2-938
 - closing 2-988
 - contents, listing 2-3173
 - copying 2-613
 - deleting 2-782
 - deleting on FTP server 2-786
 - end of, testing for 2-993
 - errors in input or output 2-994
 - Excel spreadsheets
 - loading 2-3474
 - fig 2-2633
 - figure, saving 2-2633
 - finding position within 2-1207

- getting next line 2-1017
 - getting next line (with line terminator) 2-1021
 - listing
 - in directory 2-3436
 - names in a directory 2-819
 - listing contents of 2-3173
 - locating 2-3440
 - mdl 2-2633
 - mode when opened 2-1144
 - model, saving 2-2633
 - opening 2-1145 2-2168
 - in Web browser 1-5 1-8 2-3430
 - opening in Windows applications 2-3457
 - path, getting 2-1085
 - pathname for 2-3440
 - reading
 - binary 2-1179
 - data from 2-3098
 - formatted 2-1195
 - reading image data from 2-1534
 - rewinding to beginning of 2-1194 2-1531
 - setting position within 2-1205
 - size, determining 2-821
 - sound
 - reading 2-197 2-3426
 - writing 2-198 to 2-199 2-3429
 - startup 2-1932
 - version, getting 2-1085
 - .wav
 - reading 2-3426
 - writing 2-3429
 - WK1
 - loading 2-3461
 - writing to 2-3463
 - writing binary data to 2-1228
 - writing formatted data to 2-1165
 - writing image data to 2-1544
 - See also* file
- filesep 2-1086
- fill 2-1087
 - Fill
 - contour property 2-579
 - fill3 2-1090
 - filter 2-1093
 - digital 2-1093
 - finite impulse response (FIR) 2-1093
 - infinite impulse response (IIR) 2-1093
 - two-dimensional 2-600
 - filter (timeseries) 2-1096
 - filter2 2-1099
 - find 2-1101
 - findall function 2-1106
 - findfigs 2-1107
 - finding 2-1101
 - sign of array elements 2-2715
 - zero of a function 2-1234
 - See also* detecting
 - findobj 2-1108
 - findstr 2-1111
 - finish 2-1112
 - finish.m 2-2441
 - FIR filter 2-1093
 - fitsinfo 2-1113
 - fitsread 2-1123
 - fix 2-1125
 - fixed-width font
 - axes 2-228
 - text 2-3080
 - uicontrols 2-3224
 - FixedColors, Figure property 2-1041
 - FixedWidthFontName, Root property 2-2594
 - flints 2-2063
 - flipdim 2-1126
 - fliplr 2-1127
 - flipud 2-1128
 - floating-point
 - integer, maximum 2-331
 - floating-point arithmetic, IEEE
 - smallest positive number 2-2496

- floor 2-1130
- flops 2-1131
- flow control
 - break 2-341
 - case 2-398
 - end 2-889
 - error 2-899
 - for 2-1150
 - keyboard 2-1739
 - otherwise 2-2200
 - return 2-2576
 - switch 2-3030
 - while 2-3443
- fminbnd 2-1133
- fminsearch 2-1138
- font
 - fixed-width, axes 2-228
 - fixed-width, text 2-3080
 - fixed-width, uicontrols 2-3224
- FontAngle
 - annotation textbox property 2-132
 - Axes property 2-227
 - Text property 2-122 2-3079
 - Uicontrol property 2-3223
- FontName
 - annotation textarrow property 2-122
 - annotation textbox property 2-132
 - Axes property 2-227
 - Text property 2-3080
 - Uicontrol property 2-3223
- fonts
 - bold 2-122 2-132 2-3081
 - italic 2-122 2-3079
 - specifying size 2-3080
 - TeX characters
 - bold 2-3093
 - italics 2-3093
 - specifying family 2-3093
 - specifying size 2-3093
 - units 2-122 2-132 2-3081
- FontSize
 - annotation textarrow property 2-122
 - annotation textbox property 2-132
 - Axes property 2-228
 - Text property 2-3080
 - Uicontrol property 2-3224
- FontUnits
 - Axes property 2-229
 - Text property 2-3081
 - Uicontrol property 2-3224
- FontWeight
 - annotation textarrow property 2-122
 - annotation textbox property 2-132
 - Axes property 2-229
 - Text property 2-3081
 - Uicontrol property 2-3225
- fopen 2-1143
 - serial port I/O 2-1148
- for 2-1150
- ForegroundColor
 - Uicontrol property 2-3225
 - Uimenu property 2-3262
- format 2-1152
 - precision when writing 2-1179
 - reading files 2-1196
 - specification string, matching file data to 2-2803
- Format 2-2594
- formats
 - big endian 2-1145
 - little endian 2-1145
- FormatSpacing, Root property 2-2595
- formatted data
 - reading from file 2-1195
 - writing to file 2-1165
- formatting data 2-2786
- Fourier transform
 - algorithm, optimal performance of 2-1005 2-1488 2-1490 2-2097
 - as method of interpolation 2-1616

- convolution theorem and 2-598
- discrete, n-dimensional 2-1008
- discrete, one-dimensional 2-1002
- discrete, two-dimensional 2-1007
- fast 2-1002
- inverse, n-dimensional 2-1492
- inverse, one-dimensional 2-1488
- inverse, two-dimensional 2-1490
- shifting the zero-frequency component
 - of 2-1011
- fplot 2-1160 2-1175
- fprintf 2-1165
 - serial port I/O 2-1172
- fraction, continued 2-2483
- fragmented memory 2-2201
- frame2im 2-1175
- frames 2-3210
- frames for printing 2-1176
- fread 2-1179
 - serial port I/O 2-1189
- freqspace 2-1193
- frequency response
 - desired response matrix
 - frequency spacing 2-1193
- frequency vector 2-1881
- frewind 2-1194
- fscanf 2-1195
 - serial port I/O 2-1201
- fseek 2-1205
- ftell 2-1207
- FTP
 - connecting to server 2-1208
- ftp function 2-1208
- full 2-1210
- fullfile 2-1211
- func2str 2-1212
- function 2-1214
- function handle 2-1216
- function handles
 - overview of 2-1216

- function syntax 2-1409 2-3049
- functions 2-1219
 - call history 2-2394
 - call stack for 2-702
 - checking existence of 2-938
 - clearing from workspace 2-480
 - finding using keywords 2-1882
 - help for 2-1408 2-1418
 - in memory 2-1571
 - locating 2-3440
 - pathname for 2-3440
 - that work down the first non-singleton
 - dimension 2-2709
- funm 2-1223
- fwrite 2-1228
 - serial port I/O 2-1230
- fzero 2-1234

G

- gallery 2-1239
- gamma function
 - (defined) 2-1262
 - incomplete 2-1262
 - logarithm of 2-1262
 - logarithmic derivative 2-2399
- Gaussian distribution function 2-896
- Gaussian elimination
 - (as algorithm for solving linear
 - equations) 2-1631 2-2020
 - Gauss Jordan elimination with partial
 - pivoting 2-2618
 - LU factorization 2-1901
- gca 2-1264
- gcbf function 2-1265
- gcbo function 2-1266
- gcd 2-1267
- gcf 2-1269
- gco 2-1270
- ge 2-1271

- generalized eigenvalue problem 2-864 2-2340
- generating a sequence of matrix names (M1 through M12) 2-927
- genpath 2-1273
- genvarname 2-1275
- geodesic dome 2-3044
- get 1-106 2-1280 2-1283
 - memmapfile object 2-1966
 - serial port I/O 2-1285
 - timer object 2-1287
- get (timeseries) 2-1289
- get (tscollection) 2-1292
- getabstime (timeseries) 2-1290
- getabstime (tscollection) 2-1293
- getappdata function 2-1295
- getdatasamplesize 2-1298
- getenv 2-1299
- getfield 2-1300
- getframe 2-1302
 - image resolution and 2-1303
- getinterpmethod 2-1308
- getpref function 2-1310
- getqualitydesc 2-1312
- getsamplusingtime (timeseries) 2-1313
- getsamplusingtime (tscollection) 2-1314
- gettimeseriesnames 2-1315
- gettsafteratevent 2-1316
- gettsaftererevent 2-1317
- gettsatevent 2-1318
- gettsbeforeatevent 2-1319
- gettsbeforeerevent 2-1320
- gettsbetweenevents 2-1321
- GIF files
 - reading 2-1535
 - writing 2-1545
- ginput function 2-1326
- global 2-1328
- global variable
 - defining 2-1328
- global variables, clearing from
 - workspace 2-480
- gmres 2-1330
- golden section search 2-1136
- Goup
 - defining default properties 2-1446
- gplot 2-1336
- grabcode function 2-1338
- gradient 2-1340
- gradient, numerical 2-1340
- graph
 - adjacency 2-840
- Graphics Interchange Format (GIF) files
 - reading 2-1535
- graphics objects
 - Axes 2-205
 - Figure 2-1027
 - getting properties 2-1280
 - Image 2-1499
 - Light 2-1792
 - Line 2-1804
 - Patch 2-2218
 - resetting properties 1-97 2-2566
 - Root 1-90 2-2590
 - setting properties 1-69 1-90 1-93 1-108 1-110 2-2681
 - Surface 1-90 1-94 2-2968
 - Text 1-90 2-3067
 - uicontextmenu 2-3199
 - Uicontrol 2-3209
 - Uimenu 1-103 2-3254
- graphics objects, deleting 2-782
- graphs
 - editing 2-2320
- graymon 2-1343
- greatest common divisor 2-1267
- Greek letters and mathematical symbols 2-3091
- grid 2-1344
 - aligning data to a 2-1346

- grid arrays
 - for volumetric plots 2-1983
 - multi-dimensional 2-2089
- griddata 2-1346
- griddata3 2-1350
- griddatan 2-1353
- GridLineStyle, Axes property 2-229
- group
 - hggroup function 2-1425
- gsvd 2-1356
- gt 2-1362
- gtext 2-1364
- guidata function 2-1365
- guihandles function 2-1368
- GUIs, printing 2-2372
- gunzip 2-1369 2-1371

H

- H1 line 2-1410 to 2-1411
- hadamard 2-1372
- Hadamard matrix 2-1372
 - subspaces of 2-2946
- handle graphics
 - hgtransform 2-1442
- handle graphicshggroup 2-1425
- HandleVisibility
 - areaseries property 2-151
 - Axes property 2-229
 - barseries property 2-277
 - contour property 2-580
 - errorbar property 2-909
 - Figure property 2-1042
 - hggroup property 2-1432
 - hgtransform property 2-1453
 - Image property 2-1514
 - Light property 2-1796
 - Line property 2-1813
 - lineseries property 2-1824
 - patch property 2-2241

- quivergroup property 2-2455
- rectangle property 2-2511
- Root property 2-2595
- scatter property 2-2651
- stairs series property 2-2817
- stem property 2-2849
- Surface property 2-2984
- surfaceplot property 2-3005
- Text property 2-3081
- Uicontextmenu property 2-3205
- Uicontrol property 2-3225
- Uimenu property 2-3262
- Uipushtool property 2-3296
- Uitoggletool property 2-3320
- Uitoolbar property 2-3332
- hankel 2-1373
- Hankel matrix 2-1373
- HDF
 - appending to when saving
 - (WriteMode) 2-1549
 - compression 2-1548
 - setting JPEG quality when writing 2-1549
- HDF files
 - reading images from 2-1535
 - writing images 2-1545
- HDF4
 - summary of capabilities 2-1374
- HDF5
 - high-level access 2-1376
 - summary of capabilities 2-1376
- HDF5 class
 - low-level access 2-1376
- hdf5info 2-1379
- hdf5read 2-1381
- hdf5write 2-1383
- hdfinfo 2-1387
- hdfread 2-1395
- hdfstool 2-1407
- Head1Length
 - annotation doublearrow property 2-108

- Head1Style
 - annotation doublearrow property 2-109
- Head1Width
 - annotation doublearrow property 2-110
- Head2Length
 - annotation doublearrow property 2-108
- Head2Style
 - annotation doublearrow property 2-109
- Head2Width
 - annotation doublearrow property 2-110
- HeadLength
 - annotation arrow property 2-104
 - annotation textarrow property 2-123
- HeadStyle
 - annotation arrow property 2-104
 - annotation textarrow property 2-123
- HeadWidth
 - annotation arrow property 2-105
 - annotation textarrow property 2-124
- Height
 - annotation ellipse property 2-113
- help 2-1408
 - contents file 2-1409
 - creating for M-files 2-1410
 - keyword search in functions 2-1882
 - online 2-1408
- Help browser 2-1413
 - accessing from doc 2-842
- Help Window 2-1418
- helpbrowser 2-1413
- helpdesk 2-1415
- helpdlg 2-1416
- helpwin 2-1418
- Hermite transformations, elementary 2-1267
- hess 2-1419
- Hessenberg form of a matrix 2-1419
- hex2dec 2-1422
- hex2num 2-1423
- hidden 2-1458
- Hierarchical Data Format (HDF) files
 - reading images from 2-1535
 - writing images 2-1545
- hilb 2-1459
- Hilbert matrix 2-1459
 - inverse 2-1634
- hist 2-1460
- hisc 2-1464
- HitTest
 - areaseries property 2-153
 - Axes property 2-231
 - barseries property 2-278
 - contour property 2-581
 - errorbar property 2-911
 - Figure property 2-1043
 - hggroup property 2-1434
 - hgtransform property 2-1454
 - Image property 2-1515
 - Light property 2-1798
 - Line property 2-1813
 - lineseries property 2-1824
 - Patch property 2-2242
 - quivergroup property 2-2457
 - rectangle property 2-2512
 - Root property 2-2595
 - scatter property 2-2653
 - stairs series property 2-2818
 - stem property 2-2851
 - Surface property 2-2985
 - surfaceplot property 2-3006
 - Text property 2-3082
 - Uicontrol property 2-3226
- HitTestArea
 - areaseries property 2-153
 - barseries property 2-278
 - contour property 2-581
 - errorbar property 2-911
 - quivergroup property 2-2457
 - scatter property 2-2653
 - stairs series property 2-2818
 - stem property 2-2851

hold 2-1467
 home 2-1469
 HorizontalAlignment
 Text property 2-3083
 textarrow property 2-124
 textbox property 2-132
 Uicontrol property 2-3226
 horzcat 2-1470
 horzcat (M-file function equivalent for
 [,]) 2-21
 horzcat (tscollection) 2-1472
 hostid 2-1473
 Householder reflections (as algorithm for
 solving linear equations) 2-2021
 hsv2rgb 2-1474
 HTML
 in Command Window 2-1927
 HTML browser
 in MATLAB 2-1413
 HTML files
 opening 1-5 1-8 2-3430
 hyperbolic
 cosecant 2-642
 cosecant, inverse 2-45
 cosine 2-624
 cosine, inverse 2-35
 cotangent 2-629
 cotangent, inverse 2-40
 secant 2-2669
 secant, inverse 2-170
 sine 2-2720
 sine, inverse 2-175
 tangent 2-3057
 tangent, inverse 2-184
 hyperlinks
 in Command Window 2-1927
 hyperplanes, angle between 2-2946
 hypot 2-1475

I

i 2-1478
 ICO files
 reading 2-1536
 icon images
 reading 2-1537
 idealfilter (timeseries) 2-1479
 identity matrix 2-950
 sparse 2-2762
 idivide 2-1482
 IEEE floating-point arithmetic
 smallest positive number 2-2496
 if 2-1484
 ifft 2-1488
 ifft2 2-1490
 ifftn 2-1492
 ifftshift 2-1494
 IIR filter 2-1093
 im2java 2-1496
 imag 2-1498
 image 2-1499
 Image
 creating 2-1499
 properties 2-1506
 image types
 querying 2-1524
 images
 file formats 2-1534 2-1544
 reading data from files 2-1534
 returning information about 2-1522
 writing to files 2-1544
 Images
 converting MATLAB image to Java
 Image 2-1496
 imagesc 2-1519
 imaginary 2-1498
 part of complex number 2-1498
 unit (sqrt(\xd0 1)) 2-1478 2-1718
 See also complex
 imfinfo

- returning file information 2-1522
- imformats 2-1526
- import 2-1529
- importdata 2-1531
- importing
 - Java class and package names 2-1529
- imread 2-1534
- imwrite 2-1544
- incomplete beta function
 - (defined) 2-306
- incomplete gamma function
 - (defined) 2-1262
- ind2sub 2-1560
- Index into matrix is negative or zero (error message) 2-1874
- indexing
 - logical 2-1873
- indicator of file position 2-1194
- indices, array
 - of sorted elements 2-2737
- Inf 2-1564
- inferiorto 2-1566
- infinity 2-1564
 - norm 2-2101
- info 2-1567
- information
 - returning file information 2-1522
- inheritance, of objects 2-478
- inline 2-1568
- inmem 2-1571
- inpolygon 2-1573
- input 2-1575
 - checking number of M-file arguments 2-2080
 - name of array passed as 2-1580
 - number of M-file arguments 2-2082
 - prompting users for 2-1575 2-1976
- inputdlg 2-1576
- inputname 2-1580
- inspect 2-1581
- installation, root directory of 2-1934
- instrcallback 2-1587
- instrfind 2-1589
- instrfindall 2-1591
 - example of 2-1592
- int2str 2-1594
- integer
 - floating-point, maximum 2-331
- integration
 - polynomial 2-2346
 - quadrature 2-2432
- interfaces 2-1597
- interp1 2-1599
- interp1q 2-1607
- interp2 2-1609
- interp3 2-1613
- interpft 2-1616
- interpn 2-1617
- interpolated shading and printing 2-2373
- interpolation
 - cubic method 2-1346 2-1599 2-1609 2-1613 2-1617
 - cubic spline method 2-1599 2-1609 2-1613 2-1617
 - FFT method 2-1616
 - linear method 2-1599 2-1609 2-1613 2-1617
 - multidimensional 2-1617
 - nearest neighbor method 2-1346 2-1599 2-1609 2-1613 2-1617
 - one-dimensional 2-1599
 - three-dimensional 2-1613
 - trilinear method 2-1346
 - two-dimensional 2-1609
- Interpreter
 - Text property 2-3084
 - textarrow property 2-124
 - textbox property 2-133
- interpstreamspeed 2-1620
- Interruptible

- areaseries property 2-153
- Axes property 2-231
- barseries property 2-279 to 2-280
- contour property 2-582
- errorbar property 2-912
- Figure property 2-1044
- hggroup property 2-1434
- hgtransform property 2-1454
- Image property 2-1515
- Light property 2-1798
- Line property 2-1815
- lineseries property 2-1826
- patch property 2-2242
- quivergroup property 2-2457
- rectangle property 2-2512
- Root property 2-2595
- scatter property 2-2654
- stairs series property 2-2819
- stem property 2-2851 2-2855
- Surface property 2-2985 2-3006
- Text property 2-3085
- Uicontextmenu property 2-3206
- Uicontrol property 2-3227
- Uimenu property 2-3263
- Uipushtool property 2-3297
- Uitoggletool property 2-3321
- Uitoolbar property 2-3333
- intersect 2-1624
- intmax 2-1625
- intmin 2-1626
- intwarning 2-1627
- inv 2-1631
- inverse
 - cosecant 2-42
 - cosine 2-32
 - cotangent 2-37
 - Fourier transform 2-1488 2-1490 2-1492
 - Hilbert matrix 2-1634
 - hyperbolic cosecant 2-45
 - hyperbolic cosine 2-35
 - hyperbolic cotangent 2-40
 - hyperbolic secant 2-170
 - hyperbolic sine 2-175
 - hyperbolic tangent 2-184
 - of a matrix 2-1631
 - secant 2-167
 - sine 2-172
 - tangent 2-179
 - tangent, four-quadrant 2-181
- inversion, matrix
 - accuracy of 2-548
- InvertHardCopy, Figure property 2-1044
- invhilb 2-1634
- invoke 2-1635
- involutary matrix 2-2217
- ipermute 2-1638
- iqr (timeseries) 2-1639
- is* 2-1641
- isa 2-1643
- isappdata function 2-1645
- iscell 2-1646
- iscellstr 2-1647
- ischar 2-1648
- iscom 2-1649
- isdir 2-1650
- isempty 2-1651
- isempty (timeseries) 2-1652
- isempty (tscollection) 2-1653
- isequal 2-1654
- isequalwithequalnans 2-1657
- isevent 2-1659
- isfield 2-1660
- isfinite 2-1662
- isfloat 2-1663
- isglobal 2-1664
- ishandle 2-1666
- isinf 2-1668
- isinteger 2-1669
- isinterface 2-1670
- isjava 2-1671

- iskeyword 2-1672
 - isletter 2-1674
 - islogical 2-1675
 - ismember 2-1676
 - ismethod 2-1678
 - isnan 2-1679
 - isnumeric 2-1680
 - isobject 2-1681
 - isocap 2-1682
 - isonormals 2-1689
 - isosurface 2-1692
 - calculate data from volume 2-1692
 - end caps 2-1682
 - vertex normals 2-1689
 - ispc 2-1695
 - ispref function 2-1696
 - isprime 2-1697
 - isprop 2-1698
 - isreal 2-1699
 - isscalar 2-1701
 - issorted 2-1702
 - isspace 2-1705 2-1708
 - issparse 2-1706
 - isstr 2-1707
 - isstruct 2-1711
 - isstudent 2-1712
 - isunix 2-1713
 - isvalid 2-1714
 - timer object 2-1715
 - isvarname 2-1716
 - isvector 2-1717
 - italics font
 - TeX characters 2-3093
- J**
- j 2-1718
 - Jacobi rotations 2-2784
 - Jacobian elliptic functions
 - (defined) 2-879
 - Jacobian matrix (BVP) 2-363
 - Jacobian matrix (ODE) 2-2156
 - generating sparse numerically 2-2157 2-2159
 - specifying 2-2156 2-2159
 - vectorizing ODE function 2-2157 to 2-2159
 - Java
 - class names 2-482 2-1529
 - objects 2-1671
 - Java Image class
 - creating instance of 2-1496
 - Java import list
 - adding to 2-1529
 - clearing 2-482
 - Java version used by MATLAB 2-3382
 - java_method 2-1724 2-1731
 - java_object 2-1733
 - javaaddath 2-1719
 - javachk 2-1725
 - javaclasspath 2-1727
 - javarmpath 2-1735
 - joining arrays, *see* concatenation
 - Joint Photographic Experts Group (JPEG)
 - reading 2-1536
 - writing 2-1545
 - JPEG
 - setting Bitdepth 2-1549
 - specifying mode 2-1549
 - JPEG comment
 - setting when writing a JPEG image 2-1549
 - JPEG files
 - parameters that can be set when writing 2-1549
 - reading 2-1536
 - writing 2-1545
 - JPEG quality
 - setting when writing a JPEG image 2-1549 2-1555
 - setting when writing an HDF image 2-1549

jvm
 version used by MATLAB 2-3382

K

K>> prompt
 keyboard function 2-1739
 keyboard 2-1739
 keyboard mode 2-1739
 terminating 2-2576
 KeyPressFcn
 Uicontrol property 2-3228
 KeyPressFcn, Figure property 2-1045
 keyword search in functions 2-1882
 keywords
 iskeyword function 2-1672
 kron 2-1740
 Kronecker tensor product 2-1740

L

Label, Uimenu property 2-3264
 labeling
 axes 2-3467
 matrix columns 2-824
 plots (with numeric values) 2-2112
 LabelSpacing
 contour property 2-582
 Laplacian 2-763
 largest array elements 2-1954
 lasterr 2-1742
 lasterror 2-1745
 lastwarn 2-1749
 LaTeX, see TeX 2-3091
 Layer, Axes property 2-231
 Layout Editor
 starting 2-1367
 lcm 2-1751
 LData
 errorbar property 2-912

LDataSource
 errorbar property 2-912
 ldivide (M-file function equivalent for .\) 2-7
 le 2-1759
 least common multiple 2-1751
 least squares
 polynomial curve fitting 2-2342
 problem, overdetermined 2-2303
 legend 2-1761
 properties 2-1766
 setting text properties 2-1766
 legendre 2-1769
 Legendre functions
 (defined) 2-1769
 Schmidt semi-normalized 2-1769
 length 2-1773
 serial port I/O 2-1774
 length (timeseries) 2-1775
 length (tscollection) 2-1776
 LevelList
 contour property 2-583
 LevelListMode
 contour property 2-583
 LevelStep
 contour property 2-583
 LevelStepMode
 contour property 2-583
 libfunctions 2-1777
 libfunctionsview 2-1779
 libisloaded 2-1781
 libpointer 2-1783
 libstruct 2-1785
 license 2-1788
 light 2-1792
 Light
 creating 2-1792
 defining default properties 2-1503 2-1793
 positioning in camera coordinates 2-378
 properties 2-1794
 Light object

- positioning in spherical coordinates 2-1801
 - lightangle 2-1801
 - lighting 2-1802
 - limits of axes, setting and querying 2-3469
 - line 2-1804
 - editing 2-2320
 - Line
 - creating 2-1804
 - defining default properties 2-1807
 - properties 2-1809 2-1820 2-2506
 - line numbers in M-files 2-717
 - linear audio signal 2-1803 2-2063
 - linear dependence (of data) 2-2946
 - linear equation systems
 - accuracy of solution 2-548
 - solving overdetermined 2-2422 to 2-2423
 - linear equation systems, methods for solving
 - Cholesky factorization 2-2019
 - Gaussian elimination 2-2020
 - Householder reflections 2-2021
 - matrix inversion (inaccuracy of) 2-1631
 - linear interpolation 2-1599 2-1609 2-1613 2-1617
 - linear regression 2-2342
 - linearly spaced vectors, creating 2-1850
 - LineColor
 - contour property 2-584
 - lines
 - computing 2-D stream 1-99 2-2875
 - computing 3-D stream 1-99 2-2877
 - drawing stream lines 1-99 2-2879
 - LineStyle 1-82 2-1834
 - LineStyle
 - annotation arrow property 2-105
 - annotation doublearrow property 2-110
 - annotation ellipse property 2-113
 - annotation line property 2-115
 - annotation rectangle property 2-119
 - annotation textarrow property 2-124
 - annotation textbox property 2-133
 - areaseries property 2-154
 - barseries property 2-279
 - contour property 2-584
 - errorbar property 2-913
 - Line property 2-1815
 - lineseries property 2-1826
 - patch property 2-2242
 - quivergroup property 2-2458
 - rectangle property 2-2513
 - stairs series property 2-2819
 - stem property 2-2852
 - surface object 2-2986
 - surfaceplot object 2-3007
 - text object 2-3086
- LineStyleOrder
 - Axes property 2-232
 - LineWidth
 - annotation arrow property 2-106
 - annotation doublearrow property 2-111
 - annotation ellipse property 2-114
 - annotation line property 2-116
 - annotation rectangle property 2-119
 - annotation textarrow property 2-125
 - annotation textbox property 2-133
 - areaseries property 2-154
 - Axes property 2-233
 - barseries property 2-280
 - contour property 2-585
 - errorbar property 2-913
 - Line property 2-1815
 - lineseries property 2-1826
 - Patch property 2-2243
 - quivergroup property 2-2458
 - rectangle property 2-2513
 - scatter property 2-2654
 - stairs series property 2-2820
 - stem property 2-2852
 - Surface property 2-2986
 - surfaceplot property 2-3007
 - text object 2-3087

- linkaxes 2-1839
- linkprop 2-1843
- links
 - in Command Window 2-1927
- linsolve 2-1847
- linspace 2-1850
- list boxes 2-3211
 - defining items 2-3234
- ListboxTop, Uicontrol property 2-3229
- listdlg 2-1851
- little endian formats 2-1145
- load 2-1853 2-1858
 - serial port I/O 2-1859
- loadlibrary 2-1861
- loadobj 2-1867
- Lobatto IIIa ODE solver 2-354
- local variables 2-1214 2-1328
- locking M-files 2-2033
- log 2-1869
 - saving session to file 2-814
- log10 [log010] 2-1870
- log1p 2-1871
- log2 2-1872
- logarithm
 - base ten 2-1870
 - base two 2-1872
 - complex 2-1869 to 2-1870
 - natural 2-1869
 - of beta function (natural) 2-308
 - of gamma function (natural) 2-1263
 - of real numbers 2-2494
 - plotting 2-1875
- logarithmic derivative
 - gamma function 2-2399
- logarithmically spaced vectors,
 - creating 2-1881
- logical 2-1873
- logical array
 - converting numeric array to 2-1873
 - detecting 2-1675
- logical indexing 2-1873
- logical operations
 - AND, bit-wise 2-327
 - OR, bit-wise 2-333
 - XOR 2-3494
 - XOR, bit-wise 2-337
- logical operators 2-14 2-16
- logical OR
 - bit-wise 2-333
- logical tests 2-1643
 - all 2-83
 - any 2-137
 - See also* detecting
- logical XOR 2-3494
 - bit-wise 2-337
- loglog 2-1875
- logm 2-1878
- logspace 2-1881
- lookfor 2-1882
- lossless compression
 - reading JPEG files 2-1536
- lossy compression
 - writing JPEG files with 2-1549
- Lotus WK1 files
 - loading 2-3461
 - writing 2-3463
- lower 2-1884
- lower triangular matrix 2-3150
- lowercase to uppercase 2-3361
- ls 2-1885
- lscov 2-1886
- lsqnonneg 2-1891
- lsqr 2-1894
- lt 2-1899
- lu 2-1901
- LU factorization 2-1901
 - storage requirements of (sparse) 2-2116
- luinc 2-1909

M

M-file

- debugging 2-1739
- displaying during execution 2-857
- function 2-1214
- function file, echoing 2-857
- naming conventions 2-1214
- pausing execution of 2-2257
- programming 2-1214
- script 2-1214
- script file, echoing 2-857

M-files

- checking existence of 2-938
- checking for problems 2-2023
- clearing from workspace 2-480
- creating
 - in MATLAB directory 2-2251
- debugging with profile 2-2389
- deleting 2-782
- editing 2-861
- line numbers, listing 2-717
- lint tool 2-2023
- listing names of in a directory 2-3436
- locking (preventing clearing) 2-2033
- opening 2-2168
- optimizing 2-2389
- problems, checking for 2-2023
- setting breakpoints 2-708 2-710
- unlocking (allowing clearing) 2-2075

M-Lint

- function 2-2023
- function for entire directory 2-2029
- HTML report 2-2029
- machine epsilon 2-3445
- magic 2-1916
- magic squares 2-1916

Margin

- annotation textbox property 2-134

Marker

- Line property 2-1816

- lineseries property 2-1827
- marker property 2-914
- Patch property 2-2243
- quivergroup property 2-2458
- scatter property 2-2654
- stairs series property 2-2820
- stem property 2-2853
- Surface property 2-2986
- surfaceplot property 2-3007

MarkerEdgeColor

- errorbar property 2-914
- Line property 2-1816
- lineseries property 2-1827
- Patch property 2-2244
- quivergroup property 2-2459
- scatter property 2-2655
- stairs series property 2-2821
- stem property 2-2854
- Surface property 2-2987
- surfaceplot property 2-3008

MarkerFaceColor

- errorbar property 2-915
- Line property 2-1817
- lineseries property 2-1828
- Patch property 2-2244
- quivergroup property 2-2460
- scatter property 2-2655
- stairs series property 2-2821
- stem property 2-2854
- Surface property 2-2988
- surfaceplot property 2-3009

MarkerSize

- errorbar property 2-915
- Line property 2-1817
- lineseries property 2-1828
- Patch property 2-2245
- quivergroup property 2-2460
- stairs series property 2-2822
- stem property 2-2854
- Surface property 2-2988

- surfaceplot property 2-3009
- mass matrix (ODE) 2-2160
 - initial slope 2-2161 to 2-2162
 - singular 2-2161
 - sparsity pattern 2-2161
 - specifying 2-2161
 - state dependence 2-2161
- MAT-file 2-2623
 - converting sparse matrix after loading from 2-2749
- MAT-files 2-1853
 - listing for directory 2-3436
- mat2cell 2-1920
- mat2str 2-1923
- material 2-1925
- MATLAB
 - directory location 2-1934
 - installation directory 2-1934
 - quitting 2-2441
 - startup 2-1932
 - version number, displaying 2-3375
- matlab : function 2-1927
- matlab (UNIX command) 2-1936
- matlab (Windows command) 2-1949
- matlab function for UNIX 2-1936
- matlab function for Windows 2-1949
- MATLAB startup file 2-2830
- matlab.mat 2-1853 2-2623
- matlabcolon function 2-1927
- matlabrc 2-1932
- matlabroot 2-1934
- \$matlabroot 2-1934
- matrices
 - preallocation 2-3497
- matrix 2-2
 - addressing selected rows and columns of 2-23
 - arrowhead 2-533
 - companion 2-541
 - complex unitary 2-2420
 - condition number of 2-548 2-2489
 - condition number, improving 2-255
 - converting to formatted data file 2-1165
 - converting to from string 2-2802
 - converting to vector 2-23
 - decomposition 2-2420
 - defective (defined) 2-865
 - detecting sparse 2-1706
 - determinant of 2-805
 - diagonal of 2-811
 - Dulmage-Mendelsohn decomposition 2-840
 - evaluating functions of 2-1223
 - exponential 2-945
 - flipping left-right 2-1127
 - flipping up-down 2-1128
 - Hadamard 2-1372 2-2946
 - Hankel 2-1373
 - Hermitian Toeplitz 2-3140
 - Hessenberg form of 2-1419
 - Hilbert 2-1459
 - identity 2-950
 - inverse 2-1631
 - inverse Hilbert 2-1634
 - inversion, accuracy of 2-548
 - involutary 2-2217
 - left division (arithmetic operator) 2-3
 - lower triangular 2-3150
 - magic squares 2-1916 2-2954
 - maximum size of 2-546
 - modal 2-863
 - multiplication (defined) 2-3
 - orthonormal 2-2420
 - Pascal 2-2217 2-2349
 - permutation 2-1901 2-2420
 - poorly conditioned 2-1459
 - power (arithmetic operator) 2-4
 - pseudoinverse 2-2303
 - reading files into 2-832
 - reduced row echelon form of 2-2618

- replicating 2-2558
- right division (arithmetic operator) 2-3
- rotating 90\xfb 2-2607
- Schur form of 2-2620 2-2662
- singularity, test for 2-805
- sorting rows of 2-2740
- sparse, *see* sparse matrix
- specialized 2-1239
- square root of 2-2796
- subspaces of 2-2946
- test 2-1239
- Toeplitz 2-3140
- trace of 2-811 2-3142
- transpose (arithmetic operator) 2-4
- transposing 2-20
- unimodular 2-1267
- unitary 2-3022
- upper triangular 2-3157
- Vandermonde 2-2344
- Wilkinson 2-2755 2-3456
- writing as binary data 2-1228
- writing formatted data to 2-1195
- writing to ASCII delimited file 2-836
- writing to spreadsheet 2-3463
- See also* array
- Matrix
 - hgtransform property 2-1455
- matrix functions
 - evaluating 2-1223
- matrix names, (M1 through M12) generating a sequence of 2-927
- matrix power, *see* matrix, exponential
- max 2-1954
- max (timeseries) 2-1955
- Max, Uicontrol property 2-3229
- MaxHeadSize
 - quivergroup property 2-2460
- maximum matching 2-840
- MDL-files
 - checking existence of 2-938
- mean 2-1959
- mean (timeseries) 2-1960
- median 2-1962
- median (timeseries) 2-1963
- median value of array elements 2-1962
- memmapfile 2-1969
- memory 2-1975
 - clearing 2-480
 - minimizing use of 2-2201
 - variables in 2-3449
- menu (of user input choices) 2-1976
- menu function 2-1976
- MenuBar, Figure property 2-1047
- mesh plot
 - tetrahedron 2-3062
- mesh size (BVP) 2-366
- meshc 1-94 2-1978
- meshgrid 2-1983
- MeshStyle, Surface property 2-2988
- MeshStyle, surfaceplot property 2-3009
- meshz 1-94 2-1978
- message
 - error *See* error message 2-3415
 - warning *See* warning message 2-3415
- methods 2-1985
 - inheritance of 2-478
 - locating 2-3440
- methodsview 2-1987
- mex 2-1989
- MEX-files
 - clearing from workspace 2-480
 - debugging on UNIX 2-699
 - listing for directory 2-3436
- mexext 2-1992
- mfilename 2-1993
- mget function 2-1994
- Microsoft Excel files
 - loading 2-3474
- min 2-1995
- min (timeseries) 2-1996

- Min, Uicontrol property 2-3230
 - MinColormap, Figure property 2-1047
 - minimum degree ordering 2-3043
 - MinorGridLineStyle, Axes property 2-233
 - minres 2-2000
 - minus (M-file function equivalent for -) 2-7
 - mislocked 2-2005
 - mkdir 2-2006
 - mkdir (ftp) 2-2009
 - mkpp 2-2010
 - mldivide (M-file function equivalent for \) 2-7
 - mlint 2-2023
 - mlintrpt 2-2029
 - suppressing messages 2-2032
 - mlock 2-2033
 - mmfileinfo 2-2034
 - mod 2-2037
 - modal matrix 2-863
 - mode 2-2039
 - mode objects
 - pan, using 2-2205
 - rotate3d, using 2-2611
 - zoom, using 2-3502
 - models
 - opening 2-2168
 - saving 2-2633
 - modification date
 - of a file 2-819
 - modified Bessel functions
 - relationship to Airy functions 2-77
 - modulo arithmetic 2-2037
 - MonitorPosition
 - Root property 2-2595
 - Moore-Penrose pseudoinverse 2-2303
 - more 2-2042 2-2063
 - move 2-2044
 - movefile 2-2046
 - movegui function 2-2049
 - movie 2-2051
 - movie2avi 2-2054
 - movies
 - exporting in AVI format 2-199
 - mpower (M-file function equivalent for ^) 2-7
 - mput function 2-2056
 - mrdivide (M-file function equivalent for /) 2-7
 - msgbox 2-2057
 - mtimes 2-2059
 - mtimes (M-file function equivalent for *) 2-7
 - mu-law encoded audio signals 2-1803 2-2063
 - multibandread 2-2064
 - multibandwrite 2-2069
 - multidimensional arrays 2-1773
 - concatenating 2-401
 - interpolation of 2-1617
 - longest dimension of 2-1773
 - number of dimensions of 2-2091
 - rearranging dimensions of 2-1638 2-2295
 - removing singleton dimensions of 2-2799
 - reshaping 2-2567
 - size of 2-2722
 - sorting elements of 2-2736
 - See also* array
 - multiple
 - least common 2-1751
 - multiplication
 - array (arithmetic operator) 2-3
 - matrix (defined) 2-3
 - of polynomials 2-598
 - multistep ODE solver 2-2135
 - munlock 2-2075
- ## N
- Name, Figure property 2-1048
 - namelengthmax 2-2077
 - naming conventions
 - M-file 2-1214
 - NaN 2-2078
 - NaN (Not-a-Number) 2-2078
 - returned by rem 2-2554

- nargchk 2-2080
- nargoutchk 2-2084
- native2unicode 2-2086
- ndgrid 2-2089
- ndims 2-2091
- ne 2-2092
- nearest neighbor interpolation 2-1346 2-1599
 - 2-1609 2-1613 2-1617
- newplot 2-2094
- NextPlot
 - Axes property 2-233
 - Figure property 2-1048
- nextpow2 2-2097
- nnz 2-2098
- no derivative method 2-1142
- noncontiguous fields, inserting data
 - into 2-1228
- nonzero entries
 - specifying maximum number of in sparse matrix 2-2746
- nonzero entries (in sparse matrix)
 - allocated storage for 2-2116
 - number of 2-2098
 - replacing with ones 2-2776
 - vector of 2-2100
- nonzeros 2-2100
- norm 2-2101
 - 1-norm 2-2101 2-2489
 - 2-norm (estimate of) 2-2103
 - F-norm 2-2101
 - infinity 2-2101
 - matrix 2-2101
 - pseudoinverse and 2-2303 2-2305
 - vector 2-2101
- normal vectors, computing for volumes 2-1689
- NormalMode
 - Patch property 2-2245
 - Surface property 2-2989
 - surfaceplot property 2-3010
- normest 2-2103
- not 2-2104
- not (M-file function equivalent for ~) 2-15
- notebook 2-2105
- now 2-2106
- nthroot 2-2107
- null 2-2108
- null space 2-2108
- num2cell 2-2110
- num2hex 2-2111
- num2str 2-2112
- number
 - of array dimensions 2-2091
- numbers
 - imaginary 2-1498
 - NaN 2-2078
 - plus infinity 2-1564
 - prime 2-2360
 - random 2-2472 2-2477
 - real 2-2493
 - smallest positive 2-2496
- NumberTitle, Figure property 2-1048
- numel 2-2114
- numeric format 2-1152
- numeric precision
 - format reading binary data 2-1179
- numerical differentiation formula ODE solvers 2-2136
- numerical evaluation
 - double integral 2-697
 - triple integral 2-3152
- nzmax 2-2116
- o**
- object
 - determining class of 2-1643
 - inheritance 2-478
- object classes, list of predefined 2-477 2-1643
- objects
 - Java 2-1671

- ODE file template 2-2139
- ODE solver properties
 - error tolerance 2-2147
 - event location 2-2154
 - Jacobian matrix 2-2156
 - mass matrix 2-2160
 - ode15s 2-2162
 - solver output 2-2149
 - step size 2-2153
- ODE solvers
 - backward differentiation formulas 2-2162
 - numerical differentiation formulas 2-2162
 - obtaining solutions at specific times 2-2124
 - variable order solver 2-2162
- ode15i function 2-2117
- odefile 2-2138
- odeget 2-2145
- odephas2 output function 2-2151
- odephas3 output function 2-2151
- odeplot output function 2-2151
- odeprint output function 2-2151
- odeset 2-2146
- odextend 2-2164
- off-screen figures, displaying 2-1107
- OffCallback
 - Uitoggletool property 2-3322
- OnCallback
 - Uitoggletool property 2-3323
- one-step ODE solver 2-2135
- ones 2-2167
- online documentation, displaying 2-1413
- online help 2-1408
- open 2-2168
- openfig 2-2172
- OpenGL 2-1054
 - autoselection criteria 2-1058
- opening
 - files in Windows applications 2-3457
- opening files 2-1145
- openvar 2-2179
- operating system
 - MATLAB is running on 2-546
- operating system command 1-4 1-11 2-3052
- operating system command, issuing 2-21
- operators
 - arithmetic 2-2
 - logical 2-14 2-16
 - overloading arithmetic 2-8
 - overloading relational 2-12
 - relational 2-12 2-1873
 - symbols 2-1408
- optimget 2-2180
- optimization parameters structure 2-2180 to 2-2181
- optimizing M-file execution 2-2389
- optimset 2-2181
- or 2-2185
- or (M-file function equivalent for |) 2-15
- ordeig 2-2187
- orderfields 2-2190
- ordering
 - minimum degree 2-3043
 - reverse Cuthill-McKee 2-3033 2-3044
- ordqz 2-2193
- ordschur 2-2195
- orient 2-2197
- orth 2-2199
- orthogonal-triangular decomposition 2-2420
- orthographic projection, setting and querying 2-387
- orthonormal matrix 2-2420
- otherwise 2-2200
- Out of memory (error message) 2-2201
- OuterPosition
 - Axes property 2-234
- output
 - checking number of M-file arguments 2-2084
 - controlling display format 2-1152

- in Command Window 2-2042
- number of M-file arguments 2-2082
- output points (ODE)
 - increasing number of 2-2149
- output properties (DDE) 2-741
- output properties (ODE) 2-2149
 - increasing number of output points 2-2149
- overdetermined equation systems,
 - solving 2-2422 to 2-2423
- overflow 2-1564
- overloading
 - arithmetic operators 2-8
 - relational operators 2-12
 - special characters 2-22

P

- P-files
 - checking existence of 2-938
- pack 2-2201
- pagesetupdlg 2-2203
- paging
 - of screen 2-1410
- paging in the Command Window 2-2042
- pan mode objects 2-2205
- PaperOrientation, Figure property 2-1049
- PaperPosition, Figure property 2-1049
- PaperPositionMode, Figure property 2-1049
- PaperSize, Figure property 2-1050
- PaperType, Figure property 2-1050
- PaperUnits, Figure property 2-1051
- parametric curve, plotting 2-971
- Parent
 - areaseries property 2-154
 - Axes property 2-235
 - barseries property 2-280
 - contour property 2-585
 - errorbar property 2-915
 - Figure property 2-1052
 - hggroup property 2-1435
 - hgtransform property 2-1455
 - Image property 2-1516
 - Light property 2-1798
 - Line property 2-1817
 - lineseries property 2-1828
 - Patch property 2-2245
 - quivergroup property 2-2460
 - rectangle property 2-2513
 - Root property 2-2596
 - scatter property 2-2656
 - stairs series property 2-2822
 - stem property 2-2854
 - Surface property 2-2989
 - surfaceplot property 2-3010
 - Text property 2-3089
 - Uicontextmenu property 2-3207
 - Uicontrol property 2-3231
 - Uimenu property 2-3265
 - Uipushtool property 2-3298
 - Uitoggletool property 2-3323
 - Uitoolbar property 2-3334
- parentheses (special characters) 2-19
- parseSoapResponse 2-2214
- partial fraction expansion 2-2569
- partialpath 2-2215
- pascal 2-2217
- Pascal matrix 2-2217 2-2349
- patch 2-2218
- Patch
 - converting a surface to 1-99 2-2966
 - creating 2-2218
 - defining default properties 2-2224
 - properties 2-2226
 - reducing number of faces 1-99 2-2519
 - reducing size of face 1-99 2-2711
- path 2-2250
 - adding directories to 2-65
 - building from parts 2-1211
 - current 2-2250
 - removing directories from 2-2588

- viewing 2-2255
- path2rc 2-2252
- pathdef 2-2253
- pathname
 - partial 2-2215
 - toolbox directory 2-3141
- pathnames
 - of functions or files 2-3440
 - relative 2-2215
- pathsep 2-2254
- pathtool 2-2255
- pause 2-2257
- pauses, removing 2-692
- pausing M-file execution 2-2257
- pbaspect 2-2259
- PBM
 - parameters that can be set when writing 2-1549
- PBM files
 - reading 2-1536
 - writing 2-1545
- pcg 2-2265
- pchip 2-2269
- pcode 2-2272
- pcolor 2-2273
- PCX files
 - reading 2-1536
 - writing 2-1546
- PDE, *see* Partial Differential Equations
- pdepe 2-2277
- pdeval 2-2289
- percent sign (special characters) 2-21
- percent-brace (special characters) 2-21
- perfect matching 2-840
- period (.), to distinguish matrix and array operations 2-2
- period (special characters) 2-20
- perl 2-2292
- perl function 2-2292
- Perl scripts in MATLAB 1-4 1-11 2-2292
- perms 2-2294
- permutation
 - matrix 2-1901 2-2420
 - of array dimensions 2-2295
 - random 2-2481
- permutations of n elements 2-2294
- permute 2-2295
- persistent 2-2296
- persistent variable 2-2296
- perspective projection, setting and querying 2-387
- PGM
 - parameters that can be set when writing 2-1549
- PGM files
 - reading 2-1536
 - writing 2-1546
- phase angle, complex 2-99
- phase, complex
 - correcting angles 2-3354
- pi 2-2298
- pie 2-2299
- pie3 2-2301
- pinv 2-2303
- planerot 2-2306
- platform MATLAB is running on 2-546
- playshow function 2-2307
- plot 2-2308
 - editing 2-2320
- plot (timeseries) 2-2315
- plot box aspect ratio of axes 2-2259
- plot editing mode
 - overview 2-2321
- Plot Editor
 - interface 2-2321 2-2396
- plot, volumetric
 - generating grid arrays for 2-1983
 - slice plot 1-87 1-99 2-2728
- PlotBoxAspectRatio, Axes property 2-236

- PlotBoxAspectRatioMode, Axes
 - property 2-236
- plottedit 2-2320
- plotting
 - 2-D plot 2-2308
 - 3-D plot 1-82 2-2316
 - contours (a 2-951
 - contours (ez function) 2-951
 - ez-function mesh plot 2-959
 - feather plots 2-991
 - filled contours 2-955
 - function plots 2-1160
 - functions with discontinuities 2-979
 - histogram plots 2-1460
 - in polar coordinates 2-974
 - isosurfaces 2-1692
 - loglog plot 2-1875
 - mathematical function 2-967
 - mesh contour plot 2-963
 - mesh plot 1-94 2-1978
 - parametric curve 2-971
 - plot with two y-axes 2-2327
 - ribbon plot 1-87 2-2580
 - rose plot 1-86 2-2603
 - scatter plot 2-2323
 - scatter plot, 3-D 1-87 2-2643
 - semilogarithmic plot 1-83 2-2672
 - stem plot, 3-D 1-85 2-2841
 - surface plot 1-94 2-2961
 - surfaces 1-86 2-977
 - velocity vectors 2-552
 - volumetric slice plot 1-87 1-99 2-2728
 - , *see* visualizing
- plus (M-file function equivalent for +) 2-7
- PNG
 - writing options for 2-1551
 - alpha 2-1554
 - background color 2-1553
 - chromaticities 2-1553
 - gamma 2-1553
 - interlace type 2-1551
 - resolution 2-1554
 - significant bits 2-1554
 - transparency 2-1553
- PNG files
 - reading 2-1536
 - reading alpha channel 2-1539
 - reading transparency chunk 2-1539
 - specifying background color chunk 2-1539
 - writing 2-1546
- PNM files
 - reading 2-1536
 - writing 2-1546
- Pointer, Figure property 2-1052
- PointerLocation, Root property 2-2596
- PointerShapeCData, Figure property 2-1052
- PointerShapeHotSpot, Figure
 - property 2-1053
- PointerWindow, Root property 2-2597
- pol2cart 2-2330
- polar 2-2332
- polar coordinates 2-2330
 - computing the angle 2-99
 - converting from Cartesian 2-396
 - converting to cylindrical or Cartesian 2-2330
 - plotting in 2-974
- poles of transfer function 2-2569
- poly 2-2334
- polyarea 2-2337
- polyder 2-2339
- polyeig 2-2340
- polyfit 2-2342
- polygamma function 2-2399
- polygon

- area of 2-2337
 - creating with patch 2-2218
 - detecting points inside 2-1573
- polyint 2-2346
- polynomial
 - analytic integration 2-2346
 - characteristic 2-2334 to 2-2335 2-2601
 - coefficients (transfer function) 2-2569
 - curve fitting with 2-2342
 - derivative of 2-2339
 - division 2-762
 - eigenvalue problem 2-2340
 - evaluation 2-2347
 - evaluation (matrix sense) 2-2349
 - make piecewise 2-2010
 - multiplication 2-598
- polyval 2-2347
- polyvalm 2-2349
- poorly conditioned
 - matrix 2-1459
- poorly conditioned eigenvalues 2-255
- pop-up menus 2-3211
 - defining choices 2-3234
- Portable Anymap files
 - reading 2-1536
 - writing 2-1546
- Portable Bitmap (PBM) files
 - reading 2-1536
 - writing 2-1545
- Portable Graymap files
 - reading 2-1536
 - writing 2-1546
- Portable Network Graphics files
 - reading 2-1536
 - writing 2-1546
- Portable pixmap format
 - reading 2-1537
 - writing 2-1546
- Position
 - annotation ellipse property 2-114
 - annotation line property 2-116
 - annotation rectangle property 2-119
 - Axes property 2-236
 - doublearrow property 2-111
 - Figure property 2-1053
 - Light property 2-1798
 - tarrow property 2-106
 - Text property 2-3090
 - textarrow property 2-125
 - textbox property 2-134
 - Uicontextmenu property 2-3207
 - Uicontrol property 2-3231
 - Uimenu property 2-3265
- position indicator in file 2-1207
- position of camera
 - dollying 2-374
- position of camera, setting and querying 2-385
- Position, rectangle property 2-2513
- PostScript
 - default printer 2-2365
 - levels 1 and 2 2-2365
 - printing interpolated shading 2-2373
- pow2 2-2351
- power 2-2352
 - matrix, *see* matrix exponential
 - of real numbers 2-2497
 - of two, next 2-2097
- power (M-file function equivalent for . ^) 2-7
- PPM
 - parameters that can be set when writing 2-1549
- PPM files
 - reading 2-1537
 - writing 2-1546
- ppval 2-2353
- preallocation
 - matrix 2-3497
- precision 2-1152
 - reading binary data writing 2-1179
- prefdir 2-2355

- preferences 2-2359
 - opening the dialog box 2-2359
 - prime factors 2-985
 - dependence of Fourier transform on 2-1005
 - 2-1007 to 2-1008
 - prime numbers 2-2360
 - primes 2-2360
 - print frames 2-1176
 - printdlg 1-88 1-100 2-2378
 - printdlg function 2-2378
 - printer
 - default for linux and unix 2-2365
 - printer drivers
 - GhostScript drivers 2-2362
 - interplotted shading 2-2373
 - MATLAB printer drivers 2-2362
 - printframe 2-1176
 - PrintFrame Editor 2-1176
 - printing
 - borders 2-1176
 - fig files with frames 2-1176
 - GUIs 2-2372
 - interpolated shading 2-2373
 - on MS-Windows 2-2372
 - with a variable filename 2-2375
 - with non-normal EraseMode 2-1514
 - 2-1813 2-1824 2-2235 2-2510 2-2982
 - 2-3003 2-3079
 - with print frames 2-1178
 - printing figures
 - preview 1-89 2-2379
 - printing tips 2-2371
 - printing, suppressing 2-20
 - printpreview 1-89 2-2379
 - prod 2-2387
 - product
 - cumulative 2-651
 - Kronecker tensor 2-1740
 - of array elements 2-2387
 - of vectors (cross) 2-638
 - scalar (dot) 2-638
 - profile 2-2389
 - profsave 2-2395
 - projection type, setting and querying 2-387
 - ProjectionType, Axes property 2-237
 - prompting users for input 2-1575 2-1976
 - propedit 2-2396 to 2-2397
 - proppanel 1-83 2-2398
 - pseudoinverse 2-2303
 - psi 2-2399
 - publish function 2-2401
 - push buttons 2-3211
 - PutFullMatrix 2-2406
 - pwd 2-2413
- ## Q
- qmr 2-2414
 - qr 2-2420
 - QR decomposition 2-2420
 - deleting column from 2-2425
 - qrdelete 2-2425
 - qrinsert 2-2427
 - qrupdate 2-2429
 - quad 2-2432
 - quad1 2-2435
 - quadrature 2-2432
 - quadv 2-2437
 - questdlg 1-100 2-2439
 - questdlg function 2-2439
 - quit 2-2441
 - quitting MATLAB 2-2441
 - quiver 2-2444
 - quiver3 2-2447
 - quotation mark
 - inserting in a string 2-1170
 - qz 2-2469
 - QZ factorization 2-2341 2-2469

R

- radio buttons 2-3211
- rand 2-2472
- randn 2-2477
- random
 - numbers 2-2472 2-2477
 - permutation 2-2481
 - sparse matrix 2-2782 to 2-2783
 - symmetric sparse matrix 2-2784
- randperm 2-2481
- range space 2-2199
- rank 2-2482
- rank of a matrix 2-2482
- RAS files
 - parameters that can be set when writing 2-1555
 - reading 2-1537
 - writing 2-1546
- RAS image format
 - specifying color order 2-1555
 - writing alpha data 2-1555
- Raster image files
 - reading 2-1537
 - writing 2-1546
- rational fraction approximation 2-2483
- rbbox 1-98 2-2487 2-2526
- rcond 2-2489
- rdivide (M-file function equivalent for ./) 2-7
- readasync 2-2490
- reading
 - binary files 2-1179
 - data from files 2-3098
 - formatted data from file 2-1195
 - formatted data from strings 2-2802
- readme files, displaying 1-5 2-1650 2-3439
- real 2-2493
- real numbers 2-2493
- realloc 2-2494
- realmax 2-2495
- realmin 2-2496
- realpow 2-2497
- realsqrt 2-2498
- rearranging arrays
 - converting to vector 2-23
 - removing first n singleton dimensions 2-2709
 - removing singleton dimensions 2-2799
 - reshaping 2-2567
 - shifting dimensions 2-2709
 - swapping dimensions 2-1638 2-2295
- rearranging matrices
 - converting to vector 2-23
 - flipping left-right 2-1127
 - flipping up-down 2-1128
 - rotating 90\xfb 2-2607
 - transposing 2-20
- record 2-2499
- rectangle
 - rectangle function 2-2501
- rectint 2-2516
- RecursionLimit
 - Root property 2-2597
- recycle 2-2517
- reduced row echelon form 2-2618
- reducepatch 2-2519
- reducevolume 2-2523
- reference page
 - accessing from doc 2-842
- refresh 2-2526
- regexprep 2-2541
- regexprtranslate 2-2545
- registerevent 2-2548
- regression
 - linear 2-2342
- regularly spaced vectors, creating 2-23 2-1850
- rehash 2-2550
- relational operators 2-12 2-1873
- relative accuracy
 - BVP 2-362
 - DDE 2-740

- norm of DDE solution 2-740
- norm of ODE solution 2-2148
- ODE 2-2148
- release 2-2552
- rem 2-2554
- removets 2-2555
- rename function 2-2557
- renderer
 - OpenGL 2-1054
 - painters 2-1054
 - zbuffer 2-1054
- Renderer, Figure property 2-1053
- RendererMode, Figure property 2-1057
- repeatedly executing statements 2-1150
 - 2-3443
- replicating a matrix 2-2558
- repmat 2-2558
- resample (timeseries) 2-2560
- resample (tscollection) 2-2563
- reset 2-2566
- reshape 2-2567
- residue 2-2569
- residues of transfer function 2-2569
- Resize, Figure property 2-1059
- ResizeFcn, Figure property 2-1059
- restoredefaultpath 2-2573
- rethrow 2-2574
- return 2-2576
- reverse Cuthill-McKee ordering 2-3033 2-3044
- rewinding files to beginning of 2-1194 2-1531
- RGB, converting to HSV 1-95 2-2577
- rgb2hsv 2-2577
- rgbplot 2-2578
- ribbon 2-2580
- right-click and context menus 2-3199
- rmapdata function 2-2582
- rmdir 2-2583
- rmdir (ftp) function 2-2586
- rmfield 2-2587
- rmpath 2-2588
- rmpref function 2-2589
- RMS, *see* root-mean-square
- rolling camera 2-388
- root 1-90 2-2590
- root directory 2-1934
- root directory for MATLAB 2-1934
- Root graphics object 1-90 2-2590
- root object 2-2590
- root, *see* rootobject 1-90 2-2590
- root-mean-square
 - of vector 2-2101
- roots 2-2601
- roots of a polynomial 2-2334 to 2-2335 2-2601
- rose 2-2603
- Rosenbrock
 - banana function 2-1140
 - ODE solver 2-2136
- rosser 2-2606
- rot90 2-2607
- rotate 2-2608
- rotate3d 2-2611
- rotate3d mode objects 2-2611
- rotating camera 2-382
- rotating camera target 1-96 2-384
- Rotation, Text property 2-3090
- rotations
 - Jacobi 2-2784
- round 2-2617
 - to nearest integer 2-2617
 - towards infinity 2-424
 - towards minus infinity 2-1130
 - towards zero 2-1125
- roundoff error
 - characteristic polynomial and 2-2335
 - convolution theorem and 2-598
 - effect on eigenvalues 2-255
 - evaluating matrix functions 2-1225
 - in inverse Hilbert matrix 2-1634
 - partial fraction expansion and 2-2570
 - polynomial roots and 2-2601

sparse matrix conversion and 2-2750
rref 2-2618
rrefmovie 2-2618
rsf2csf 2-2620
rubberband box 1-98 2-2487
run 2-2622
Runge-Kutta ODE solvers 2-2135
running average 2-1094

S

save 2-2623 2-2630
 serial port I/O 2-2631
saveas 2-2633
saveobj 2-2637
savepath 2-2639
saving
 ASCII data 2-2623
 session to a file 2-814
 workspace variables 2-2623
scalar product (of vectors) 2-638
scaled complementary error function
 (defined) 2-896
scatter 2-2640
scatter3 2-2643
scattered data, aligning
 multi-dimensional 2-2089
 two-dimensional 2-1346
scattergroup
 properties 2-2646
Schmidt semi-normalized Legendre
 functions 2-1769
schur 2-2662
Schur decomposition 2-2662
Schur form of matrix 2-2620 2-2662
screen, paging 2-1410
ScreenDepth, Root property 2-2597
ScreenPixelsPerInch, Root property 2-2598
ScreenSize, Root property 2-2598
script 2-2665
scrolling screen 2-1410
search path 2-2588
 adding directories to 2-65
 MATLAB's 2-2250
 modifying 2-2255
 viewing 2-2255
search, string 2-1111
sec 2-2666
secant 2-2666
 hyperbolic 2-2669
 inverse 2-167
 inverse hyperbolic 2-170
secd 2-2668
sech 2-2669
Selected
 areaserie property 2-155
 Axes property 2-238
 barseries property 2-280
 contour property 2-585
 errorbar property 2-915
 Figure property 2-1060
 hggroup property 2-1435
 hgtransform property 2-1455
 Image property 2-1516
 Light property 2-1799
 Line property 2-1817
 lineseries property 2-1828
 Patch property 2-2245
 quivergroup property 2-2460
 rectangle property 2-2514
 Root property 2-2599
 scatter property 2-2656
 stairs property 2-2822
 stem property 2-2855
 Surface property 2-2989
 surfaceplot property 2-3010
 Text property 2-3090
 Uicontrol property 2-3232
selecting areas 1-98 2-2487
SelectionHighlight

- areaserie property 2-155
- Axes property 2-238
- barseries property 2-280
- contour property 2-585
- errorbar property 2-916
- Figure property 2-1060
- hggroup property 2-1435
- hgtransform property 2-1455
- Image property 2-1516
- Light property 2-1799
- Line property 2-1818
- lineseries property 2-1829
- Patch property 2-2246
- quivergroup property 2-2461
- rectangle property 2-2514
- scatter property 2-2656
- stairs series property 2-2822
- stem property 2-2855
- Surface property 2-2989
- surfaceplot property 2-3010
- Text property 2-3090
- Uicontrol property 2-3232
- SelectionType, Figure property 2-1061
- selectmoveresize 2-2671
- semicolon (special characters) 2-20
- send 2-2675
- sendmail 2-2676
- Separator
 - Uipushtool property 2-3298
 - Uitoggetool property 2-3323
- Separator, Uimenu property 2-3265
- sequence of matrix names (M1 through M12)
 - generating 2-927
- serial 2-2678
- serialbreak 2-2680
- server (FTP)
 - connecting to 2-1208
- session
 - saving 2-814
- set 1-108 2-2681 2-2685
 - serial port I/O 2-2686
 - timer object 2-2689
- set (timeseries) 2-2692
- set (tscollection) 2-2693
- set operations
 - difference 2-2697
 - exclusive or 2-2706
 - intersection 2-1624
 - membership 2-1676
 - union 2-3338
 - unique 2-3340
- setabstime (timeseries) 2-2694
- setabstime (tscollection) 2-2695
- setappdata 2-2696
- setdiff 2-2697
- setenv 2-2698
- setfield 2-2699
- setinterpmethod 2-2701
- setpref function 2-2703
- setstr 2-2704
- settimeseriesnames 2-2705
- setxor 2-2706
- shading 2-2707
- shading colors in surface plots 1-95 2-2707
- ShareColors, Figure property 2-1062
- shared libraries
 - MATLAB functions
 - calllib 2-371
 - libfunctions 2-1777
 - libfunctionsview 2-1779
 - libisloaded 2-1781
 - libpointer 2-1783
 - libstruct 2-1785
 - loadlibrary 2-1861
 - unloadlibrary 2-3345
- shell script 1-4 1-11 2-3052 2-3343
- shiftdim 2-2709
- shifting array
 - circular 2-469
- ShowArrowHead

- quivergroup property 2-2461
- ShowHiddenHandles, Root property 2-2599
- showplottool 2-2710
- ShowText
 - contour property 2-585
- shrinkfaces 2-2711
- shutdown 2-2441
- sign 2-2715
- signum function 2-2715
- simplex search 2-1142
- Simpson's rule, adaptive recursive 2-2433
- Simulink
 - printing diagram with frames 2-1176
 - version number, displaying 2-3375
- sin 2-2716
- sind 2-2718
- sine 2-2716
 - hyperbolic 2-2720
 - inverse 2-172
 - inverse hyperbolic 2-175
- single 2-2719
- single quote (special characters) 2-20
- singular value
 - decomposition 2-2482 2-3022
 - largest 2-2101
 - rank and 2-2482
- sinh 2-2720
- size
 - array dimesions 2-2722
 - serial port I/O 2-2727
- size (timeseries) 2-2725
- size (tscollection) 2-2726
- size of array dimensions 2-2722
- size of fonts, see also FontSize property 2-3093
- size vector 2-2567
- SizeData
 - scatter property 2-2656
- skipping bytes (during file I/O) 2-1228
- slice 2-2728
- slice planes, contouring 2-593
- sliders 2-3212
- SliderStep, Uicontrol property 2-3232
- smallest array elements 2-1995
- smooth3 2-2734
- smoothing 3-D data 1-99 2-2734
- soccer ball (example) 2-3044
- solution statistics (BVP) 2-367
- sort 2-2736
- sorting
 - array elements 2-2736
 - complex conjugate pairs 2-633
 - matrix rows 2-2740
- sortrows 2-2740
- sound 2-2743 to 2-2744
 - converting vector into 2-2743 to 2-2744
 - files
 - reading 2-197 2-3426
 - writing 2-198 2-3429
 - playing 1-79 2-3424
 - recording 1-79 2-3427
 - resampling 1-79 2-3424
 - sampling 1-79 2-3427
- source control on UNIX platforms
 - checking out files
 - function 2-451
- source control system
 - viewing current system 2-494
- source control systems
 - checking in files 2-448
 - undo checkout 1-10 2-3336
- spalloc 2-2745
- sparse 2-2746
- sparse matrix
 - allocating space for 2-2745
 - applying function only to nonzero elements
 - of 2-2763
 - density of 2-2098
 - detecting 2-1706
 - diagonal 2-2751

- finding indices of nonzero elements
 - of 2-1101
 - identity 2-2762
 - minimum degree ordering of 2-500
 - number of nonzero elements in 2-2098
 - permuting columns of 2-533
 - random 2-2782 to 2-2783
 - random symmetric 2-2784
 - replacing nonzero elements of with ones 2-2776
 - results of mixed operations on 2-2747
 - solving least squares linear system 2-2421
 - specifying maximum number of nonzero elements 2-2746
 - vector of nonzero elements 2-2100
 - visualizing sparsity pattern of 2-2793
- sparse storage
 - criterion for using 2-1210
- spaugment 2-2748
- spconvert 2-2749
- spdiags 2-2751
- special characters
 - descriptions 2-1408
 - overloading 2-22
- specular 2-2761
- SpecularColorReflectance
 - Patch property 2-2246
 - Surface property 2-2989
 - surfaceplot property 2-3010
- SpecularExponent
 - Patch property 2-2246
 - Surface property 2-2990
 - surfaceplot property 2-3011
- SpecularStrength
 - Patch property 2-2246
 - Surface property 2-2990
 - surfaceplot property 2-3011
- speye 2-2762
- spfun 2-2763
- sph2cart 2-2765
- sphere 2-2766
- spherical coordinates
 - defining a Light position in 2-1801
- spherical coordinates 2-2765
- spinmap 2-2768
- spline 2-2769
- spline interpolation (cubic)
 - one-dimensional 2-1600 2-1610 2-1613 2-1617
- Spline Toolbox 2-1605
- spones 2-2776
- spparms 2-2777
- sprand 2-2782
- sprandn 2-2783
- sprandsym 2-2784
- sprank 2-2785
- spreadsheets
 - loading WK1 files 2-3461
 - loading XLS files 2-3474
 - reading into a matrix 2-832
 - writing from matrix 2-3463
 - writing matrices into 2-836
- sprintf 2-2786
- sqrt 2-2795
- sqrtn 2-2796
- square root
 - of a matrix 2-2796
 - of array elements 2-2795
 - of real numbers 2-2498
- squeeze 2-2799
- sscanf 2-2802
- stack, displaying 2-702
- standard deviation 2-2831
- start
 - timer object 2-2827
- startat
 - timer object 2-2828
- startup 2-2830
- startup file 2-2830
- startup files 2-1932

- State
 - Uitoggletool property 2-3323
- Stateflow
 - printing diagram with frames 2-1176
- static text 2-3212
- std 2-2831
- std (timeseries) 2-2833
- stem 2-2835
- stem3 2-2841
- step size (DDE)
 - initial step size 2-744
 - upper bound 2-745
- step size (ODE) 2-743 2-2153
 - initial step size 2-2153
 - upper bound 2-2153
- stop
 - timer object 2-2861
- stopasync 2-2862
- stopwatch timer 2-3123
- storage
 - allocated for nonzero entries
 - (sparse) 2-2116
 - sparse 2-2746
- storage allocation 2-3497
- str2cell 2-440
- str2double 2-2864
- str2func 2-2865
- str2mat 2-2867
- str2num 2-2868
- strcat 2-2870
- stream lines
 - computing 2-D 1-99 2-2875
 - computing 3-D 1-99 2-2877
 - drawing 1-99 2-2879
- stream2 2-2875
- stream3 2-2877
- stretch-to-fill 2-206
- strfind 2-2902
- string
 - converting from vector to 2-447
 - converting matrix into 2-1923 2-2112
 - converting to lowercase 2-1884
 - converting to numeric array 2-2868
 - converting to uppercase 2-3361
 - dictionary sort of 2-2740
 - finding first token in 2-2919
 - searching and replacing 2-2918
 - searching for 2-1111
- String
 - Text property 2-3090
 - textarrow property 2-125
 - textbox property 2-134
 - Uicontrol property 2-3233
- string matrix to cell array conversion 2-440
- strings 2-2904
 - converting to matrix (formatted) 2-2802
 - inserting a quotation mark in 2-1170
 - writing data to 2-2786
- strjust 1-51 1-63 2-2906
- strmatch 2-2907
- strread 2-2910
- strrep 1-51 1-63 2-2918
- strtok 2-2919
- strtrim 2-2922
- struct 2-2923
- struct2cell 2-2928
- structfun 2-2929
- structure array
 - field names of 2-1025
 - getting contents of field of 2-1300
 - remove field from 2-2587
 - setting contents of a field of 2-2699
- structures
 - dynamic fields 2-20
- strvcat 2-2932
- Style
 - Light property 2-1799
 - Uicontrol property 2-3234
- sub2ind 2-2934

- subfunction 2-1214
 - subplot 2-2936
 - subsasgn 1-54 2-2943
 - subscripts
 - in axis title 2-3138
 - in text strings 2-3094
 - subsindex 2-2945
 - subspace 1-20 2-2946
 - suboref 1-54 2-2947
 - suboref (M-file function equivalent for $A(i, j, k, \dots)$) 2-21
 - substruct 2-2949
 - subtraction (arithmetic operator) 2-2
 - subvolume 2-2951
 - sum 2-2954
 - cumulative 2-653
 - of array elements 2-2954
 - sum (timeseries) 2-2957
 - superiorto 2-2959
 - superscripts
 - in axis title 2-3138
 - in text strings 2-3094
 - support 2-2960
 - surf2patch 2-2966
 - surface 2-2968
 - Surface
 - and contour plotter 2-981
 - converting to a patch 1-99 2-2966
 - creating 1-90 1-94 2-2968
 - defining default properties 2-2505 2-2972
 - plotting mathematical functions 2-977
 - properties 2-2973 2-2993
 - surface normals, computing for
 - volumes 2-1689
 - surf1 2-3016
 - surfnorm 2-3020
 - svd 2-3022
 - svds 2-3025
 - swapbytes 2-3028
 - switch 2-3030
 - symamd 2-3032
 - symbfact 2-3036
 - symbols
 - operators 2-1408
 - symbols in text 2-3091
 - symmlq 2-3038
 - symmmd 2-3043
 - symrcm 2-3044
 - synchronize 2-3047
 - syntax 2-1409
 - syntax, command 2-3049
 - syntax, function 2-3049
 - syntaxes
 - of M-file functions, defining 2-1214
 - system 2-3052
 - UNC pathname error 2-3052
 - system directory, temporary 2-3060
- ## T
- table lookup, *see* interpolation
 - Tag
 - areaseries property 2-155
 - Axes property 2-238
 - barseries property 2-281
 - contour property 2-586
 - errorbar property 2-916
 - Figure property 2-1062
 - hggroup property 2-1435
 - hgtransform property 2-1456
 - Image property 2-1516
 - Light property 2-1799
 - Line property 2-1818
 - lineseries property 2-1829
 - Patch property 2-2247
 - quivergroup property 2-2461
 - rectangle property 2-2514
 - Root property 2-2599
 - scatter property 2-2657
 - stairs series property 2-2823

- stem property 2-2855
- Surface property 2-2990
- surfaceplot property 2-3011
- Text property 2-3095
- Uicontextmenu property 2-3207
- Uicontrol property 2-3235
- Uimenu property 2-3266
- Uipushtool property 2-3298
- Uitoggletool property 2-3324
- Uitoolbar property 2-3334
- Tagged Image File Format (TIFF)
 - reading 2-1537
 - writing 2-1547
- tan 2-3054
- tand 2-3056
- tangent 2-3054
 - four-quadrant, inverse 2-181
 - hyperbolic 2-3057
 - inverse 2-179
 - inverse hyperbolic 2-184
- tanh 2-3057
- tar 2-3059
- target, of camera 2-389
- tcpip 2-3363
- tempdir 2-3060
- tempname 2-3061
- temporary
 - files 2-3061
 - system directory 2-3060
- tensor, Kronecker product 2-1740
- terminating MATLAB 2-2441
- test matrices 2-1239
- test, logical, *see* logical tests *and* detecting
- tetrahedron
 - mesh plot 2-3062
- tetramesh 2-3062
- TeX commands in text 2-3091
- text 2-3067
 - editing 2-2320
 - subscripts 2-3094
 - superscripts 2-3094
- Text
 - creating 1-90 2-3067
 - defining default properties 2-3071
 - fixed-width font 2-3080
 - properties 2-3072
 - text mode for opened files 2-1144
 - TextBackgroundColor
 - annotation textbarrow property 2-125
 - TextColor
 - annotation textbarrow property 2-126
 - TextEdgeColor
 - annotation textbarrow property 2-126
 - TextLineWidth
 - annotation textarrow property 2-126
 - TextList
 - contour property 2-586
 - TextListMode
 - contour property 2-587
 - TextMargin
 - annotation textbarrow property 2-126
 - textread 1-75 2-3098
 - TextRotation, textarrow property 2-126
 - textscan 1-75 2-3104
 - TextStep
 - contour property 2-587
 - TextStepMode
 - contour property 2-588
 - textwrap 2-3122
 - TickDir, Axes property 2-239
 - TickDirMode, Axes property 2-239
 - TickLength, Axes property 2-239
 - TIFF
 - compression 2-1555
 - encoding 2-1550
 - ImageDescription field 2-1555
 - maxvalue 2-1550
 - parameters that can be set when
 - writing 2-1555
 - reading 2-1537

- resolution 2-1556
- writemode 2-1556
- writing 2-1547
- TIFF image format
 - specifying compression 2-1555
- tiling (copies of a matrix) 2-2558
- time
 - CPU 2-634
 - elapsed (stopwatch timer) 2-3123
 - required to execute commands 2-923
- time and date functions 2-891
- timer
 - properties 2-3124
 - timer object 2-3124
- timerfind
 - timer object 2-3130
- timerfindall
 - timer object 2-3132
- times (M-file function equivalent for .*) 2-7
- timeseries 2-3134
- timestamp 2-819
- title 2-3137
 - with superscript 2-3138
- Title, Axes property 2-240
- todatenum 2-3139
- toeplitz 2-3140
- Toeplitz matrix 2-3140
- toggle buttons 2-3212
- token 2-2919
 - See also* string
- Toolbar
 - Figure property 2-1062
- Toolbox
 - Spline 2-1605
- toolbox directory, pathname 2-3141
- toolboxdir 2-3141
- TooltipString
 - Uicontrol property 2-3235
 - Uipushtool property 2-3299
 - Uitogletool property 2-3324
- trace 2-3142
- trace of a matrix 2-811 2-3142
- trailing blanks
 - removing 2-754
- transform
 - hgtransform function 2-1442
- transform, Fourier
 - discrete, n-dimensional 2-1008
 - discrete, one-dimensional 2-1002
 - discrete, two-dimensional 2-1007
 - inverse, n-dimensional 2-1492
 - inverse, one-dimensional 2-1488
 - inverse, two-dimensional 2-1490
 - shifting the zero-frequency component of 2-1011
- transformation
 - See also* conversion 2-412
- transformations
 - elementary Hermite 2-1267
- transmitting file to FTP server 1-81 2-2056
- transparency chunk
 - in PNG files 2-1539
- transpose
 - array (arithmetic operator) 2-4
 - matrix (arithmetic operator) 2-4
- transpose (M-file function equivalent for .\q) 2-7
- transpose (timeseries) 2-3143
- trapz 2-3145
- treelayout 2-3147
- treepplot 2-3148
- triangulation
 - 2-D plot 2-3154
- tricubic interpolation 2-1346
- tril 2-3150
- trilinear interpolation 2-1346
- trimesh 2-3151
- triple integral
 - numerical evaluation 2-3152
- triplequad 2-3152

triplot 2-3154
 trisurf 2-3156
 triu 2-3157
 true 2-3158
 truth tables (for logical operations) 2-14
 try 2-3159
 tscollection 2-3160
 tsdata.event 2-3163
 tsearch 2-3164
 tsearchn 2-3165
 tsprops 2-3166
 tstool 2-3172
 type 2-3173
 Type

- areaserie property 2-156
- Axes property 2-240
- barseries property 2-281
- contour property 2-588
- errorbar property 2-917
- Figure property 2-1063
- hggroup property 2-1436
- hgtransform property 2-1456
- Image property 2-1517
- Light property 2-1799
- Line property 2-1818
- lineseries property 2-1829
- Patch property 2-2247
- quivergroup property 2-2462
- rectangle property 2-2514
- Root property 2-2599
- scatter property 2-2657
- stairs series property 2-2823
- stem property 2-2856
- Surface property 2-2990
- surfaceplot property 2-3011
- Text property 2-3095
- Uicontextmenu property 2-3208
- Uicontrol property 2-3235
- Uimenu property 2-3266
- Uipushtool property 2-3299

- Uitoggletool property 2-3324
- Uitoolbar property 2-3334
- typecast 2-3174

U

UData

- errorbar property 2-917
- quivergroup property 2-2463

UDataSource

- errorbar property 2-917
- quivergroup property 2-2463

Uibuttongroup

- defining default properties 2-3182

uibuttongroup function 2-3178

Uibuttongroup Properties 2-3182

uicontextmenu 2-3199

UiContextMenu

- Uicontrol property 2-3235

UiContextMenu

- areaserie property 2-156
- Axes property 2-240
- barseries property 2-281
- contour property 2-588
- errorbar property 2-917
- Figure property 2-1063
- hggroup property 2-1436
- hgtransform property 2-1456
- Image property 2-1517
- Light property 2-1800
- Line property 2-1818
- lineseries property 2-1829
- Patch property 2-2247
- quivergroup property 2-2462
- rectangle property 2-2514
- scatter property 2-2657
- stairs series property 2-2823
- stem property 2-2856
- Surface property 2-2991
- surfaceplot property 2-3012

- Text property 2-3096
- Uicontextmenu Properties 2-3201
- uicontrol 2-3209
- Uicontrol
 - defining default properties 2-3215
 - fixed-width font 2-3224
 - types of 2-3209
- Uicontrol Properties 2-3215
- uigetdir 2-3238
- uigetfile 2-3242
- uigetpref function 2-3249
- uiimport 2-3253
- uimenu 2-3254
- Uimenu
 - creating 1-103 2-3254
 - defining default properties 2-3256
 - Properties 2-3256
- Uimenu Properties 2-3256
- uint16 2-3267
- uint32 2-3267
- uint64 2-3267
- uint8 2-1595 2-3267
- uiopen 2-3269
- Uipanel
 - defining default properties 2-3273
- uipanel function 2-3271
- Uipanel Properties 2-3273
- uipushtool 2-3289
- Uipushtool
 - defining default properties 2-3291
- Uipushtool Properties 2-3291
- uiputfile 2-3301
- uiresume 2-3306
- uisave 2-3307
- uisetcolor function 2-3308
- uisetfont 2-3309
- uisetpref function 2-3311
- uistack 2-3312
- uitoggletool 2-3313
- Uitoggletool
 - defining default properties 2-3315
- Uitoggletool Properties 2-3315
- uitoolbar 2-3326
- Uitoolbar
 - defining default properties 2-3328
- Uitoolbar Properties 2-3328
- uiwait 2-3306
- uminus (M-file function equivalent for unary
\xd0) 2-7
- UNC pathname error and dos 2-848
- UNC pathname error and system 2-3052
- unconstrained minimization 2-1138
- undefined numerical results 2-2078
- undocheckout 2-3336
- unicode2native 2-3337
- unimodular matrix 2-1267
- union 2-3338
- unique 2-3340
- unitary matrix (complex) 2-2420
- Units
 - annotation ellipse property 2-114
 - annotation rectangle property 2-119
 - arrow property 2-106
 - Axes property 2-241
 - doublearrow property 2-111
 - Figure property 2-1063
 - line property 2-116
 - Root property 2-2600
 - Text property 2-3095
 - textarrow property 2-127
 - textbox property 2-134
 - Uicontrol property 2-3235
- unix 2-3343
- UNIX
 - Web browser 2-844
- unloadlibrary 2-3345
- unlocking M-files 2-2075
- unmkpp 2-3346
- unregisterallevents 2-3347
- unregisterevent 2-3349

untar 2-3352
 unwrap 2-3354
 unzip 2-3359
 up vector, of camera 2-391
 updating figure during M-file execution 2-853
 uplus (M-file function equivalent for unary
 +) 2-7
 upper 2-3361
 upper triangular matrix 2-3157
 uppercase to lowercase 2-1884
 url
 opening in Web browser 1-5 1-8 2-3430
 urlread 2-3362
 urlwrite 2-3364
 usejava 2-3366
 UserData
 areaseries property 2-156
 Axes property 2-241
 barseries property 2-282
 contour property 2-588
 errorbar property 2-918
 Figure property 2-1064
 hggroup property 2-1436
 hgtransform property 2-1457
 Image property 2-1517
 Light property 2-1800
 Line property 2-1819
 lineseries property 2-1830
 Patch property 2-2248
 quivergroup property 2-2462
 rectangle property 2-2515
 Root property 2-2600
 scatter property 2-2658
 stairs series property 2-2824
 stem property 2-2856
 Surface property 2-2991
 surfaceplot property 2-3012
 Text property 2-3096
 Uicontextmenu property 2-3208
 Uicontrol property 2-3236

Uimenu property 2-3266
 Uipushtool property 2-3299
 Uitoggletool property 2-3324
 Uitoolbar property 2-3335

V

Value, Uicontrol property 2-3236
 vander 2-3368
 Vandermonde matrix 2-2344
 var 2-3369
 var (timeseries) 2-3370
 varargin 2-3372
 varargout 2-3373
 variable numbers of M-file arguments 2-3373
 variable-order solver (ODE) 2-2162
 variables
 checking existence of 2-938
 clearing from workspace 2-480
 global 2-1328
 graphical representation of 2-3465
 in workspace 2-3465
 listing 2-3449
 local 2-1214 2-1328
 name of passed 2-1580
 opening 2-2168 2-2179
 persistent 2-2296
 saving 2-2623
 sizes of 2-3449
 VData
 quivergroup property 2-2463
 VDataSource
 quivergroup property 2-2464
 vector
 dot product 2-849
 frequency 2-1881
 length of 2-1773
 product (cross) 2-638
 vector field, plotting 2-552
 vectorize 2-3374

- vectorizing ODE function (BVP) 2-363
- vectors, creating
 - logarithmically spaced 2-1881
 - regularly spaced 2-23 2-1850
- velocity vectors, plotting 2-552
- ver 2-3375
- verctrl function (Windows) 2-3378
- version 2-3382
- version numbers
 - displaying 2-3375
- vertcat 2-3384
- vertcat (M-file function equivalent for [2-21
- vertcat (timeseries) 2-3386
- vertcat (tscollection) 2-3387
- VertexNormals
 - Patch property 2-2248
 - Surface property 2-2991
 - surfaceplot property 2-3012
- VerticalAlignment, Text property 2-3096
- VerticalAlignment, textarrow
 - property 2-127
- VerticalAlignment, textbox property 2-134
- Vertices, Patch property 2-2248
- video
 - saving in AVI format 2-199
- view 2-3388
 - azimuth of viewpoint 2-3389
 - coordinate system defining 2-3389
 - elevation of viewpoint 2-3389
- view angle, of camera 2-393
- View, Axes property (obsolete) 2-241
- viewing
 - a group of object 2-380
 - a specific object in a scene 2-380
- viewmtx 2-3391
- Visible
 - areaseries property 2-156
 - Axes property 2-242
 - barseries property 2-282
 - contour property 2-589
 - errorbar property 2-918
 - Figure property 2-1064
 - hggroup property 2-1436
 - hgtransform property 2-1457
 - Image property 2-1517
 - Light property 2-1800
 - Line property 2-1819
 - lineseries property 2-1830
 - Patch property 2-2248
 - quivergroup property 2-2462
 - rectangle property 2-2515
 - Root property 2-2600
 - scatter property 2-2658
 - stairs series property 2-2824
 - stem property 2-2856
 - Surface property 2-2991
 - surfaceplot property 2-3012
 - Text property 2-3097
 - Uicontextmenu property 2-3208
 - Uicontrol property 2-3237
 - Uimenu property 2-3266
 - Uipushtool property 2-3299
 - Uitoggletool property 2-3325
 - Uitoolbar property 2-3335
- visualizing
 - cell array structure 2-438
 - sparse matrices 2-2793
- volumes
 - calculating isosurface data 2-1692
 - computing 2-D stream lines 1-99 2-2875
 - computing 3-D stream lines 1-99 2-2877
 - computing isosurface normals 2-1689
 - contouring slice planes 2-593
 - drawing stream lines 1-99 2-2879
 - end caps 2-1682
 - reducing face size in isosurfaces 1-99 2-2711
 - reducing number of elements in 1-99 2-2523
- voronoi 2-3398

Voronoi diagrams
 multidimensional vizualization 2-3404
 two-dimensional vizualization 2-3398
 voronoin 2-3404

W

wait
 timer object 2-3408
 waitbar 2-3409
 waitfor 2-3411
 waitforbuttonpress 2-3412
 warndlg 2-3413
 warning 2-3415
 warning message (enabling, suppressing, and displaying) 2-3415
 waterfall 2-3419
 .wav files
 reading 2-3426
 writing 2-3429
 waverecord 2-3427
 wavinfo 2-3423
 wavplay 1-79 2-3424
 wavread 2-3423 2-3426
 wavrecord 1-79 2-3427
 wavwrite 2-3429
 WData
 quivergroup property 2-2464
 WDataSource
 quivergroup property 2-2465
 web 2-3430
 Web browser
 displaying help in 2-1413
 pointing to file or url 1-5 1-8 2-3430
 specifying for UNIX 2-844
 weekday 2-3434
 well conditioned 2-2489
 what 2-3436
 whatsnew 2-3439
 which 2-3440

while 2-3443
 white space characters, ASCII 2-1705 2-2919
 whitebg 2-3447
 who, whos
 who 2-3449
 wilkinson 2-3456
 Wilkinson matrix 2-2755 2-3456
 WindowButtonDownFcn, Figure
 property 2-1064
 WindowButtonMotionFcn, Figure
 property 2-1065
 WindowButtonUpFcn, Figure property 2-1065
 Windows Cursor Resources (CUR)
 reading 2-1535
 Windows Icon resources
 reading 2-1536
 Windows Paintbrush files
 reading 2-1536
 writing 2-1546
 WindowStyle, Figure property 2-1066
 winopen 2-3457
 winqueryreg 2-3458
 WK1 files
 loading 2-3461
 writing from matrix 2-3463
 wk1info 2-3460
 wk1read 2-3461
 wk1write 2-3463
 workspace 2-3465
 changing context while debugging 2-696
 2-718
 clearing items from 2-480
 consolidating memory 2-2201
 predefining variables 2-2830
 saving 2-2623
 variables in 2-3449
 viewing contents of 2-3465
 workspace variables
 reading from disk 2-1853
 writing

- binary data to file 2-1228
- formatted data to file 2-1165
- WVisual, Figure property 2-1067
- WVisualMode, Figure property 2-1070

X

X

- annotation arrow property 2-107 2-111
- annotation line property 2-116
- annotation textarrow property 2-127

X Windows Dump files

- reading 2-1537
- writing 2-1547

- x-axis limits, setting and querying 2-3469

- XAxisLocation, Axes property 2-242

- XColor, Axes property 2-242

XData

- areaseries property 2-157
- barseries property 2-282
- contour property 2-589
- errorbar property 2-918
- Image property 2-1517
- Line property 2-1819
- lineseries property 2-1830
- Patch property 2-2248
- quivergroup property 2-2465
- scatter property 2-2658
- stairs series property 2-2824
- stem property 2-2857
- Surface property 2-2991
- surfaceplot property 2-3012

XDataMode

- areaseries property 2-157
- barseries property 2-282
- contour property 2-589
- errorbar property 2-918
- lineseries property 2-1830
- quivergroup property 2-2466
- stairs series property 2-2824

- stem property 2-2857
- surfaceplot property 2-3013

XDataSource

- area property 2-2658
- areaseries property 2-157
- barseries property 2-283
- contour property 2-589
- errorbar property 2-919
- lineseries property 2-1830
- quivergroup property 2-2466
- stairs series property 2-2825
- stem property 2-2857
- surfaceplot property 2-3013

- XDir, Axes property 2-243

- XDisplay, Figure property 2-1070

- XGrid, Axes property 2-243

- xlabel 1-83 2-3467

- XLabel, Axes property 2-244

- xlim 2-3469

- XLim, Axes property 2-245

- XLimMode, Axes property 2-245

XLS files

- loading 2-3474

- xlsfinfo 2-3472

- xlsread 2-3474

- xlswrite 2-3484

- XMinorGrid, Axes property 2-245 to 2-246

- xmlread 2-3488

- xmlwrite 2-3493

- xor 2-3494

- XOR, printing 2-151 2-276 2-579 2-909 2-1452
2-1514 2-1813 2-1824 2-2236 2-2455 2-2510
2-2651 2-2816 2-2849 2-2982 2-3003 2-3079

- XScale, Axes property 2-246

- xslt 2-3495

- XTick, Axes property 2-246

- XTickLabel, Axes property 2-246

- XTickLabelMode, Axes property 2-248

- XTickMode, Axes property 2-247

- XVisual, Figure property 2-1070

XVisualMode, Figure property 2-1072

XWD files

reading 2-1537

writing 2-1547

xyz coordinates, *see* Cartesian coordinates

Y

Y

annotation arrow property 2-107 2-111
2-117

annotation textarrow property 2-128

y-axis limits, setting and querying 2-3469

YAxisLocation, Axes property 2-242

YColor, Axes property 2-242

YData

areaseries property 2-158

barseries property 2-283

contour property 2-590

errorbar property 2-919

Image property 2-1518

Line property 2-1819

lineseries property 2-1831

Patch property 2-2249

quivergroup property 2-2467

scatter property 2-2659

stairs series property 2-2825

stem property 2-2858

Surface property 2-2992

surfaceplot property 2-3014

YDataMode

contour property 2-591

quivergroup property 2-2467

surfaceplot property 2-3014

YDataSource

areaseries property 2-158

barseries property 2-284

contour property 2-591

errorbar property 2-920

lineseries property 2-1831

quivergroup property 2-2467

scatter property 2-2659

stairs series property 2-2826

stem property 2-2858

surfaceplot property 2-3014

YDir, Axes property 2-243

YGrid, Axes property 2-243

ylabel 1-83 2-3467

YLabel, Axes property 2-244

ylim 2-3469

YLim, Axes property 2-245

YLimMode, Axes property 2-245

YMinorGrid, Axes property 2-245 to 2-246

YScale, Axes property 2-246

YTick, Axes property 2-246

YTickLabel, Axes property 2-246

YTickLabelMode, Axes property 2-248

YTickMode, Axes property 2-247

Z

z-axis limits, setting and querying 2-3469

ZColor, Axes property 2-242

ZData

contour property 2-591

Line property 2-1819

lineseries property 2-1832 2-2660

Patch property 2-2249

quivergroup property 2-2468

stemseries property 2-2859

Surface property 2-2992

surfaceplot property 2-3015

ZDataSource

contour property 2-592

lineseries property 2-1832 2-2660 2-2859

surfaceplot property 2-3015

ZDir, Axes property 2-243

zero of a function, finding 2-1234

zeros 2-3497

ZGrid, Axes property 2-243

zip 2-3499
zlabel 1-83 2-3467
zlim 2-3469
ZLim, Axes property 2-245
ZLimMode, Axes property 2-245
ZMinorGrid, Axes property 2-245 to 2-246
zoom 2-3501
zoom mode objects 2-3502
ZScale, Axes property 2-246
ZTick, Axes property 2-246
ZTickLabel, Axes property 2-246
ZTickLabelMode, Axes property 2-248
ZTickMode, Axes property 2-247