MATLAB® 7
Function Reference: Volume 3 (P-Z)

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support
suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB Function Reference

Revision History

December 1996	First printing	For MATLAB 5.0 (Release 8)
June 1997	Online only	Revised for MATLAB 5.1 (Release 9)
October 1997	Online only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online only	Revised for MATLAB 5.3 (Release 11)
June 1999	Second printing	For MATLAB 5.3 (Release 11)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for 6.5 (Release 13)
June 2004	Online only	Revised for 7.0 (Release 14)
September 2006	Online only	Revised for 7.3 (Release 2006b)
March 2007	Online only	Revised for 7.4 (Release 2007a)
September 2007	Online only	Revised for Version 7.5 (Release 2007b)
March 2008	Online only	Revised for Version 7.6 (Release 2008a)
October 2008	Online only	Revised for Version 7.7 (Release 2008b)
March 2009	Online only	Revised for Version 7.8 (Release 2009a)
September 2009	Online only	Revised for Version 7.9 (Release 2009b)

Contents

1 Function Reference

Function Reference

Desktop Tools and Development Environment

Startup and Shutdown

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

startup	Startup file for user-defined options
userpath	View or change user portion of search path

Command Window and History

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

Help for Using MATLAB

builddocsearchdb	Build searchable documentation database
demo	Access product demos via Help browser
doc	Reference page in Help browser
docsearch	Help browser search
echodemo	Run M-file demo step-by-step in Command Window
help	Help for functions in Command Window
helpbrowser	Open Help browser to access 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 or Help browser
whatsnew	Release Notes for MathWorks™ products

Workspace, Search Path, and File Operations

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 class
openvar	Open workspace variable in Variable Editor or other graphical editing tool
pack	Consolidate workspace memory
uiimport	Open Import Wizard to import data
which	Locate functions and files
who, whos	List variables in workspace
workspace	Open Workspace browser to manage workspace

Search Path

addpath	Add folders to search path
genpath	Generate path string
path	View or change search path

path2rc	Save current search path to `pathdef.m` file
pathsep	Search path separator for current platform
pathtool	Open Set Path dialog box to view and change search path
restoredefaultpath	Restore default search path
rmpath	Remove folders from search path
savepath	Save current search path
userpath	View or change user portion of search path

File Operations

See also "Data Import and Export" on page 1-13 functions.

cd	Change current folder
copyfile	Copy file or folder
delete	Remove files or graphics objects
dir	Folder listing
exist	Check existence of variable, function, directory, or class
fileattrib	Set or get attributes of file or folder
filebrowser	Open Current Folder browser, or select it if already open
isdir	Determine whether input is folder
lookfor	Search for keyword in all help entries
ls	Folder contents
matlabroot	Root folder
mkdir	Make new folder

movefile	Move file or folder
pwd	Identify current folder
recycle	Set option to move deleted files to recycle folder
rehash	Refresh function and file system path caches
rmdir	Remove folder
tempdir	Name of system's temporary folder
toolboxdir	Root folder for specified toolbox
type	Display contents of file
visdiff	Compare two text files, MAT-Files, or binary files
what	List MATLAB files in folder
which	Locate functions and files

Programming Tools

M-File Editing and Debugging

clipboard	Copy and paste strings to and from system clipboard
datatipinfo	Produce short description of input variable

dbclear	Clear breakpoints
dbcont	Resume execution
dbdown	Reverse workspace shift performed by `dbup`, while in debug mode
dbquit	Quit debug mode
dbstack	Function call stack
dbstatus	List all breakpoints
dbstep	Execute one or more lines from current breakpoint
dbstop	Set breakpoints
dbtype	List M-file with line numbers
dbup	Shift current workspace to workspace of caller, while in debug mode
edit	Edit or create M-file
keyboard	Input from keyboard

M-File Performance

bench	MATLAB benchmark
mlint	Check M-files for possible problems
mlintrpt	Run `mlint` for file or folder, reporting results in 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

Source Control

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

Publishing

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

System

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

MATLAB Version and License

ismac	Determine if version is for Mac OS® X platform
ispc	Determine if version is for Windows (PC) platform
isstudent	Determine if version is Student Version
isunix	Determine if version is for UNIX platform
javachk	Generate error message based on Sun™ Java™ feature support

license	Return license number or perform licensing task
prefdir	Folder containing preferences, history, and layout files
usejava	Determine whether Sun Java feature is supported in MATLAB software
ver	Version information for MathWorks products
verLessThan	Compare toolbox version to specified version string
version	Version number for MATLAB and libraries

Data Import and Export

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	File separator for current platform
fullfile	Build full file name from parts

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

File Opening, Loading, and Saving

daqread	Read Data Acquisition Toolbox™ (.daq) file
importdata	Load data from file
load	Load workspace variables from disk
open	Open file in appropriate application
save	Save workspace variables to disk
uiimport	Open Import Wizard to import data
winopen	Open file in appropriate application (Windows)

Memory Mapping

disp (memmapfile)	Information about memmapfile object
get (memmapfile)	Memmapfile object properties
memmapfile	Construct memmapfile object

Low-Level File I/O

fclose	Close one or all open files
feof	Test for end-of-file
ferror	Information about file I/O errors
fgetl	Read line from file, removing newline characters

fgets	Read line from file, keeping newline characters
fopen	Open file, or obtain information about open files
fprintf	Write data to text file
fread	Read data from binary file
frewind	Move file position indicator to beginning of open file
fscanf	Read data from a text file
fseek	Move to specified position in file
ftell	Position in open file
fwrite	Write data to binary file

Text Files

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

XML Documents

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

Spreadsheets

Microsoft Excel (p. 1-16)	Read and write Microsoft Excel spreadsheet
Lotus 1-2-3 (p. 1-16)	Read and write Lotus WK1 spreadsheet

Microsoft Excel

xlsfinfo	Determine whether file contains a Microsoft® Excel® spreadsheet
xlsread	Read Microsoft Excel spreadsheet file
xlswrite	Write Microsoft Excel spreadsheet file

Lotus 1-2-3

wk1finfo	Determine whether file contains 1-2-3 WK1 worksheet
wk1read	Read Lotus 1-2-3 WK1 spreadsheet file into matrix
wk1write	Write matrix to Lotus 1-2-3 WK1 spreadsheet file

Scientific Data

Common Data Format

cdfepoch	Convert MATLAB formatted dates to CDF formatted dates
cdfinfo	Information about Common Data Format (CDF) file
cdfread	Read data from Common Data Format (CDF) file
cdfwrite	Write data to Common Data Format (CDF) file
todatenum	Convert CDF epoch object to MATLAB datenum

Network Common Data Form

File Operations

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

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

Dimensions

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

Variables

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

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

Attributes

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

Flexible Image Transport System

fitsinfo	Information about FITS file
fitsread	Read data from FITS file

Hierarchical Data Format

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

hdfinfo	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 Video

Reading and Writing Files (p. 1-20)	Input/output data to audio and video file formats
Recording and Playback (p. 1-21)	Record and listen to audio
Utilities (p. 1-22)	Convert audio signal

Reading and Writing Files

addframe (avifile)	Add frame to Audio/Video Interleaved (AVI) file
aufinfo	Information about NeXT/SUN (`.au`) sound file
auread	Read NeXT/SUN (`.au`) sound file
auwrite	Write NeXT/SUN (`.au`) sound 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 (avifile)	Close Audio/Video Interleaved (AVI) file
mmfileinfo	Information about multimedia file
mmreader	Create multimedia reader object for reading video files
mmreader.isPlatformSupported	Determine whether `mmreader` is available on current platform
movie2avi	Create Audio/Video Interleaved (AVI) movie from MATLAB movie
read (mmreader)	Read video frame data from multimedia reader object
wavfinfo	Information about WAVE (`.wav`) sound file
wavread	Read WAVE (`.wav`) sound file
wavwrite	Write WAVE (`.wav`) sound file

Recording and Playback

audiodevinfo	Information about audio device
audioplayer	Create `audioplayer` object
audiorecorder	Create `audiorecorder` object
sound	Convert vector into sound
soundsc	Scale data and play as sound
wavplay	Play recorded sound on PC-based audio output device
wavrecord	Record sound using PC-based audio input device

Utilities

beep	Produce beep sound
lin2mu	Convert linear audio signal to mu-law
mu2lin	Convert mu-law audio signal to linear

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
Tiff	MATLAB Gateway to LibTIFF library routines

Internet Exchange

URL, Zip, Tar, E-Mail (p. 1-22)	Send e-mail, read from given URL, extract from tar or zip file, compress and decompress files
FTP (p. 1-23)	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	Download content at URL into MATLAB string
urlwrite	Download content at URL and save to file
zip	Compress files into zip file

FTP

ascii	Set FTP transfer type to ASCII
binary	Set FTP transfer type to binary
cd (ftp)	Change current directory on FTP server
close (ftp)	Close connection to FTP server
delete (ftp)	Remove file on FTP server
dir (ftp)	Directory contents on FTP server
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

Mathematics

Arrays and Matrices

Basic Information

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

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 or largest array dimension
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)
.^	Array power (element-wise)
.\	Left array divide (element-wise)
./	Right array divide (element-wise)

Elementary Matrices and Arrays

blkdiag	Construct block diagonal matrix from input arguments
diag	Diagonal matrices and diagonals of matrix
eye	Identity matrix
freqspace	Frequency spacing for frequency response
ind2sub	Subscripts from linear index
linspace	Generate linearly spaced vectors
logspace	Generate logarithmically spaced vectors
meshgrid	Generate X and Y arrays for 3-D plots
ndgrid	Generate arrays for N-D functions and interpolation
ones	Create array of all ones
rand	Uniformly distributed pseudorandom numbers
randi	Uniformly distributed pseudorandom integers
randn	Normally distributed pseudorandom numbers
RandStream	Random number stream

sub2ind	Single index from subscripts
zeros	Create array of all zeros

Array Operations

See "Linear Algebra" on page 1-30 and "Elementary Math" on page 1-34 for other array operations.

accumarray	Construct array with accumulation
arrayfun	Apply function to each element of array
bsxfun	Apply element-by-element binary operation to two arrays with singleton expansion enabled
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
inline	Construct inline object
ipermute	Inverse permute dimensions of N-D array
permute	Rearrange dimensions of N-D array
repmat	Replicate and tile array
reshape	Reshape array
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
vectorize	Vectorize expression
vertcat	Concatenate arrays vertically

Specialized Matrices

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

Linear Algebra

Matrix Analysis (p. 1-31)	Compute norm, rank, determinant, condition number, etc.
Linear Equations (p. 1-31)	Solve linear systems, least squares, LU factorization, Cholesky factorization, etc.
Eigenvalues and Singular Values (p. 1-32)	Eigenvalues, eigenvectors, Schur decomposition, Hessenburg matrices, etc.
Matrix Logarithms and Exponentials (p. 1-33)	Matrix logarithms, exponentials, square root
Factorization (p. 1-33)	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
ilu	Sparse incomplete LU factorization
inv	Matrix inverse

ldl	Block LDL' factorization for Hermitian indefinite matrices
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	Eigenvalues and eigenvectors
eigs	Largest eigenvalues and eigenvectors of 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
ss2tf	Convert state-space filter parameters to transfer function form
svd	Singular value decomposition
svds	Find singular values and vectors

Matrix Logarithms and Exponentials

expm	Matrix exponential
logm	Matrix logarithm
sqrtm	Matrix square root

Factorization

balance	Diagonal scaling to improve eigenvalue accuracy
cdf2rdf	Convert complex diagonal form to real block diagonal form
chol	Cholesky factorization
cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations

cholupdate	Rank 1 update to Cholesky factorization
gsvd	Generalized singular value decomposition
ilu	Sparse incomplete LU factorization
ldl	Block LDL' factorization for Hermitian indefinite matrices
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

Rounding and Remainder (p. 1-38)	Rounding, modulus, and remainder
Discrete Math (p. 1-38)	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 nth 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	Check if input is real array
j	Imaginary unit
real	Real part of complex number

sign	Signum function
unwrap	Correct phase angles to produce smoother phase plots

Rounding and Remainder

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

Discrete Math

factor	Prime factors
factorial	Factorial function
gcd	Greatest common divisor
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

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
interpn	N-D data interpolation (table lookup)
meshgrid	Generate X and Y arrays for 3-D plots
mkpp	Make piecewise polynomial
ndgrid	Generate arrays for N-D functions and interpolation
padecoef	Padé approximation of time delays
pchip	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)
ppval	Evaluate piecewise polynomial
spline	Cubic spline data interpolation
tsearch	Search for enclosing Delaunay triangle
tsearchn	N-D closest simplex search
unmkpp	Piecewise polynomial details

Delaunay Triangulation and Tessellation

baryToCart (TriRep)	Converts point coordinates from barycentric to Cartesian
cartToBary (TriRep)	Convert point coordinates from cartesian to barycentric
circumcenters (TriRep)	Circumcenters of specified simplices
delaunay	Delaunay triangulation
delaunay3	3-D Delaunay tessellation
delaunayn	N-D Delaunay tessellation
DelaunayTri	Contruct Delaunay triangulation
DelaunayTri	Delaunay triangulation in 2-D and 3-D
edgeAttachments (TriRep)	Simplices attached to specified edges
edges (TriRep)	Triangulation edges
faceNormals (TriRep)	Unit normals to specified triangles
featureEdges (TriRep)	Sharp edges of surface triangulation
freeBoundary (TriRep)	Facets referenced by only one simplex
incenters (TriRep)	Incenters of specified simplices
inOutStatus (DelaunayTri)	Status of triangles in 2-D constrained Delaunay triangulation
isEdge (TriRep)	Test if vertices are joined by edge
nearestNeighbor (DelaunayTri)	Point closest to specified location
neighbors (TriRep)	Simplex neighbor information
pointLocation (DelaunayTri)	Simplex containing specified location
size (TriRep)	Size of triangulation matrix
tetramesh	Tetrahedron mesh plot
trimesh	Triangular mesh plot
triplot	2-D triangular plot

TriRep	Triangulation representation
TriRep	Triangulation representation
TriScatteredInterp	Interpolate scattered data
TriScatteredInterp	Interpolate scattered data
trisurf	Triangular surface plot
vertexAttachments (TriRep)	Return simplices attached to specified vertices

Convex Hull

convexHull (DelaunayTri)	Convex hull
convhull	Convex hull
convhulln	N-D convex hull
patch	Create one or more filled polygons
plot	2-D line plot
trisurf	Triangular surface plot

Voronoi Diagrams

patch	Create one or more filled polygons
plot	2-D line plot
voronoi	Voronoi diagram
voronoiDiagram (DelaunayTri)	Voronoi diagram
voronoin	N-D Voronoi diagram

Domain Generation

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

Cartesian Coordinate System Conversion

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

Nonlinear Numerical Methods

Ordinary Differential Equations (p. 1-44)	Solve stiff and nonstiff differential equations, define the problem, set solver options, evaluate solution
Delay Differential Equations (p. 1-45)	Solve delay differential equations with constant and general delays, set solver options, evaluate solution
Boundary Value Problems (p. 1-45)	Solve boundary value problems for ordinary differential equations, set solver options, evaluate solution
Partial Differential Equations (p. 1-46)	Solve initial-boundary value problems for parabolic-elliptic PDEs, evaluate solution

Optimization (p. 1-46)	Find minimum of single and multivariable functions, solve nonnegative least-squares constraint problem
Numerical Integration (Quadrature) (p. 1-46)	Evaluate Simpson, Lobatto, and vectorized quadratures, evaluate double and triple integrals

Ordinary Differential Equations

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

Delay Differential Equations

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

Boundary Value Problems

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

Partial Differential Equations

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

Optimization

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

Numerical Integration (Quadrature)

dblquad	Numerically evaluate double integral over a rectangle
quad	Numerically evaluate integral, adaptive Simpson quadrature
quad2d	Numerically evaluate double integral over planar region
quadgk	Numerically evaluate integral, adaptive Gauss-Kronrod quadrature

quadl	Numerically evaluate integral, adaptive Lobatto quadrature
quadv	Vectorized quadrature
triplequad	Numerically evaluate triple integral

Specialized Math

airy	Airy functions
besselh	Bessel function of third kind (Hankel function)
besseli	Modified Bessel function of first kind
besselj	Bessel function of first kind
besselk	Modified Bessel function of second kind
bessely	Bessel function of second kind
beta	Beta function
betainc	Incomplete beta function
betaincinv	Beta inverse cumulative distribution 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
gammaincinv	Inverse incomplete gamma function
legendre	Associated Legendre functions
psi	Psi (polygamma) function

Sparse Matrices

Elementary Sparse Matrices (p. 1-48)	Create random and nonrandom sparse matrices
Full to Sparse Conversion (p. 1-49)	Convert full matrix to sparse, sparse matrix to full
Sparse Matrix Manipulation (p. 1-49)	Test matrix for sparseness, get information on sparse matrix, allocate sparse matrix, apply function to nonzero elements, visualize sparsity pattern
Reordering Algorithms (p. 1-49)	Random, column, minimum degree, Dulmage-Mendelsohn, and reverse Cuthill-McKee permutations
Linear Algebra (p. 1-50)	Compute norms, eigenvalues, factorizations, least squares, structural rank
Linear Equations (Iterative Methods) (p. 1-50)	Methods for conjugate and biconjugate gradients, residuals, lower quartile
Tree Operations (p. 1-51)	Elimination trees, tree plotting, factorization analysis

Elementary Sparse Matrices

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

Full to Sparse Conversion

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

Sparse Matrix Manipulation

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

Reordering Algorithms

amd	Approximate minimum degree permutation
colamd	Column approximate minimum degree permutation

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

Linear Algebra

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

Linear Equations (Iterative Methods)

bicg	Biconjugate gradients method
bicgstab	Biconjugate gradients stabilized method

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

Tree Operations

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

Math Constants

eps	Floating-point relative accuracy
i	Imaginary unit

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

Data Analysis

Basic Operations

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

Descriptive Statistics

corrcoef	Correlation coefficients
cov	Covariance matrix

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

Filtering and Convolution

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

Interpolation and Regression

interp1	1-D data interpolation (table lookup)
interp2	2-D data interpolation (table lookup)
interp3	3-D data interpolation (table lookup)
interpn	N-D data interpolation (table lookup)
mldivide \, mrdivide /	Left or right matrix division
polyfit	Polynomial curve fitting
polyval	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
ifft2	2-D 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

Utilities

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

timeseries	Create `timeseries` object
tsdata.event	Construct event object for `timeseries` object
tsprops	Help on `timeseries` object properties
tstool	Open Time Series Tools GUI

Data Manipulation

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

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

Event Data

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

Descriptive Statistics

iqr (timeseries)	Interquartile range of `timeseries` data
max (timeseries)	Maximum value of `timeseries` data
mean (timeseries)	Mean value of `timeseries` data
median (timeseries)	Median value of `timeseries` data

min (timeseries)	Minimum value of `timeseries` data
std (timeseries)	Standard deviation of `timeseries` data
sum (timeseries)	Sum of `timeseries` data
var (timeseries)	Variance of `timeseries` data

Time Series Collections

Utilities

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

Data Manipulation

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

Programming and Data Types

Data Types

Numeric Types

arrayfun	Apply function to each element of array
cast	Cast variable to different data type
cat	Concatenate arrays along specified dimension
class	Determine class name 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

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

Characters and Strings

See “Strings” on page 1-74 for all string-related functions.

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

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

Structures

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

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

Cell Arrays

cell	Construct cell array
cell2mat	Convert cell array of matrices to single matrix
cell2struct	Convert cell array to structure array
celldisp	Cell array contents
cellfun	Apply function to each cell in cell array
cellplot	Graphically display structure of cell array
cellstr	Create cell array of strings from character array
class	Determine class name 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	Determine class name of object
feval	Evaluate function
func2str	Construct function name string from function handle
functions	Information about function handle
function_handle (@)	Handle used in calling functions indirectly
isa	Determine whether input is object of given class
isequal	Test arrays for equality
str2func	Construct function handle from function name string

Java Classes and Objects

cell	Construct cell array
class	Determine class name 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 class
fieldnames	Field names of structure, or public fields of object
im2java	Convert image to Java image
import	Add package or class to current import list
inmem	Names of M-files, MEX-files, Sun Java classes in memory
isa	Determine whether input is object of given class
isjava	Determine whether input is Sun Java object
javaaddpath	Add entries to dynamic Sun Java class path
javaArray	Construct Sun Java array
javachk	Generate error message based on Sun Java feature support
javaclasspath	Get and set Sun Java class path
javaMethod	Call Sun Java method
javaMethodEDT	Call Sun Java method from Event Dispatch Thread (EDT)
javaObject	Construct Sun Java object
javaObjectEDT	Construct Sun Java object on Event Dispatch Thread (EDT)
javarmpath	Remove entries from dynamic Sun Java class path
methods	Class method names
methodsview	View class methods

usejava	Determine whether Sun Java feature is supported in MATLAB software
which	Locate functions and files

Data Type Identification

is*	Detect state
isa	Determine whether input is object of given class
iscell	Determine whether input is cell array
iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array
isfield	Determine whether input is structure array field
isfloat	Determine whether input is floating-point array
ishghandle	True for Handle Graphics® object handles
isinteger	Determine whether input is integer array
isjava	Determine whether input is Sun Java object
islogical	Determine whether input is logical array
isnumeric	Determine whether input is numeric array
isobject	Is input MATLAB object
isreal	Check if input is real array

isstr	Determine whether input is character array
isstruct	Determine whether input is structure array
validateattributes	Check validity of array
who, whos	List variables in workspace

Data Type Conversion

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
str2mat	Form blank-padded character matrix from strings
struct2cell	Convert structure to cell array

Operators and Special Characters

Arithmetic Operators (p. 1-72)	Plus, minus, power, left and right divide, transpose, etc.
Relational Operators (p. 1-72)	Equal to, greater than, less than or equal to, etc.
Logical Operators (p. 1-72)	Element-wise and short circuit and, or, not
Special Characters (p. 1-73)	Array constructors, line continuation, comments, etc.

Arithmetic Operators

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

Relational Operators

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

Logical Operators

See also “Logical Operations” on page 1-77 for functions like `xor`, `all`, `any`, etc.

&&	Logical AND
||	Logical OR
&	Logical AND for arrays
|	Logical OR for arrays
~	Logical NOT

Special Characters

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

Strings

Description of Strings in MATLAB

strings	String handling

String Creation

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

String Identification

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
validatestring	Check validity of text string

String Manipulation

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

String Parsing

findstr	Find string within another, longer string
regexp, regexpi	Match regular expression
regexprep	Replace string using regular expression
regexptranslate	Translate string into regular expression
sscanf	Read formatted data from string
strfind	Find one string within another
strread	Read formatted data from string
strtok	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 Operations

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 Operations

all	Determine whether all array elements are nonzero or `true`
and	Find logical AND of array or scalar inputs
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

not	Find logical NOT of array or scalar input
or	Find logical OR of array or scalar inputs
true	Logical 1 (true)
xor	Logical exclusive-OR

See "Operators and Special Characters" on page 1-71 for logical operators.

Relational Operations

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

See "Operators and Special Characters" on page 1-71 for relational operators.

Set Operations

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 Operations

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-Files and Scripts

addOptional (inputParser)	Add optional argument to `inputParser` schema
addParamValue (inputParser)	Add parameter-value argument to `inputParser` schema
addRequired (inputParser)	Add required argument to `inputParser` schema
createCopy (inputParser)	Create copy of `inputParser` object
depdir	List dependent directories of M-file or P-file
depfun	List dependencies of M-file or P-file
echo	Echo M-files during execution
end	Terminate block of code, or indicate last array index
function	Declare M-file function
input	Request user input
inputname	Variable name of function input
inputParser	Construct input parser object

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
parse (inputParser)	Parse and validate named inputs
pcode	Create protected M-file (P-file)
script	Script M-file description
syntax	Two ways to call MATLAB functions
varargin	Variable length input argument list
varargout	Variable length output argument list

Evaluation

ans	Most recent answer
arrayfun	Apply function to each element of array
assert	Generate error when condition is violated
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
tic, toc	Measure performance using stopwatch timer

Timer

delete (timer)	Remove timer object from memory
disp (timer)	Information about timer object
get (timer)	Timer object properties
isvalid (timer)	Determine whether timer object is valid
set (timer)	Configure or display timer 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

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

persistent	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	Handle error detected in try-catch 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
parfor	Parallel `for`-loop
return	Return to invoking function
switch	Switch among several cases, based on expression

try	Execute statements and catch resulting errors
while	Repeatedly execute statements while condition is true

Error Handling

addCause (MException)	Record additional causes of exception
assert	Generate error when condition is violated
catch	Handle error detected in try-catch statement
disp (MException)	Display `MException` object
eq (MException)	Compare `MException` objects for equality
error	Display message and abort function
ferror	Information about file I/O errors
getReport (MException)	Get error message for exception
intwarning	Control state of integer warnings
isequal (MException)	Compare `MException` objects for equality
last (MException)	Last uncaught exception
lastwarn	Last warning message
MException	Capture error information
ne (MException)	Compare `MException` objects for inequality
rethrow (MException)	Reissue existing exception
throw (MException)	Issue exception and terminate function

try	Execute statements and catch resulting errors
warning	Warning message

MEX Programming

dbmex	Enable MEX-file debugging (on UNIX platforms)
inmem	Names of M-files, MEX-files, Sun Java classes in memory
mex	Compile MEX-function from C/ C++ or Fortran source code
mex.getCompilerConfigurations	Get compiler configuration information for building MEX-files
mexext	Binary MEX-file name extension

Object-Oriented Programming

Classes and Objects

class	Determine class name of object
classdef	Class definition keywords
exist	Check existence of variable, function, directory, or class
inferiorto	Specify inferior class relationship
isobject	Is input MATLAB object
loadobj	Modify load process for object
methods	Class method names
methodsview	View class methods
properties	Class property names
subsasgn	Subscripted assignment
subsindex	Subscript indexing with object
subsref	Redefine subscripted reference for objects
superiorto	Establish superior class relationship

Handle Classes

Events and Listeners

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

Meta-Classes

meta.class	`meta.class` class describes MATLAB classes
meta.class.fromName	Return `meta.class` object associated with named class
meta.DynamicProperty	`meta.DynamicProperty` class describes dynamic property of MATLAB object
meta.event	`meta.event` class describes MATLAB class events
meta.method	`meta.method` class describes MATLAB class methods
meta.package	`meta.package` class describes MATLAB packages
meta.package.fromName	Return `meta.package` object for specified package
meta.package.getAllPackages	Get all top-level packages

meta.property	`meta.property` class describes MATLAB class properties
metaclass	Obtain `meta.class` object

Graphics

Basic Plots and Graphs

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

semilogx, semilogy	Semilogarithmic plots
subplot	Create axes in tiled positions

Plotting Tools

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

Annotating Plots

annotation	Create annotation objects
clabel	Contour plot elevation labels
datacursormode	Enable or disable interactive data cursor mode
datetick	Date formatted tick labels
gtext	Mouse placement of text in 2-D view
legend	Graph legend for lines and patches
rectangle	Create 2-D rectangle object
texlabel	Produce TeX format from character string

title	Add title to current axes
xlabel, ylabel, zlabel	Label *x*-, *y*-, and *z*-axis

Specialized Plotting

Area, Bar, and Pie Plots

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

Contour Plots

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

Direction and Velocity Plots

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

Discrete Data Plots

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

Function Plots

ezcontour	Easy-to-use contour plotter
ezcontourf	Easy-to-use filled contour plotter
ezmesh	Easy-to-use 3-D mesh plotter

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
ezsurfc	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

cylinder	Generate cylinder
delaunay	Delaunay triangulation
delaunay3	3-D Delaunay tessellation
delaunayn	N-D Delaunay tessellation
dsearch	Search Delaunay triangulation for nearest point
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
waterfall	Waterfall plot

Scatter/Bubble Plots

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

Animation

frame2im	Return image data associated with movie frame
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	Return image data associated with movie frame
im2frame	Convert image to movie frame

im2java	Convert image to Java image
image	Display image object
imagesc	Scale data and display image object
imfinfo	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

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

Handle Graphics

Axes Operations (p. 1-101)	Operate on axes objects
Object Property Operations (p. 1-101)	Query, set, and link object properties

Graphics Object Identification

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

Object Creation

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

Plot Objects

Annotation Arrow Properties	Define annotation arrow properties
Annotation Doublearrow Properties	Define annotation doublearrow properties
Annotation Ellipse Properties	Define annotation ellipse properties
Annotation Line Properties	Define annotation line properties
Annotation Rectangle Properties	Define annotation rectangle properties
Annotation Textarrow Properties	Define annotation textarrow properties
Annotation Textbox Properties	Define annotation textbox properties
Areaseries Properties	Define areaseries properties

Barseries Properties	Define barseries properties
Contourgroup Properties	Define contourgroup properties
Errorbarseries Properties	Define errorbarseries properties
Image Properties	Define image properties
Lineseries Properties	Define lineseries properties
Quivergroup Properties	Define quivergroup properties
Scattergroup Properties	Define scattergroup properties
Stairseries Properties	Define stairseries properties
Stemseries Properties	Define stemseries properties
Surfaceplot Properties	Define surfaceplot properties

Figure Windows

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

Object Property Operations

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

3-D Visualization

Surface and Mesh Plots

Surface and Mesh Creation

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

meshgrid	Generate X and Y arrays for 3-D plots

Color Operations

brighten	Brighten or darken colormap
caxis	Color axis scaling
colorbar	Colorbar showing color scale
colordef	Set default property values to display different color schemes
colormap	Set and get current colormap
colormapeditor	Start colormap editor
ColorSpec (Color Specification)	Color specification
contrast	Grayscale colormap for contrast enhancement
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

View Control

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

Aspect Ratio and Axis Limits

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

Object Manipulation

pan	Pan view of graph interactively
reset	Reset graphics object properties to their defaults
rotate	Rotate object in specified direction
rotate3d	Rotate 3-D view using mouse
selectmoveresize	Select, move, resize, or copy axes and uicontrol graphics objects
zoom	Turn zooming on or off or magnify by factor

Region of Interest

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

Lighting

camlight	Create or move light object in camera coordinates
diffuse	Calculate diffuse reflectance
light	Create light object
lightangle	Create or position `light` object in spherical coordinates
lighting	Specify lighting algorithm
material	Control reflectance properties of surfaces and patches
specular	Calculate specular reflectance

Transparency

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

Volume Visualization

coneplot	Plot velocity vectors as cones in 3-D vector field
contourslice	Draw contours in volume slice planes
curl	Compute curl and angular velocity of vector field
divergence	Compute divergence of vector field
flow	Simple function of three variables

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 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

GUI Development

Predefined Dialog Boxes

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

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

User Interface Deployment

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

User Interface Development

addpref	Add preference
getappdata	Value of application-defined data
getpref	Preference

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

User Interface Objects

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

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

Objects from Callbacks

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

GUI Utilities

align	Align user interface controls (`uicontrols`) and axes
getpixelposition	Get component position in pixels
listfonts	List available system fonts
selectmoveresize	Select, move, resize, or copy axes and uicontrol graphics objects
setpixelposition	Set component position in pixels

textwrap	Wrapped string matrix for given uicontrol
uistack	Reorder visual stacking order of objects

Program Execution

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

External Interfaces

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

Shared Libraries

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

loadlibrary	Load shared library into MATLAB software
unloadlibrary	Unload shared library from memory

Java

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

methodsview	View class methods
usejava	Determine whether Sun Java feature is supported in MATLAB software

.NET

enableNETfromNetworkDrive	Enable access to .NET commands from network drive
NET.addAssembly	Make .NET assembly visible to MATLAB
NET.Assembly	Members of .NET assembly
NET.convertArray	Convert numeric MATLAB array to .NET array
NET.createArray	Create single or multidimensional .NET array
NET.createGeneric	Create instance of specialized .NET generic type
NET.GenericClass	Represent parameterized generic type definitions
NET.GenericClass	Constructor for NET.GenericClass class
NET.invokeGenericMethod	Invoke generic method of object
NET.NetException	.NET exception
NET.setStaticProperty	Static property or field name

Component Object Model and ActiveX

actxcontrol	Create Microsoft® ActiveX® control in figure window
actxcontrollist	List currently installed Microsoft ActiveX controls

actxcontrolselect	Create Microsoft ActiveX control from GUI
actxGetRunningServer	Handle to running instance of Automation server
actxserver	Create COM server
addproperty	Add custom property to COM object
delete (COM)	Remove COM control or server
deleteproperty	Remove custom property from COM object
enableservice	Enable, disable, or report status of MATLAB Automation server
eventlisteners	List event handler functions associated with COM object events
events (COM)	List of events COM object can trigger
Execute	Execute MATLAB command in Automation server
Feval (COM)	Evaluate MATLAB function in Automation server
fieldnames	Field names of structure, or public fields of object
get (COM)	Get property value from interface, or display properties
GetCharArray	Character array from Automation server
GetFullMatrix	Matrix from Automation server workspace
GetVariable	Data from variable in Automation server workspace
GetWorkspaceData	Data from Automation server workspace
inspect	Open Property Inspector

interfaces	List custom interfaces exposed by COM server object
invoke	Invoke method on COM object or interface, or display methods
isa	Determine whether input is object of given class
iscom	Determine whether input is COM or ActiveX object
isevent	Determine whether input is COM object event
isinterface	Determine whether input is COM interface
ismethod	Determine whether input is COM object method
isprop	Determine whether input is COM object property
load (COM)	Initialize control object from file
MaximizeCommandWindow	Open Automation server window
methods	Class method names
methodsview	View class methods
MinimizeCommandWindow	Minimize size of Automation server window
move	Move or resize control in parent window
propedit (COM)	Open built-in property page for control
PutCharArray	Store character array in Automation server
PutFullMatrix	Matrix in Automation server workspace
PutWorkspaceData	Data in Automation server workspace

Quit (COM)	Terminate MATLAB Automation server
registerevent	Associate event handler for COM object event at run time
release	Release COM interface
save (COM)	Serialize control object to file
set (COM)	Set object or interface property to specified value
unregisterallevents	Unregister all event handlers associated with COM object events at run time
unregisterevent	Unregister event handler associated with COM object event at run time

Web Services

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

Serial Port Devices

clear (serial)	Remove serial port object from MATLAB workspace
delete (serial)	Remove serial port object from memory
fgetl (serial)	Read line of text from device and discard terminator

fgets (serial)	Read line of text from device and include terminator
fopen (serial)	Connect serial port object to device
fprintf (serial)	Write text to device
fread (serial)	Read binary data from device
fscanf (serial)	Read data from device, and format as text
fwrite (serial)	Write binary data to device
get (serial)	Serial port object properties
instrcallback	Event information when event occurs
instrfind	Read serial port objects from memory to MATLAB workspace
instrfindall	Find visible and hidden serial port objects
isvalid (serial)	Determine whether serial port objects are valid
length (serial)	Length of serial port object array
load (serial)	Load serial port objects and variables into MATLAB workspace
readasync	Read data asynchronously from device
record	Record data and event information to file
save (serial)	Save serial port objects and variables to MAT-file
serial	Create serial port object
serialbreak	Send break to device connected to serial port
set (serial)	Configure or display serial port object properties

size (serial)	Size of serial port object array
stopasync	Stop asynchronous read and write operations

2

Alphabetical List

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

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

campan
campos
camproj
camroll
camtarget
camup
camva
camzoom
cartToBary
cart2pol
cart2sph
case
cast
cat
catch
caxis
cd
convexHull
cd (ftp)
cdf2rdf
cdfepoch
cdfinfo
cdfread
cdfwrite
ceil
cell
cell2mat
cell2struct
celldisp
cellfun
cellplot
cellstr
cgs
char
checkin
checkout
chol

computeStrip
computeTile
computer
cond
condeig
condest
coneplot
conj
continue
contour
contour3
contourc
contourf
Contourgroup Properties
contourslice
contrast
conv
conv2
convhull
convhulln
convn
copyfile
copyobj
corrcoef
cos
cosd
cosh
cot
cotd
coth
cov
cplxpair
cputime
create (RandStream)
createClassFromWsdl
createCopy (inputParser)
createSoapMessage

ddesd
ddeset
deal
deblank
dec2base
dec2bin
dec2hex
decic
deconv
del2
DelaunayTri
DelaunayTri
delaunay
delaunay3
delaunayn
delete
delete (COM)
delete (ftp)
delete (handle)
delete (serial)
delete (timer)
deleteproperty
delevent
delsample
delsamplefromcollection
demo
depdir
depfun
det
detrend
detrend (timeseries)
deval
diag
dialog
diary
diff
diffuse

enableservice
end
eomday
eps
eq
eq (MException)
erf, erfc, erfcx, erfinv, erfcinv
error
errorbar
Errorbarseries Properties
errordlg
etime
etree
etreeplot
eval
evalc
evalin
event.EventData
event.PropertyEvent
event.listener
event.proplistener
eventlisteners
events
events (COM)
Execute
exifread
exist
exit
exp
expint
expm
expm1
export2wsdlg
eye
ezcontour
ezcontourf
ezmesh

filesep
fill
fill3
filter
filter (timeseries)
filter2
find
findall
findfigs
findobj
findobj (handle)
findprop (handle)
findstr
finish
fitsinfo
fitsread
fix
flipdim
fliplr
flipud
floor
flow
fminbnd
fminsearch
fopen
fopen (serial)
for
format
fplot
fprintf
fprintf (serial)
frame2im
fread
fread (serial)
freeBoundary
freqspace
frewind

getabstime (tscollection)
getappdata
GetCharArray
getdatasamplesize
getDefaultStream (RandStream)
getdisp (hgsetget)
getenv
getfield
getframe
GetFullMatrix
getinterpmethod
getpixelposition
getpref
getqualitydesc
getReport (MException)
getsampleusingtime (timeseries)
getsampleusingtime (tscollection)
getTag
getTagNames
gettimeseriesnames
gettsafteratevent
gettsafterevent
gettsatevent
gettsbeforeatevent
gettsbeforeevent
gettsbetweenevents
GetVariable
getVersion
GetWorkspaceData
ginput
global
gmres
gplot
grabcode
gradient
graymon
grid

griddata
griddata3
griddatan
gsvd
gt
gtext
guidata
guide
guihandles
gunzip
gzip
hadamard
handle
hankel
hdf
hdf5
hdf5info
hdf5read
hdf5write
hdfinfo
hdfread
hdftool
help
helpbrowser
helpdesk
helpdlg
helpwin
hess
hex2dec
hex2num
hgexport
hggroup
Hggroup Properties
hgload
hgsave
hgsetget
hgtransform

Hgtransform Properties
hidden
hilb
hist
histc
hold
home
horzcat
horzcat (tscollection)
hostid
hsv2rgb
hypot
i
idealfilter (timeseries)
idivide
if
ifft
ifft2
ifftn
ifftshift
ilu
im2frame
im2java
imag
image
Image Properties
imagesc
imapprox
imfinfo
imformats
import
importdata
imread
imwrite
incenters
inOutStatus
ind2rgb

ind2sub
Inf
inferiorto
info
inline
inmem
inpolygon
input
inputdlg
inputname
inputParser
inspect
instrcallback
instrfind
instrfindall
int2str
int8, int16, int32, int64
interfaces
interp1
interp1q
interp2
interp3
interpft
interpn
interpstreamspeed
intersect
intmax
intmin
intwarning
inv
invhilb
invoke
ipermute
iqr (timeseries)
is*
isa
isappdata

iscell
iscellstr
ischar
iscom
isdir
isEdge
isempty
isempty (timeseries)
isempty (tscollection)
isequal
isequal (MException)
isequalwithequalnans
isevent
isfield
isfinite
isfloat
isglobal
ishandle
ishghandle
ishold
isinf
isinteger
isinterface
isjava
isKey (Map)
iskeyword
isletter
islogical
ismac
ismember
ismethod
isnan
isnumeric
isobject
isocaps
isocolors
isonormals

NET.createGeneric
NET.GenericClass
NET.GenericClass
NET.invokeGenericMethod
NET.NetException
NET.setStaticProperty
netcdf
netcdf.abort
netcdf.close
netcdf.copyAtt
netcdf.create
netcdf.defDim
netcdf.defVar
netcdf.delAtt
netcdf.endDef
netcdf.getAtt
netcdf.getConstant
netcdf.getConstantNames
netcdf.getVar
netcdf.inq
netcdf.inqAtt
netcdf.inqAttID
netcdf.inqAttName
netcdf.inqDim
netcdf.inqDimID
netcdf.inqLibVers
netcdf.inqVar
netcdf.inqVarID
netcdf.open
netcdf.putAtt
netcdf.putVar
netcdf.reDef
netcdf.renameAtt
netcdf.renameDim
netcdf.renameVar
netcdf.setDefaultFormat
netcdf.setFill

ordeig
orderfields
ordqz
ordschur
orient
orth
otherwise
pack
padecoef
pagesetupdlg
pan
pareto
parfor
parse (inputParser)
parseSoapResponse
pascal
patch
Patch Properties
path
path2rc
pathsep
pathtool
pause
pbaspect
pcg
pchip
pcode
pcolor
pdepe
pdeval
peaks
perl
perms
permute
persistent
pi
pie

shading
shg
shiftdim
showplottool
shrinkfaces
sign
sin
sind
single
sinh
size
size (Map)
size (serial)
size (timeseries)
size
size (tscollection)
slice
smooth3
snapnow
sort
sortrows
sound
soundsc
spalloc
sparse
spaugment
spconvert
spdiags
specular
speye
spfun
sph2cart
sphere
spinmap
spline
spones
spparms

treeplot
tril
trimesh
triplequad
triplot
TriRep
TriRep
TriScatteredInterp
TriScatteredInterp
trisurf
triu
true
try
tscollection
tsdata.event
tsearch
tsearchn
tsprops
tstool
type
typecast
uibuttongroup
Uibuttongroup Properties
uicontextmenu
Uicontextmenu Properties
uicontrol
Uicontrol Properties
uigetdir
uigetfile
uigetpref
uiimport
uimenu
Uimenu Properties
uint8, uint16, uint32, uint64
uiopen
uipanel
Uipanel Properties

vander
var
var (timeseries)
varargin
varargout
vectorize
ver
verctrl
verLessThan
version
vertcat
vertcat (timeseries)
vertcat (tscollection)
vertexAttachments
view
viewmtx
visdiff
volumebounds
voronoi
voronoiDiagram
voronoin
wait
waitbar
waitfor
waitforbuttonpress
warndlg
warning
waterfall
wavfinfo
wavplay
wavread
wavrecord
wavwrite
web
weekday
what
whatsnew

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, `[]`).

The MATLAB software 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

- 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:

- When running MATLAB on The Open Group UNIX platforms, ask your system manager to increase your swap space.
- On Microsoft Windowsplatforms, 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

Padé approximation of time delays

Syntax

```
[num,den] = padecoef(T,N)
```

Description

`[num,den] = padecoef(T,N)` returns the `N`th-order Padé approximation of the continuous-time delay T in transfer function form. The row vectors `num` and `den` contain the numerator and denominator coefficients in descending powers of T. Both are `N`th-order polynomials.

Class support for input T:

```
float: double, single
```

Class Support

Input T support floating-point values of type `single` or `double`.

References

[1] Golub, G. H. and C. F. Van Loan *Matrix Computations*, 3rd ed. Johns Hopkins University Press, Baltimore: 1996, pp. 572–574.

See Also

`pade`

pagesetupdlg

Purpose

Page setup dialog box

Syntax

```
dlg = pagesetupdlg(fig)
```

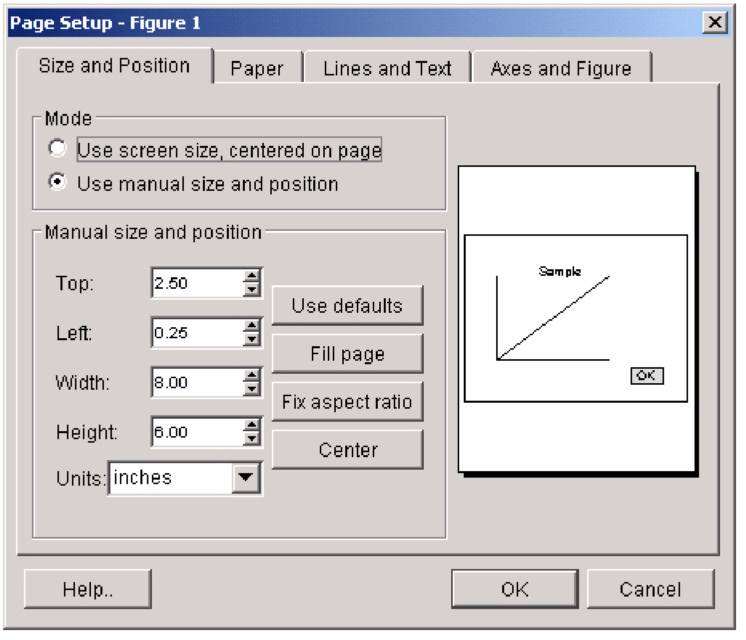
Note This function is obsolete. Use `printpreview` instead.

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.



See Also

printdlg, printpreview, printopt

Purpose

Pan view of 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 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 pan *mode object* for the figure `figure_handle` for you to customize the mode's behavior.

Using Pan Mode Objects

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, a read-only property that cannot be set

Pan Mode Callbacks

You can program the following callbacks for pan mode operations.

- `ButtonDownFilter <function_handle>` — Function to intercept `ButtonDown` events

  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 Graphics object callbacks):

  ```
  function [res] = myfunction(obj,event_obj)
  % obj          handle to the object that has been clicked on
  % event_obj    event data (empty in this release)
  % res [output] 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>` — Function to execute before panning

  Set this callback to if you need to execute code when a pan operation begins. The function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

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

  The event data struct has the following field:

`Axes`	The handle of the axes that is being panned

- ActionPostCallback <function_handle> — Function to execute after panning

  Set this callback if you need to execute code when a pan operation ends. The function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked on
  % event_obj   object containing struct of event data (same as the
  %             event data of the 'ActionPreCallback' callback)
  ```

Pan Mode Utility Functions

The following functions in pan mode query and set certain of its properties.

- `flags = isAllowAxesPan(h,axes)` — Function querying permission to pan axes

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

- `setAllowAxesPan(h,axes,flag)` — Function to set permission to pan axes

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

- `info = getAxesPanMotion(h,axes)` — Function to get style of pan operations

  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)` — Function to set style of pan operations

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

Examples

Example 1 — Entering Pan Mode

Plot a graph and turn on Pan mode:

```
plot(magic(10));
pan on
% pan on the plot
```

Example 2 — Constrained Pan

Constrain pan to *x*-axis using `set`:

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

Example 3 — Constrained Pan in Subplots

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 — Coding a ButtonDown Callback

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.
% Indicate what the target is
disp(['Clicked ' get(obj,'Type') ' object'])
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5 — Coding Pre- and Post-Callback Behavior

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 [%.2f %.2f].',newLim));
```

Example 6 — Creating a Context Menu for Pan Mode

Coding a context menu that lets the user to switch to Zoom mode by right-clicking:

```
figure; plot(magic(10));
hCM = uicontextmenu;
hMenu = uimenu('Parent',hCM,'Label','Switch to zoom',...
        'Callback','zoom(gcbf,''on'')');
hPan = pan(gcf);
set(hPan,'UIContextMenu',hCM);
pan('on')
```

You cannot add items to the built-in pan context menu, but you can replace it with your own.

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.

Note **Do not change figure callbacks within an interactive mode.** While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a GUI that updates a figure's callbacks, the GUI should some keep track of which interactive mode is active, if any, before attempting to do this.

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 carries 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 1-105 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 `NaN`s. 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.

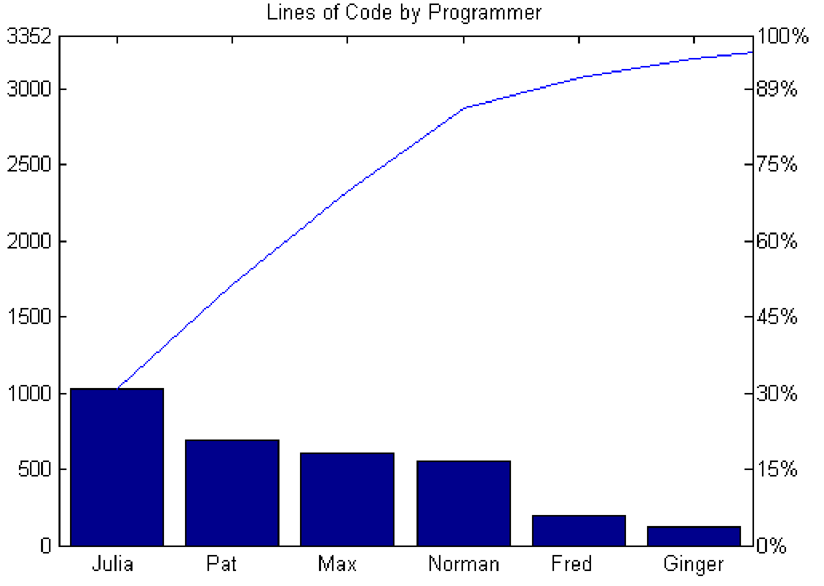
Examples

Example 1:

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

```
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')
```



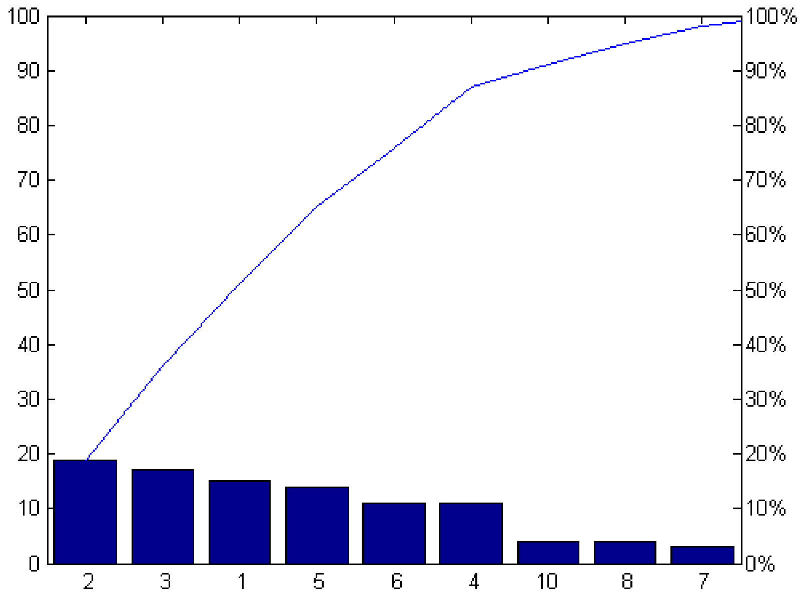
Example 2:

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.

You cannot place datatips (use the Datacursor tool) on graphs created with `pareto`.

See Also

`hist`, `bar`

Purpose

Parallel `for`-loop

Syntax

```
parfor loopvar = initval:endval; statements; end
parfor (loopvar = initval:endval, M); statements; end
```

Description

`parfor loopvar = initval:endval; statements; end` executes a series of MATLAB commands denoted here as `statements` for values of `loopvar` between `initval` and `endval`, inclusive, which specify a vector of increasing integer values. Unlike a traditional `for`-loop, there is no guarantee of the order in which the loop iterations are executed.

The general format of a `parfor` statement is:

```
parfor loopvar = initval:endval
    <statements>
end
```

Certain restrictions apply to the `statements` to ensure that the iterations are independent, so that they can execute in parallel. If you have the Parallel Computing Toolbox™ software, the iterations of `statements` can execute in parallel on separate MATLAB workers on your multi-core computer or computer cluster.

To execute the loop body in parallel, you must open a pool of MATLAB workers using the `matlabpool` function, which is available in Parallel Computing Toolbox.

`parfor (loopvar = initval:endval, M); statements; end` executes statements in a loop using a maximum of `M` MATLAB workers to evaluate statements in the body of the `parfor`-loop. Input variable `M` must be a nonnegative integer. By default, MATLAB uses up to as many workers as it finds available.

When any of the following are true, MATLAB does not execute the loop in parallel:

- There are no workers in a MATLAB pool
- You set `M` to zero

- You do not have Parallel Computing Toolbox

If you have Parallel Computing Toolbox, you can read more about `parfor` and `matlabpool` by typing

```
doc distcomp/parfor
doc distcomp/matlabpool
```

Examples

Perform three large eigenvalue computations using three computers or cores:

```
matlabpool(3)
parfor i=1:3, c(:,i) = eig(rand(1000)); end
```

See Also

`for`

parse (inputParser)

Purpose

Parse and validate named inputs

Syntax

```
p.parse(arglist)
parse(p, arglist)
```

Description

`p.parse(arglist)` parses and validates the inputs named in `arglist`.

`parse(p, arglist)` is functionally the same as the syntax above.

For more information on the `inputParser` class, see in the MATLAB Programming Fundamentals documentation.

Examples

Write an M-file function called `publish_ip`, based on the MATLAB `publish` function, to illustrate the use of the `inputParser` class. Construct an instance of `inputParser` and assign it to variable `p`:

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

Add arguments to the schema. See the reference pages for the `addRequired`, `addOptional`, and `addParamValue` methods for help with this:

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

Call the `parse` method of the object to read and validate each argument in the schema:

```
p.parse(script, varargin{:});
```

Execution of the `parse` method validates each argument and also builds a structure from the input arguments. The name of the structure is

Results, which is accessible as a property of the object. To get the value of any input argument, type

```
p.Results.argname
```

Continuing with the publish_ip exercise, add the following lines to your M-file:

```
% Parse and validate all input arguments.
p.parse(script, varargin{:});

% Display the value for maxHeight.
disp(sprintf('\nThe maximum height is %d.\n', p.Results.maxHeight))

% Display all arguments.
disp 'List of all arguments:'
disp(p.Results)
```

When you call the program, MATLAB assigns those values you pass in the argument list to the appropriate fields of the Results structure. Save the M-file and execute it at the MATLAB command prompt with this command:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', 'C:/matlab/test', ...
    'maxWidth', 500, 'maxHeight', 300);

The maximum height is 300.

List of all arguments:
              format: 'ppt'
           maxHeight: 300
            maxWidth: 500
           outputDir: 'C:/matlab/test'
              script: 'ipscript.m'
```

See Also

inputParser, addRequired(inputParser), addOptional(inputParser), addParamValue(inputParser), createCopy(inputParser)

parseSoapResponse

Purpose Convert response string from SOAP server into MATLAB types

Syntax `parseSoapResponse(response)`

Description `parseSoapResponse(response)` extracts data from `response` a string returned by a SOAP server from the `callSoapService` function, and converts it to appropriate MATLAB classes (types).

Examples This example uses `parseSoapResponse` in conjunction with other SOAP functions to retrieve information about books from a library database, specifically, the author's name for a given book title.

Note The example does not use an actual endpoint; therefore, you cannot run it. The example only illustrates how to use the SOAP functions.

```
% Create the message:
message = createSoapMessage(...
'urn:LibraryCatalog',...
'getAuthor',...
{'In the Fall'},...
{'nameToLookUp'},...
{'{http://www.w3.org/2001/XMLSchema}string'},...
'rpc');
%
% Send the message to the service and get the response:
response = callSoapService(...
'http://test/soap/services/LibraryCatalog',...
'urn:LibraryCatalog#getAuthor',...
message)
%
% Extract MATLAB data from the response
author = parseSoapResponse(response)
```

MATLAB returns:

```
author = Kate Alvin
```

where author is a char class (type).

See Also

`callSoapService`, `createClassFromWsdl`, `createSoapMessage`, `urlread`, `xmlread`

in the MATLAB External Interfaces documentation

pascal

Purpose

Pascal matrix

Syntax

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

Description

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

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

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

Examples

`pascal(4)` returns

```
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`

Purpose

Create one or more filled polygons

Syntax

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

Description

`patch(X,Y,C)` adds a filled 2-D patch object to the current axes. A patch object is one or more polygons defined by the coordinates of its vertices. The elements of `X` and `Y` specify the vertices of a polygon . If `X` and `Y` are `m`-by-`n` matrices, MATLAB draws `n` polygons with `m` vertices. `C` determines the color of the patch. For more information on color input requirements, see "Coloring Patches" on page 2-2711.

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

See in for more information on using patch objects.

`patch(X,Y,Z,C)` creates a patch in 3-D coordinates. 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 might be only partly filled. In that case, it is better to divide the face into smaller polygons.

`patch(FV)` creates a patch using structure `FV`, which contains the fields `vertices`, `faces`, and optionally `facevertexcdata`. These fields correspond to the `Vertices`, `Faces`, and `FaceVertexCData` patch properties. Specifying only unique vertices and their connection matrix can reduce the size of the data for patches having many faces. For an example of how to specify patches with this method, see "Specifying Patch Object Shapes" on page 2-2708.

`patch(X,Y,C,'PropertyName',propertyvalue...)` follows the `X`, `Y`, `(Z)`, and `C` arguments with property name/property value pairs to specify additional 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). See `Patch Properties` for a full list of modifiable properties.

`patch('PropertyName',propertyvalue,...)` specifies all properties using property name/property value pairs. This form lets you 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 lets you specify the patch using the `Faces` and `Vertices` properties instead of *x*-, *y*-, and *z*-coordinates. See "Specifying Patch Object Shapes" on page 2-2708 for more information.

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

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.

Examples

Specifying Patch Object Shapes

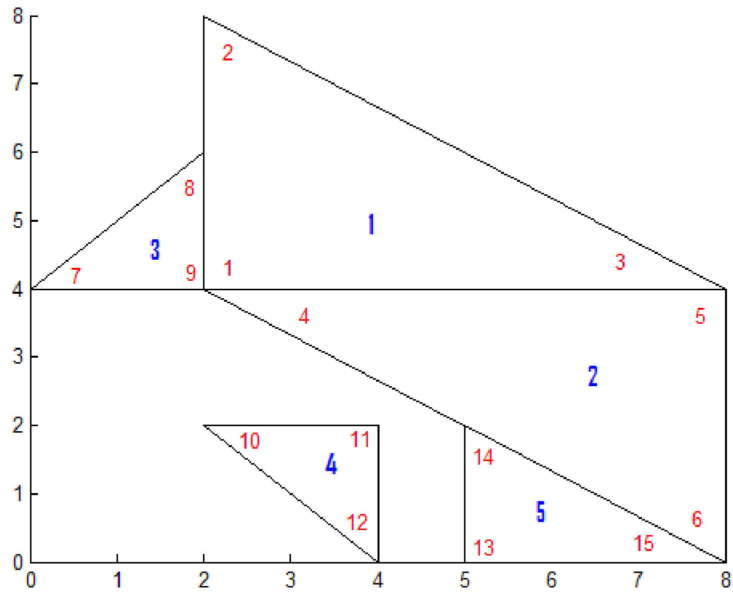
The next two examples create a patch object using two methods:

- 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`, and `FaceVertexCData` properties)

Create five triangular faces, each having three vertices, by specifying the *x*-, *y*-, and *z*-coordinates of each vertex:

```
xdata = [2 2 0 2 5;
         2 8 2 4 5;
         8 8 2 4 8];
ydata = [4 4 4 2 0;
```

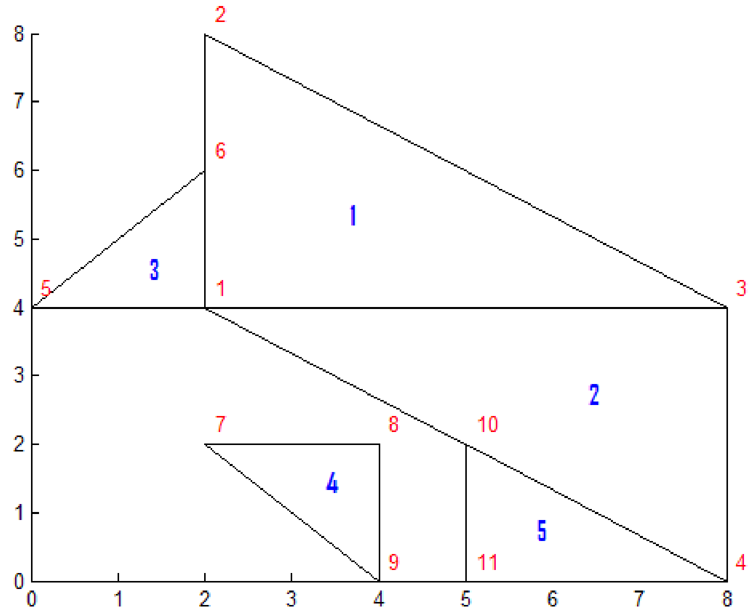
```
          8 4 6 2 2;
          4 0 4 0 0];
zdata = ones(3,5);

% Red numbers denote the vertex indices.
% For this example:
% xindices = [1 4 7 10 13;
%              2 5 8 11 14;
%              3 6 9 12 15];
% Blue numbers denote the face numbers.
patch(xdata,ydata,zdata,'w')
```



Create the five triangular faces, specifying faces and vertices:

```
% 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. More than one face may use a given vertex.
% For this example, five triangles have 11 total vertices,
% instead of 15. Each row contains the x- and y-coordinates
% of each vertex.
verts = [2 4; ...
        2 8; ...
        8 4; ...
        8 0; ...
        0 4; ...
        2 6; ...
        2 2; ...
        4 2; ...
        4 0; ...
        5 2; ...
        5 0 ];

% There are five faces, defined by connecting the
% vertices in the order indicated.
faces = [ ...
        1  2  3; ...
        1  3  4; ...
        5  6  1; ...
        7  8  9; ...
        11 10 4 ];

% Create the patch by specifying the Faces, Vertices,
% and FaceVertexCData properties as well as the
% FaceColor property. Red numbers denote the vertex
% numbers, as defined in faces. Blue indicate face numbers.
p =
patch('Faces',faces,'Vertices',verts,'FaceColor','w');
```




```
% Using the previous values for verts and faces, you can
% create the same patch object using a structure:
patchinfo.Vertices = verts;
patchinfo.Faces = faces;
patchinfo.FaceColor = 'w';

patch(patchinfo);
```

Coloring Patches

There are many ways to customize your patch objects using colors. The appropriate input depends on:

- Whether you want to change the edge colors
- How you specified the patch faces:
  - Using face/vertex values

- Using *x*-, *y*-, and *z*-coordinates

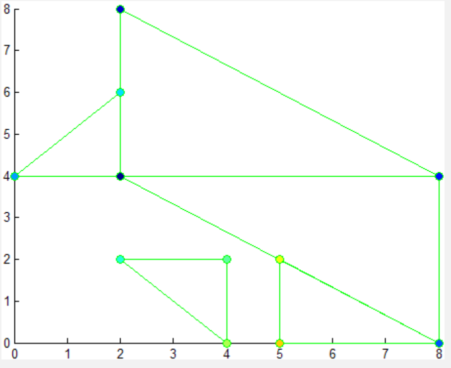
The following sections present the various options available.

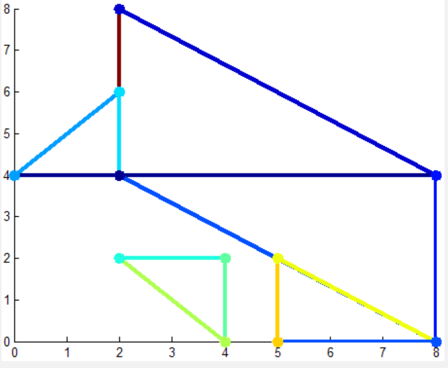
Specifying Edge Colors

The following options apply to the edge colors of your patch object. The settings are independent of the face colors, but the colors themselves depend on the colors specified at each vertex. Markers show the color at each vertex. Specify the colors using the EdgeColor property. To explore the options using the Sample Input Code, first start with a base patch object:

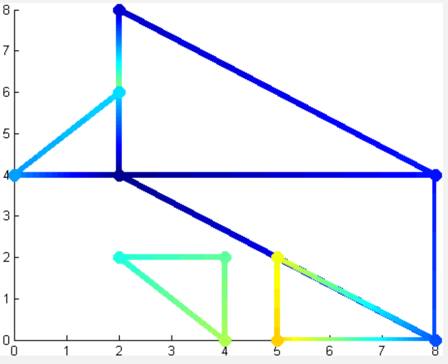
```
xdata = [2     2     0     2     5;
         2     8     2     4     5;
         8     8     2     4     8];
ydata = [4     4     4     2     0;
         8     4     6     2     2;
         4     0     4     0     0];
cdata = [15     0     4     6    10;
         1     2     5     7     9;
         2     3     0     8     3];
p = patch(xdata,ydata,cdata,'Marker','o','MarkerFaceColor','flat','Fac
```

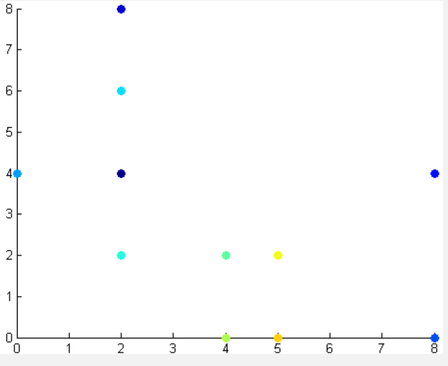
For more detailed information on how the EdgeColor property works, see the Patch Properties page.

Desired Look	EdgeColor Value	Sample Code
All edges have the same color, around all faces. This option does not rely on the FaceColor value. 	ColorSpec	`set(p,'EdgeColor','g')`

Desired Look	EdgeColor Value	Sample Code
Each edge corresponds to the color of the vertex that precedes the edge, with one color per edge. This option requires that the `FaceColor` property be `flat` or `interp`. By default, if you specify `CData` when creating the patch object, its `FaceColor` property is `interp`. 	`'flat'`	`set(p,'EdgeColor','flat'...` `'LineWidth',3)`

Desired Look	EdgeColor Value	Sample Code

Each edge corresponds to the vertex colors, interpolated between vertices. This option requires that the `FaceColor` property be `flat` or `interp`. By default, if you specify `CData` when creating the patch object, its `FaceColor` property is `interp`. 	`'interp'`	`set(gcf,'Renderer','zbuffer')`
		`set(p,'EdgeColor','interp',...`
		`'LineWidth',5)`

Edges have no color. This option does not rely on the `FaceColor` value. If set, markers retain vertex colors. 	`'none'`	`set(p,'EdgeColor','none')`

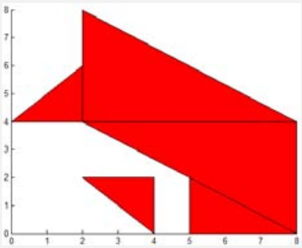
Specifying Face Colors Using Face/Vertex Input Matrics

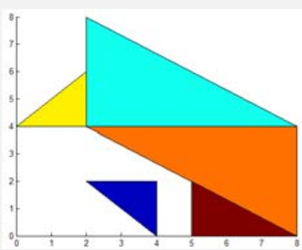
The following options apply to the face colors of your patch object when you specify the faces using face/vertex input matrices. To explore the options, first start with a base patch object:

```
% For this example, there are five triangles (m = 5) sharing
% eleven unique vertices (k = 11).
verts = [2 4; ...
        2 8; ...
        8 4; ...
        8 0; ...
        0 4; ...
        2 6; ...
        2 2; ...
        4 2; ...
        4 0; ...
        5 2; ...
        5 0 ];
faces = [1  2  3; ...
         1  3  4; ...
         5  6  1; ...
         7  8  9; ...
         11 10 4];
p = patch('Faces',faces,'Vertices',verts,'FaceColor','b');
```

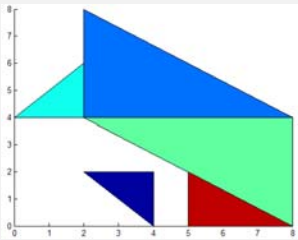
For more information on the relevant properties, see `FaceColor`, `FaceVertexCData`, and `CDataMapping`.

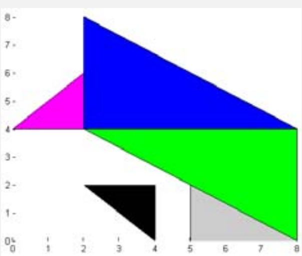
Desired Look	Parameter Values	Sample Code

All faces have the same color.

	• `FaceColor`: `ColorSpec`
	• `FaceVertexCData`: `[]` (no input)
	An empty array is the default value, and `patch` ignores any input until you set `FaceColor` to `'flat'` or `'interp'`.
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`set(p,'FaceColor','r')`
		or
		`set(p,'FaceColor',[1 0 0])`

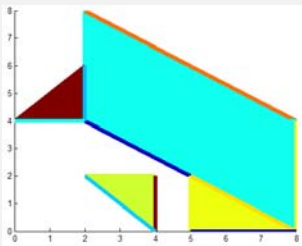
Each face has a single, unique color, indexed from a selected section of the colormap.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: `m`-by-1 matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata)`

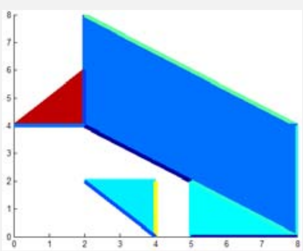
Desired Look	Parameter Values	Sample Code

Each face has a single, unique color, indexed from the whole colormap.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: m-by-1 matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','direct'`.
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'CDataMapping','direct')`

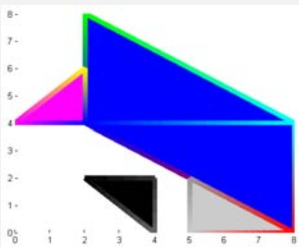
Each face has a single, unique color, determined by truecolor value input.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: m-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata = [0 0 1 0 0.8;`
		`0 1 0 0 0.8;`
		`1 0 1 0 0.8]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata)`

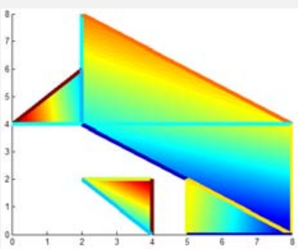
Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, indexed from a selected section of the colormap. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: k-by-1 matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2...`
		`60 12 23 40 13 26 24]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

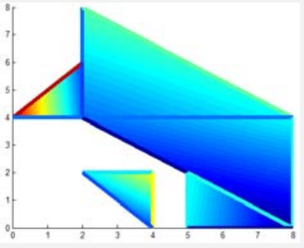
Each unique vertex has a single, unique color, indexed from the whole colormap. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: k-by-1 matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','direct'`.
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2...`
		`60 12 23 40 13 26 24]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'CDataMapping','direct',...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

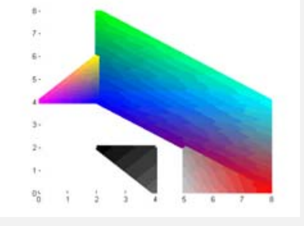
Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, determined by truecolor value input. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: k-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata = [0 0 1;`
		`0 1 0;`
		`0 1 1;`
		`1 0 0;`
		`1 0 1;`
		`1 1 0;`
		`0 0 0;`
		`0.2 0.2 0.2;`
		`0.4 0.4 0.4;`
		`0.6 0.6 0.6;`
		`0.8 0.8 0.8];`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'EdgeColor','interp',...`
		`'LineWidth',5)`

Each unique vertex has a single, unique color, indexed from a selected section of the colormap. Edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'interp'`
	• `FaceVertexCData`: k-by-1 matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2...`
		`60 12 23 40 13 26 24]';`
		`set(p,'FaceColor','interp',...`
		`'FaceVertexCData',cdata,...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, indexed from the whole colormap. Edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'interp'`
	• `FaceVertexCData`: `k`-by-1 matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','direct'`.
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2...`
		` 60 12 23 40 13 26 24]';`
		`set(p,'FaceColor','interp',...`
		`'FaceVertexCData',cdata,...`
		`'CDataMapping','direct',...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

Each unique vertex has a single, unique color, determined by truecolor value input. Edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'interp'`
	• `FaceVertexCData`: `k`-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata = [0 0 1;`
		`         0 1 0;`
		`         0 1 1;`
		`         1 0 0;`
		`         1 0 1;`
		`         1 1 0;`
		`         0 0 0;`
		`         0.2 0.2 0.2;`
		`         0.4 0.4 0.4;`
		`         0.6 0.6 0.6;`
		`         0.8 0.8 0.8];`
		`set(p,'FaceColor','interp',...`
		`'FaceVertexCData',cdata,...`
		`'EdgeColor','interp',...`
		`'LineWidth',5)`

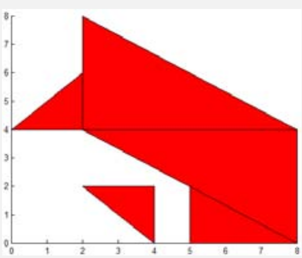
Specifying Face Colors Using x-, y-, and z-Coordinate Input

The following options apply to the face colors of your patch object when you specify the faces using *x*-, *y*-, and *z*-coordinates. To explore the options, first start with a base patch object:

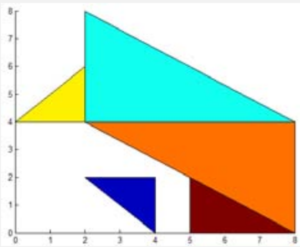
```
% For this example, there are five (m=5) triangles (n=3).
% The total number of vertices is mxn, or k = 15.
xdata = [2 2 0 2 5;
         2 8 2 4 5;
         8 8 2 4 8];
ydata = [4 4 4 2 0;
         8 4 6 2 2;
         4 0 4 0 0];
zdata = ones(3,5);
p = patch(xdata,ydata,zdata,'b')
```

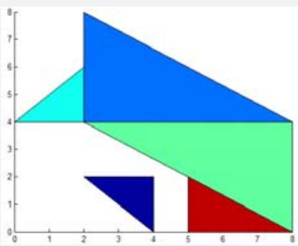
For more information on the relevant properties, see `FaceColor`, `CData`, and `CDataMapping`.

Desired Look	Parameter Values	Sample Code

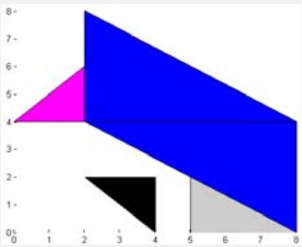
All faces have the same color.

	• `FaceColor`: `ColorSpec`
	• `FaceVertexCData`: `[]` (no input)
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`set(p,'FaceColor','r')`
		or
		`set(p,'FaceColor',[1 0 0])`

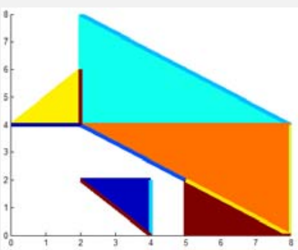
Desired Look	Parameter Values	Sample Code

Each face has a single, unique color, indexed from a selected section of the colormap.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: m-by-1 matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata)`

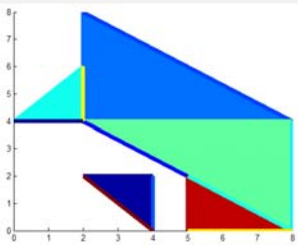
Each face has a single, unique color, indexed from the whole colormap.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: m-by-1 matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','direct'`.
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'CDataMapping','direct')`

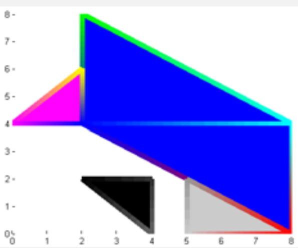
Desired Look	Parameter Values	Sample Code

Each face has a single, unique color, determined by truecolor value input.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: m-by-1-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata(:,:,1) = [0 0 1 0 0.8];`
		`cdata(:,:,2) = [0 0 0 0 0.8];`
		`cdata(:,:,3) = [1 1 1 0 0.8];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata)`

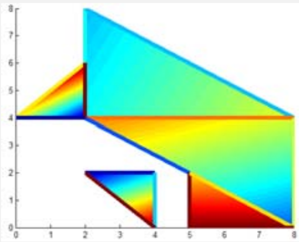
Each unique vertex has a single, unique color, indexed from a selected section of the colormap. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: `m`-by-`n` matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60;`
		`         12 23 40 13 26;`
		`         24 8 1 65 42];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

Desired Look	Parameter Values	Sample Code

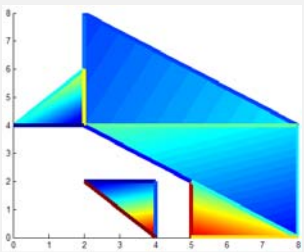
Each unique vertex has a single, unique color, indexed from the whole colormap. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: `m-by-n` matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','direct'`.
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60;`
		`12 23 40 13 26;`
		`24 8 1 65 42];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'CDataMapping','direct',...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

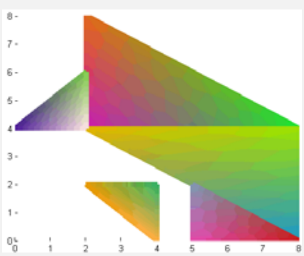
Each vertex has a single, unique color, determined by truecolor value input. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: `m-by-n-by-3` matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata(:,:,1) = [0 0 1 0 0.8;`
		`0 0 1 0.2 0.6;`
		`0 1 0 0.4 1];`
		
		`cdata(:,:,2) = [0 0 0 0 0.8;`
		`1 1 1 0.2 0.6;`
		`1 0 0 0.4 0];`
		
		`cdata(:,:,3) = [1 1 1 0 0.8;`
		`0 1 0 0.2 0.6;`
		`1 0 1 0.4 0];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'EdgeColor','interp',...`
		`'LineWidth',5)`

Desired Look	Parameter Values	Sample Code

Each vertex has a single, unique color, indexed from a selected section of the colormap. Edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'interp'`
	• `FaceVertexCData`: `m-by-n` matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60;`
		`         12 23 40 13 26;`
		`         24 8 1 65 42];`
		`set(p,'FaceColor','interp',...`
		`'CData',cdata,...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

Desired Look	Parameter Values	Sample Code

Each vertex has a single, unique color, indexed from the whole colormap. Edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'interp'`
	• `FaceVertexCData`: m-by-n matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','direct'`.
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60;`
		`        12 23 40 13 26;`
		`        24 8 1 65 42];`
		`set(p,'FaceColor','interp',...`
		`'CData',cdata,...`
		`'CDataMapping','direct',...`
		`'EdgeColor','flat',...`
		`'LineWidth',5)`

Each vertex has a single, unique color, determined by truecolor value input. Edges may have `'flat'` or `'interp'` color.

	• `FaceColor`: `'interp'`
	• `FaceVertexCData`: m-by-n-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata(:,:,1) = [0.8 0.1 0.2`
		`    0.9 0.3 1;`
		`    0.1 0.5 0.9;`
		`    0.9 1 0.5;`
		`    0.6 0.9 0.8];`
		
		`cdata(:,:,2) =[0.1 0.6 0.7;`
		`    0.4 0.1 0.7;`
		`    0.9 0.8 0.3;`
		`    0.7 0.9 0.6;`
		`    0.9 0.6 0.1];`
		
		`cdata(:,:,3) =[0.7 0.8 0.4;`
		`    0.1 0.6 0.3;`
		`    0.2 0.3 0.7;`
		`    0.0 0.9 0.7;`
		`    0.0 0.0 0.1];`
		`set(p,'FaceColor','interp',...`
		`'CData',cdata,...`
		`'EdgeColor','interp',...`
		`'LineWidth',5)`

See Also

area | caxis | fill | fill3 | isosurface | surface | FaceColor | CData | CDataMapping | FaceVertexCData

Tutorials

-

Purpose

Patch properties

Modifying Properties

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

- 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 .

See 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 the MATLAB software 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 axes 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.

Annotation
hg.Annotation object Read Only

Control the display of patch objects in legends. The Annotation property enables you to specify whether this patch object is represented in a figure legend.

Querying the Annotation property returns the handle of an hg.Annotation object. The hg.Annotation object has a property called LegendInformation, which contains an hg.LegendEntry object.

Once you have obtained the hg.LegendEntry object, you can set its IconDisplayStyle property to control whether the patch object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this patch object in a legend (default)
`off`	Do not include this patch object in a legend
`children`	Same as `on` because patch objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the `IconDisplayStyle` property

See for more information and examples.

Selecting which objects to display in legend

Some graphics functions create multiple objects. For example, `contour3` uses patch objects to create a 3D contour graph. You can use the `Annotation` property set select a subset of the objects for display in the legend.

```
[X,Y] = meshgrid(-2:.1:2);
[Cm hC] = contour3(X.*exp(-X.^2-Y.^2));
hA = get(hC,'Annotation');
hLL = get([hA{:}],'LegendInformation');
% Set the IconDisplayStyle property to display
% the first, fifth, and ninth patch in the legend
set([hLL{:}],{'IconDisplayStyle'},...
   {'on','off','off','off','on','off','off','off','on'}')
```

```
% Assign DisplayNames for the three patch
that are displayed in the legend
set(hC([1,5,9]),{'DisplayName'},{'bottom','middle','top'}')
legend show
```

BackFaceLighting

`unlit | lit | {reverselit}`

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

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

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

BeingDeleted

`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

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

BusyAction

`cancel | {queue}`

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

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

ButtonDownFcn

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

Button press callback 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.

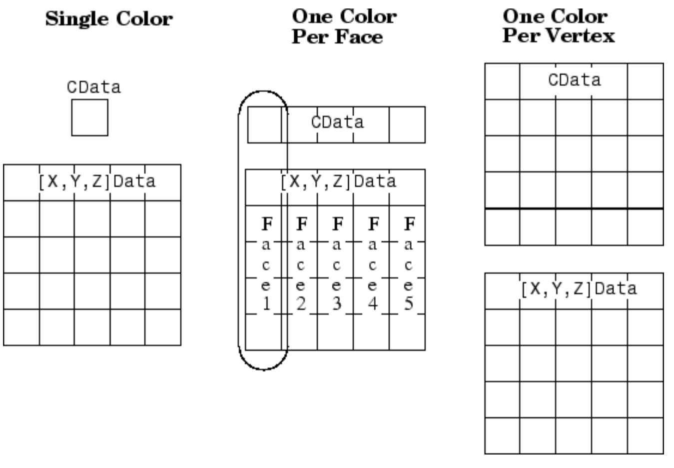
See for information on how to use function handles to define the callback function.

CData

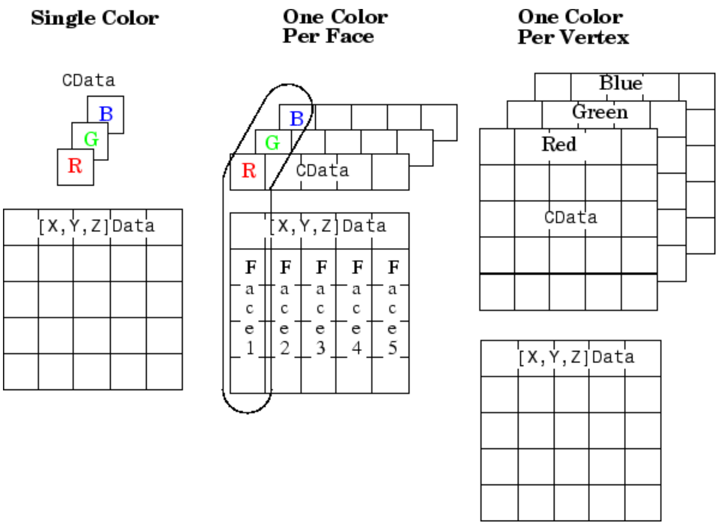
scalar, vector, or matrix

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

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



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



Note that if `CData` contains `NaN`s, 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

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 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.

`DisplayName`
string (default is empty string)

String used by legend for this patch object. The `legend` function uses the string defined by the `DisplayName` property to label this patch object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this patch object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See for more examples.

`EdgeAlpha`

`{scalar = 1} | flat | interp`

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

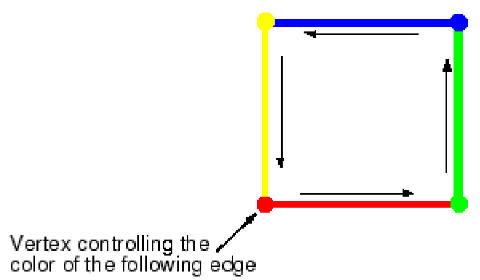
- scalar — A single non-NaN scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) means fully opaque and `0` means completely transparent.
- `flat` — The alpha data (`FaceVertexAlphaData`) of each vertex controls the transparency of the edge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

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

`EdgeColor`
`{ColorSpec} | none | flat | interp`

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

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



- `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

  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.

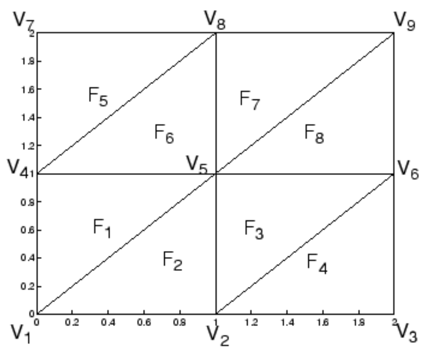
- phong — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

Faces

m-by-n matrix

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.



Faces property

F_1	V_1	V_4	V_5
F_2	V_1	V_5	V_2
F_3	V_2	V_5	V_6
F_4	V_2	V_6	V_3
F_5	V_4	V_7	V_8
F_6	V_4	V_8	V_5
F_7	V_5	V_8	V_9
F8	V_5	V_9	V_6

Vertices property

V_1	X_1	Y_1	Z_1
V_2	X_2	Y_2	Z_2
V_3	X_3	Y_3	Z_3
V_4	X_4	Y_4	Z_4
V_5	X_5	Y_5	Z_5
V_6	X_6	Y_6	Z_6
V_7	X_7	Y_7	Z_7
V_8	X_8	Y_8	Z_8
V_9	X_9	Y_9	Z_9

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.

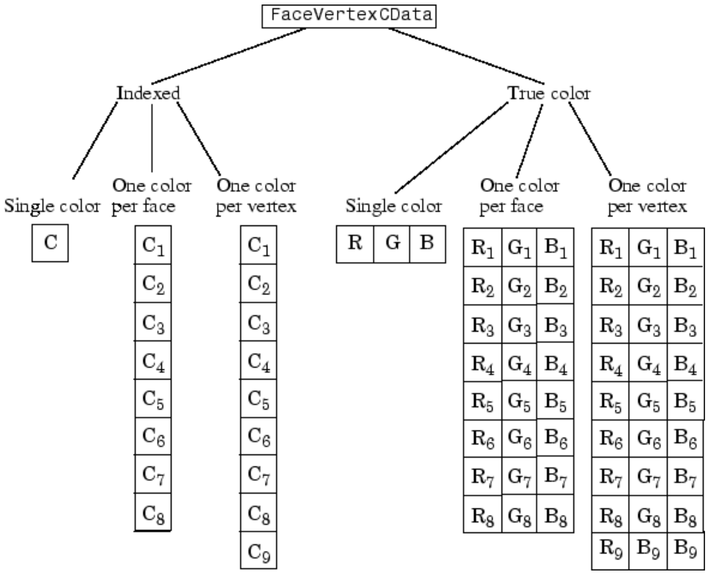
For indexed colors, `FaceVertexCData` can be

- 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
- 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.

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 = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`

character (see table)

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

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} | 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 = $^1/_{72}$ inch). Note that MATLAB draws the `point` marker at 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 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.

`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 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`

Purpose

View or change search path

GUI Alternatives

As an alternative to the `path` function, use the Set Path dialog box.

Syntax

```
path
path('newpath')
path(path,'newpath')
path('newpath',path)
p = path
```

Description

`path` displays the MATLAB search path, which is stored in `pathdef.m`.

`path('newpath')` changes the search path to `newpath`, where `newpath` is a string array of folders.

`path(path,'newpath')` adds the `newpath` folder to the end of the search path. If `newpath` is already on the search path, then `path(path, 'newpath')` moves `newpath` to the end of the search path.

`path('newpath',path)` adds the `newpath` folder to the top of the search path. If `newpath` is already on the search path, then `path('newpath', path)` moves `newpath` to the top of the search path. To add multiple folders in one statement, instead use `addpath`.

`p = path` returns the search path to string variable `p`.

Examples

Display the search path:

```
path
```

MATLAB returns, for example

```
MATLABPATH

 H:\My Documents\MATLAB
 C:\Program Files\MATLAB\R200nn\toolbox\matlab\general
 C:\Program Files\MATLAB\R200nn\toolbox\matlab\ops
 C:\Program Files\MATLAB\R200nn\toolbox\matlab\lang
```

```
 C:\Program Files\MATLAB\R200nn\toolbox\matlab\elmat
 C:\Program Files\MATLAB\R200nn\toolbox\matlab\elfun
...
```

`R200nn` represents the folder for the MATLAB release, for example, `R2009b`.

Add a new folder to the search path on Microsoft Windows platforms:

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

Add a new folder to the search path on UNIX[14] platforms:

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

Temporarily add the folder `my_mfiles` to the search path, run `my_function` in `my_mfiles`, then restore the previous search path:

```
p = path
path('my_mfiles')
my_function
path(p)
```

See Also

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

Topics in the User Guide:

-
-

14. UNIX is a registered trademark of The Open Group in the United States and other countries.

Purpose

Save current 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

Search path separator for current platform

Syntax

```
c = pathsep
```

Description

`c = pathsep` returns the search path separator character for this platform. The search path separator is the character that separates path names in the `pathdef.m` file, as returned by the `path` function. The character is a semicolon (`;`). For versions of MATLAB software earlier than version 7.7 (R2008b), the character on UNIX[15] platforms was a colon (`:`). Use `pathsep` to work programmatically with the content of the search path file.

See Also

`fileparts`, `filesep`, `fullfile`, `path`

15. UNIX is a registered trademark of The Open Group in the United States and other countries.

Purpose

Open Set Path dialog box to view and change search path

GUI Alternatives

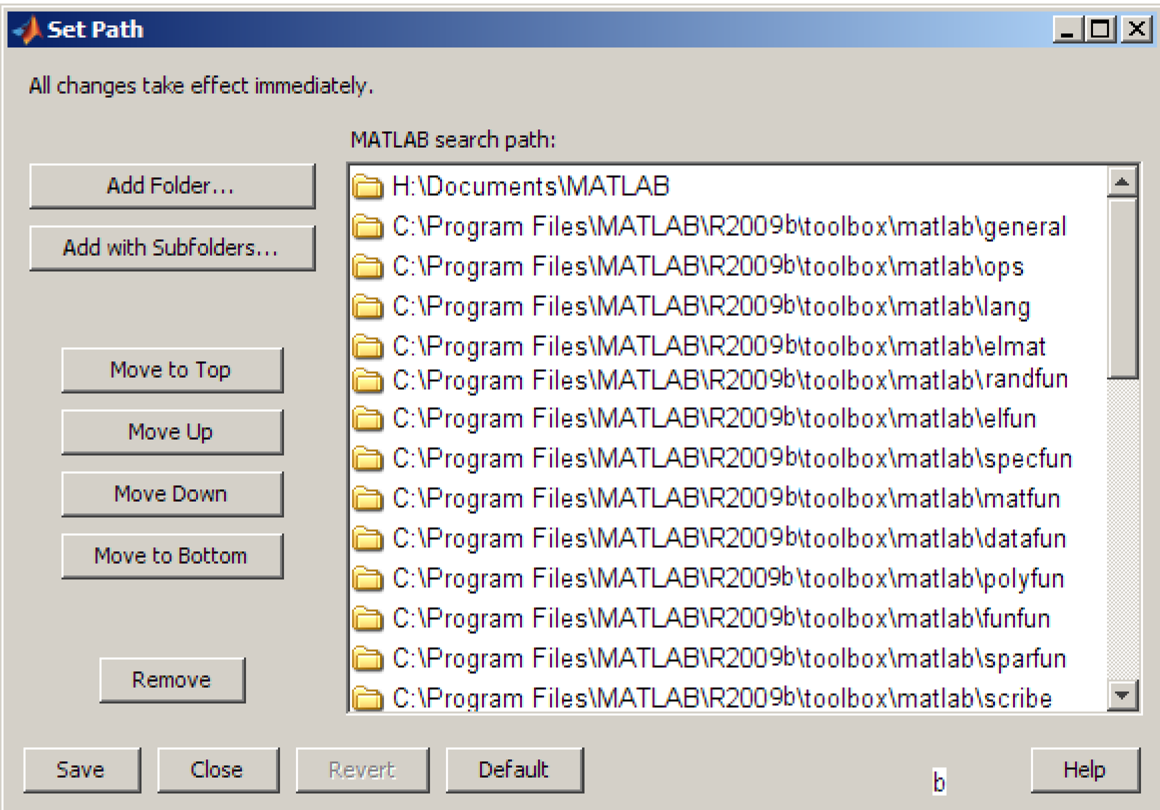
As an alternative to the `pathtool` function, select **File > Set Path** in the MATLAB desktop.

Syntax

`pathtool`

Description

`pathtool` opens the Set Path dialog box, a graphical user interface you use to view and modify the MATLAB search path.



See Also

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

Purpose

Halt execution temporarily

Syntax

```
pause
pause(n)
pause on
pause off
pause query
state = pause('query')
oldstate = pause(newstate)
```

Description

`pause`, by itself, causes the currently executing M-file to stop and wait for you to press any key before continuing. Pausing must be enabled for this to take effect. (See `pause on`, below). `pause` without arguments also blocks execution of Simulink models, but not repainting of them.

`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. Pausing must be enabled for this to take effect.

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

`pause on` enables the pausing of MATLAB execution via the `pause` and `pause(n)` commands. Pausing remains enabled until you enter `pause off` in your M-file or at the command line.

`pause off` disables the pausing of MATLAB execution via the `pause` and `pause(n)` commands. This allows normally interactive scripts to run unattended. Pausing remains disabled until you enter `pause on` in your M-file or at the command line, or start a new MATLAB session.

`pause query` displays `'on'` if pausing is currently enabled. Otherwise, it displays `'off'`.

`state = pause('query')` returns `'on'` in character array `state` if pausing is currently enabled. Otherwise, the value of `state` is `'off'`.

`oldstate = pause(newstate)`, enables or disables pausing, depending on the `'on'` or `'off'` value in `newstate`, and returns the former setting (also either `'on'` or `'off'`) in character array `oldstate`.

Remarks

While MATLAB is paused, the following continue to execute:

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

See Also

`keyboard`, `input`, `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`, the MATLAB software 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,

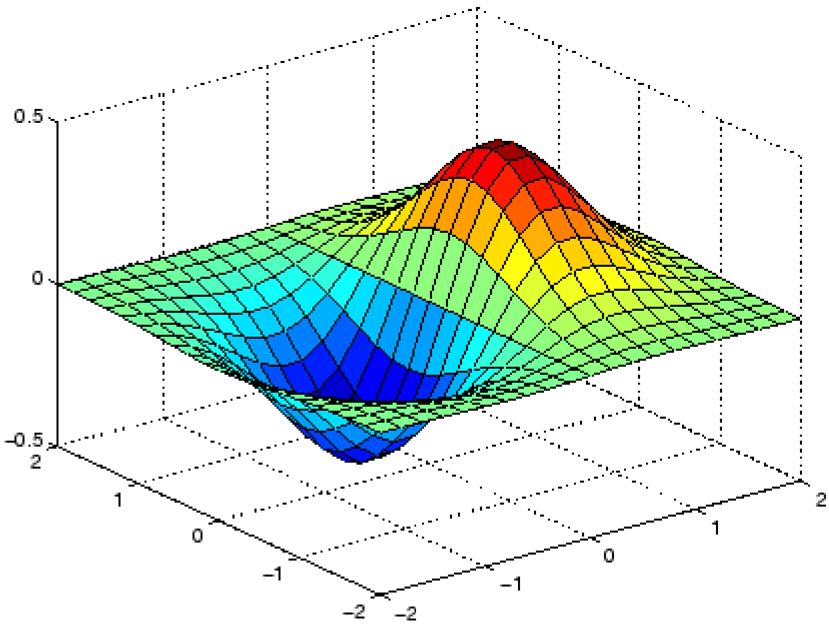
```
pbaspect(pbaspect)
```

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 $-2 \le x \le 2$, $-2 \le y \le 2$,

```
[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.

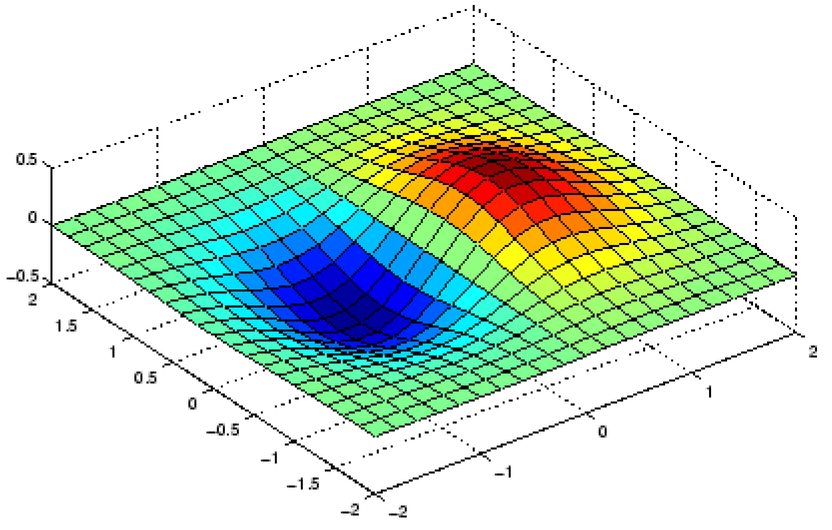
```
pbaspect
ans =
     1   1   1
```

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.

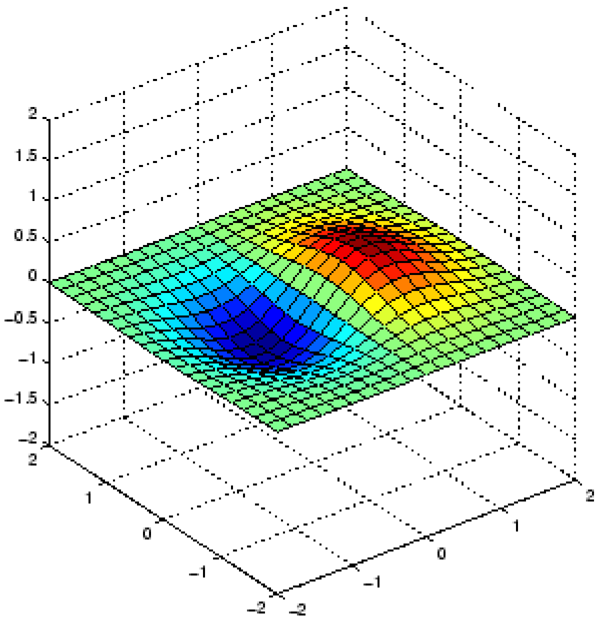
```
daspect([1 1 1])
```




```
pbaspect
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.

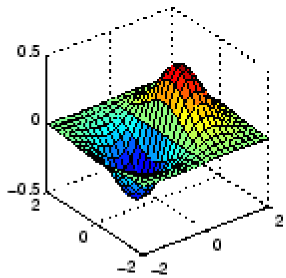
```
pbaspect([1 1 1])
```

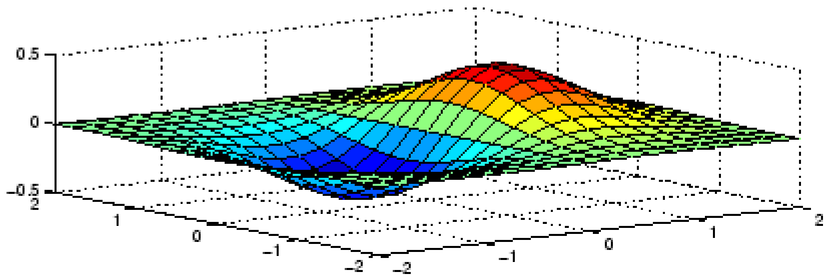


Notice how MATLAB changed the axes limits because of the constraints introduced by specifying both the plot box and data aspect ratios.

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)
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.

, 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 `norm(b-A*x)/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 `inv(M)*A*x = 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\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	pcg converged to the desired tolerance tol within maxit iterations.
1	pcg iterated maxit times but did not converge.
2	Preconditioner M was ill-conditioned.
3	pcg stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during pcg 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 norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

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

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

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

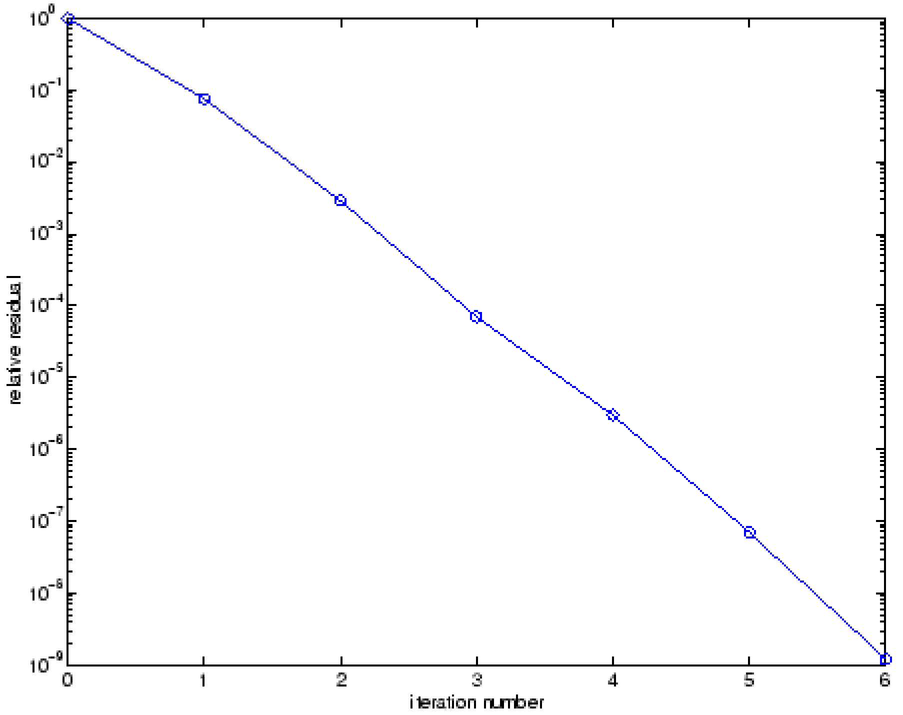
```
A = delsq(numgrid('C',25));
b = ones(length(A),1);
[x,flag] = pcg(A,b)
```

`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 $x_k \le x \le x_{k+1}$, $P(x)$ 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

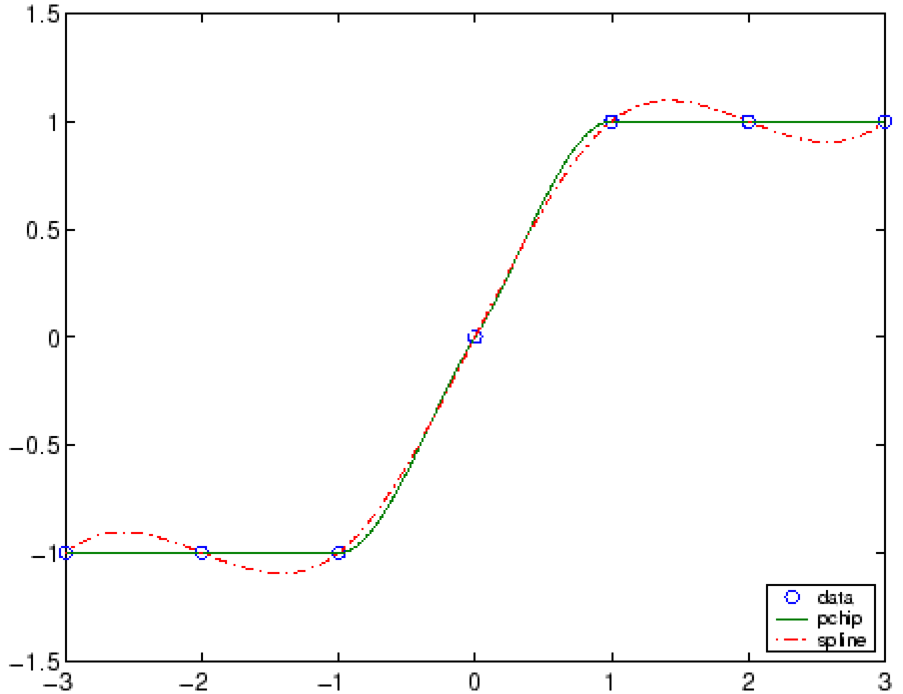
spline constructs $S(x)$ in almost the same way pchip constructs $P(x)$.

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 protected M-file (P-file)

Syntax

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

Description

`pcode` *`fun`* obfuscates (i.e., *shrouds*) M-file `fun.m` for the purpose of protecting its proprietary source code. The encrypted M-code is written to P-file `fun.p` in the current directory. The original M-file can be anywhere on the search path.

If the input file resides within a package and/or class directory, then the same package and class directories are applied to the output file. See example 2, below.

`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 cannot be created.

See in the MATLAB Programming Fundamentals documentation for more information.

Examples

Example 1 – PCoding Multiple Files

Convert selected M-files from the `sparfun` directory into pcode files:

```
dir([matlabroot '\toolbox\matlab\sparfun\spr*.m'])

   .   ..   sprand.m   sprandn.m  sprandsym.m  sprank.m

cd C:\work\pcodetest
pcode([matlabroot '\toolbox\matlab\sparfun\spr*.m'])

dir C:\work\pcodetest

.       sprand.m     sprandn.m    sprandsym.m  sprank.m
```

```
..       sprand.p     sprandn.p    sprandsym.p  sprank.p
```

Example 2 – Parsing Files That Belong to a Package and/or Class

This example takes an input file that is part of a package and class, and generates a pcode file for it in a separate directory. M-file `test.m` resides in the following package and class directory:

```
C:\work\+mypkg\@char\test.m
```

Set your current working directory to empty directory `math\pcodetest`. This is where you will generate the pcode file. This directory has no package or class structure associated with it at this time:

```
cd C:\math\pcodetest
dir
    .   ..
```

Generate pcode for `test.m`. Because the input file is part of a package and class, MATLAB creates directories `+mypkg` and `@char` so that the output file belongs to the same:

```
pcode C:\work\+mypkg\@char\test.m
dir('C:\math\pcodetest\+mypkg\@char')
     .    ..     test.p
```

Example 3 – PCoding In Place

When you generate a pcode file `inplace`, MATLAB writes the output file to the same directory as the input file:

```
pcode C:\work\+mypkg\@char\test.m -inplace
dir C:\work\+mypkg\@char

.        ..       test.m  test.p
```

See Also

`depfun`, `depdir`,

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])`.

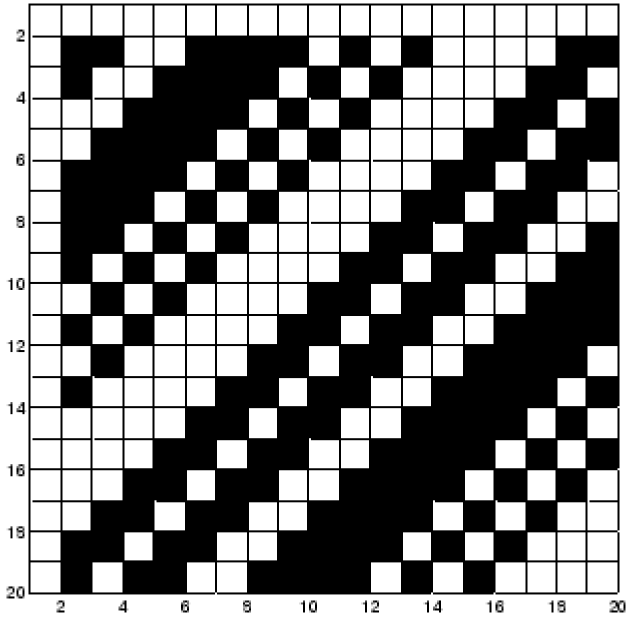
When you use `shading faceted` or `shading flat`, the constant color of each cell is the color associated with the corner having the smallest *x-y* coordinates. Therefore, `C(i,j)` determines the color of the cell in the *i*th row and *j*th column. The last row and column of C are not used.

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

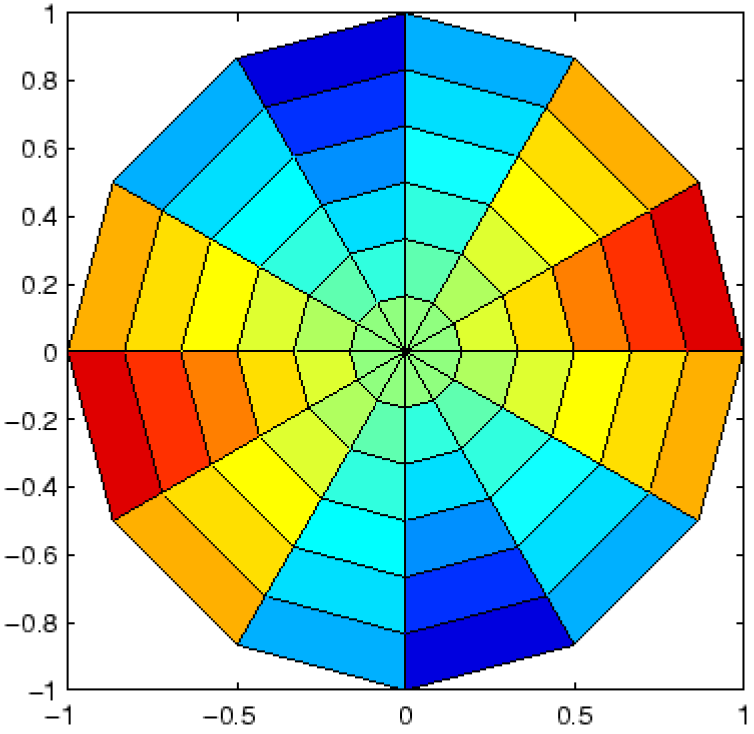
A Hadamard matrix has elements that are +1 and -1. A colormap with only two entries is appropriate when displaying a pseudocolor plot of this matrix.

```
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`

pdepe

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)
[sol,tsol,sole,te,ie] = pdepe(m,pdefun,icfun,bcfun,xmesh,
   tspan,options)
```

Arguments

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

, 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:

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

The PDEs hold for $t_0 \le t \le t_f$ and $a \le x \le b$. 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

$$u(x, t_0) = u_0(x) \tag{2-3}$$

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

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

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

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

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

  ```
  [c,f,s] = pdefun(x,t,u,dudx)
  ```

  The input arguments are scalars `x` and `t` and vectors `u` and `dudx` that approximate the solution u and its partial derivative with respect to x, respectively. `c`, `f`, and `s` are column vectors. `c` stores the diagonal elements of the matrix c (Equation 2-2).
- `icfun` evaluates the initial conditions. It has the form

  ```
  u = 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 p and q of the boundary conditions (Equation 2-4). It has the form

  ```
  [pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
  ```

ul is the approximate solution at the left boundary xl = a and ur is the approximate solution at the right boundary xr = b. pl and ql are column vectors corresponding to p and q evaluated at xl, similarly pr and qr correspond to xr. When $m > 0$ and $a = 0$, 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 pl and ql.

pdepe returns the solution as a multidimensional array sol. u_i = ui = sol(:,:,i) is an approximation to the ith 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.

[sol,tsol,sole,te,ie] = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options) with the 'Events' property in options set to a function handle Events, solves as above while also finding where event functions g(t,u(x,t))are zero. For each function you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Three column vectors are returned by events: [value,isterminal,direction] = events(m,t,xmesh,umesh). xmesh contains the spatial mesh and umesh is the solution at the mesh points. Use pdeval to evaluate the solution between mesh points. For the I-th event function, value(i) is the value of the function, `ISTERMINAL(I) = 1` if the integration is to terminate at a zero of this event function and 0 otherwise. `direction(i) = 0` if all zeros are to be computed (the default), +1 if only zeros where the event function is increasing, and -1 if only zeros where the event function is decreasing. Output `tsol` is a column vector of times specified in `tspan`, prior to first terminal event. `SOL(j,:,:)` is the solution at `T(j)`. `TE` is a vector of times at which events occur. `SOLE(j,:,:)` is the solution at `TE(j)` and indices in vector `IE` specify which event occurred.

If `UI = SOL(j,:,i)` approximates component i of the solution at time `TSPAN(j)` and mesh points `XMESH`, `pdeval` evaluates the approximation and its partial derivative $\partial u_i / \partial x$ at the array of points `XOUT` and returns them in `UOUT` and `DUOUTDX: [UOUT,DUOUTDX] = PDEVAL(M,XMESH,UI,XOUT)`

Note The partial derivative $\partial u_i / \partial x$ is evaluated here rather than the flux. The flux is continuous, but at a material interface the partial derivative may have a jump.

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 $0 \le x \le 1$ for times $t \ge 0$.

The PDE satisfies the initial condition

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

and boundary conditions

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

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

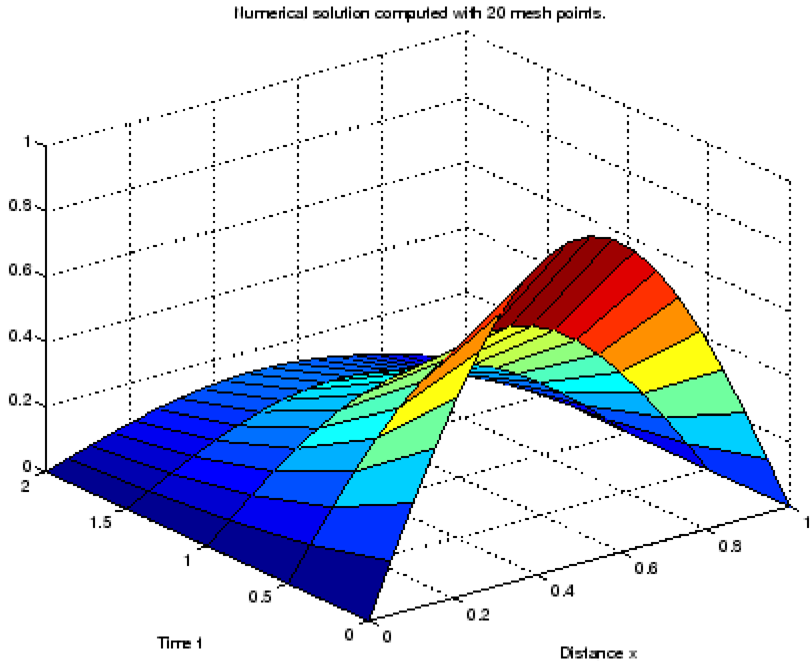
It is convenient to use subfunctions to place all the functions required by `pdepe` in a single M-file.

```
function pdex1
```

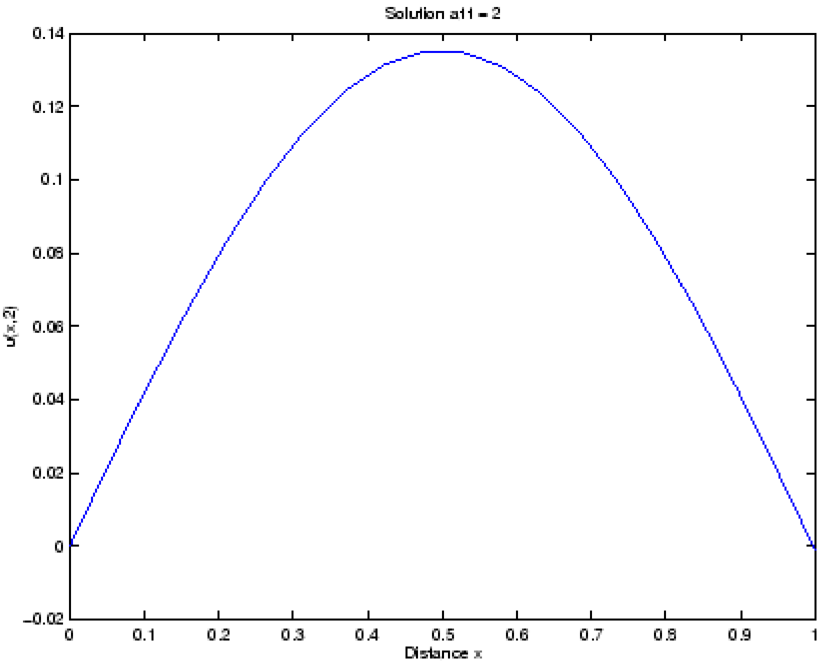
```
m = 0;
x = linspace(0,1,20);
t = linspace(0,2,5);

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 $0 \le x \le 1$ for times $t \ge 0$.

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

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

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

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

and the right boundary condition is

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

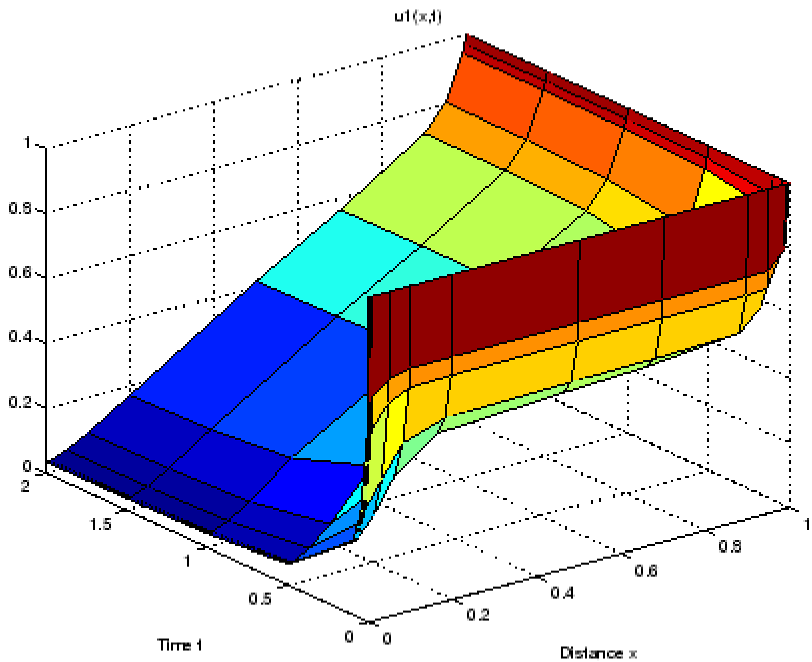
The solution changes rapidly for small t. The program selects the step size in time to resolve this sharp change, but to see this behavior in the plots, the example must select the output times accordingly. There are boundary layers in the solution at both ends of [0,1], so the example places mesh points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

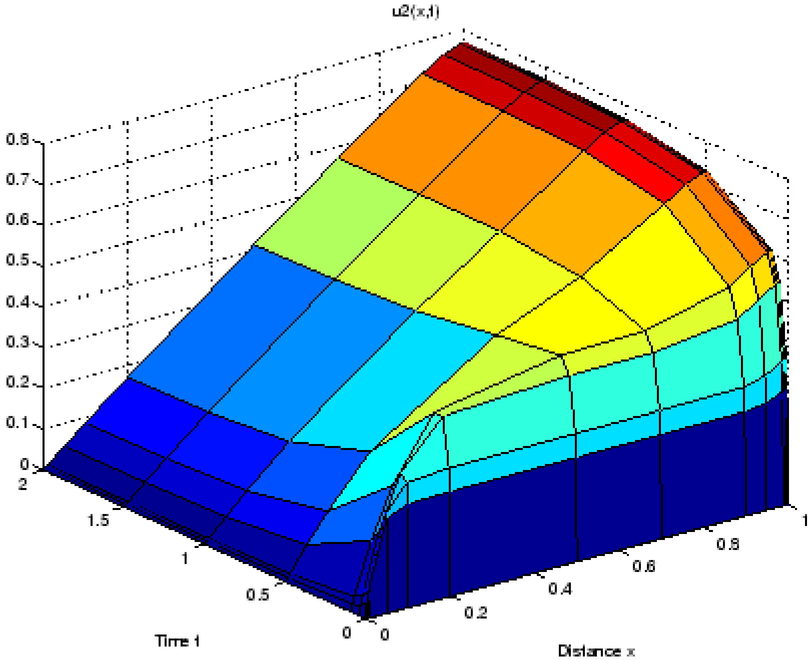
```
function pdex4
m = 0;
x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];

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,ql,pr,qr] = pdex4bc(xl,ul,xr,ur,t)
pl = [0; ul(2)];
ql = [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

```
[uout,duoutdx] = pdeval(m,x,ui,xout)
```

Arguments

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

Description

`[uout,duoutdx] = pdeval(m,x,ui,xout)` approximates the solution u_i and its partial derivative $\partial u_i / \partial x$ 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 $\partial u_i / \partial x$ rather than the flux f. Although the flux is continuous, the partial derivative may have a jump at a material interface.

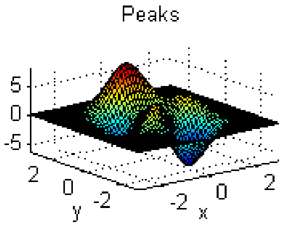
See Also

`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`

Purpose

Call Perl script using appropriate operating system executable

Syntax

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

Description

`perl('perlfile')` calls the Perl script `perlfile`, using the appropriate operating system Perl executable. Perl is included with the MATLAB software on Microsoft Windows systems, and thus MATLAB users can run M-files containing the `perl` function. On UNIX [16] systems, MATLAB calls the Perl interpreter available with the operating system.

`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`.

`[result, status] = perl(...)` returns the results of attempted Perl call to `result` and its exit status to `status`.

It is sometimes beneficial to use Perl scripts instead of MATLAB code. The `perl` function allows you to run those scripts from 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

Examples

Given the Perl script, `hello.pl`:

```
$input = $ARGV[0];
```

16. UNIX is a registered trademark of The Open Group in the United States and other countries.

```
print "Hello $input.";
```

At the MATLAB command line, type:

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

MATLAB displays:

```
ans =
Hello World.
```

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(2:4:6)` 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`, `shiftdim`, `reshape`

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 the MATLAB software 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. MATLAB also errors if you declare any of a function's input or output arguments as persistent within that same function. For example, the following persistent declaration is invalid:

```
function myfun(argA, argB, argC)
persistent argB
```

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`

Purpose

Ratio of circle's circumference to its diameter

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

Find the sine of π:

```
sin(pi)
```

returns

```
ans =

    1.2246e-16
```

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

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 the handle `axes_handle` instead of into the current axes (`gca`).

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

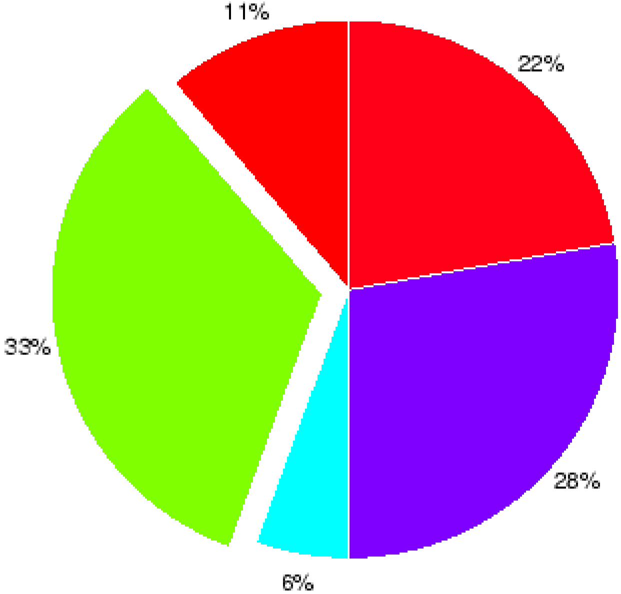
Remarks

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

Examples

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

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



See Also

`pie3`

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 the handle `axes_handle` instead of into the current axes (`gca`).

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

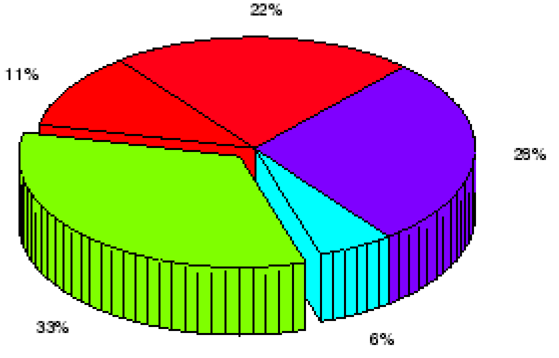
Remarks

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

Examples

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

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



See Also

`pie`

pinv

Purpose

Moore-Penrose pseudoinverse of matrix

Syntax

```
B = pinv(A)
B = pinv(A,tol)
```

Definition

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

```
A*B*A = A
B*A*B = B
A*B is Hermitian
B*A is Hermitian
```

The computation is based on `svd(A)` and any singular values less than `tol` are treated as zero.

Description

`B = pinv(A)` returns the Moore-Penrose pseudoinverse of `A`.

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

Examples

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

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

```
minimize norm(A*x-b)
```

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

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

and

```
y = A\b
```

These two are distinguished by the facts that 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 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*ones(8,1),

```
b =
     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*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 `norm(A*x-b)` and `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`

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`

playshow

Purpose Run M-file demo (deprecated; use `echodemo` instead)

Syntax `playshow filename`

Description `playshow filename` runs `filename`, which is a demo. Replace `playshow` filename with `echodemo` filename. Note that other arguments supported by `playshow` are not supported by `echodemo`.

See Also `demo`, `echodemo`, `helpbrowser`

Purpose

2-D line plot

Syntax

```
plot(Y)
plot(X1,Y1,...,Xn,Yn)
plot(X1,Y1,LineSpec,...,Xn,Yn,LineSpec)
plot(X1,Y1,LineSpec,'PropertyName',PropertyValue)
plot(axes_handle,X1,Y1,LineSpec,'PropertyName',PropertyValue)
h = plot(X1,Y1,LineSpec,'PropertyName',PropertyValue)
```

Description

`plot(Y)` plots the columns of `Y` versus the index of each value when `Y` is a real number. For complex `Y`, `plot(Y)` is equivalent to `plot(real(Y),imag(Y))`.

`plot(X1,Y1,...,Xn,Yn)` plots each vector `Yn` versus vector `Xn` on the same axes. If one of `Yn` or `Xn` is a matrix and the other is a vector, plots the vector versus the matrix row or column with a matching dimension to the vector. If `Xn` is a scalar and `Yn` is a vector, plots discrete `Yn` points vertically at `Xn`. If `Xn` or `Yn` are complex, imaginary components are ignored. `plot` automatically chooses colors and line styles in the order specified by `ColorOrder` and `LineStyleOrder` properties of current axes.

`plot(X1,Y1,LineSpec,...,Xn,Yn,LineSpec)` plots lines defined by the `Xn,Yn,LineSpec` triplets, where `LineSpec` specifies the line type, marker symbol, and color. You can mix `Xn,Yn,LineSpec` triplets with `Xn,Yn` pairs: `plot(X1,Y1,X2,Y2,LineSpec,X3,Y3)`.

`plot(X1,Y1,LineSpec,'`*`PropertyName`*`',PropertyValue)` manipulates plot characteristics by setting `lineseries properties` (of `lineseries` graphics objects created by `plot`). Enter properties as one or more name and value pairs.

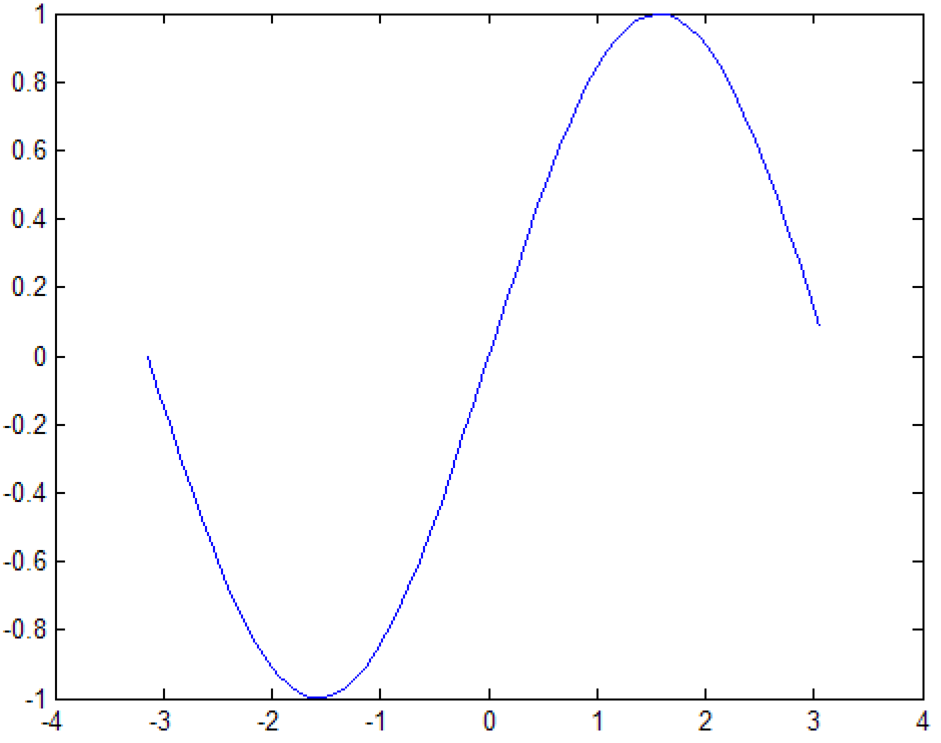
`plot(axes_handle,X1,Y1,LineSpec,'`*`PropertyName`*`',PropertyValue)` plots using axes with the handle `axes_handle` instead of the current axes (`gca`).

`h = plot(X1,Y1,LineSpec,'`*`PropertyName`*`',PropertyValue)` returns a column vector of handles to `lineseries` objects (instead of `lineseries` objects), one handle per line.

Examples

Plot a sine curve:

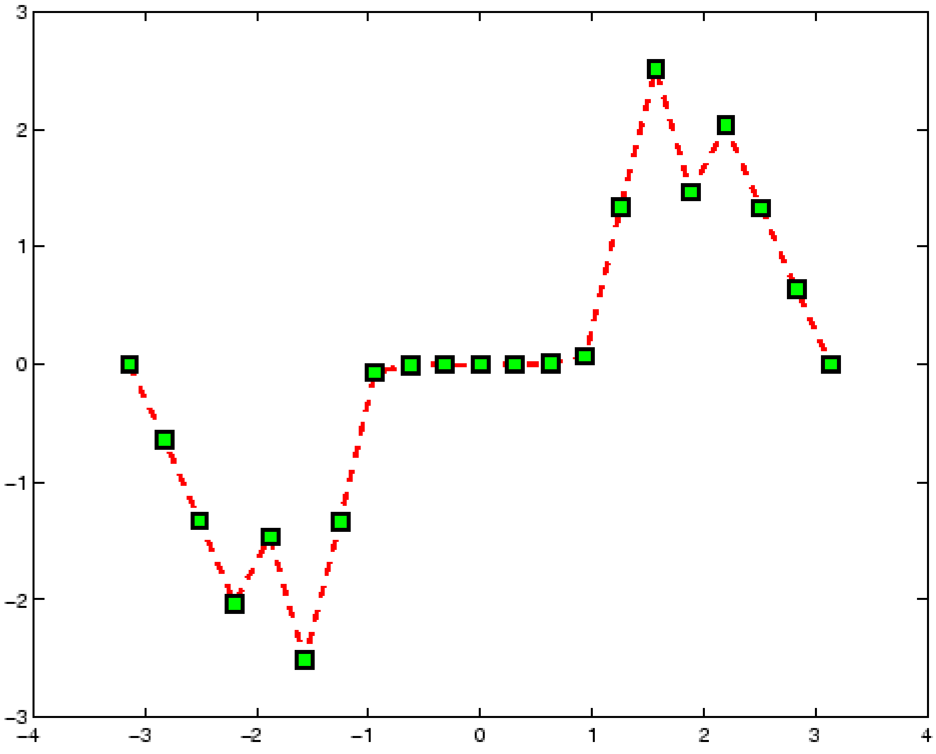
```
x = -pi:.1:pi;
y = sin(x);
plot(x,y)
```



Create line plot using specific line width, marker color, and marker size:

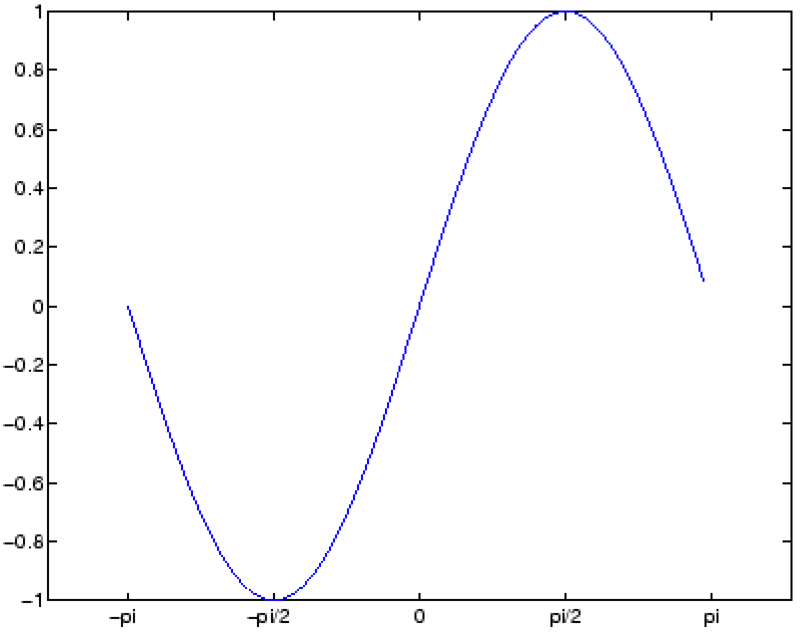
```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'--rs','LineWidth',2,...
                'MarkerEdgeColor','k',...
```

```
'MarkerFaceColor','g',...
'MarkerSize',10)
```



Modify axis tick marks and tick labels:

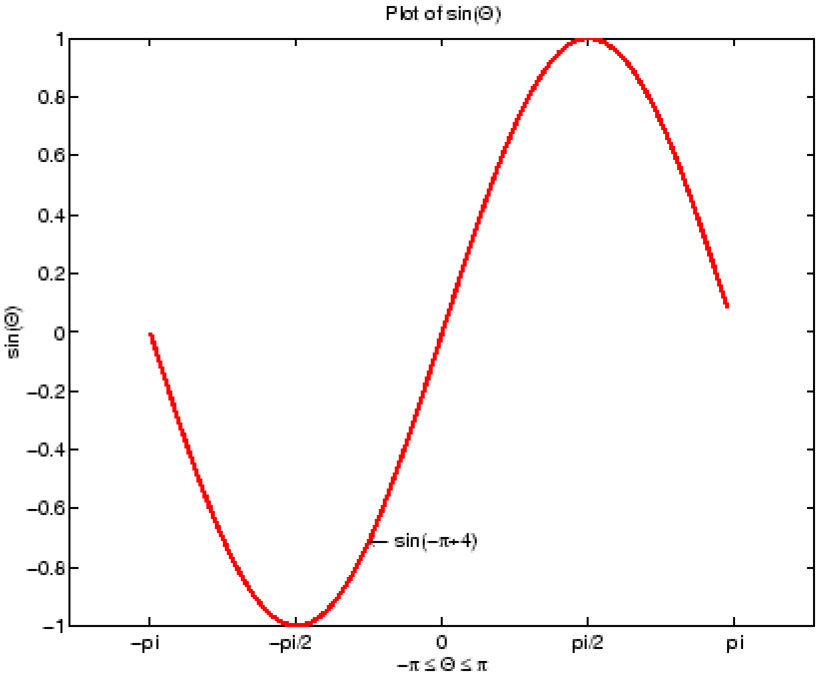
```
x = -pi:.1:pi;
y = sin(x);
plot(x,y)
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```



Add a plot title, axis labels, and annotations:

```
x = -pi:.1:pi;
y = sin(x);
p = plot(x,y)
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
xlabel('-\pi \leq \Theta \leq \pi')
ylabel('sin(\Theta)')
title('Plot of sin(\Theta)')
% \Theta appears as a Greek symbol (see String)
% Annotate the point (-pi/4, sin(-pi/4))
text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi\div4)',...
     'HorizontalAlignment','left')
% Change the line color to red and
% set the line width to 2 points
```

```
set(p,'Color','red','LineWidth',2)
```



Plot multiple line plots on the same axes:

```
plot(rand(12,1))
% hold axes and all lineseries properties, such as
% ColorOrder and LineStyleOrder, for the next plot
hold all
plot(randn(12,1))
```

Set line color to be always black and line style order to cycle through solid, dash-dot, dash-dash, and dotted line styles:

```
set(0,'DefaultAxesColorOrder',[0 0 0],...
      'DefaultAxesLineStyleOrder','-|-.|--|:')
plot(rand(12,1))
hold all
plot(rand(12,1))
hold all
plot(rand(12,1))
```

Alternatives

To plot variables in the MATLAB workspace:

1 In the MATLAB workspace browser, select one or more variables.

2 Choose the plot type from the 

menu.

See Also

`axis` | `axes` | `bar` | `gca` | `grid` | `hold` | `legend` | `line` | `lineseries properties` | `LineSpec` | `LineWidth` | `loglog` | `MarkerEdgeColor` | `MarkerFaceColor` | `MarkerSize` | `plot3` | `plotyy` | `semilogx` | `semilogy` | `subplot` | `title` | `xlabel` | `xlim` | `ylabel` | `ylim`

How To

- Editing Plot Characteristics
- Creating Line Plots
- Annotating Graphs
- Creating Graphics from the Workspace Browser
-

Purpose

Plot time series

Syntax

```
plot(ts)
plot(tsc.tsname)
plot(function)
```

Description

`plot(ts)` plots the time-series data `ts` against time and interpolates values between samples by using either zero-order-hold (`'zoh'`) or linear interpolation.

`plot(tsc.tsname)` plots the `timeseries` object `tsname` that is part of the `tscollection` `tsc`.

`plot(function)` 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.

Remarks

Time-series events, when defined, are marked in the plot by a red circular marker.

When you resize a timeseries plot to be narrower, the *x*-axis ticks and labels are readjusted so that they do not overlap one another. Unlike plots of regular variables, this behavior cannot be overridden (for example, by setting the axes `XTickMode` to `'Auto'`).

Examples

`plot(ts,'-r*')` uses a regular line with the color red and marker `'*'` to render the plot.

`plot(ts,'ko','MarkerSize',3)` uses black circular markers of size `3` to render the plot.

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,LineSpec,...)
plot3(...,'PropertyName',PropertyValue,...)
h = plot3(...)
```

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

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

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

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

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

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

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

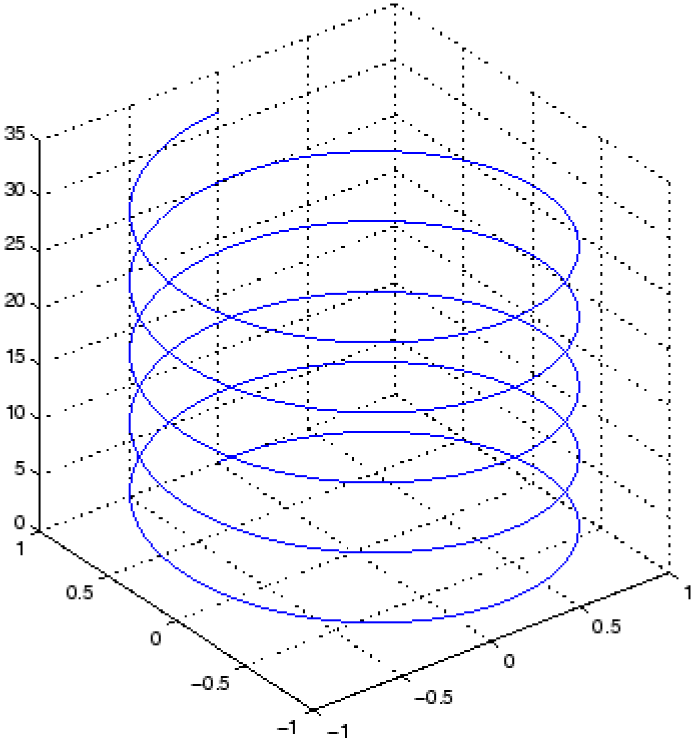
```
plot3(X1,Y1,Z1,X2,Y2,Z2,LineSpec,X3,Y3,Z3)
```

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

Examples

Plot a three-dimensional helix.

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



See Also

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

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 in the MATLAB Graphics documentation.

Syntax

```
plotbrowser('on')
plotbrowser('off')
plotbrowser('toggle')
plotbrowser
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`

plotedit

Purpose

Interactively edit and annotate plots

Syntax

```
plotedit on
plotedit off
plotedit
plotedit(h)
plotedit('state')
plotedit(h,'state')
```

Description

`plotedit on` starts plot edit mode for the current figure, allowing you to use a graphical interface to annotate and edit plots easily. In plot edit mode, you can label axes, change line styles, and add text, line, and arrow annotations.

`plotedit off` ends plot mode for the current figure.

`plotedit` toggles the plot edit mode for the current figure.

`plotedit(h)` toggles the plot edit mode for the figure specified by figure handle `h`.

`plotedit('state')` specifies the `plotedit` state for the current figure. Values for `state` can be as shown.

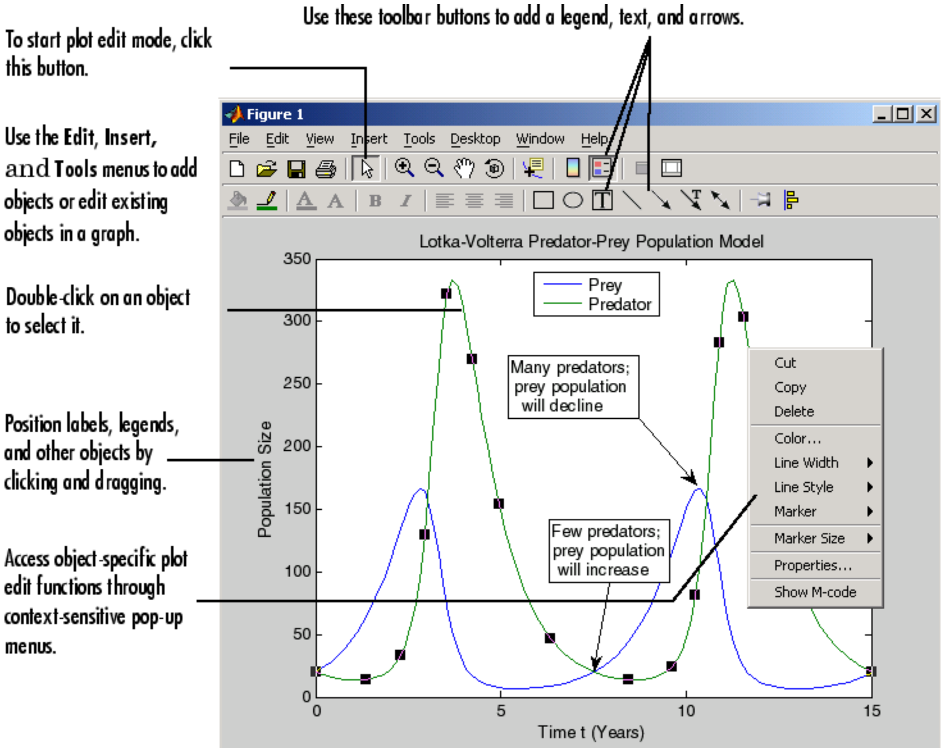
Value for state	Description
`on`	Starts plot edit mode
`off`	Ends plot edit mode
`showtoolsmenu`	Displays the **Tools** menu in the menu bar
`hidetoolsmenu`	Removes the **Tools** menu from the menu bar

Note `hidetoolsmenu` is intended for GUI developers who do not want the **Tools** menu to appear in applications that use the figure window.

`plotedit(h,'state')` specifies the `plotedit state` for figure handle `h`.

Remarks

Plot Editing Mode Graphical Interface Components



Examples

Start plot edit mode for figure 2.

```
plotedit(2)
```

End plot edit mode for figure 2.

```
plotedit(2, 'off')
```

Hide the **Tools** menu for the current figure:

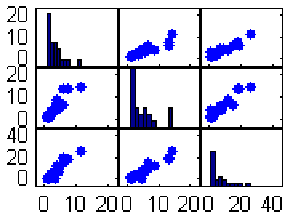
```
plotedit('hidetoolsmenu')
```

See Also

`axes`, `line`, `open`, `plot`, `print`, `saveas`, `text`, `propedit`

Purpose

Scatter plot matrix



Syntax

```
plotmatrix(X,Y)
plotmatrix(X)
plotmatrix(...,'LineSpec')
[H,AX,BigAx,P] = plotmatrix(...)
```

Description

`plotmatrix(X,Y)` scatter plots the columns of `X` against the columns of `Y`. If `X` is p-by-m and `Y` is p-by-n, `plotmatrix` produces an n-by-m matrix of axes.

`plotmatrix(X)` is the same as `plotmatrix(X,X)`, except that the diagonal is replaced by `hist(X(:,i))`.

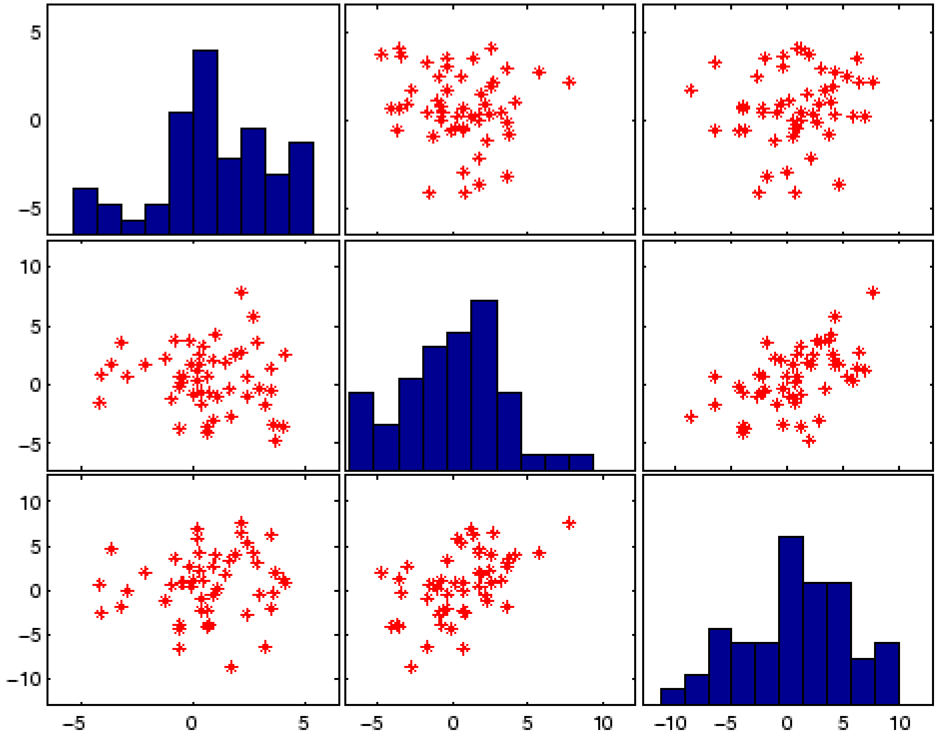
`plotmatrix(...,'LineSpec')` uses a `LineSpec` to create the scatter plot. The default is `'.'`.

`[H,AX,BigAx,P] = plotmatrix(...)` returns a matrix of handles to the objects created in `H`, a matrix of handles to the individual subaxes in `AX`, a handle to a big (invisible) axes that frames the subaxes in `BigAx`, and a matrix of handles for the histogram plots in `P`. `BigAx` is left as the current axes so that a subsequent `title`, `xlabel`, or `ylabel` command is centered with respect to the matrix of axes.

Examples

Generate plots of random data.

```
x = randn(50,3); y = x*[-1 2 1;2 0 1;1 -2 3;]';
plotmatrix(y,'*r')
```

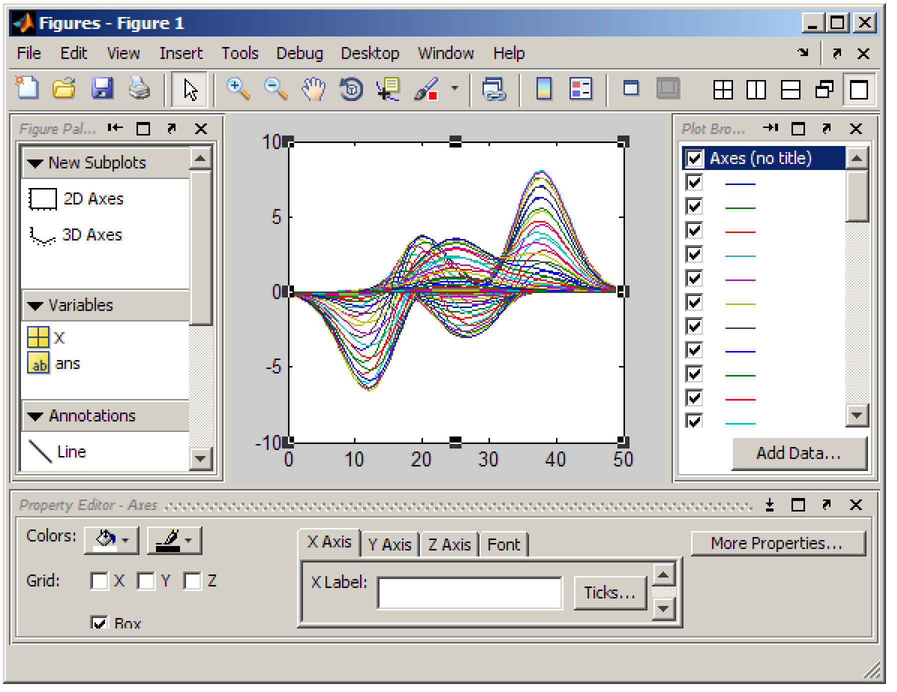


See Also

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 in the MATLAB Graphics documentation.

Syntax

```
plottools('on')
plottools('off')
plottools
plottools(figure_handle,...)
plottools(...,'tool')
```

plottools

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 on the current figure.

`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 restores 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 in the MATLAB Graphics documentation and 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:

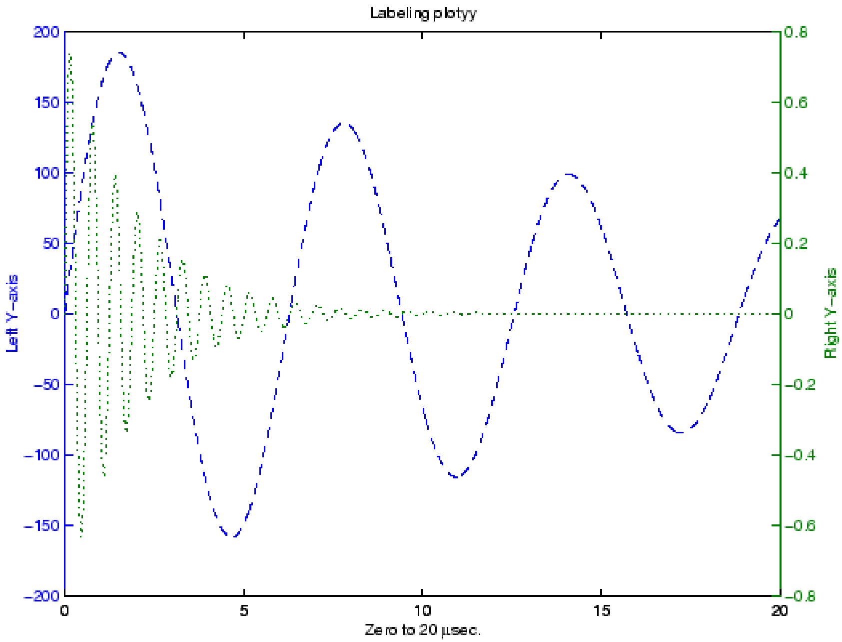
```
set(get(AX(1),'Ylabel'),'String','Slow Decay')
set(get(AX(2),'Ylabel'),'String','Fast Decay')
```

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`, `linkaxes`, `linkprop`, `loglog`, `semilogx`, `semilogy`, `XAxisLocation`, `YAxisLocation`

See for more information.

DelaunayTri.pointLocation

Purpose

Simplex containing specified location

Syntax

```
SI = pointLocation(DT,QX)
SI = pointLocation(DT,QX,QY)
SI = pointLocation(DT,QX,QY,QZ)
[SI, BC] = pointLocation(DT,...)
```

Description

`SI = pointLocation(DT,QX)` returns the indices `SI` of the enclosing simplex (triangle/tetrahedron) for each query point location in `QX`. The enclosing simplex for point `QX(k,:)` is `SI(k)`. `pointLocation` returns `NaN` for all points outside the convex hull.

`SI = pointLocation(DT,QX,QY)` and `SI = pointLocation(DT,QX,QY,QZ)` allow the query point locations to be specified in alternative column vector format when working in 2-D and 3-D.

`[SI, BC] = pointLocation(DT,...)` returns the barycentric coordinates `BC`.

Inputs

`DT`	Delaunay triangulation.
`QX`	Matrix of size `mpts`-by-`ndim`, `mpts` being the number of query points.

Outputs

`SI`	Column vector of length `mpts` containing the indices of the enclosing simplex for each query point. `mpts` is the number of query points.
`BC`	`BC` is a `mpts`-by-`ndim` matrix, each row `BC(i,:)` represents the barycentric coordinates of `QX(i,:)` with respect to the enclosing simplex `SI(i)`.

Examples

Example 1

Create a 2-D Delaunay triangulation:

```
X = rand(10,2);
dt = DelaunayTri(X);
```

Find the triangles that contain specified query points:

```
qrypts = [0.25 0.25; 0.5 0.5];
triids = pointLocation(dt, qrypts)
```

Example 2

Create a 3-D Delaunay triangulation:

```
x = rand(10,1);
y = rand(10,1);
z = rand(10,1);
dt = DelaunayTri(x,y,z);
```

Find the triangles that contain specified query points and evaluate the barycentric coordinates:

```
qrypts = [0.25 0.25 0.25; 0.5 0.5 0.5];
[tetids, bcs] = pointLocation(dt, qrypts)
```

See Also

`nearestNeighbor`

pol2cart

Purpose Transform polar or cylindrical coordinates to Cartesian

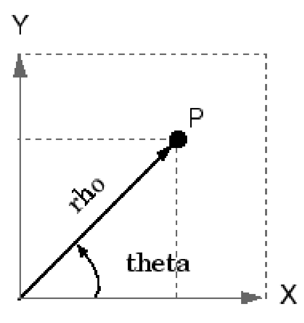
Syntax

```
[X,Y] = pol2cart(THETA,RHO)
[X,Y,Z] = pol2cart(THETA,RHO,Z)
```

Description `[X,Y] = pol2cart(THETA,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.

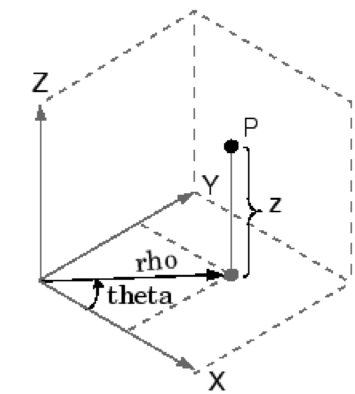
xyz, `[X,Y,Z] = pol2cart(THETA,RHO,Z)` transforms the cylindrical coordinate data stored in corresponding elements of `THETA`, `RHO`, and `Z` to three-dimensional Cartesian, or coordinates. The arrays `THETA`, `RHO`, and `Z` must be the same size (or any can be scalar). The values in `THETA` must be in radians.

Algorithm The mapping from polar and cylindrical coordinates to Cartesian coordinates is:



Polar to Cartesian Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
```



Cylindrical to Cartesian Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
z = z
```

See Also

`cart2pol`, `cart2sph`, `sph2cart`

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,LineSpec)
polar(axes_handle,...)
h = polar(...)
```

Description

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

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

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

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

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

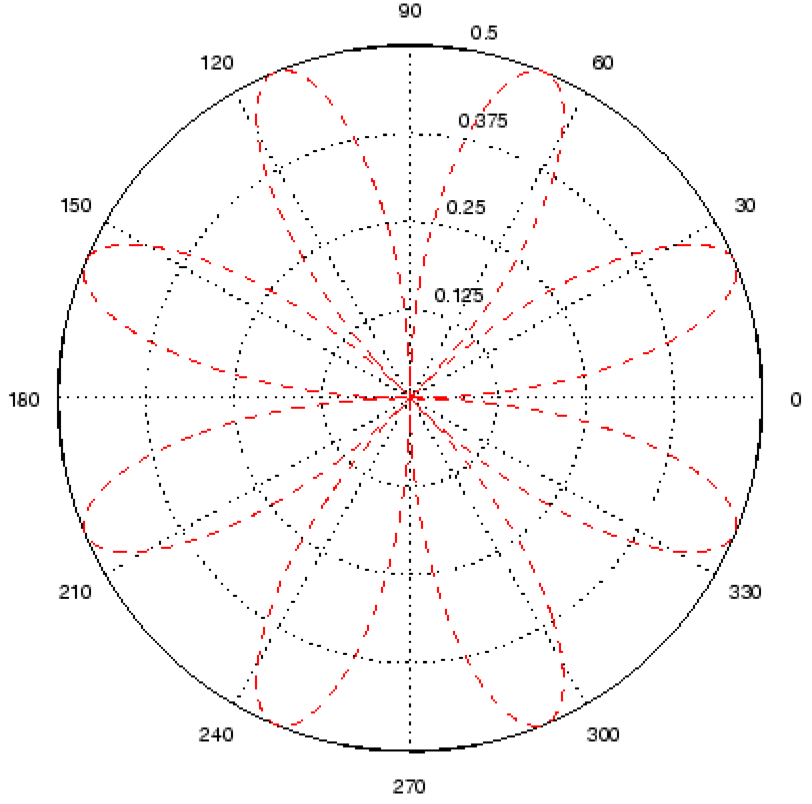
Remarks

Negative `r` values reflect through the origin, rotating by `pi` (since `(theta,r)` transforms to `(r*cos(theta), r*sin(theta))`). If you want different behavior, you can manipulate `r` prior to plotting. For example, you can make `r` equal to `max(0,r)` or `abs(r)`.

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 $c_1 s^n + \ldots + c_n s + c_{n+1}$

`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

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

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.

$$(\lambda - \lambda_1)(\lambda - \lambda_2)\ldots(\lambda - \lambda_n)$$

It is possible to prove that `poly(A)` produces the coefficients in the characteristic polynomial of a matrix within roundoff error of `A`. This is true even if the eigenvalues of `A` are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial, which do not use the eigenvalues, do not have such satisfactory numerical properties.

See Also

`conv`, `polyval`, `residue`, `roots`

Purpose

Area of polygon

Syntax

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

Description

`A = polyarea(X,Y)` returns the area of the polygon specified by the vertices in the vectors `X` and `Y`.

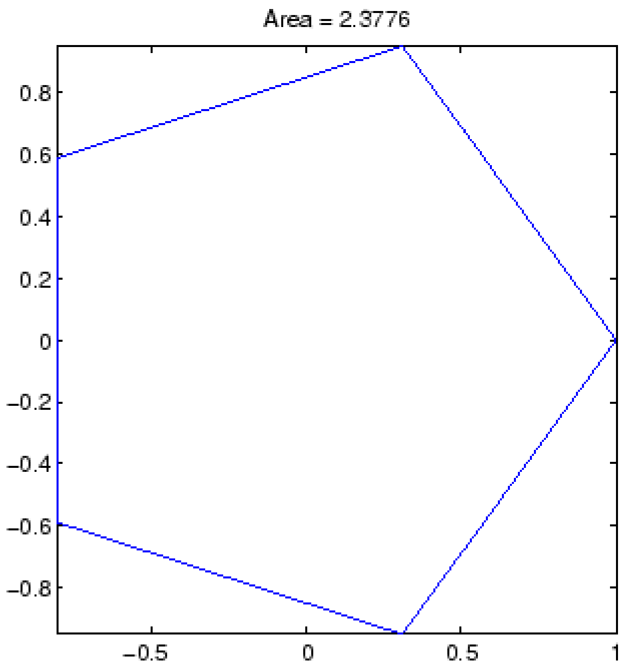
If `X` and `Y` are matrices of the same size, then `polyarea` returns the area of polygons defined by the columns `X` and `Y`.

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

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

Examples

```
L = linspace(0,2.*pi,6); xv = cos(L)';yv = sin(L)';
xv = [xv ; xv(1)]; yv = [yv ; yv(1)];
A = polyarea(xv,yv);
plot(xv,yv); title(['Area = ' num2str(A)]); axis image
```



See Also `convhull`, `inpolygon`, `rectint`

Purpose

Polynomial derivative

Syntax

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

Description

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

`k = polyder(p)` returns the derivative of the polynomial `p`.

`k = polyder(a,b)` returns the derivative of the product of the polynomials `a` and `b`.

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

Examples

The derivative of the product

$$(3x^2 + 6x + 9)(x^2 + 2x)$$

is obtained with

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

This result represents the polynomial

$$12x^3 + 36x^2 + 42x + 18$$

See Also

`conv`, `deconv`

polyeig

Purpose

Polynomial eigenvalue problem

Syntax

```
[X,e] = polyeig(A0,A1,...Ap)
e = polyeig(A0,A1,..,Ap)
[X, e, s] = polyeig(A0,A1,..,AP)
```

Description

`[X,e] = polyeig(A0,A1,...Ap)` 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 `A0,A1,...Ap` are input matrices of order `n`. The output consists of a matrix `X` of size `n`-by-`n*p` whose columns are the eigenvectors, and a vector `e` of length `n*p` containing the eigenvalues.

If `lambda` is the `j`th eigenvalue in `e`, and `x` is the `j`th column of eigenvectors in `X`, then `(A0 + lambda*A1 + ... + lambda^p*Ap)*x` is approximately `0`.

`e = polyeig(A0,A1,..,Ap)` is a vector of length `n*p` whose elements are the eigenvalues of the polynomial eigenvalue problem.

`[X, e, s] = polyeig(A0,A1,..,AP)` also returns a vector `s` of length `p*n` containing condition numbers for the eigenvalues. At least one of `A0` and `AP` 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 `a0`, `a1` ..., `ap` is the standard polynomial problem: `roots([ap ... a1 a0])`.

If both `A0` and `Ap` 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 `A0` and `Ap` is singular, the problem is well posed, but some of the eigenvalues might be zero or infinite.

Note that scaling `A0,A1,..,Ap` to have `norm(Ai)` 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 Francoise Tisseur, “Perturbation theory for homogeneous polynomial eigenvalue problems,” *Linear Algebra Appl.*, Vol. 358, pp. 71-94, 2003.

[2] Tisseur, Francoise and Karl Meerbergen, “The quadratic eigenvalue problem,” *SIAM Rev.*, Vol. 43, Number 2, pp. 235-286, 2001.

[3] Francoise 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

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

`[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 $\mu_1 = \mathrm{mean}(x)$ and $\mu_2 = \mathrm{std}(x)$. `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, `erf(x)`, by a polynomial in `x`. This is a risky project because `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 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);
```

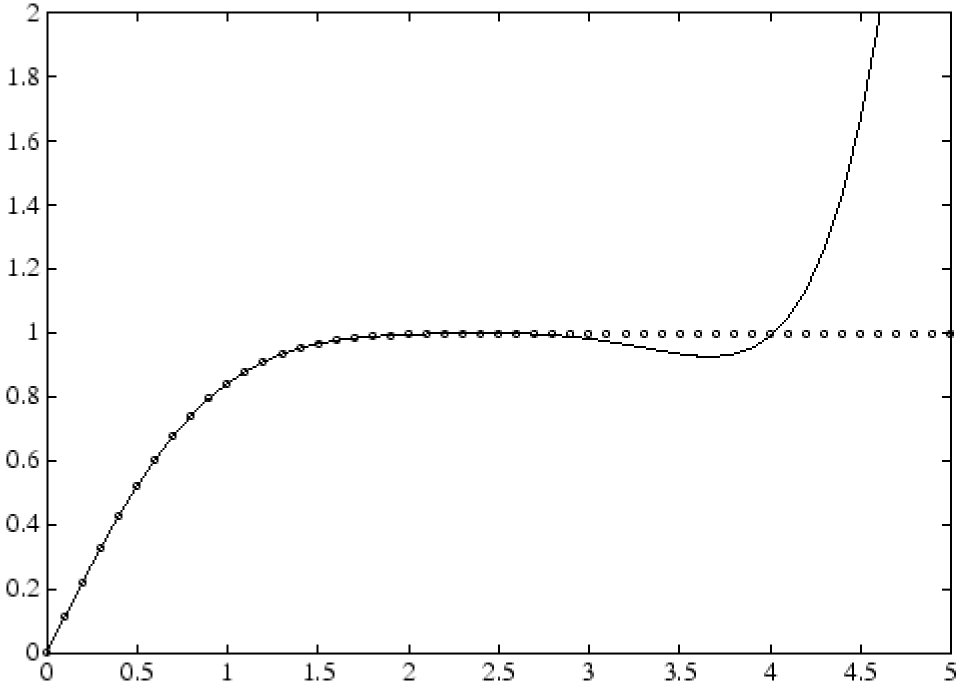
A table showing the data, fit, and error is

```
 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`, `lscov`

polyint

Purpose Integrate polynomial analytically

Syntax

```
polyint(p,k)
polyint(p)
```

Description

`polyint(p,k)` returns a polynomial representing the integral of polynomial `p`, using a scalar constant of integration `k`.

`polyint(p)` assumes a constant of integration `k=0`.

See Also `polyder`, `polyval`, `polyvalm`, `polyfit`

Purpose

Polynomial evaluation

Syntax

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

Description

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

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

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

`[y,delta] = polyval(p,x,S)` uses the optional output structure `S` generated by `polyfit` to generate error estimates `delta`. `delta` is an estimate of the standard deviation of the error in predicting a future observation at `x` by `p(x)`. If the coefficients in `p` are least squares estimates computed by `polyfit`, and the errors in the data input to `polyfit` are independent, normal, and have constant variance, then `y±delta` contains at least 50% of the predictions of future observations at `x`.

`y = polyval(p,x,[],mu)` or `[y,delta] = polyval(p,x,S,mu)` use $\hat{x} = (x - \mu_1)/\mu_2$ in place of `x`. In this equation, $\mu_1 = \mathrm{mean}(x)$ and $\mu_2 = \mathrm{std}(x)$. The centering and scaling parameters $\mathrm{mu} = [\mu_1, \mu_2]$ are optional output computed by `polyfit`.

Remarks

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

Examples

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

```
p = [3 2 1];
polyval(p,[5 7 9])
```

which results in

```
ans =

    86   162   262
```

For another example, see `polyfit`.

See Also

`polyfit`, `polyvalm`, `polyder`, `polyint`

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
```

```
      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 `.^`

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 = power(A,B)` is called for the syntax `'A .^ B'` when `A` or `B` is an object.

Note that for a negative value `X` and a non-integer value `Y`, if the `abs(Y)` is less than one, the `power` function returns the complex roots. To obtain the remaining real roots, use the `nthroot` function.

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,..,Dr]`-valued, and `xx` is a vector of length `N`, then `V` has size `[D1,...,Dr, N]`, with `V(:,...,:,J)` the value of `f` at `xx(J)`.

If `f` is `[D1,..,Dr]`-valued, and `xx` has size `[N1,...,Ns]`, then `V` has size `[D1,...,Dr, N1,...,Ns]`, with `V(:,...,:, J1,...,Js)` the value of `f` at `xx(J1,...,Js)`.

If `pp` was constructed by `interp1` using the orienatation of non-scalar function values specified for that function, then:

If `f` is `[D1,..,Dr]`-valued, and `xx` is a vector of length `N`, then `V` has size `[N,D1,...,Dr]`, with `V(J,:,...,:)` the value of `f` at `xx(J)`.

If `f` is `[D1,..,Dr]`-valued, and `xx` has size `[N1,...,Ns]`, then `V` has size `[N1,...,Ns,D1,...,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

Folder containing preferences, history, and layout files

Syntax

```
prefdir
f = prefdir
f = prefdir(1)
```

Description

`prefdir` returns the folder 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

`f = prefdir` assigns to `f` the name of the folder containing preferences and related files.

`f = prefdir(1)` creates a folder for preferences and related files if one does not exist. If the folder does exist, the name is assigned to `f`.

Remarks

You must have write access to the preferences folder, or MATLAB generates an error in the Command Window when you try to change preferences.

The folder might be a hidden folder, for example, `myname/.matlab/R2009a`. For more information, see .

The preferences folder MATLAB uses and how preferences migrate when you use a new version of MATLAB depend on the version. In R14SP3, there was a change to the way that the preference folders were named and how they migrated, affecting R13 through R14SP2. The differences are relevant if you run multiple versions of MATLAB and one version is prior to R14SP3:

- For R2009b back through and including R2006a, and R14SP3, MATLAB uses the name of the release for the preference folder. For example, `R2009b`, `R2009a`, ... through `R14SP3`. When you install R2009b, MATLAB migrates the files in the `R2009a` preferences folder to the `R2009b` preferences folder. While running R2009b through R14SP3, any changes made to files in those preferences folders (`R2009b` through `R14SP3`) are used only in their respective versions. As an example, commands you run in R2009b will *not* appear in the Command History when you run R2009a, and so on. The converse is also true.

  Upon startup, MATLAB 7.9 (R2009b) looks for, and if found uses, the `R2009b` preferences folder. If not found, MATLAB creates an `R2009b` preferences folder. This happens when the R2009b preferences folder is deleted or does not exist for some other reason. MATLAB then looks for the `R2009a` preferences folder, and if found, migrates the R2009a preferences to the R2009b preferences. If it does not find the `R2009a` preferences folder, it uses the default preferences for R2009b. This process also applies when starting MATLAB 7.8 (R2009a) through 7.1 (R14SP3).

  If you want to use default preferences for R2009b, and do not want MATLAB to migrate preferences from R2009a, the `R2009b` preferences folder *must exist but be empty* when you start MATLAB. If you want to maintain some of your R2009b customizations, but restore the defaults for others, in the `R2009b` preferences folder, 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 in the MATLAB Desktop Tools and Development Environment documentation.

- The R14 through R14SP2 releases all share the `R14` preferences folder. While running R14SP1, for example, any changes made to files in the preferences folder, `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 or later versions because those versions each use their own preferences folders.

- All R13 releases use the R13 preferences folder. While running R13SP1, for example, any changes made to files in the preferences folder, 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 folders, and the converse is true.

Examples

View the location of the preferences folder:

```
prefdir
```

MATLAB returns:

```
ans =

C:\WINNT\Profiles\my_user_name\MATHWORKS\Application Data\MathWorks\MATLAB\R2009a
```

Run dir for the folder to see the files for customizing MathWorks products:

```
.                   history.m
..                  matlab.prf
cwdhistory.m        MATLABDesktop.xml
shortcuts.xml       MATLAB EditorDesktop.xml
...
```

In MATLAB, run cd(prefdir) to make the preferences folder become the current folder.

On Windows platforms, go directly to the preferences folder in Microsoft Windows Explorer by running winopen(prefdir).

See Also

preferences, winopen

in the MATLAB Desktop Tools and Development Environment documentation

Purpose Open Preferences dialog box

GUI Alternatives As an alternative to the `preferences` function, select **File > Preferences** in the MATLAB desktop or any desktop tool.

Syntax

```
preferences
```

Description `preferences` displays the Preferences dialog box, from which you can make changes to options for MATLAB and related products.

See Also `prefdir`

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

Contents

GUI Alternative

Select **File > Print** from the figure window to open the Print dialog box and **File > Print Preview** to open the Print Preview GUI. For details, see "How to Print or Export" in the MATLAB Graphics documentation.

Syntax

```
print
print filename
print -ddriver
print -dformat
print -dformat filename
print -smodelname
print -options
print(...)
[pcmd,dev] = printopt
```

Description

`print` and `printopt` produce hard-copy output. All arguments to the `print` command are optional. You can use them in any combination or order.

`print` sends the contents of the current figure, including bitmap representations of any user interface controls, to the printer using the device and system printing command defined by `printopt`.

`print filename` directs the output to the PostScript file designated by `filename`. If `filename` does not include an extension, `print` appends an appropriate extension.

`print -ddriver` prints the figure using the specified printer *`driver`*, (such as color PostScript). If you omit `-ddriver`, `print` uses the default value stored in `printopt.m`. The table in "Printer Drivers" on page 2-2871 lists all supported device types.

`print -dformat` copies the figure to the system Clipboard (Microsoft Windows platforms only). To be valid, the *`format`* for this operation must be either `-dmeta` (Windows Enhanced Metafile) or `-dbitmap` (Windows Bitmap).

`print -dformat filename` exports the figure to the specified file using the specified graphics *`format`* (such as TIFF). The table of "Graphics Format Files" on page 2-2875 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.) "Printing Options" on page 2-2879 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 file names and handles. See "Batch Processing" on page 2-2889 for an example. Also see "Specifying the Figure to Print" on page 2-2886 for further examples.

`[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 hard-copy 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	Print Command	Driver or Format
Mac and UNIX	`lpr -r`	`-dps2`
Windows	COPY /B %s LPT1:	`-dwin`

Printer Drivers

The following table shows the more widely used printer drivers supported by MATLAB software. If you do not specify a driver, the default setting shown in the previous table is used. For a list of all supported printer drivers, type `print -d` at the MATLAB prompt. Some things to remember:

- As indicated in "Description" on page 2-2869 the `-d` switch specifies a printer driver or a graphics file format:
  - Specifying a printer driver without a file name or printer name (the `-P` option) sends the output formatted by the specified driver to your default printer, which may not be what you want to do.

    Note On Windows systems, when you use the `-P` option to identify a printer to use, if you specify any driver other than `-dwin` or `-dwinc`, MATLAB writes the output to a file with an appropriate extension but does not send it to the printer. You can then copy that file to a printer.

  - Specifying a `-dmeta` or a `-dbitmap` graphics format without a file name places the graphic on the system Clipboard, if possible (Windows platforms only).

- Specifying any other graphics format without a file name creates a file in the current folder with a name such as `figureN.`*`fmt`*, where `N` is 1, 2, 3, ... and *`fmt`* indicates the format type, for example, `eps` or `png`.

- Several drivers come from a product called Ghostscript, which is shipped with MATLAB software. The last column indicates when Ghostscript is used.
- Not all drivers are supported on all platforms. Non support is noted in the first column of the table.
- If you specify a particular printer with the `-P` option and do not specify a driver, a default driver for that printer is selected, either by the operating system or by MATLAB , depending on the platform:
  - On MATLAB, the driver associated with this particular printing device is used.
  - On Macintosh and UNIX platforms, the driver specified in `printop.m` is used

See Selecting the Printer in the Graphics documentation for more information.

Note The MathWorks™ is planning to leverage existing operating system (OS) support for printer drivers and devices. As a result, the ability to specify certain print devices using the `print -d` command, and certain graphics formats using the `print -d` command and/or the `saveas` command, will be removed in a future release. In the following table, the affected formats have an asterisk (*) next to the `print` command option string. The asterisks provide a link to the Web site which supplies a form for users to give feedback about these changes.

Printer Driver	Print Command Option String	Ghostscript
Canon BubbleJet BJ10e	`-dbj10e` *	Yes

Printer Driver	Print Command Option String	Ghostscript
Canon BubbleJet BJ200 color	`-dbj200` *	Yes
Canon Color BubbleJet BJC-70/BJC-600/BJC-4000	`-dbjc600` *	Yes
Canon Color BubbleJet BJC-800	`-dbjc800` *	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
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

Printer Driver	Print Command Option String	Ghostscript
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
HP PaintJet XL300 color (not supported on Windows)	`-dpjxl300` *	Yes
HPGL for HP 7475A and other compatible plotters. (Renderer cannot be set to Z-buffer.)	`-dhpgl` *	No
IBM 9-pin Proprinter	`-dibmpro` *	Yes

Printer Driver	Print Command Option String	Ghostscript
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

Tip 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 printing is the default for UNIX platforms. You can change this default by editing the `printopt.m` file. Likewise, if you want color PostScript printing to be the default instead of black-and-white PostScript printing, 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 file name. 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, TIFF and PNG files, but is not supported for Ghostscript raster formats. For more information, see “Printing and Exporting without a Display” on page 2-2878 and “Resolution Considerations” on page 2-2881.

Note When you print to a file, the file name must have fewer than 128 characters, including path name. When you print to a file in your current folder, the filename must have fewer than 126 characters, because MATLAB places `'./'` or `'.\'`’ at the beginning of the filename when referring to it.

The following table shows the supported output formats for exporting from figures 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 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

Graphics Format	Bitmap or Vector	Print Command Option String	MATLAB or Ghostscript
HDF 24-bit	Bitmap	`-dhdf`	MATLAB
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
PDF Color PDF file format	Vector	`-dpdf`	Ghostscript
PGM Portable Graymap (plain format)	Bitmap	`-dpgm`	Ghostscript
PGM Portable Graymap (raw format)	Bitmap	`-dpgmraw`	Ghostscript

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. The `-dtiffn` variant writes an uncompressed TIFF. JPEG is a lossy, highly compressed format that is supported on all platforms for image processing and for inclusion into HTML documents on the 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.

Printing and Exporting without a Display

On a UNIX platform (including Macintosh), where you can start in MATLAB `nodisplay` mode (`matlab -nodisplay`), you can print using most of the drivers you can use with a display and export to most of the same file formats. The PostScript and Ghostscript devices all function in `nodisplay` mode on UNIX platforms. The graphic devices `-djpeg`, `-dpng`, `-dtiff` (compressed TIFF bitmaps), and `-tiff` (EPS with TIFF preview) work as well, but under `nodisplay` they use Ghostscript to generate output instead of using the drivers built into MATLAB. However, Ghostscript ignores the `-r` option when generating `-djpeg`, `-dpng`, `-dtiff`, and `-tiff` image files. This means that you cannot vary the resolution of image files when running in `nodisplay` mode.

Naturally, the Windows only `-dwin` and `-dwinc` output formats cannot be used on UNIX or Mac platforms with or without a display.

The same holds true on Windows platforms with the `-noFigureWindows` startup option. The `-dwin`, `-dwinc`, and `-dsetup` options operate as usual under `-noFigureWindows`. However, the `printpreview` GUI does not function in this mode.

The formats which you cannot generate in `nodisplay` mode on UNIX and Mac platforms are:

- bitmap (`-dbitmap`) — Windows bitmap file (except for Simulink models)
- bmp (`-dbmp...`) — Monochrome and color bitmaps
- hdf (`-dhdf`) — Hierarchical Data Format
- svg (`-dsvg`) — Scalable Vector Graphics file (except for Simulink models)
- tiffn (`-dtiffn`) — TIFF image file, no compression

In addition, uicontrols do not print or export in `nodisplay` mode.

Printing Options

This table summarizes options that you can specify for `print`. The second column links to tutorials in in the MATLAB Graphics documentation that provide operational details. Also see “Resolution Considerations” on page 2-2881 for information on controlling output resolution.

Option	Description
`-adobecset`	PostScript devices only. Use PostScript default character set encoding. See “Early PostScript 1 Printers”.
`-append`	PostScript devices only. Append figure to existing PostScript file. See “Settings That Are Driver Specific”.

Option	Description
`-cmyk`	PostScript devices only. Print with CMYK colors instead of RGB. See “Setting CMYK Color”.
`-ddriver`	Printing only. Printer driver to use. See “Printer Drivers” on page 2-2871 table.
`-dformat`	Exporting only. Graphics format to use. See “Graphics Format Files” table.
`-dsetup`	Windows printing only. Display the (platform-specific) Print Setup dialog. Settings you make in it are saved, but nothing is printed.
`-fhandle`	Handle of figure to print. Note that you cannot specify both this option and the `-s`*`windowtitle`* option. See “Which Figure Is Printed”.
`-loose`	PostScript and Ghostscript printing only. Use loose bounding box for PostScript output. 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 `-zbuffer` or `-painters`. See “Selecting a Renderer”.
`-painters`	Render using the Painter’s algorithm. Note that you cannot specify this method in conjunction with `-zbuffer` or `-opengl`. See “Selecting a Renderer”.
`-P`*`printer`*	Specify name of printer to use. See “Selecting the Printer”.

Option	Description
`-rnumber`	PostScript and built-in raster formats, and Ghostscript vector format 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. For details, see "Resolution Considerations" on page 2-2881 and "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 printing 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".

Resolution Considerations

Use `-rnumber` to specify the resolution of the generated output. In general, using a higher value will yield higher quality output but at the cost of larger output files. It affects the resolution and output size of all MATLAB built-in *raster* formats (which are identified in column four of the table in "Graphics Format Files" on page 2-2875).

Note Built-in graphics formats are generated directly from MATLAB without conversion through the Ghostscript library. Also, in headless (`nodisplay`) mode, writing to certain image formats is not done by built-in drivers, as it is when a display is being used. These formats are `-djpeg`, `-dtiff`, and `-dpng`. Furthermore, the `-dhdf` and `-dbmp` formats cannot be generated in headless mode (but you can substitute `-dbmp16m` for `-dbmp`). See “Printing and Exporting without a Display” on page 2-2878 for details on printing when not using a display.

Unlike the built-in MATLAB formats, graphic output generated via Ghostscript does not directly obey `-r` option settings. However, the intermediate PostScript file generated by MATLAB as input for the Ghostscript processor is affected by the `-r` setting and thus can indirectly influence the quality of the final Ghostscript generated output.

The effect of the `-r` option on output quality can be subtle at ordinary magnification when using the OpenGL or ZBuffer renderers and writing to one of the MATLAB built-in raster formats, or when generating vector output that contains an embedded raster image (for example, PostScript or PDF). The effect of specifying higher resolution is more apparent when viewing the output at higher magnification or when printed, since a larger `-r` setting provides more data to use when scaling the image.

When generating fully vectorized output (as when using the Painters renderer to output a vector format such as PostScript or PDF), the resolution setting affects the degree of detail of the output; setting resolution higher generates crisper output (but small changes in the resolution may have no observable effect). For example, the gap widths of lines that do not use a solid (`'-'`) linestyle can be affected.

Paper Sizes

MATLAB printing 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	8.5 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
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

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 Microsoft 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 Microsoft 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 `print` , which 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 on-screen 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 keep a bounding box from being too 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.

If you print or export in `nodisplay` mode, none of the `uicontrols` the figure has will be visible. If you run code that adds `uicontrols` to a figure when the figure is invisible, the controls will not print until the figure is made visible.

Printing Interpolated Shading with PostScript Drivers

You can print MATLAB 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. For example:

```
h = figure('IntegerHandle','off')
print h -dps2
```

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 file name to be passed in as a variable:

```
filename = 'mydata';
print('-f3', '-dpsc', filename);
```

(Because a file name is specified, the figure will be printed to a file.)

This example combines the previous two examples and uses the function form to print using both a handle variable and a file name variable:

```
h = figure; plot(1:4,5:8)
filename = 'mydata';
print(h, '-dpsc', filename);
```

(Because a file name 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 the 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 "Printing Options" on page 2-2879 and "Printing Interpolated Shading with PostScript Drivers" on page 2-2886 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 file names 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 -depsc -tiff -r300 picture1
```

saves the current figure at 300 dpi, in a color Encapsulated PostScript file named `picture1.eps`. The `-tiff` option creates a 72 dpi TIFF preview, which many word processor applications can display on screen after you import the EPS file. This enables you to view the picture on screen within your word processor and print the document to a PostScript printer using a resolution of 300 dpi.

See Also

`figure`, `hgsave`, `imwrite`, `orient`, `printdlg`, `printopt`, `saveas`

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 modal dialog box from which you can print the figure window identified by the handle `fig`. Note that uimenus do not print.

`printdlg('-crossplatform',fig)` displays the standard cross-platform MATLAB printing dialog rather than the built-in printing dialog box for Microsoft Windows computers. Insert this option before the `fig` argument.

`printdlg('-setup',fig)` forces the printing dialog to appear in a setup mode. Here one can set the default printing options without actually printing.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

See Also

`pagesetupdlg`, `printpreview`

Purpose

Preview figure to print

Contents

GUI Alternative

Use **File > Print Preview** on the figure window menu to access the Print Preview dialog box, described below. For details, see in the MATLAB Graphics documentation.

Syntax

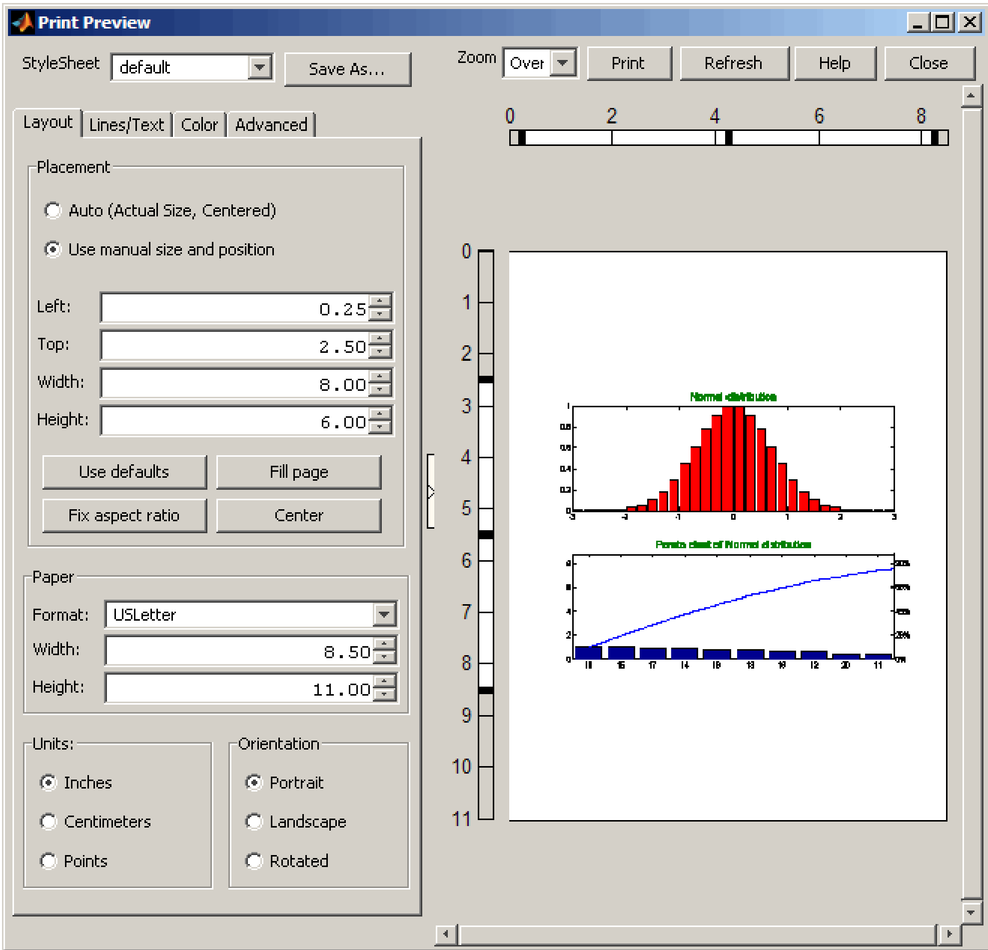
```
printpreview
printpreview(f)
```

Description

`printpreview` 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.

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

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**.



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:

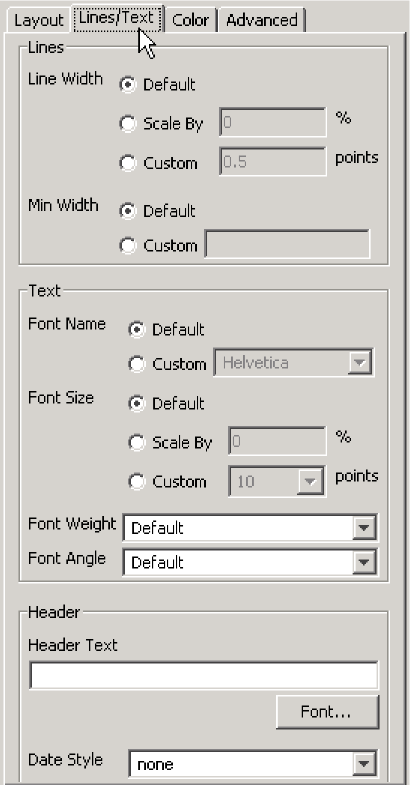
Group	Option	Description
Placement	**Auto**	Let MATLAB decide placement of plot on page
	Use manual...	Specify position parameters for plot on page
	Top, Left, Width, Height	Standard position parameters in current units
	Use defaults	Revert to default position
	Fill page	Expand figure to fill printable area (see note below)
	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
Orientation	**Portrait**	Upright paper orientation

Group	Option	Description
	Landscape	Sideways paper orientation
	Rotated	Currently the same as **Landscape**

Note Selecting the **Fill page** option changes the `PaperPosition` property to fill the page, allowing objects in normalized units to expand to fill the space. If an object within the figure has an absolute size, for example a table, it can overflow the page when objects with normalized units expand. To avoid having objects fall off the page, do not use **Fill page** under such circumstances.

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:

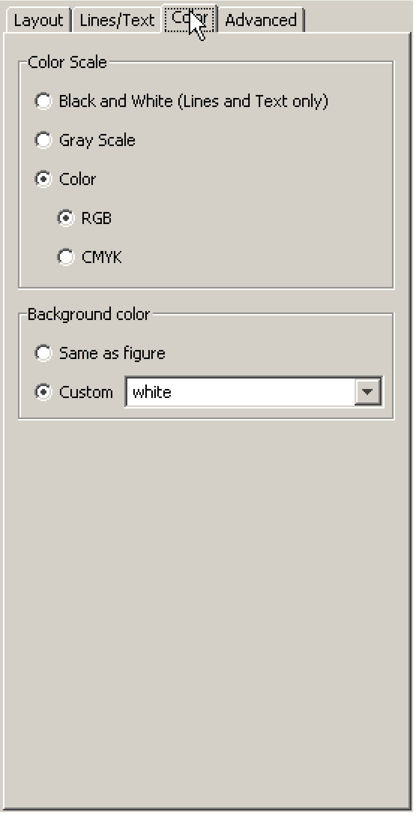


Group	Option	Description
Lines	**Line Width**	Scale all lines by a percentage from 0 upward (100 being no change), print lines at a specified point size, or default line widths used on the plot
	Min Width	Smallest line width (in points) to use when printing; defaults to 0.5 point
Text	**Font Name**	Select a system font for all text on plot, or default to fonts currently used on the plot

Group	Option	Description
	Font Size	Scale all text by a percentage from 0 upward (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 Tex**t	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:

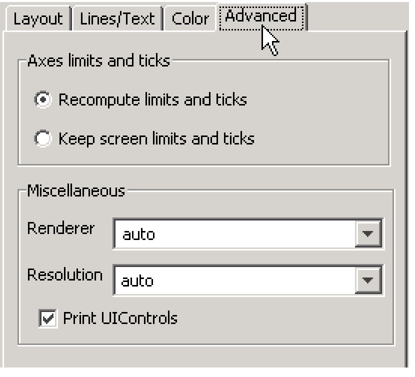


Group	Option	Description
Color Scale	**Black and White**	Select to print lines and text in black and white, but use color for patches and other objects
	Gray Scale	Convert colors to shades of gray on printed pages

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)

Group	Option	Description
	Keep limits and ticks	Use the *x*- and *y*-axes ticks and limits shown on the plot when printing the previewed figure
Miscellaneous	**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
```

```
105
 72
```

See Also cumprod, diff, sum

profile

Purpose

Profile execution time for function

GUI Alternatives

As an alternative to the `profile` function, select **Desktop > Profiler** to open the Profiler.

Syntax

```
profile on
profile -history
profile -nohistory
profile -history -historysize integer
profile -timer clock
profile -history -historysize integer -timer clock
profile off
profile resume
profile clear
profile viewer
S = profile('status')
stats = profile('info')
```

Description

The `profile` function helps you debug and optimize M-files by tracking their execution time. For each M-function, M-subfunction, or MEX-function in the file, `profile` records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. Some people use `profile` simply to see the child functions; see also `depfun` for that purpose. To open the Profiler graphical user interface, use the `profile viewer` syntax. By default, Profiler time is CPU time. The total time reported by the Profiler is not the same as the time reported using the `tic` and `toc` functions or the time you would observe using a stopwatch.

Note If your system uses Intel multi-core chips, you may want to restrict the active number of CPUs to 1 for the most accurate and efficient profiling. See for details on how to do this.

`profile on` starts the Profiler, clearing previously recorded profile statistics. Note the following:

- You can specify all, none, or a subset, of the `history`, `historysize` and `timer` options with the `profile on` syntax.
- You can specify options in any order, including before or after `on`.
- If the Profiler is currently on and you specify `profile` with one of the options, MATLAB software returns an error message and the option has no effect. For example, if you specify `profile timer real`, MATLAB returns the following error: `The profiler has already been started. TIMER cannot be changed.`
- To change options, first specify `profile off`, and then specify `profile on` or `profile resume` with new options.

`profile -history` records the exact sequence of function calls. The `profile` function records, by default, up to 1,000,000 function entry and exit events. For more than 1,000,000 events, `profile` continues to record other profile statistics, but not the sequence of calls. To change the number of function entry and exit events that the `profile` function records, use the **`historysize`** option. By default, the `history` option is not enabled.

`profile -nohistory` 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 be collected.

`profile -history -historysize integer` specifies the number of function entry and exit events to record. By default, `historysize` is set to 1,000,000.

`profile -timer clock` specifies the type of time to use. Valid values for *`clock`* are:

- `'cpu'` — The Profiler uses computer time (the default).
- `'real'` — The Profiler uses wall-clock time.

For example, `cpu` time for the `pause` function is typically small, but `real` time accounts for the actual time paused, and therefore would be larger.

`profile` **`-history -historysize`** *`integer`* **`-timer`** *`clock`* specifies all of the 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 that they appear in the structure.

Field	Values	Default Value
`ProfilerStatus`	`'on'` or `'off'`	off
`DetailLevel`	`'mmex'`	'mmex'
`Timer`	`'cpu'` or `'real'`	'cpu'
`HistoryTracking`	`'on'` or `'off'`	'off'
`HistorySize`	integer	1000000

`stats = profile('`**`info`**`')` displays a structure containing the results. Use this function to access the data generated by `profile`. The table lists the fields in the order that they appear in the structure.

Field	Description
`FunctionTable`	Structure array containing statistics about each function called

Field	Description
`FunctionHistory`	Array containing function call history
`ClockPrecision`	Precision of the `profile` function'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 that 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

Field	Description
ExecutedLines	Array containing line-by-line details for the function being profiled. Column 1: Number of the line that executed. If a line was not executed, it does not appear in this matrix. Column 2: Number of times the line was executed 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.
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 when 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 `0` means entrance into a function and `1` 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

`depdir`, `depfun`, `mlint`, `profsave`

Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation

in the Parallel Computing Toolbox 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 subfolder of the current folder 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 folder specified by `dirname`.

Examples

Run `profile` and save the results.

```
profile on
plot(magic(5))
profile off
profsave(profile('info'),'myprofile_results')
```

See Also

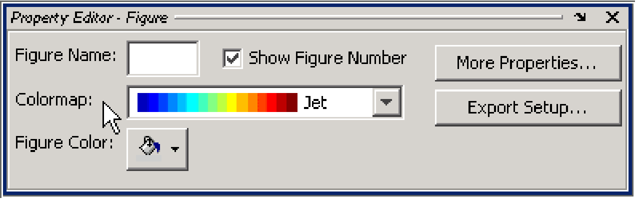
`profile`

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`, `plotedit`, `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.

Remarks

COM functions are available on Microsoft Windows systems only.

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 (COM)`

properties

Purpose Class property names

Syntax

```
properties('classname')
properties(obj)
p = properties(...)
```

Description

`properties('classname')` displays the names of the public properties for the MATLAB class named by `classname`. The `properties` function also displays inherited properties.

`properties(obj)` `obj` can be either a scalar object or an array of objects. When `obj` is scalar, `properties` also returns dynamic properties. See for information on using dynamic properties.

`p = properties(...)` returns the property names in a cell array of strings.

Definitions

A property is public when its `GetAccess` attribute value is `public` and its `Hidden` attribute value is `false` (default values for these attributes). See for a complete list of attributes.

`properties` is also a MATLAB class-definition keyword. See `classdef` for more information on class definition keywords.

Examples

Retrieve the names of the public properties of class `memmapfile` and store the result in a cell array of strings:

```
p = properties('memmapfile');
p
ans =

    'writable'
    'offset'
    'format'
    'repeat'
    'filename'
```

Construct an instance of the `MException` class and get its properties names:

```
me = MException('Msg:ID','MsgText');
properties(me)
Properties for class MException:

    identifier
    message
    cause
    stack
```

Alternatives

You can use the Workspace browser to browse current property values. See for more information on using the Workspace browser.

See Also

`fieldnames` | `events` | `methods`

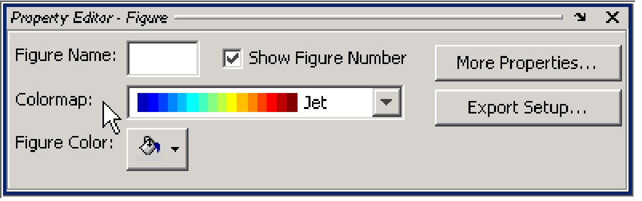
Tutorials

-

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 in the MATLAB Graphics documentation.

Syntax

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

Description

`propertyeditor('on')` displays the Property Editor on the current figure.

`propertyeditor('off')` hides the Property Editor on the current figure.

`propertyeditor('toggle')` or `propertyeditor` toggles the visibility of the property editor on the current figure.

`propertyeditor(figure_handle,...)` displays or hides the Property Editor on the figure specified by `figure_handle`.

See Also

`plottools`, `plotbrowser`, `figurepalette`, `inspect`

Purpose Psi (polygamma) function

Syntax

```
Y = psi(X)
Y = psi(k,X)
Y = psi(k0:k1,X)
```

Description `Y = 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

$$\psi(x) = \mathbf{digamma}(x) = \frac{d(\log(\Gamma(x)))}{dx} = \frac{d(\Gamma(x))/dx}{\Gamma(x)}$$

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

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

Publish M-file containing cells, save output to specified file type

Syntax

```
publish('file')
publish('file','format')
publish('file', options)
my_doc = publish('file',...)
```

Description

`publish('file')` publishes `file.m` by running it in the base workspace, one cell at a time. It saves the code, comments, and results to an HTML output file. MATLAB stores the output file, `file.html`, along with other supporting output files in an `html` subfolder of the folder containing `file.m`.

`publish('file','format')` publishes `file.m` by running it in the base workspace, one cell at a time. It saves the code, comments, and results to an output file, `file.format`. Regardless of the format, MATLAB stores this output file, along with other supporting output files, in an `html` subfolder of the folder containing `file.m`.

`publish('file', options)` publishes `file.m` using the structure *`options`*.

`my_doc = publish('file',...)` returns the output resulting from publishing `file.m` to `my_doc`.

Inputs

`file`

Specifies the M-file to publish.

`format`

Specifies the format to which you want to publish the file. Valid formats appear in the "Options for the publish Function" table later in this section.

`options`

A structure with the fields listed in the following table.

Options for the publish Function

<table>
<tr><th>Field</th><th>Values</th></tr>
<tr><td><code>format</code></td><td>Specifies the output format for the published document, as follows:
• <code>'doc'</code>—Specifies Microsoft Word output format.
• <code>'latex'</code>—Specifies LaTeX output format.
• <code>'ppt'</code>—Specifies Microsoft PowerPoint output format.
• <code>'xml'</code>—Specifies Extensible Markup Language output format.
• <code>'pdf'</code>—Specifies Portable Document Format output format.
If you specify <code>'pdf'</code>, then you must specify <code>imageFormat</code> as <code>'.bmp'</code> (the default) or <code>'.jpg'</code>.
• <code>'html'</code> (default)—Specifies Hypertext Markup Language output format.
If you specify <code>html</code>, MATLAB includes the M-file code at the end of the published HTML file as comments, even when you set the <code>showCode</code> option to <code>false</code>. Because MATLAB includes the M-file code as comments, the code does not display in a Web browser. Use the <code>grabcode</code> function to extract the code from the HTML file.</td></tr>
<tr><td><code>stylesheet</code></td><td>Specifies the Extensible Stylesheet Language (XSL) file that you want MATLAB to use when you specify a <code>format</code> of <code>'html'</code>, <code>'xml'</code>, or <code>'latex'</code> as follows:
• <code>''</code> (default)—The MATLAB default stylesheet
• <code>XSL file name</code>—The full path of the XSL file</td></tr>
<tr><td><code>outputDir</code></td><td>Specifies the folder to which you want MATLAB to publish the output document and its associated image files, as follows:
• <code>''</code> (default)—MATLAB places output in an <code>html</code> subfolder of the current folder, which MATLAB creates.
• <code>full path</code>—MATLAB places output in the specified folder</td></tr>
</table>

Options for the publish Function (Continued)

<table>
<tr><th>Field</th><th>Values</th></tr>
<tr><td><code>imageFormat</code></td><td>Specifies the file type for images that MATLAB produces when publishing M-files.
<code>'png'</code> (default unless <code>format</code> is <code>latex</code> or <code>pdf</code>)
<code>'epsc2'</code> (default when <code>format</code> is <code>latex</code>)
<code>'bmp'</code> (default when <code>format</code> is <code>'pdf'</code>)
Alternatively, <code>'.jpg'</code> when the <code>format</code> is <code>'pdf'</code>
Any format supported by <code>print</code> when <code>figureSnapMethod</code> is <code>print</code>, unless <code>format</code> is <code>pdf</code>
Any format supported by <code>imwrite</code> when <code>figureSnapMethod</code> is <code>getframe</code>, <code>entireFigureWindow</code>, or <code>entireGUIWindow</code>, unless <code>format</code> is <code>pdf</code></td></tr>
<tr><td><code>figureSnapMethod</code></td><td>Specifies how figure windows and GUI dialog boxes that the M-file code creates appear in published documents. Window decorations are the title bar, toolbar, menu bar, and window border.
<table>
<tr><th>Values</th><th colspan="2">Window Decorations for ...</th><th colspan="2">Background Color for ...</th></tr>
<tr><th></th><th>GUIs</th><th>Figures</th><th>GUIs</th><th>Figures</th></tr>
<tr><td>’entireGUIWindow’ (default)</td><td>Included</td><td>Excluded</td><td>Match screen</td><td>White</td></tr>
<tr><td>’print’</td><td>Excluded</td><td>Excluded</td><td>White</td><td>White</td></tr>
<tr><td>’getframe’</td><td>Excluded</td><td>Excluded</td><td>Match screen</td><td>Match screen</td></tr>
<tr><td>’entireFigureWindow’</td><td>Included</td><td>Included</td><td>Match screen</td><td>Match screen</td></tr>
</table></td></tr>
</table>

Options for the publish Function (Continued)

Field	Values
`useNewFigure`	A logical value that specifies whether MATLAB creates a Figure window for figures that the M-file code generates, as follows: • `true` (default) specifies that if the M-file code generates a figure, then MATLAB create a Figure window with a white background and at the default size before publishing. • `false` specifies that you do not want MATLAB to create a figure window. This value enables you to use a figure with different properties for publishing. Open a Figure window, change the size and background color, for example, and then publish. Figures in your published document use the characteristics of the figure you opened before publishing.
`maxHeight`	Specifies the maximum height, in pixels, for an image that the M-file code generates, as follows: • `[]` (default)—Height is unrestricted. Always used when the `format` is `pdf`. Height is unrestricted. Always used when the `format` is `pdf`. • Any positive integer—Height is the specified value. Height is the specified value.
`maxWidth`	Specifies the maximum width, in pixels, for an image that the M-file code generates, as follows: • `[]` (default)—Width is unrestricted. Always used when the `format` is `pdf`. • Any positive integer—Width is the specified value.

Options for the publish Function (Continued)

Field	Values
`showCode`	Logical value that specifies whether MATLAB includes the M-file code in the published document: • `true` (default) • `false`
`evalCode`	Logical value that specifies whether MATLAB runs the code that it is publishing. • `true` (default) Use this option if you want to run the code. If set to `true` and you are publishing a function M-file that requires inputs, you must also specify the `codeToEvaluate` option. • `false` Use this option if you do not want to run the code, but do want to present it (without output) in the published document.
`catchError`	Logical value that specifies what MATLAB does if there is an error in the code that it is publishing: • `true` (default)—MATLAB continues publishing and includes the error in the published file • `false`—MATLAB displays the error and publishing ends
`codeToEvaluate`	Specifies the code that MATLAB is to evaluate. By default, MATLAB evaluates the code in the M-file you are publishing.

Options for the publish Function (Continued)

Field	Values
`createThumbnail`	Logical value that specifies whether MATLAB creates a thumbnail image of the published document: • `true` (default) • `false`
`maxOutputLines`	Value that specifies the maximum number of output lines per M-file cell that you want to publish before truncating the output. • `Inf` (default)—Includes all output lines • nonnegative integer—Includes at most the number of lines specified

Examples

Copy `sine_wave.m`, publish it to HTML, and then view the published document:

```
copyfile(fullfile(docroot,'techdoc','matlab_env','examples', ...
'sine_wave.m'),'.','f')

% When you run the command that follows, MATLAB runs sine_wave.m,
% and saves the code, comments, and results to
% /html/sine_wave.html:

publish('sine_wave.m', 'html')

% View the published output file in the Web browser:
web('html/sine_wave.html')
```

Copy `sine_wave.m`, publish it to Microsoft Word format by using a structure, and then view the published document:

```
copyfile(fullfile(docroot,'techdoc','matlab_env','examples', ...
```

```
'sine_wave.m'),'.','f')

% Define the structure, options_doc_nocode,
% to exclude code from the output
% and publish to Microsoft World format:
options_doc_nocode.format='doc'
options_doc_nocode.showCode=false

% Publish sine_wave.m:
publish('sine_wave.m',options_doc_nocode)

% View the published output file in Microsoft Word:
winopen('html/sine_wave.doc')
```

Copy `collatz.m`, create a structure to specify the input values, publish it to HTML, and then view the published document:

```
copyfile(fullfile(docroot,'techdoc','matlab_env','examples', ...
'collatz.m'),'.','f')

% Create a structure, opts, that contains the code that you
% want collatz.m to evaluate when it runs:
opts.codeToEvaluate = 'n = 3; collatz(n)';

% In the MATLAB Web browser, display the results of
% publishing collatz.m when it runs with the values
% specified in opts:
web(publish('collatz',opts))
```

Copy `sine_wave.m`, publish it capturing window decorations, and then view the published document:

```
copyfile(fullfile(docroot,'techdoc','matlab_env','examples', ...
'sine_wave.m'),'.','f')

% Create an options file that causes the published document
%  to capture window decorations:
```

```
function_options.format='html';
function_options.figureSnapMethod='entireGUIWindow';

% Publish the script using the options file:
publish('sine_wave.m',function_options);

% View the output in the MATLAB Web browser
web('html/sine_wave.html')
```

Publish an M-File demo file to PDF, and then open the published document:

```
open(publish('sparsity',struct('format','pdf','outputDir',tempname)))
```

Alternatives

To publish an M-File from the desktop:

1. Open the M-File you want to publish in the Editor.
2. Choose one of the following:
   - To publish with default options, choose **File > Publish *`filename`***.
   - To publish with customized options, choose **File > Publish Configuration for filename > Edit Publish Configurations for filename**, and then adjust the **Publish settings**.

See Also

`grabcode` | `notebook`

How To

-
-

Purpose

Store character array in Automation server

Syntax

MATLAB Client

```
h.PutCharArray('varname', 'workspace', 'string')
PutCharArray(h, 'varname', 'workspace', 'string')
invoke(h, 'PutCharArray', 'varname', 'workspace', 'string')
```

IDL Method Signature

```
PutCharArray([in] BSTR varname, [in] BSTR workspace,
[in] BSTR string)
```

Microsoft 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`.

MATLAB Client

```
h = actxserver('matlab.application');
h.PutCharArray('str', 'base', ...
   'He jests at scars that never felt a wound.')

S = h.GetCharArray('str', 'base')
S =
    He jests at scars that never felt a wound.
```

Visual Basic .NET Client

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```
Dim Matlab As Object
Try
    Matlab = GetObject(, "matlab.application")
Catch e As Exception
    Matlab = CreateObject("matlab.application")
End Try
MsgBox("MATLAB window created; now open it...")
```

Open the MATLAB window, then click **Ok**.

```
Matlab.PutCharArray("str", "base", _
       "He jests at scars that never felt a wound.")
MsgBox("In MATLAB, type" & vbCrLf _
  & "str")
```

In the MATLAB window type `str`; MATLAB displays:

```
str =
He jests at scars that never felt a wound.
```

Click **Ok**.

```
MsgBox("closing MATLAB window...")
```

Click **Ok** to close and terminate MATLAB.

```
Matlab.Quit()
```

See Also

`GetCharArray`, `PutWorkspaceData`, `GetWorkspaceData`, `Execute`

PutFullMatrix

Purpose Matrix in Automation server workspace

Syntax

MATLAB Client

```
h.PutFullMatrix('varname', 'workspace', xreal, ximag)
PutFullMatrix(h, 'varname', 'workspace', xreal, ximag)
```

IDL Method Signature

```
PutFullMatrix([in] BSTR varname, [in] BSTR
workspace, [in] SAFEARRAY(double) xreal, [in]
SAFEARRAY(double) ximag)
```

Microsoft Visual Basic Client

```
PutFullMatrix([in] varname As String, [in] workspace As
String, [in] xreal As Double, [in] ximag As Double)
```

Description `h.PutFullMatrix('varname', 'workspace', xreal, ximag)` stores a matrix in the specified `workspace` of the server attached to handle `h` and assigns it to variable `varname`. Use `xreal` and `ximag` for the real and imaginary parts of the matrix. The matrix cannot be a scalar, an empty array, or have more than two dimensions. The values for *`workspace`* are `base` or `global`.

`PutFullMatrix(h, 'varname', 'workspace', xreal, ximag)` is an alternate syntax.

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 Use a MATLAB client to write a matrix to the base workspace of the server:

```
h = actxserver('matlab.application');
h.PutFullMatrix('M', 'base', rand(5), zeros(5))
%Use one output for real values only
xreal = h.GetFullMatrix('M', 'base', zeros(5), zeros(5))
```

Use a Visual Basic .NET client to write a matrix to the base workspace of the server:

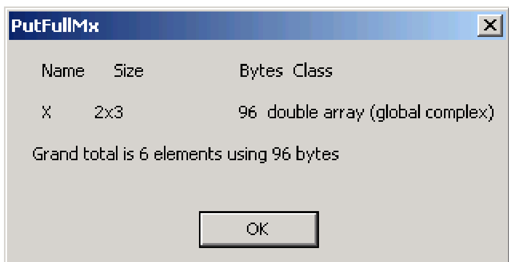
```
Dim MatLab As Object
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim ZReal(4, 4) As Double
Dim ZImag(4, 4) As Double
Dim i, j As Integer

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)
```

Use a MATLAB client to write a matrix to the global workspace of the server:

```
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')
```

Use a Visual Basic .NET client to write a matrix to the global workspace of the server:

```
Dim MatLab As Object
Dim XReal(1, 2) As Double
Dim XImag(1, 2) As Double
Dim result As String
Dim i, j As Integer

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("X", "global", XReal, XImag)
result = Matlab.Execute("whos global")
MsgBox(result)
```



See Also

GetFullMatrix | PutWorkspaceData | Execute

How To

-
-

Purpose

Data in Automation server workspace

Syntax

MATLAB Client

```
h.PutWorkspaceData('varname', 'workspace', data)
PutWorkspaceData(h, 'varname', 'workspace', data)
```

IDL Method Signature

```
PutWorkspaceData([in] BSTR varname, [in] BSTR
workspace, [in] VARIANT data)
```

Microsoft Visual Basic Client

```
PutWorkspaceData(varname As String, workspace
As String, data As Object)
```

Description

`h.PutWorkspaceData('varname', 'workspace', data)` stores `data` in the `workspace` of the server attached to handle `h` and assigns it to `varname`. The values for *`workspace`* are `base` or `global`.

`PutWorkspaceData(h, 'varname', 'workspace', data)` is an alternate syntax.

Use `PutWorkspaceData` to pass numeric and character array data respectively to the server. Do *not* use `PutWorkspaceData` on sparse arrays, structures, or function handles. Use the `Execute` method for these data types.

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 a MATLAB client and put it in the base workspace of the MATLAB Automation server:

```
h = actxserver('matlab.application');
for i = 0:6
   data(i+1)  = i * 15;
end
```

```
h.PutWorkspaceData('A', 'base', data)
```

Create an array in a Visual Basic client and put it in the base workspace of the MATLAB Automation server:

1 Create the Visual Basic application. Use the `MsgBox` command to control flow between MATLAB and the application:

```
Dim Matlab As Object
Dim data(6) As Double
Dim i As Integer
MatLab = CreateObject("matlab.application")
For i = 0 To 6
   data(i)  = i * 15
Next i
MatLab.PutWorkspaceData("A", "base", data)
MsgBox("In MATLAB, type" & vbCrLf & "A")
```

2 Open the MATLAB window and type `A`. MATLAB displays:

```
A =
     0    15    30    45    60    75    90
```

3 Click **Ok** to close and terminate MATLAB.

See Also

`GetWorkspaceData` | `PutFullMatrix` | `PutCharArray` | `Execute`

How To

-
-

Purpose

Identify current folder

Syntax

```
pwd
currentFolder = pwd
```

Description

`pwd` displays the MATLAB current folder.

`currentFolder = pwd` returns the current folder as a string to `currentFolder`.

Alternatives

- Use the Current Folder field in the MATLAB desktop toolbar.
- Use address bar in the Current Folder browser.

See Also

`cd` | `dir`

How To

-
-
-

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 in the MATLAB Programming documentation for more information.

, 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 norm(b-A*x)/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 inv(M)*A*x = 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\x and mfun(x,'transp') returns M'\x.

qmr(A,b,tol,maxit,M1,M2,x0) specifies the initial guess. If x0 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 tol within maxit iterations.
1	qmr iterated maxit 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 qmr 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 norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

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

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

Examples

Example 1

```
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
```

When you enter

```
x1=run_qmr;
```

MATLAB software 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.

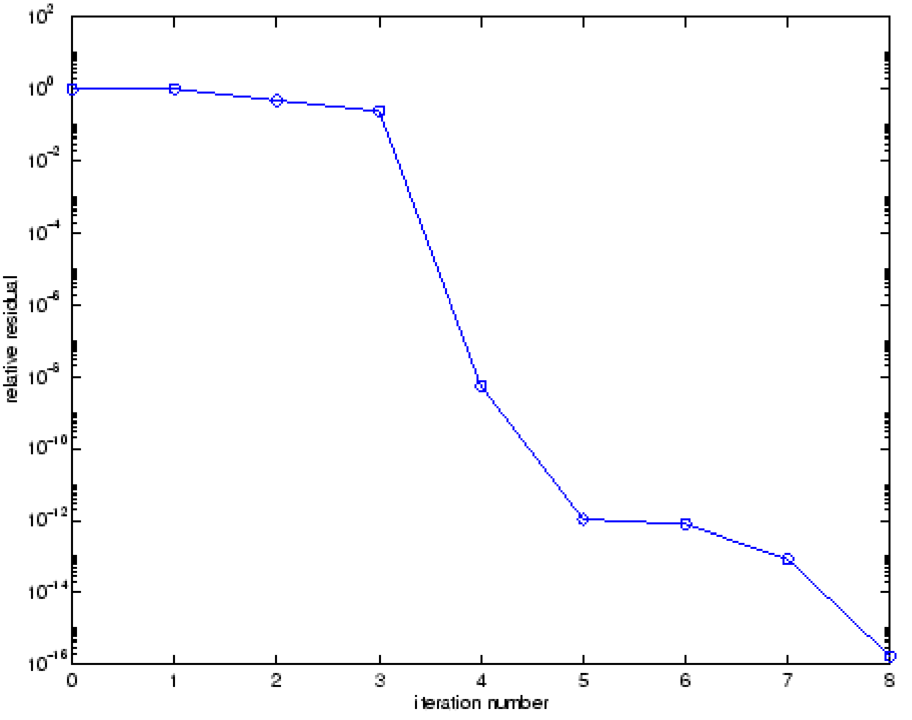
```
[L1,U1] = luinc(A,1e-5);
[x1,flag1] = qmr(A,b,1e-6,20,L1,U1)
```

`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. resvec2(1) = norm(b) and resvec2(9) = norm(b-A*x2). You can follow the progress of qmr by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
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 Nöel 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] = qr(A)
[Q,R] = qr(A,0)
[Q,R,E] = qr(A)
[Q,R,E] = qr(A,0)
X = qr(A)
X = qr(A,0)
R = qr(A)
```

Description

`[Q,R] = qr(A)`, where `A` is `m`-by-`n`, produces an `m`-by-`n` upper triangular matrix `R` and an `m`-by-`m` unitary matrix `Q` so that `A = Q*R`.

`[Q,R] = qr(A,0)` produces the economy-size decomposition. If `m > n`, only the first `n` columns of `Q` and the first `n` rows of `R` are computed. If `m<=n`, this is the same as `[Q,R] = qr(A)`.

If A is full:

`[Q,R,E] = qr(A)` produces unitary `Q`, upper triangular `R` and a permutation matrix `E` so that `A*E = Q*R`. The column permutation `E` is chosen so that `abs(diag(R))` is decreasing.

`[Q,R,E] = qr(A,0)` produces an economy-size decomposition in which `E` is a permutation vector, so that `A(:,E) = Q*R`.

`X = qr(A)` and `X = qr(A,0)` return a matrix `X` such that `triu(X)` is the upper triangular factor `R`.

If A is sparse:

`R = qr(A)` computes a `Q`-less `QR` decomposition and returns the upper triangular factor `R`. Note that `R = CHOL(A'*A)`. Since `Q` is often nearly full, this is preferred to `[Q,R] = QR(A)`.

`R = qr(A,0)` produces economy-size `R`. If `m>n`, `R` has only `n` rows. If `m<=n`, this is the same as `R = qr(A)`.

`[Q,R,E] = qr(A)` produces unitary `Q`, upper triangular `R` and a permutation matrix `E` so that `A*E = Q*R`. The column permutation E is chosen to reduce fill-in in R.

`[Q,R,E] = qr(A,0)` produces an economy-size decomposition in which `E` is a permutation vector, so that `A(:,E) = Q*R`.

`[C,R] = qr(A,B)`, where `B` has as many rows as `A`, returns `C = Q'*B`. The least-squares solution to `A*X = B` is `X = R\C`.

`[C,R,E] = qr(A,B)`, also returns a fill-reducing ordering. The least-squares solution to `A*X = B` is `X = E*(R\C)`.

`[C,R] = qr(A,B,0)` produces economy-size results. If `m>n`, `C` and `R` have only `n` rows. If `m<=n`, this is the same as `[C,R] = qr(A,B)`.

`[C,R,E] = qr(A,B,0)` additionally produces a fill-reducing permutation vector `E`. In this case, the least-squares solution to `A*X = B` is `X(E,:) = R\C`.

Examples

Find the least squares approximate solution to `A*x = b` with the `Q`-less `QR` decomposition and one step of iterative refinement:

```
if issparse(A), R = qr(A);
else R = triu(qr(A)); end
x = R\(R'\(A'*b));
r = b - A*x;
e = R\(R'\(A'*r));
x = x + e;
```

See Also

`lu` | `ldl`

qrdelete

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)
```

```
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`

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
```

```
        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`

qrupdate

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*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'*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
```

In this case, the upper triangular entries of R, excluding the first row, are on the order of sqrt(eps).

Consider the update vectors

```
u = [-1 0 0 0 0]'; v = ones(4,1);
```

Instead of computing the rather trivial QR factorization of this rank one update to A from scratch with

```
[QT,RT] = qr(A + u*v')

QT =

     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

RT =

  1.0e-007 *

   -0.1490         0         0         0
         0   -0.1490         0         0
         0         0   -0.1490         0
         0         0         0   -0.1490
         0         0         0         0
```

we may use qrupdate.

```
[Q1,R1] = qrupdate(Q,R,u,v)

Q1 =

   -0.0000   -0.0000   -0.0000   -0.0000    1.0000
    1.0000   -0.0000   -0.0000   -0.0000    0.0000
```

```
    0.0000    1.0000   -0.0000   -0.0000    0.0000
    0.0000    0.0000    1.0000   -0.0000    0.0000
   -0.0000   -0.0000   -0.0000    1.0000    0.0000

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`

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 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`.

, 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.

The list below contains information to help you determine which quadrature function in `MATLAB` to use:

- The quad function may be most efficient for low accuracies with nonsmooth integrands.
- The quadl function may be more efficient than quad at higher accuracies with smooth integrands.
- The quadgk function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The quadv function vectorizes quad for an array-valued fun.
- If the interval is infinite, [a,Inf), then for the integral of fun(x) to exist, fun(x) must decay as x approaches infinity, and quadgk requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but quadgk can be used if fun(x) decays fast enough.
- The quadgk function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint c like log|x-c| or $|x-c|^p$ for p >= -1/2. If the function is singular at points inside (a,b), write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with quadgk, and add the results.

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.

`'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

`quad2d`, `dblquad`, `quadgk`, `quadl`, `quadv`, `trapz`, `triplequad`, `function_handle (@)`,

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`.

quad2d

Purpose

Numerically evaluate double integral over planar region

Syntax

```
q = quad2d(fun,a,b,c,d)
[q,errbnd] = quad2d(...)
q = quad2d(fun,a,b,c,d,param1,val1,param2,val2,...)
```

Description

`q = quad2d(fun,a,b,c,d)` approximates the integral of `fun(x,y)` over the planar region $a \le x \le b$ and $c(x) \le y \le d(x)$. `fun` is a function handle, `c` and `d` may each be a scalar or a function handle.

All input functions must be vectorized. The function `Z=fun(X,Y)` must accept 2-D matrices `X` and `Y` of the same size and return a matrix `Z` of corresponding values. The functions `ymin=c(X)` and `ymax=d(X)` must accept matrices and return matrices of the same size with corresponding values.

`[q,errbnd] = quad2d(...)`. `errbnd` is an approximate upper bound on the absolute error, `|Q - I|`, where `I` denotes the exact value of the integral.

`q = quad2d(fun,a,b,c,d,param1,val1,param2,val2,...)` performs the integration as above with specified values of optional parameters:

`AbsTol`	absolute error tolerance
`RelTol`	relative error tolerance

`quad2d` attempts to satisfy `ERRBND <= max(AbsTol,RelTol*|Q|)`. This is absolute error control when `|Q|` is sufficiently small and relative error control when `|Q|` is larger. A default tolerance value is used when a tolerance is not specified. The default value of `AbsTol` is 1e-5. The default value of `RelTol` is `100*eps(class(Q))`. This is also the minimum value of `RelTol`. Smaller `RelTol` values are automatically increased to the default value.

`MaxFunEvals`	Maximum allowed number of evaluations of `fun` reached.

The MaxFunEvals parameter limits the number of vectorized calls to fun. The default is 2000.

FailurePlot	Generate a plot if MaxFunEvals is reached.

Setting FailurePlot to true generates a graphical representation of the regions needing further refinement when MaxFunEvals is reached. No plot is generated if the integration succeeds before reaching MaxFunEvals. These (generally) 4-sided regions are mapped to rectangles internally. Clusters of small regions indicate the areas of difficulty. The default is false.

Singular	Problem may have boundary singularities

With Singular set to true, quad2d will employ transformations to weaken boundary singularities for better performance. The default is true. Setting Singular to false will turn these transformations off, which may provide a performance benefit on some smooth problems.

Examples

Example 1

Integrate $y\sin(x)+x\cos(y)$ over $\pi \le x \le 2\pi$, $0 \le y \le \pi$. The true value of the integral is $-\pi^2$.

```
Q = quad2d(@(x,y) y.*sin(x)+x.*cos(y),pi,2*pi,0,pi)
```

Example 2

Integrate $[(x+y)^{1/2}(1+x+y)^2]^{-1}$ over the triangle $0 \le x \le 1$ and $0 \le y \le 1-x$. The integrand is infinite at (0,0). The true value of the integral is $\pi/4-1/2$.

```
fun = @(x,y) 1./(sqrt(x + y) .* (1 + x + y).^2 )
```

In Cartesian coordinates:

```
ymax = @(x) 1 - x;
```

```
Q = quad2d(fun,0,1,0,ymax)
```

In polar coordinates:

```
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
rmax = @(theta) 1./(sin(theta) + cos(theta));
Q = quad2d(polarfun,0,pi/2,0,rmax)
```

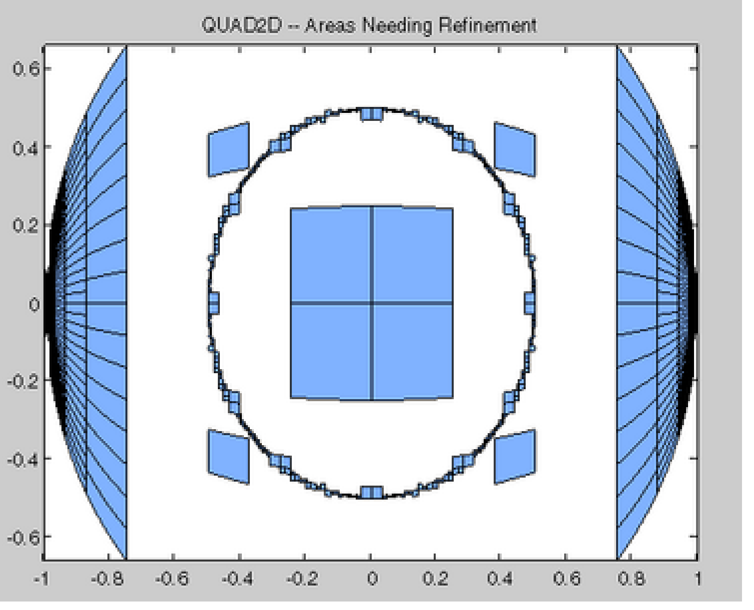
Limitations

quad2d begins by mapping the region of integration to a rectangle. Consequently, it may have trouble integrating over a region that does not have four sides or has a side that cannot be mapped smoothly to a straight line. If the integration is unsuccessful, some helpful tactics are leaving Singular set to its default value of true, changing between Cartesian and polar coordinates, or breaking the region of integration into pieces and adding the results of integration over the pieces.

For example:

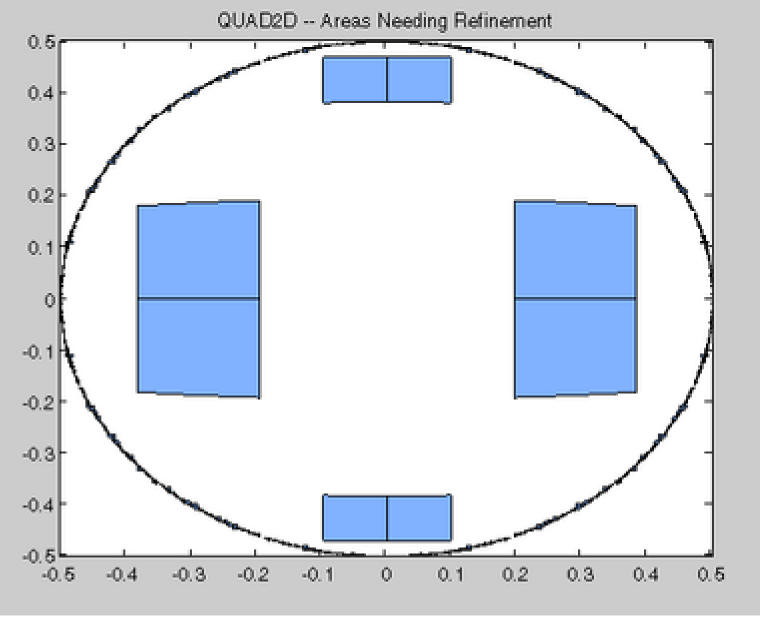
```
fun = @(x,y)abs(x.^2 + y.^2 - 0.25);
c = @(x)-sqrt(1 - x.^2);
d = @(x)sqrt(1 - x.^2);
quad2d(fun,-1,1,c,d,'AbsTol',1e-8,...
    'FailurePlot',true,'Singular',false)
Warning: Reached the maximum number of function ...
     evaluations (2000). The result fails the  ...
     global error test.
```

The failure plot shows two areas of difficulty, near the points (-1,0) and (1,0) and near the circle $x^2 + y^2 = 0.25$:



Changing the value of `Singular` to `true` will cope with the geometric singularities at `(-1,0)` and `(1,0)`. The larger shaded areas may need refinement but are probably not areas of difficulty.

```
Q = quad2d(fun,-1,1,c,d,'AbsTol',1e-8, ...
     'FailurePlot',true,'Singular',true)
Warning: Reached the maximum number of function ...
     evaluations (2000). The result passes the ...
     global error test.
```

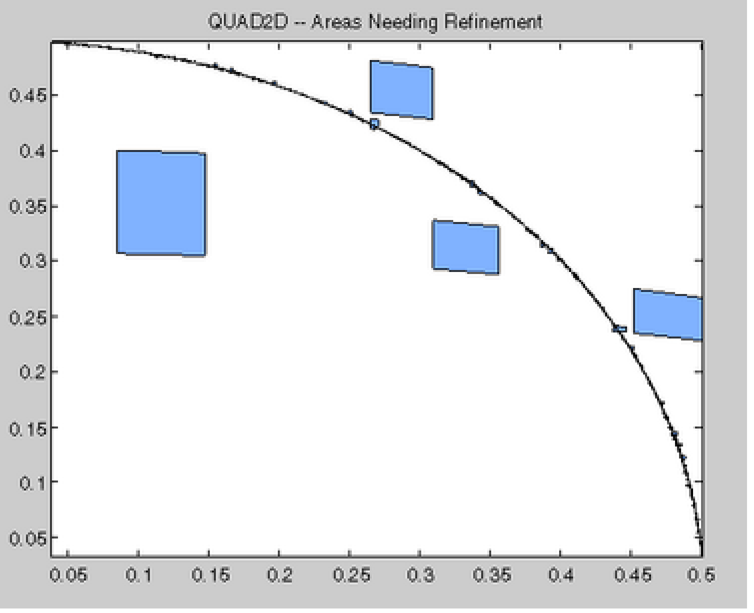


From here you can take advantage of symmetry:

```
Q = 4*quad2d(fun,0,1,0,d,'Abstol',1e-8,...
     'Singular',true, 'FailurePlot',true)
```

However, the code is still working very hard near the singularity. It may not be able to provide higher accuracy:

```
Q = 4*quad2d(fun,0,1,0,d,'Abstol',1e-10,...
     'Singular',true,'FailurePlot',true)
Warning: Reached the maximum number of function ...
     evaluations (2000). The result passes the ...
     global error test.
```



At higher accuracy, a change in coordinates may work better.

```
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
Q = 4*quad2d(polarfun,0,pi/2,0,1,'AbsTol',1e-10)
```

It is best to put the singularity on the boundary by splitting the region of integration into two parts:

```
Q1 = 4*quad2d(polarfun,0,pi/2,0,0.5,'AbsTol',5e-11);
Q2 = 4*quad2d(polarfun,0,pi/2,0.5,1,'AbsTol',5e-11);
Q = Q1 + Q2
```

References

[1] L.F. Shampine "Vectorized Adaptive Quadrature in MATLAB," *Journal of Computational and Applied Mathematics*, 211, 2008, pp.131–140.

See Also

`dblquad`, `quad`, `quadl`, `quadv`, `quadgk`, `triplequad`, `function_handle` (`@`),

quadgk

Purpose Numerically evaluate integral, adaptive Gauss-Kronrod quadrature

Syntax

```
q = quadgk(fun,a,b)
[q,errbnd] = quadgk(fun,a,b,tol)
[q,errbnd] = quadgk(fun,a,b,param1,val1,param2,val2,...)
```

Description

`q = quadgk(fun,a,b)` attempts to approximate the integral of a scalar-valued function `fun` from `a` to `b` using high-order global adaptive quadrature and default error tolerances. 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`. `fun` must be a function handle. See in the MATLAB Programming documentation for more information. Limits `a` and `b` can be `-Inf` or `Inf`. If both are finite, they can be complex. If at least one is complex, the integral is approximated over a straight line path from `a` to `b` in the complex plane.

, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`[q,errbnd] = quadgk(fun,a,b,tol)` returns an approximate bound on the absolute error, `|Q - I|`, where `I` denotes the exact value of the integral.

`[q,errbnd] = quadgk(fun,a,b,param1,val1,param2,val2,...)` performs the integration with specified values of optional parameters. The available parameters are

Parameter	Description	
'AbsTol'	Absolute error tolerance. The default value of 'AbsTol' is 1.e-10 (double), 1.e-5 (single).	quadgk attempts to satisfy errbnd <= max(AbsTol,RelTol*|Q|). This is absolute error control when |Q| is sufficiently small and relative error control when |Q| is larger. For pure absolute error control use 'AbsTol' > 0 and'RelTol'= 0. For pure relative error control use 'AbsTol' = 0. Except when using pure absolute error control, the minimum relative tolerance is 'RelTol' >= 100*eps(class(Q)).
'RelTol'	Relative error tolerance. The default value of 'RelTol' is 1.e-6 (double), 1.e-4 (single).	

Parameter	Description	
'Waypoints'	Vector of integration waypoints.	If fun(x) has discontinuities in the interval of integration, the locations should be supplied as a Waypoints vector. When a, b, and the waypoints are all real, only the waypoints between a and b are used, and they are used in sorted order. Note that waypoints are not intended for singularities in fun(x). Singular points should be handled by making them endpoints of separate integrations and adding the results. If a, b, or any entry of the waypoints vector is complex, the integration is performed over a sequence of straight line paths in the complex plane, from a to the first waypoint, from the first waypoint to the second, and so forth, and finally from the last waypoint to b.
'MaxIntervalCount'	Maximum number of intervals allowed. The default value is 650.	The 'MaxIntervalCount' parameter limits the number of intervals that quadgk uses at any one time after the first iteration. A warning is issued if quadgk returns early because of this limit. Routinely increasing this value is not recommended, but

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The quad function may be most efficient for low accuracies with nonsmooth integrands.
- The quadl function may be more efficient than quad at higher accuracies with smooth integrands.
- The quadgk function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The quadv function vectorizes quad for an array-valued fun.
- If the interval is infinite, [a,Inf), then for the integral of fun(x) to exist, fun(x) must decay as x approaches infinity, and quadgk requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but quadgk can be used if fun(x) decays fast enough.
- The quadgk function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint c like log|x-c| or $|x-c|^p$ for p >= -1/2. If the function is singular at points inside (a,b), write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with quadgk, and add the results.

Examples

Integrand with a singularity at an integration end point

Write an M-file function myfun that computes the integrand:

```
function y = myfun(x)
y = exp(x).*log(x);
```

Then pass @myfun, a function handle to myfun, to quadgk, along with the limits of integration, 0 to 1:

```
Q = quadgk(@myfun,0,1)

Q =

   -1.3179
```

Alternatively, you can pass the integrand to `quadgk` as an anonymous function handle `F`:

```
F = (@(x)exp(x).*log(x));
Q = quadgk(F,0,1);
```

Oscillatory integrand on a semi-infinite interval

Integrate over a semi-infinite interval with specified tolerances, and return the approximate error bound:

```
[q,errbnd] = quadgk(@(x)x.^5.*exp(-x).*sin(x),0,inf,'RelTol',1e-8,'Abs

q =

  -15.0000


errbnd =

  9.4386e-009
```

Contour integration around a pole

Use `Waypoints` to integrate around a pole using a piecewise linear contour:

```
Q = quadgk(@(z)1./(2*z - 1),-1-i,-1-i,'Waypoints',[1-i,1+i,-1+i])

Q =

   0.0000 + 3.1416i
```

Algorithm

quadgk implements adaptive quadrature based on a Gauss-Kronrod pair (15^{th} and 7^{th} order formulas).

Diagnostics

quadgk may issue one of the following warnings:

'Minimum step size reached' indicates that 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.

'Reached the limit on the maximum number of intervals in use' indicates that the integration was terminated before meeting the tolerance requirements and that continuing the integration would require more than MaxIntervalCount subintervals. The integral may not exist, or it may be difficult to approximate numerically. Increasing MaxIntervalCount usually does not help unless the tolerance requirements were nearly met when the integration was previously terminated.

'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.

References

[1] L.F. Shampine "Vectorized Adaptive Quadrature in MATLAB," *Journal of Computational and Applied Mathematics*, 211, 2008, pp.131–140.

See Also

quad2d, dblquad, quad, quadl, quadv, triplequad, function_handle (@),

quadl

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 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.

, 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.

The list below contains information to help you determine which quadrature function in `MATLAB` to use:

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function may be more efficient than `quad` at higher accuracies with smooth integrands.

- The `quadgk` function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, `[a,Inf)`, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or `|x-c|`^{p} for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Examples

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

```
Q = quadl(@myfun,0,2);
```

where the M-file `myfun.m` is

```
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

`quad2d`, `dblquad`, `quad`, `quadgk`, `triplequad`, `function_handle` (`@`),

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 in the MATLAB Programming documentation for more information. The function `Y = fun(x)` should accept a scalar argument `x` and return an array result `Y`, whose components are the integrands evaluated at `x`. Limits `a` and `b` must be finite.

, 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`.

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.

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

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

The list below contains information to help you determine which quadrature function in `MATLAB` to use:

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.

- The `quadl` function may be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, `[a,Inf)`, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or `|x-c|`^{p} for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Example

For the parameterized array-valued function `myarrayfun`, defined by

```
function Y = myarrayfun(x,n)
Y = 1./((1:n)+x);
```

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) = quadv(@(x)myscalarfun(x,k),0,1);
end
```

where `myscalarfun` is:

```
function y = myscalarfun(x,k)
y = 1./(k+x);
```

See Also

`quad`, `quad2d`, `quadgk`, `quadl`, `dblquad`, `triplequad`, `function_handle` (`@`)

questdlg

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)
button = questdlg('qstring','title', ..., options)
```

Description

`button = questdlg('qstring')` displays a modal dialog box 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 without making a choice, `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.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`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'`.

When *`default`* is specified, but is not set to one of the button names, pressing the **Enter** key displays a warning and the dialog remains open.

`button = questdlg('qstring','title', ..., options)` replaces the string *`default`* with a structure, `options`. The structure specifies which button string is the default answer, and whether to use TeX to interpret the question string, `qstring`. Button strings and dialog titles cannot use TeX interpretation. The `options` structure must include the fields `Default` and `Interpreter`, both strings. It can include other fields, but `questdlg` does not use them. You can set `Interpreter` to `'none'` or `'tex'`. If the `Default` field does not contain a valid button name, a command window warning is issued and the dialog box does not respond to pressing the **Enter** key.

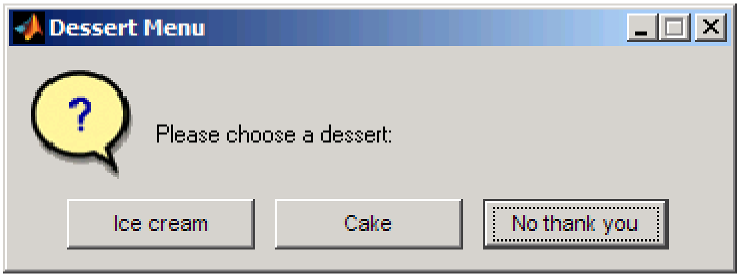
Examples

Example 1

Create a dialog that requests a dessert preference and encode the resulting choice as an integer.

```
% Construct a questdlg with three options
choice = questdlg('Please choose a dessert:', ...
 'Dessert Menu', ...
 'Ice cream','Cake','No thank you','No thank you');
% Handle response
switch choice
    case 'Ice cream'
        disp([choice ' coming right up.'])
        dessert = 1;
        break
    case 'Cake'
        disp([choice ' coming right up.'])
        dessert = 2;
       break
```

```
        case 'No thank you'
            disp('I''ll bring you your check.')
            dessert = 0;
end
```

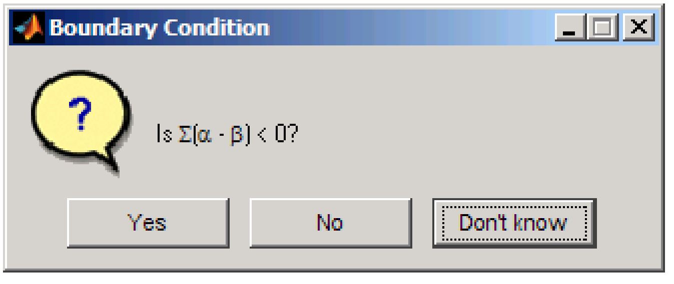


The case statements can contain white space but are case-sensitive.

Example 2

Specify an options structure to use the TeX interpreter to format a question.

```
options.Interpreter = 'tex';
% Include the desired Default answer
options.Default = 'Don''t know';
% Create a TeX string for the question
qstring = 'Is \Sigma(\alpha - \beta) < 0?';
choice = questdlg(qstring,'Boundary Condition',...
    'Yes','No','Don''t know',options)
```



See Also

`dialog`, `errordlg`, `helpdlg`, `inputdlg`, `listdlg`, `msgbox`, `warndlg`

`figure`, `textwrap`, `uiwait`, `uiresume`

Predefined Dialog Boxes for related functions

quit

Purpose

Terminate MATLAB program

GUI Alternatives

As an alternative to the quit function, use the Close box or select **File > Exit MATLAB** in the MATLAB desktop.

Syntax

```
quit
quit cancel
quit force
```

Description

`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 `finish.m`, if `finish.m` exists. The workspace is not automatically saved by `quit`. To save the workspace or perform other actions when quitting, create a `finish.m` file to perform those actions. For example, you can display a custom dialog box to confirm quitting using a `finish.m` file—see the following examples for details. If an error occurs while `finish.m` is running, `quit` is canceled so that you can correct your `finish.m` file without losing your workspace.

`quit cancel` is for use in `finish.m` and cancels quitting. It has no effect anywhere else.

`quit force` bypasses `finish.m` and terminates MATLAB. Use this to override `finish.m`, for example, if an errant `finish.m` will not let you quit.

Remarks

When using Handle Graphics objects in `finish.m`, use `uiwait`, `waitfor`, or `drawnow` so that figures are visible. See the reference pages for these functions for more information.

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**.

Examples

Two sample `finish.m` files are included with MATLAB. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` to use it.

- `finishsav.m`—Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m`—Displays a dialog allowing you to cancel quitting; it uses `quit cancel` and contains the following code:

```
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`

Quit (COM)

Purpose Terminate MATLAB Automation server

Syntax

MATLAB Client

```
h.Quit
Quit(h)
invoke(h, 'Quit')
```

IDL Method Signature

```
void Quit(void)
```

Microsoft Visual Basic Client

```
Quit
```

Description `Quit` terminates the MATLAB server session attached to handle `h`.

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.

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(...,LineSpec)
quiver(...,LineSpec,'filled')
quiver(axes_handle,...)
h = quiver(...)
hlines = quiver('v6',...)
```

Description

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

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

`quiver(x,y,u,v)` plots vectors as arrows at the coordinates specified in each corresponding pair of elements in `x` and `y`. The matrices `x`, `y`, `u`, and `v` must all be the same size and contain corresponding position and velocity components. However, `x` and `y` can also be vectors, as explained in the next section. 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(...,LineSpec)` specifies line style, marker symbol, and color using any valid `LineSpec`. `quiver` draws the markers at the origin of the vectors.

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

`quiver(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into 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.

Note The v6 option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

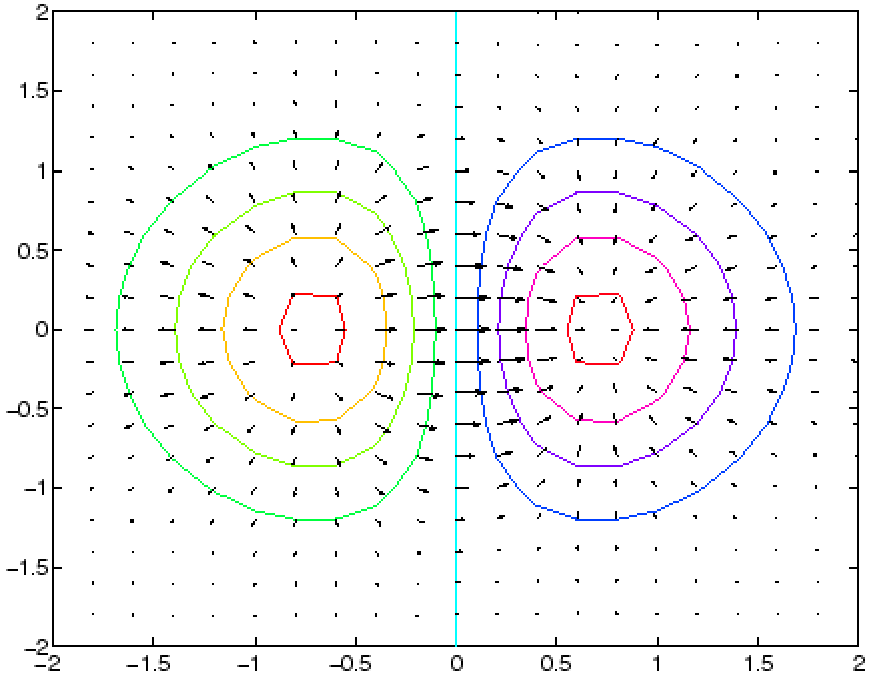
See Plot Objects and Backward Compatibility for more information.

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 1-94 for related functions

Two-Dimensional Quiver Plots for more examples

`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(...,LineSpec)
quiver3(...,LineSpec,'filled')
quiver3(axes_handle,...)
h = quiver3(...)
```

Description

A three-dimensional quiver plot displays vectors with components `(u,v,w)` at the points `(x,y,z)`, where `u,v,w,x,y,` and `z` all have real (non-complex) values.

`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, and then multiplies them by `scale`. `scale = 2` doubles their relative length, and `scale = 0.5` halves them. Use `scale = 0` to plot the vectors without the automatic scaling.

`quiver3(...,LineSpec)` specifies line type and color using any valid `LineSpec`.

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

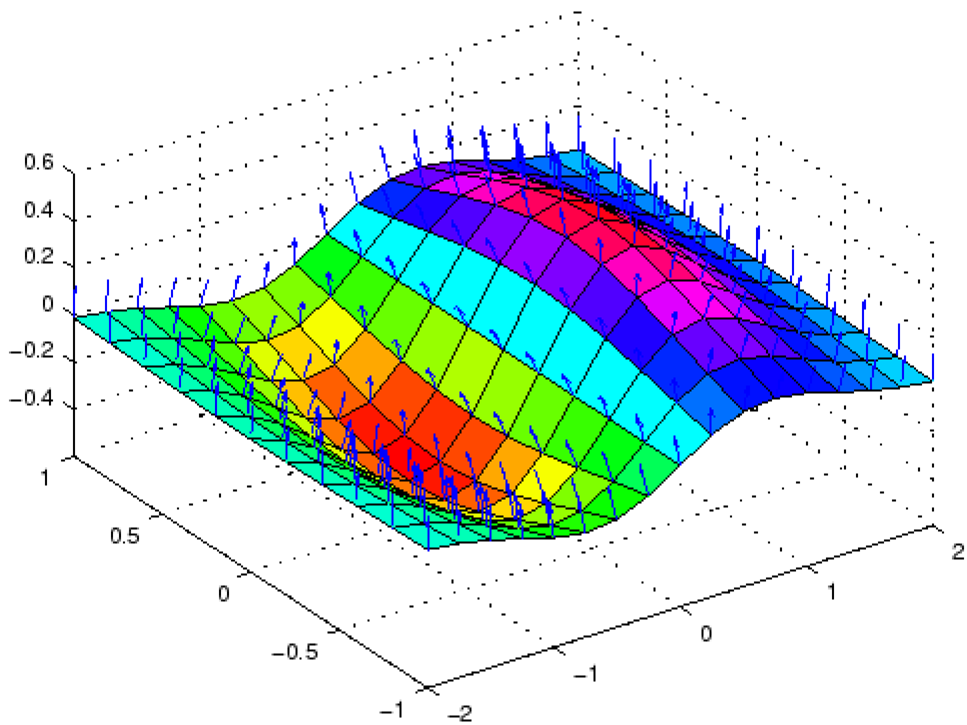
`quiver3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`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 1-94 for related functions

Three-Dimensional Quiver Plots for more examples

Quivergroup Properties

Purpose

Define 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.

`Annotation`

`hg.Annotation` object Read Only

Control the display of quivergroup objects in legends. The `Annotation` property enables you to specify whether this quivergroup object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the quivergroup object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the quivergroup object in a legend as one entry, but not its children objects
`off`	Do not include the quivergroup or its children in a legend (default)
`children`	Include only the children of the quivergroup as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See for more information and examples.

`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 if the objects are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

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

This property can be

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

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

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

`Children`
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

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

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`

ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

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

`CreateFcn`

string or function handle

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

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

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

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 for information on how to use function handles to define the callback function.

`DeleteFcn`

string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

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

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

See the `BeingDeleted` property for related information.

`DisplayName`
string (default is empty string)

String used by legend for this quivergroup object. The `legend` function uses the string defined by the `DisplayName` property to label this quivergroup object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this quivergroup object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See for more examples.

`EraseMode`

`{normal} | none | xor | background`

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

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

Printing with Nonnormal Erase Modes

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

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

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

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

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

`HitTest`
`{on} | off`

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

`HitTestArea`
`on | {off}`

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click th eobject's lines or markers (excluding the baseline, if any) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineStyle`

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

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

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

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

`LineWidth`

scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`

character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the

`Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Five-pointed star (pentagram)
`h`	Six-pointed star (hexagram)
`none`	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`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`
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`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. You can define `Tag` as any string.

For example, you might create an areaseries object and set the `Tag` property.

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

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

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

`Type`

string (read only)

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

`UserData`

array

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

`Visible`

`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`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

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.

`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 is evaluated in the base workspace to generate the `XData`.

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

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

See the `refreshdata` reference page for more information.

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

`YData`

vector or matrix

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 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 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] = qz(A,B)
[AA,BB,Q,Z,V,W] = qz(A,B)
qz(A,B,flag)
```

Description

The `qz` function gives access to intermediate results in the computation of generalized eigenvalues.

`[AA,BB,Q,Z] = 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] = qz(A,B)` also produces matrices `V` and `W` whose columns are generalized eigenvectors.

`qz(A,B,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`, $\alpha = \mathrm{diag}(AA)$ and $\beta = \mathrm{diag}(BB)$, are the generalized eigenvalues that satisfy

$$A * V * \beta = B * V * \alpha$$
$$\beta * W' * A = \alpha * W' * B$$

The eigenvalues produced by

$$\lambda = \mathrm{eig}(A, B)$$

are the ratios of the αs and βs.

$$\lambda = \alpha./\beta$$

If AA is not triangular, it is necessary to further reduce the 2-by-2 blocks to obtain the eigenvalues of the full system.

See Also

eig

Purpose

Uniformly distributed pseudorandom numbers

Syntax

```
r = rand(n)
rand(m,n)
rand([m,n])
rand(m,n,p,...)
rand([m,n,p,...])
rand
rand(size(A))
r = rand(..., 'double')
r = rand(..., 'single')
```

Description

`r = rand(n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval (0,1). `rand(m,n)` or `rand([m,n])` returns an `m`-by-`n` matrix. `rand(m,n,p,...)` or `rand([m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `rand` returns a scalar. `rand(size(A))` returns an array the same size as `A`.

`r = rand(..., 'double')` or `r = rand(..., 'single')` returns an array of uniform values of the specified class.

Note Note: The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by rand is determined by the internal state of the uniform pseudorandom number generator that underlies `rand`, `randi`, and `randn`. The default random number stream properties can be set using `@RandStream` methods. See `@RandStream` for details about controlling the default stream.

Resetting the default stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties. Since the random number generator is initialized to the same state every time MATLAB software starts up, `rand`, `randn`, and `randi` will generate the same sequence of numbers in each session until the state is changed.

Note In versions of MATLAB prior to 7.7, you controlled the internal state of the random number stream used by `rand` by calling `rand` directly with the `'seed'`, `'state'`, or `'twister'` keywords. That syntax is still supported for backwards compatibility, but is deprecated. For version 7.7, use the default stream as described in the `@RandStream` reference documentation.

Examples

Generate values from the uniform distribution on the interval `[a, b]`.

```
r = a + (b-a).*rand(100,1);
```

Replace the default stream at MATLAB startup, using a stream whose seed is based on `clock`, so that `rand` will return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB session.

```
RandStream.setDefaultStream ...
     (RandStream('mt19937ar','seed',sum(100*clock)));
rand(1,5)
```

Save the current state of the default stream, generate 5 values, restore the state, and repeat the sequence.

```
defaultStream = RandStream.getDefaultStream;
savedState = defaultStream.State;
u1 = rand(1,5)
defaultStream.State = savedState;
u2 = rand(1,5) % contains exactly the same values as u1
```

See Also

`randi`, `randn`, `@RandStream`, `rand (RandStream)`, `getDefaultStream (RandStream)`, `sprand`, `sprandn`, `randperm`

Purpose Uniformly distributed random numbers

Class `@RandStream`

Syntax

```
r = rand(s,n)
rand(s,m,n)
rand(s,[m,n])
rand(s,m,n,p,...)
rand(s,[m,n,p,...])
rand(s)
rand(s,size(A))
r = rand(..., 'double')
r = rand(..., 'single')
```

Description

`r = rand(s,n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval (0,1). The values are drawn from the random stream `s`. `rand(s,m,n)` or `rand(s,[m,n])` returns an `m`-by-`n` matrix. `rand(s,m,n,p,...)` or `rand(s,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `rand(s)` returns a scalar. `rand(s,size(A))` returns an array the same size as `A`.

`r = rand(..., 'double')` or `r = rand(..., 'single')` returns an array of uniform values of the specified class.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `rand` is determined by the internal state of the random number stream `s`. Resetting that stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also

rand, @RandStream, randi (RandStream), randn (RandStream), randperm (RandStream)

Purpose

Uniformly distributed pseudorandom integers

Syntax

```
randi(imax)
r = randi(imax,n)
randi(imax,m,n)
randi(imax,[m,n])
randi(imax,m,n,p,...)
randi(imax,[m,n,p,...])
randi(imax,size(A))
r = randi([imin,imax],...)
r = randi(..., classname)
```

Description

`randi(imax)` returns a random integer on the interval1:`imax`. `r = randi(imax,n)` returns an `n`-by-`n` matrix containing pseudorandom integer values drawn from the discrete uniform distribution on `1:imax`. `randi(imax,m,n)` or `randi(imax,[m,n])` returns an `m`-by-`n` matrix. `randi(imax,m,n,p,...)` or `randi(imax,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `randi(imax,size(A))` returns an array the same size as `A`.

`r = randi([imin,imax],...)` returns an array containing integer values drawn from the discrete uniform distribution on `imin:imax`.

`r = randi(..., classname)` returns an array of integer values of class `classname`. `classname` does not support 64-bit integers.

Note Note: The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randi` is determined by the internal state of the uniform pseudorandom number generator that underlies `rand`, `randi`, and `randn`. `randi` uses one uniform value from that default stream to generate each integer value. Control the default stream using its properties and methods. See `@RandStream` for details about the default stream.

Resetting the default stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties. Since the random number generator is initialized to the same state every time MATLAB software starts up, `rand`, `randn`, and `randi` will generate the same sequence of numbers in each session until the state is changed.

Examples

Generate integer values from the uniform distribution on the set 1:10.

```
r = randi(10,100,1);
```

Generate an integer array of integers drawn uniformly from 1:10.

```
r = randi(10,100,1,'uint32');
```

Generate integer values drawn uniformly from -10:10.

```
r = randi([-10 10],100,1);
```

Replace the default stream at MATLAB startup, using a stream whose seed is based on `clock`, so that `randi` will return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB session.

```
RandStream.setDefaultStream ...
     (RandStream('mt19937ar','seed',sum(100*clock)));
randi(100,1,5)
```

Save the current state of the default stream, generate 5 integer values, restore the state, and repeat the sequence.

```
defaultStream = RandStream.getDefaultStream;
savedState = defaultStream.State;
i1 = randi(10,1,5)
defaultStream.State = savedState;
i2 = randi(10,1,5) %contains exactly the same values as i1
```

See Also

`rand`, `randn`, `@RandStream`, `randi (RandStream)`, `getDefaultStream (RandStream)`

randi (RandStream)

Purpose Uniformly distributed pseudorandom integers

Class `@RandStream`

Syntax

```
r = randi(s,imax,n)
randi(s,imax,m,n)
randi(s,imax,[m,n])
randi(s,imax,m,n,p,...)
randi(s,imax,[m,n,p,...])
randi(s,imax)
randi(s,imax,size(A))
r = randi(s,[imin,imax],...)
r = randi(..., classname)
```

Description

`r = randi(s,imax,n)` returns an `n`-by-`n` matrix containing pseudorandom integer values drawn from the discrete uniform distribution on `1:imax`. `randi` draws those values from the random stream `s`. `randi(s,imax,m,n)` or `randi(s,imax,[m,n])` returns an `m`-by-`n` matrix. `randi(s,imax,m,n,p,...)` or `randi(s,imax,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `randi(s,imax)` returns a scalar. `randi(s,imax,size(A))` returns an array the same size as `A`.

`r = randi(s,[imin,imax],...)` returns an array containing integer values drawn from the discrete uniform distribution on `imin:imax`.

`r = randi(..., classname)` returns an array of integer values of class `classname`. `classname` does not support 64-bit integers.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randi` is determined by the internal state of the random stream `s`. `randi` uses one uniform value from `s` to generate each integer value. Resetting `s` to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also

`rand`, `@RandStream`, `rand (RandStream)`, `randn (RandStream)`, `randperm (RandStream)`

randn

Purpose

Normally distributed pseudorandom numbers

Syntax

```
r = randn(n)
randn(m,n)
randn([m,n])
randn(m,n,p,...)
randn([m,n,p,...])
randn(size(A))
r = randn(..., 'double')
r = randn(..., 'single')
```

Description

`r = randn(n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard normal distribution. `randn(m,n)` or `randn([m,n])` returns an `m`-by-`n` matrix. `randn(m,n,p,...)` or `randn([m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `randn` returns a scalar. `randn(size(A))` returns an array the same size as `A`.

`r = randn(..., 'double')` or `r = randn(..., 'single')` returns an array of normal values of the specified class.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randn` is determined by the internal state of the uniform pseudorandom number generator that underlies `rand`, `randi`, and `randn`. `randn` uses one or more uniform values from that default stream to generate each normal value. Control the default stream using its properties and methods. See `@RandStream` for details about the default stream.

Resetting the default stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties. Since the random number generator is initialized to the same state every time MATLAB software starts up, `rand`, `randn`, and `randi` will generate the same sequence of numbers in each session until the state is changed.

Note In versions of MATLAB prior to 7.7, you controlled the internal state of the random number stream used by `randn` by calling `randn` directly with the `'seed'` or `'state'` keywords. That syntax is still supported for backwards compatibility, but is deprecated. For version 7.7, use the default stream as described in the `@RandStream` reference documentation.

Examples

Generate values from a normal distribution with mean 1 and standard deviation 2.

```
r = 1 + 2.*randn(100,1);
```

Generate values from a bivariate normal distribution with specified mean vector and covariance matrix.

```
mu = [1 2];
Sigma = [1 .5; .5 2]; R = chol(Sigma);
z = repmat(mu,100,1) + randn(100,2)*R;
```

Replace the default stream at MATLAB startup, using a stream whose seed is based on `clock`, so that `randn` will return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB session.

```
RandStream.setDefaultStream ...
     (RandStream('mt19937ar','seed',sum(100*clock)));
randn(1,5)
```

Save the current state of the default stream, generate 5 values, restore the state, and repeat the sequence.

```
defaultStream = RandStream.getDefaultStream;
savedState = defaultStream.State;
```

```
z1 = randn(1,5)
defaultStream.State = savedState;
z2 = randn(1,5) % contains exactly the same values as z1
```

See Also

rand, randi, @RandStream, randn (RandStream), getDefaultStream (RandStream)

Purpose

Normally distributed pseudorandom numbers

Class

`@RandStream`

Syntax

```
randn(s,m,n)
randn(s,[m,n])
randn(s,m,n,p,...)
randn(s,[m,n,p,...])
randn(s)
randn(s,size(A))
r = randn(..., 'double')
r = randn(..., 'single')
```

Description

`r = randn(s,n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard normal distribution. `randn` draws those values from the random stream s. `randn(s,m,n)` or `randn(s,[m,n])` returns an `m`-by-`n` matrix. `randn(s,m,n,p,...)` or `randn(s,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `randn(s)` returns a scalar. `randn(s,size(A))` returns an array the same size as `A`.

`r = randn(..., 'double')` or `r = randn(..., 'single')` returns an array of uniform values of the specified class.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randn` is determined by the internal state of the random stream `s`. `randn` uses one or more uniform values from `s` to generate each normal value. Resetting that stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also

`randn`, `@RandStream`, `rand (RandStream)`, `randi (RandStream)`

randpermg

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 the state of the default random number stream.

Examples `randperm(6)` might be the vector

```
[3  2  6  4  1  5]
```

or it might be some other permutation of `1:6`.

See Also `permute`

Purpose Random permutation

Class `@RandStream`

Syntax `randperm(s,n)`

Description `randperm(s,n)` generates a random permutation of the integers from 1 to `n`. For example, `randperm(s,6)` might be `[2 4 5 6 1 3]`. `randperm(s,n)` uses random values drawn from the random number stream `s`.

See Also `permute`, `@RandStream`

Purpose

Random number stream

Constructor

`RandStream (RandStream)`

Description

Pseudorandom numbers in MATLAB come from one or more random number streams. The simplest way to generate arrays of random numbers is to use `rand`, `randn`, or `randi`. These functions all rely on the same stream of uniform random numbers, known as the default stream. You can create other stream objects that act separately from the default stream, and you can use their `rand`, `randi`, or `randn` methods to generate arrays of random numbers. You can also create a random number stream and make it the default stream.

To create a single random number stream, use either the `RandStream` constructor or the `RandStream.create` factory method. To create multiple independent random number streams, use `RandStream.create`.

`stream = RandStream.getDefaultStream` returns the default random number stream, that is, the one currently used by the `rand`, `randi`, and `randn` functions.

`prevstream = RandStream.setDefaultStream(stream)` returns the current default stream, and designates the random number stream `stream` as the new default to be used by the `rand`, `randi`, and `randn` functions.

A random number stream `s` has properties that control its behavior. Access or assign to a property using `p= s.Property` or `s.Property = p`. The following table lists defined properties:

Properties

Property	Description
Type	(Read-only) Generator algorithm used by the stream. The list of possible generators is given by `RandStream.list`.

Property	Description
Seed	(Read-only) Seed value used to create the stream.
NumStreams	(Read-only) Number of streams in the group in which the current stream was created.
StreamIndex	(Read-only) Index of the current stream from among the group of streams with which it was created.
State	Internal state of the generator. You should not depend on the format of this property. The value you assign to `S.State` must be a value read from `S.State` previously.
Substream	Index of the substream to which the stream is currently set. The default is 1. Multiple substreams are not supported by all generator types; the multiplicative lagged Fibonacci generator (`mlfg6331_64`) and combined multiple recursive generator (`mrg32k3a`) support multiple streams.
RandnAlg	Algorithm used by `randn(s, ...)` to generate normal pseudorandom values. Possible values are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.

Property	Description
Antithetic	Logical value indicating whether `S` generates antithetic pseudorandom values. For uniform values, these are the usual values subtracted from 1. The default is false.
FullPrecision	Logical value indicating whether `S` generates values using its full precision. Some generators can create pseudorandom values faster, but with fewer random bits, if `FullPrecision` is false. The default is true.

Methods

Method	Description
`RandStream`	Create a random number stream
`RandStream.create`	Create multiple independent random number streams
`get`	Get the properties of a random stream object
`list`	List available random number generator algorithms
`set`	Set random stream property
`RandStream.getDefaultStream`	Get the default random number stream
`RandStream.setDefaultStream`	Set the default random number stream
`reset`	Reset a stream to its initial internal state

Method	Description
rand	Pseudorandom numbers from a uniform distribution
randn	Pseudorandom numbers from a standard normal distribution
randi	Pseudorandom integers from a uniform discrete distribution
randperm	Random permutation of a set of values

See Also

rand, randn, randi, rand (RandStream), randn (RandStream), randi (RandStream)

RandStream (RandStream)

Purpose

Random number stream

Class

`@RandStream`

Syntax

```
s = RandStream('gentype')
[...]=RandStream('gentype','param1',val1,'param2',val2,...)
```

Description

`s = RandStream('gentype')` creates a random number stream that uses the uniform pseudorandom number generator algorithm specified by `gentype`.`[...]=RandStream('gentype','param1',val1,'param2',val2,...)` allows you to specify optional parameter name/value pairs to control creation of the stream. Options for `gentype` are given by `RandStream.list`.

Parameters are for `RandStream` are:

Parameter	Description
`Seed`	Nonnegative scalar integer with which to initialize all streams. Default is 0. Seeds must be an integer between 0 and 2^{32}.
`RandnAlg`	Algorithm used by `randn(s, ...)` to generate normal pseudorandom values. Possible values are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.

Examples

Construct a random stream object using the combined multiple recursive generator and generate 5 uniformly distributed values from that stream.

```
stream=RandStream('mrg32k3a');
rand(stream,1,5)
```

Construct a random stream object using the multiplicative lagged Fibonacci generator and generate 5 normally distributed values using the polar algorithm.

```
stream=RandStream('mlfg6331_64','RandnAlg','Polar');
randn(stream,1,5)
```

See Also

`@RandStream`, `rand (RandStream)`, `randn (RandStream)`, `randi (RandStream)`, `getDefaultStream (RandStream)`

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 software 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 =
```

```
    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

$$\frac{n}{d} = d_1 + \cfrac{1}{d_2 + \cfrac{1}{\left(d_3 + \ldots + \cfrac{1}{d_k}\right)}}$$

The ds are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when `X = sqrt(2)`. For `x = sqrt(2)` , the error with `k` terms is about `2.68*(.173)^k`, so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.

See Also

`format`

Purpose

Create rubberband box for area selection

Syntax

```
rbbox
rbbox(initialRect)
rbbox(initialRect,fixedPoint)
rbbox(initialRect,fixedPoint,stepSize)
finalRect = rbbox(...)
```

Description

`rbbox` initializes and tracks a rubberband box in the current figure. It sets the initial rectangular size of the box to 0, anchors the box at the figure's `CurrentPoint`, and begins tracking from this point.

`rbbox(initialRect)` specifies the initial location and size of the rubberband box as `[x y width height]`, where `x` and `y` define the lower left corner, and `width` and `height` define the size. `initialRect` is in the units specified by the current figure's `Units` property, and measured from the lower left corner of the figure window. The corner of the box closest to the pointer position follows the pointer until `rbbox` receives a button-up event.

`rbbox(initialRect,fixedPoint)` specifies the corner of the box that remains fixed. All arguments are in the units specified by the current figure's `Units` property, and measured from the lower left corner of the figure window. `fixedPoint` is a two-element vector, `[x y]`. The tracking point is the corner diametrically opposite the anchored corner defined by `fixedPoint`.

`rbbox(initialRect,fixedPoint,stepSize)` specifies how frequently the rubberband box is updated. When the tracking point exceeds `stepSize` figure units, `rbbox` redraws the rubberband box. The default stepsize is `1`.

`finalRect = rbbox(...)` returns a four-element vector, `[x y width height]`, where `x` and `y` are the *x* and *y* components of the lower left corner of the box, and `width` and `height` are the dimensions of the box.

Remarks

`rbbox` is useful for defining and resizing a rectangular region:

- 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 1-104 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.

read (mmreader)

Purpose Read video frame data from multimedia reader object

Syntax

```
video = read(obj)
video = read(obj, index)
```

Description

video = read(*obj*) reads in all video frames from the associated file. *video* is an *H*-by-*W*-by-*B*-by-*F* matrix where *H* is the image frame height, *W* is the image frame width, *B* is the number of bands in the image (e.g., 3 for RGB), and *F* is the number of frames read in. *video* contains uint8 data representing RGB24 video frames.

video = read(*obj*, *index*) reads only the frames specified by *index*, where the first frame number is 1. *index* can be a single index, or a two-element array representing an index range of the video stream.

For example:

```
video = read(obj, 1);         % first frame only
video = read(obj, [1 10]);    % first 10 frames
video = read(obj, Inf);       % last frame only
video = read(obj, [50 Inf]);  % frame 50 through end of file
```

For information about specifying the index in variable frame rate files, see mmreader.

Examples

Read and play back the movie file xylophone.mpg.

```
xyloObj = mmreader('xylophone.mpg');

nFrames = xyloObj.NumberOfFrames;
vidHeight = xyloObj.Height;
vidWidth = xyloObj.Width;

% Preallocate movie structure.
mov(1:nFrames) = ...
    struct('cdata', zeros(vidHeight, vidWidth, 3, 'uint8'),...
           'colormap', []);
```

```
% Read one frame at a time.
for k = 1 : nFrames
    mov(k).cdata = read(xyloObj, k);
end

% Size a figure based on the video's width and height.
hf = figure;
set(hf, 'position', [150 150 vidWidth vidHeight])

% Play back the movie once at the video's frame rate.
movie(hf, mov, 1, xyloObj.FrameRate);
```

See Also

get (hgsetget), mmreader, movie, set (hgsetget)

Tiff.read

Purpose

Read entire image

Syntax

```
imageData = tiffobj.read()
[Y,Cb,Cr] = tiffobj.read()
```

Description

`imageData = tiffobj.read()` reads the image data from the current image file directory (IFD) in the TIFF file associated with the `Tiff` object, `tiffobj`.

`[Y,Cb,Cr] = tiffobj.read()`reads the YCbCr component data from the current directory in the TIFF file. Depending upon the values of the `YCbCrSubSampling` tag, the size of the `Cb` and `Cr` channels might differ from the `Y` channel.

Examples

Open a `Tiff` object and read data from the TIFF file:

```
t = Tiff('mytif.tif', 'r');
imageData = t.read();
```

See Also

`Tiff.write`

Tutorials

-
-

Purpose

Read data asynchronously from device

Syntax

```
readasync(obj)
readasync(obj,size)
```

Description

`readasync(obj)` initiates an asynchronous read operation on the serial port object, `obj`.

`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` on a Windows platform. It 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)
out =
2.0399999619E0
fclose(s)
```

See Also

Functions

`fopen`, `stopasync`

Properties

`BytesAvailable`, `BytesAvailableFcn`, `ReadAsyncMode`, `Status`, `TransferStatus`

Tiff.readEncodedStrip

Purpose Read data from specified strip

Syntax

```
stripData = tiffobj.readEncodedStrip(stripNumber)
[Y,Cb,Cr] = tiffobj.readEncodedStrip(stripNumber)
```

Description

`stripData = tiffobj.readEncodedStrip(stripNumber)` reads data from the strip specified by `stripNumber`. Strip numbers are one-based numbers.

`[Y,Cb,Cr] = tiffobj.readEncodedStrip(stripNumber)` reads YCbCr component data from the specified strip. The size of the chrominance components `Cb` and `Cr` might differ from the size of the luminance component `Y` depending on the value of the `YCbCrSubSampling` tag.

`readEncodeStrip` clips the last strip, if the strip extends past the `ImageLength` boundary.

Examples

Open a `Tiff` object and read a strip of data. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r');
%
% Check if image is tiled or stipped.
if ~t.isTiled()
  data = t.readEncodedStrip(1);
end
```

References

This method corresponds to the `TIFFReadEncodedStrip` function in the LibTIFF C API. To use this method, you must be familiar with LibTIFF version 3.7.1, as well as the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.readEncodedTile` | `Tiff.isTiled`

Tutorials

-
-

Purpose

Read data from specified tile

Syntax

```
tileData = tiffobj.readEncodedTile(tileNumber)
[Y,Cb,Cr] = tiffobj.readEncodedTile(tileNumber)
```

Description

`tileData = tiffobj.readEncodedTile(tileNumber)` reads data from the tile specified by `tileNumber`. Tile numbers are one-based numbers.

`[Y,Cb,Cr] = tiffobj.readEncodedTile(tileNumber)` reads YCbCr component data from the specified tile. The size of the chrominance components `Cb` and `Cr` might differ from the size of the luminance component `Y`, depending on the value of the `YCbCrSubSampling` tag.

`readEncodedTile` clips tiles on the last row or right-most column of an image if the tile extends past the `ImageLength` and `ImageLength` boundaries.

Examples

Open a `Tiff` object and read a tile of data. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r');
%
% Check if image is tiled or stipped.
if t.isTiled()
  data = t.readEncodedTile(1);
end
```

References

This method corresponds to the `TIFFReadEncodedTile` function in the LibTIFF C API. To use this method, you must be familiar with LibTIFF version 3.7.1, as well as the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.readEncodedStrip` | `Tiff.isTiled`

Tutorials

-
-

Purpose Real part of complex number

Syntax

```
X = real(Z)
```

Description `X = real(Z)` returns the real part of the elements of the complex array `Z`.

Examples `real(2+3*i)` is `2`.

See Also `abs`, `angle`, `conj`, `i`, `j`, `imag`

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

```
n = realmax
```

Description

`n = realmax` returns the largest floating-point number representable on your computer. Anything larger overflows.

`realmax('double')` is the same as `realmax` with no arguments.

`realmax('single')` is the largest single precision floating point number representable on your computer. Anything larger overflows to `single(Inf)`.

Examples

`realmax` is one bit less than 2^{1024} or about `1.7977e+308`.

Algorithm

The `realmax` function is equivalent to `pow2(2-eps,maxexp)`, where `maxexp` is the largest possible floating-point exponent.

Execute `type realmax` to see `maxexp` for various computers.

See Also

`eps`, `realmin`, `intmax`

realmin

Purpose Smallest positive normalized floating-point number

Syntax

```
n = realmin
```

Description `n = realmin` returns the smallest positive normalized floating-point number on your computer. Anything smaller underflows or is an IEEE “denormal.”

`REALMIN('double')` is the same as `REALMIN` with no arguments.

`REALMIN('single')` is the smallest positive normalized single precision floating point number on your computer.

Examples `realmin` is `2^(-1022)` or about `2.2251e-308`.

Algorithm The `realmin` function is equivalent to `pow2(1,minexp)` where `minexp` is the smallest possible floating-point exponent.

Execute `type realmin` to see `minexp` for various computers.

See Also `eps`, `realmax`, `intmin`

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

```
record(obj)
record(obj,'switch')
```

Description

`record(obj)` toggles the recording state for the serial port object, `obj`.

`record(obj,'switch')` initiates or terminates recording for `obj`. *`switch`* can be `on` or `off`. If *`switch`* is `on`, recording is initiated. If *`switch`* is `off`, recording is terminated.

Remarks

Before you can record information to disk, `obj` must be connected 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 record information while `obj` is not connected to the device. Each serial port object must record information to a separate file. Recording is automatically terminated when `obj` is disconnected from the device with `fclose`.

The `RecordName` and `RecordMode` properties are read-only while `obj` is recording, and must be configured before using `record`.

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.

Example

This example creates the serial port object `s` on a Windows platform. It connects `s` to the device, configures `s` to record information to a file, writes and reads text data, and then disconnects `s` from the device.

```
s = serial('COM1');
fopen(s)
s.RecordDetail = 'verbose';
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 2-D rectangle object

Syntax

```
rectangle
rectangle('Position',[x,y,w,h])
rectangle(...,'Curvature',[x,y])
h = rectangle(...)
```

Description

`rectangle` draws a rectangle with `Position [0,0,1,1]` and `Curvature [0,0]` (i.e., no curvature).

`rectangle('Position',[x,y,w,h])` draws the rectangle from the point `x,y` and having a width of `w` and a height of `h`. Specify values in axes data units.

Note that, to display a rectangle in the specified proportions, you need to set the axes data aspect ratio so that one unit is of equal length along both the x and y axes. You can do this with the command `axis equal` or `daspect([1,1,1])`.

`rectangle(...,'Curvature',[x,y])` specifies the curvature of the rectangle sides, enabling it to vary from a rectangle to an ellipse. The horizontal curvature `x` is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature `y` is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of `x` and `y` can range from `0` (no curvature) to `1` (maximum curvature). A value of `[0,0]` creates a rectangle with square sides. A value of `[1,1]` creates an ellipse. If you specify only one value for `Curvature`, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

`h = rectangle(...)` returns the handle of the rectangle object created.

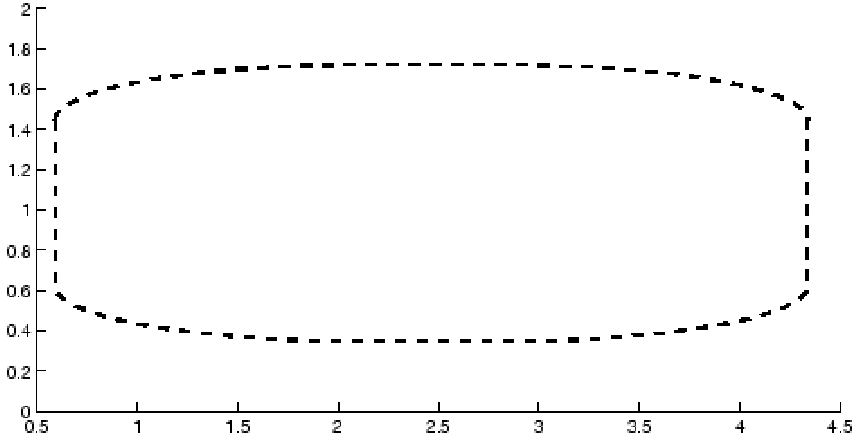
Remarks

Rectangle objects are 2-D and can be drawn in an axes only if the view is `[0 90]` (i.e., `view(2)`). Rectangles are children of axes and are defined in coordinates of the axes data.

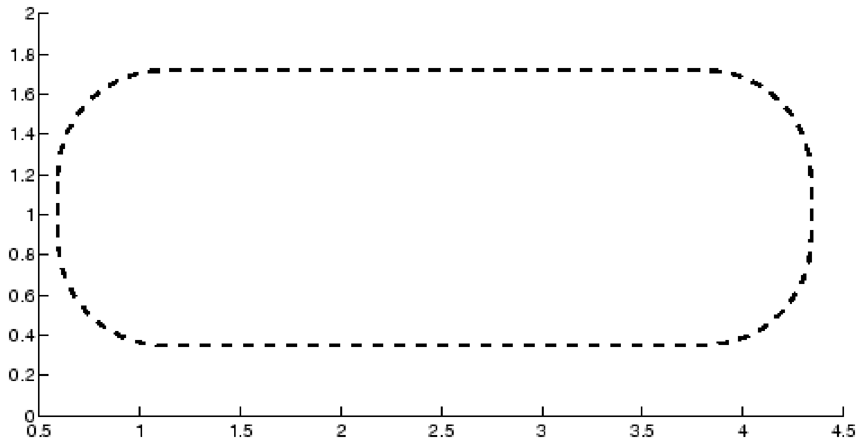
Examples

This example sets the data aspect ratio to `[1,1,1]` so that the rectangle is displayed in the specified proportions (`daspect`). Note that the horizontal and vertical curvature can be different. Also, note the effects of using a single value for `Curvature`.

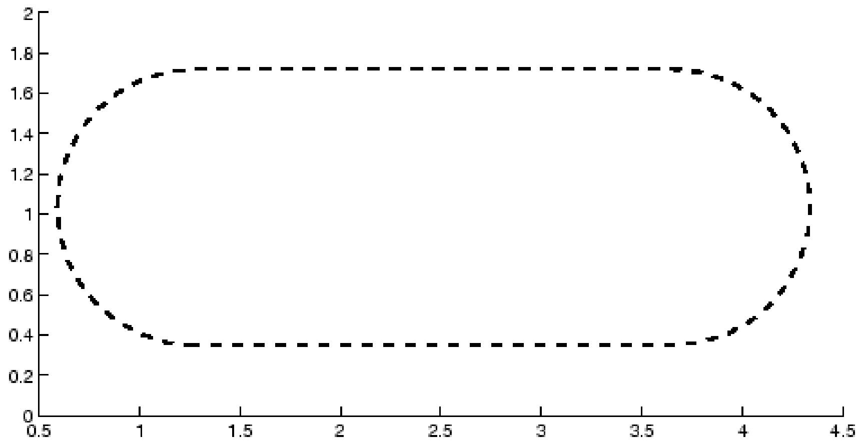
```
rectangle('Position',[0.59,0.35,3.75,1.37],...
          'Curvature',[0.8,0.4],...
         'LineWidth',2,'LineStyle','--')
daspect([1,1,1])
```



Specifying a single value of `[0.4]` for `Curvature` produces



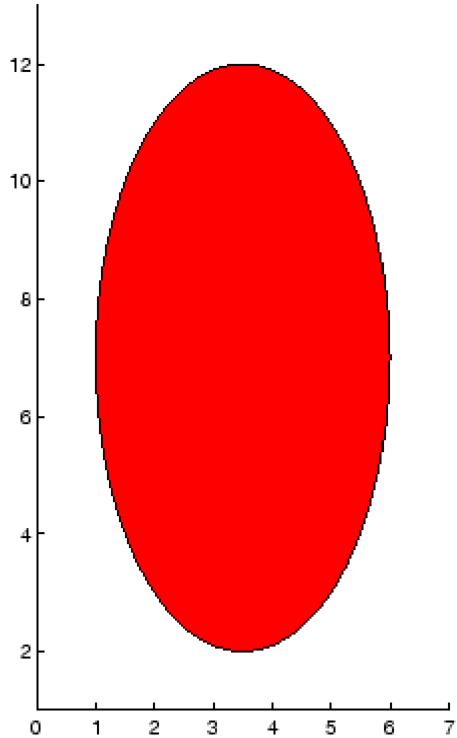
A `Curvature` of `[1]` produces a rectangle with the shortest side completely round:



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])
```

```
ylim([1,13])
```



Setting Default Properties

You can set default rectangle properties on the axes, figure, and 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`

"Object Creation" on page 1-99 for related functions

See the `annotation` function for information about the rectangle annotation object.

`Rectangle Properties` for property descriptions

Rectangle Properties

Purpose

Define rectangle properties

Modifying Properties

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

- 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 .

See 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.

`Annotation`

`hg.Annotation` object Read Only

Control the display of rectangle objects in legends. The `Annotation` property enables you to specify whether this rectangle object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the rectangle object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this rectangle object in a legend (default)

IconDisplayStyle Value	Purpose
`off`	Do not include this rectangle object in a legend
`children`	Same as `on` because rectangle objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the `IconDisplayStyle` property

See for more information and examples.

`BeingDeleted`

`on | {off}` read only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. The MATLAB software sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

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

`BusyAction`
`cancel | {queue}`

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

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

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

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is over the 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. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property)

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
```

```
    sel_typ = get(gcbf,'SelectionType')
    switch sel_typ
       case 'normal'
          disp('User clicked left-mouse button')
          set(src,'Selected','on')
       case 'extend'
          disp('User did a shift-click')
          set(src,'Selected','on')
       case 'alt'
          disp('User did a control-click')
          set(src,'Selected','on')
          set(src,'SelectionHighlight','off')
    end
 end
```

Suppose `h` is the handle of a rectangle object and that the `button_down` function is on your MATLAB path. The following statement assigns the function above to the `ButtonDownFcn`:

```
set(h,'ButtonDownFcn',@button_down)
```

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

`Children`
vector of handles

The empty matrix; rectangle objects have no children.

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips rectangles to the axes plot box by default. If you set `Clipping` to `off`, rectangles are displayed outside the axes plot box. This can occur if you create a rectangle, set `hold` to `on`, freeze axis scaling (`axis` set to `manual`), and then create a larger rectangle.

`CreateFcn`

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

Callback function executed during object creation. This property defines a callback function 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',@rect_create)
```

defines a default value for the rectangle `CreateFcn` property on the root level that sets the axes `DataAspectRatio` whenever you create a rectangle object. The callback function must be on your MATLAB path when you execute the above statement.

```
function rect_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
 axh = get(src,'Parent');
 set(axh,'DataAspectRatio',[1,1,1]))
end
```

MATLAB executes this function after setting all rectangle properties. Setting this property on an existing rectangle object has no effect. The function must define at least two input arguments (handle of object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See 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

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

Delete rectangle callback function. A callback function that executes when you delete the rectangle object (e.g., when you issue a delete command or clear the axes cla or figure clf). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

`DisplayName`

string (default is empty string)

String used by legend for this rectangle object. The `legend` function uses the string defined by the `DisplayName` property to label this rectangle object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this rectangle object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See for more examples.

`EdgeColor`
`{ColorSpec} | none`

Color of the rectangle edges. This property specifies the color of the rectangle edges as a color or specifies that no edges be drawn.

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

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

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

Printing with Nonnormal Erase Modes

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

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

`FaceColor`
`ColorSpec | {none}`

Color of rectangle face. This property specifies the color of the rectangle face, which is not colored by default.

`HandleVisibility`
`{on} | callback | off`

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

Handles are always visible when `HandleVisibility` is `on`.

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

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

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

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

You can set the Root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their 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 for more information on parenting graphics objects.

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

Location and size of rectangle. This property specifies the location and size of the rectangle in the data units of the axes. The point defined by `x`, `y` specifies one corner of the rectangle, and `width` and `height` define the size in units along the *x*-and *y*-axes respectively.

`Selected`
: `on | off`

Is object selected? When this property is `on` MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
: `{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing handles at each vertex. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
: string

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

`Type`
: string (read only)

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

`UIContextMenu`
handle of a uicontextmenu object

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

`UserData`
matrix

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

`Visible`
`{on} | off`

Rectangle visibility. By default, all rectangles are visible. When set to `off`, the rectangle is not visible, but still exists, and you can `get` and `set` its properties.

Purpose

Rectangle intersection area

Syntax

```
area = rectint(A,B)
```

Description

`area = rectint(A,B)` returns the area of intersection of the rectangles specified by position vectors `A` and `B`.

If `A` and `B` each specify one rectangle, the output `area` is a scalar.

`A` and `B` can also be matrices, where each row is a position vector. `area` is then a matrix giving the intersection of all rectangles specified by `A` with all the rectangles specified by `B.` That is, if `A` is `n`-by-`4` and `B` is `m`-by-`4`, then `area` is an `n`-by-`m` matrix where `area(i,j)` is the intersection area of the rectangles specified by the `i`th row of `A` and the `j`th row of `B`.

Note A position vector is a four-element vector `[x,y,width,height]`, where the point defined by `x` and `y` specifies one corner of the rectangle, and `width` and `height` define the size in units along the x and y axes respectively.

See Also

`polyarea`

recycle

Purpose

Set option to move deleted files to recycle folder

Syntax

```
recycle
stat = recycle
previousStat = recycle state
previousStat = recycle('state')
```

Description

`recycle` displays the current state, `on` or `off`, for recycling files you remove using the `delete` function. When the value is `on`, deleted files move to a different location. The location varies by platform—see . When the value is `off`, the `delete` function permanently removes the files. For details, see the Remarks section.

`stat = recycle` returns the current state for recycling files to the character array `stat`.

`previousStat = recycle` *`state`* sets the recycle option for MATLAB to the specified state, either `on` or `off`. The `previousStat` value is the recycle state before running the statement.

`previousStat = recycle('state')` is the function form of the syntax.

Remarks

The preference for **Deleting files** sets the state of the `recycle` function at startup. When you change the preference, it changes the state of `recycle`. When you change the state of `recycle`, it does not change the preference. Use `recycle` to override the behavior of the preference. For example, regardless of the setting for the **Deleting files** preference, to remove `thisfile.m` permanently, run:

```
recycle('off')
delete('thisfile.m')
```

After setting the `recycle` state to `off`, all files you delete using the `delete` function are deleted permanently until you do one of the following:

- Run `recycle('on')`

- Restart MATLAB. Upon startup, MATLAB sets the state for `recycle` to match the **Deleting files** preference.

Examples

Start from a state where file recycling is off. Verify the current recycle state:

```
recycle
ans =
   off
```

Turn file recycling on. Delete a file and move it to the recycle bin or temporary folder:

```
recycle on;
delete myfile.txt
```

See Also

`delete`, `dir`, `ls`, `rmdir`

reducepatch

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. The MATLAB software interprets the reduction factor `r` in one of two ways depending on its value:

- If `r` is less than 1, `r` is interpreted as a fraction of the original number of faces. For example, if you specify `r` as 0.2, then the number of faces is reduced to 20% of the number in the original patch.
- If `r` is greater than or equal to 1, then `r` is the target number of faces. For example, if you specify `r` as 400, then the number of faces is reduced until there are 400 faces remaining.

`nfv = reducepatch(p,r)` returns the reduced set of faces and vertices but does not set the `Faces` and `Vertices` properties of patch `p`. The struct `nfv` contains the faces and vertices after reduction.

`nfv = reducepatch(fv,r)` performs the reduction on the faces and vertices in the struct `fv`.

`nfv = reducepatch(p) or nfv = reducepatch(fv)` uses a reduction value of `0.5`.

`reducepatch(...,'fast')` assumes the vertices are unique and does not compute shared vertices.

`reducepatch(...,'verbose')` prints progress messages to the command window as the computation progresses.

`nfv = reducepatch(f,v,r)` performs the reduction on the faces in `f` and the vertices in `v`.

`[nf,nv] = reducepatch(...)` returns the faces and vertices in the arrays `nf` and `nv`.

Remarks

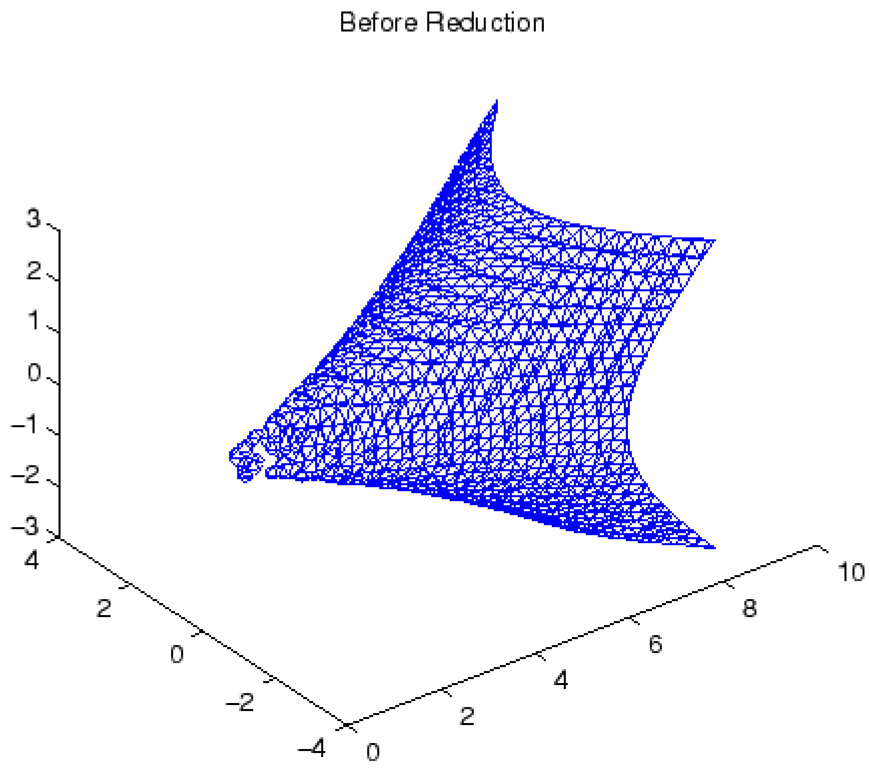
If the patch contains nonshared vertices, MATLAB computes shared vertices before reducing the number of faces. If the faces of the patch are not triangles, MATLAB triangulates the faces before reduction. The faces returned are always defined as triangles.

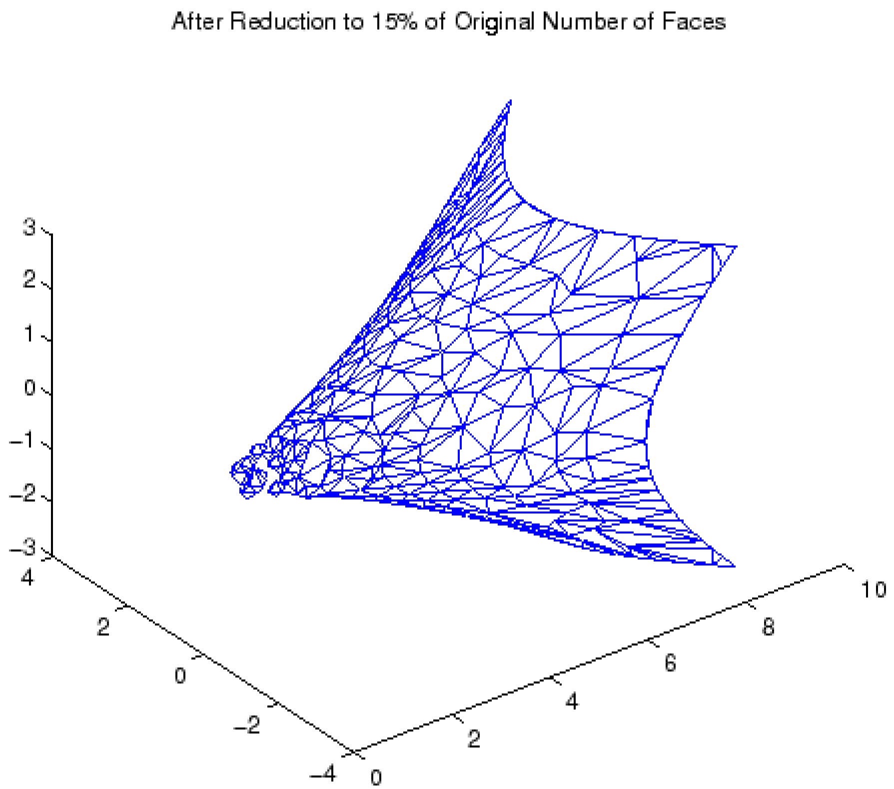
The number of output triangles may not be exactly the number specified with the reduction factor argument (`r`), particularly if the faces of the original patch are not triangles.

Examples

This example illustrates the effect of reducing the number of faces to only 15% of the original value.

```
[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
3
2
1
0
–1
–2
–3
4
2
0
–2
–4
0
2
4
6
8
10



See Also

`isosurface`, `isocaps`, `isonormals`, `smooth3`, `subvolume`, `reducevolume`

"Volume Visualization" on page 1-106 for related functions

Vector Field Displayed with Cone Plots for another example

reducevolume

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 `Rx`th element in the *x* direction, every `Ry`th element in the *y* direction, and every `Rz`th element in the *z* direction. If a scalar `R` is used to indicate the amount or reduction instead of a three-element vector, the MATLAB software assumes the reduction to be `[R R R]`.

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The reduced volume is returned in `nv`, and the coordinates of the reduced volume are returned in `nx`, `ny`, and `nz`.

`[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p] = size(V)`.

`nv = reducevolume(...)` returns only the reduced volume.

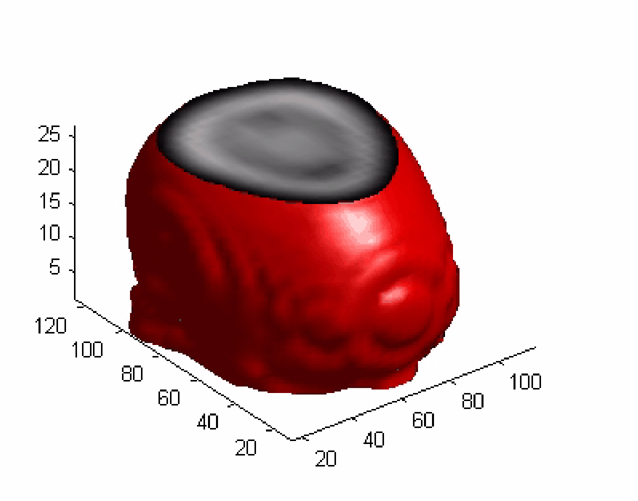
Examples

This example uses a data set that is a collection of MRI slices of a human skull. This data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then reduced (`reducevolume`) so that what remains is every fourth element in the *x* and *y* directions and every element in the *z* direction.
- The reduced data is smoothed (`smooth3`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with an interpolated face color draws the end caps (`FaceColor`, `isocaps`).
- The view of the object is set (`view`, `axis`, `daspect`).

- A 100-element grayscale colormap provides coloring for the end caps (colormap).
- Adding a light to the right of the camera illuminates the object (camlight, lighting).

```
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 1-106 for related functions

Purpose Redraw current figure

Syntax

```
refresh
refresh(h)
```

Description `refresh` erases and redraws the current figure.

`refresh(h)` redraws the figure identified by `h`.

See Also "Figure Windows" on page 1-100 for related functions

refreshdata

Purpose

Refresh data in graph when data source is specified

Syntax

```
refreshdata
refreshdata(figure_handle)
refreshdata(object_handles)
refreshdata(object_handles,'workspace')
```

Description

`refreshdata` evaluates any data source properties (`XDataSource`, `YDataSource`, or `ZDataSource`) on all objects in graphs in the current figure. If the specified data source has changed, the MATLAB software 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.

Remarks

The Linked Plots feature (see documentation for `linked`) sets up data sources for graphs and synchronizes them with the workspace variables they display. When you use this feature, you do not also need to call `refreshdata`, as it is essentially automatically triggered every time a data source changes.

If you are not using the Linked Plots feature, you need to set the `XDataSource`, `YDataSource`, and/or `ZDataSource` properties of a graph in order to use `refreshdata`. You can do that programmatically, as shown in the examples below, or use the Property Editor, one of the plotting tools. In the Property Editor, select the graph (e.g., a lineseries object) and type in (or select from the drop-down choices) the name(s) of the workspace variable(s) from which you want the plot to refresh, in the fields labelled **X Data Source**, **Y Data Source**, and/or **Z Data Source**. The call to `refreshdata` causes the graph to update.

Examples

Plot a sine wave, identify data sources, and then modify its `YDataSource`:

```
x = 0:.1:8;
y = sin(x);
h = plot(x,y)
set(h,'YDataSource','y')
set(h,'XDataSource','x')
y = sin(x.^3);
refreshdata
```

Create a surface plot, identify a `ZDataSource` for it, and change the data to a different size.

```
Z = peaks(5);
h = surf(Z)
set(h,'ZDataSource','Z')
pause(3)
Z = peaks(25);
refreshdata
```

See Also

The `[X,Y,Z]DataSource` properties of plot objects.

Purpose

Match regular expression

Syntax

```
regexp('str', 'expr')
[start_idx, end_idx, extents, matches, tokens, names,
   splits] = regexp('str', 'expr')
[v1, v2, ...] = regexp('str', 'expr', q1, q2, ...)
[v1 v2 ...] = regexp('str', 'expr', ..., options)
```

Each of these syntaxes applies 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. See in the MATLAB Programming Fundamentals documentation for more information.

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 2-3088.

`[start_idx, end_idx, extents, matches, tokens, names, splits] = 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.

Return Values for Regular Expressions

Default Order	Description	Qualifier
1	Row vector containing the starting index of each substring of `str` that matches `expr`.	**`start`**
2	Row vector containing the ending index of each substring of `str` that matches `expr`.	**`end`**
3	Cell array containing the starting and ending indices of each substring of `str` that matches a token in `expr`. (This is a `double` array when used with `'once'`.)	**`tokenExtents`**
4	Cell array containing the text of each substring of `str` that matches `expr`. (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 `regexp`. (This is a cell array of strings when used with `'once'`.)	**`tokens`**
6	Structure array containing the name and text of each *named* token captured by `regexp`. If there are no named tokens in `expr`, `regexp` 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`**
7	Cell array containing those parts of the input string that are delimited by substrings returned when using the `regexp` `'match'` option.	**`split`**

Tip When using the `split` option, `regexp` always returns one more string than it does with the `match` option. Also, you can always put the original input string back together from the substrings obtained from both `split` and `match`. See "Example 4 — Splitting the Input String" on page 2-3090.

`[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 2-3086 below.
'**`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 11 — Displaying Parsing Warnings" on page 2-3096.

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 7 — Using the Case-Sensitive Mode" on page 2-3093 illustrates this mode.

Mode Keyword	Flag	Description
'**matchcase**'	(?-i)	Letter case must match when matching patterns to a string. (The default for regexp).
'**ignorecase**'	(?i)	Do not consider letter case when matching patterns to a string. (The default for regexpi).

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 8 — Using the Dot Matching Mode" on page 2-3094 illustrates the Dot Matching mode.

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 9 — Using the Anchor Type Mode" on page 2-3094 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 parsing string. Note that spacing mode applies to the parsing string (the second input argument that contains the metacharacters (e.g., `\w`) and not the string being parsed. "Example 10 — Using the Spacing Mode" on page 2-3095 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 in the MATLAB Programming Fundamentals 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';
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 — Splitting the Input String

Find the substrings delimited by the ^ character:

```
s1 = ['Use REGEXP to split ^this string into ' ...
      'several ^individual pieces'];

s2 = regexp(s1, '\^', 'split');

s2(:)
ans =
    'Use REGEXP to split '
    'this string into several '
    'individual pieces'
```

The `split` option returns those parts of the input string that are not returned when using the `'match'` option. Note that when you match the beginning or ending characters in a string (as is done in this example), the first (or last) return value is always an empty string:

```
str = 'She sells sea shells by the seashore.';

[matchstr splitstr] = regexp(str, '[Ss]h.', 'match', ...
                             'split')
matchstr =
    'She'    'she'    'sho'
splitstr =
     ''    ' sells sea '    'lls by the sea'    're.'
```

For any string that has been split, you can reassemble the pieces into the initial string using the command

```
j = [splitstr; [matchstr {''}]]; [j{:}]

ans =
   She sells sea shells by the seashore.
```

Example 5 — 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>']
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 in the MATLAB Programming Fundamentals documentation for information on using tokens.

Example 6 — 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 7 — 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 8 — Using the Dot Matching Mode

Parse the following string that contains a newline (\n) character:

```
str = sprintf('abc\ndef')
str =
   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 9 — 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 10 — Using the Spacing Mode

Create a file called `regexp_str.txt` containing the following text.

```
(?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.
.       # Match a character of any type.
)       # Finish capturing said token.
\1      # Backreference to match what token #1 matched.

\w*     # Finally, match the remainder of the word.
```

Because the first line enables freespacing mode, MATLAB ignores all spaces and comments that appear in the file. 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 11 — 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('$.', '[a-]','warnings')
Warning: Unbound range.
 [a-]
   |
```

See Also

, `regexprep`, `regexptranslate`, `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`. See in the MATLAB Programming Fundamentals documentation for more information.

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 2-3098.

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 and the example in the MATLAB Programming Fundamentals 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
`mode`	See `mode` descriptions on the `regexp` reference page.
`N`	Replace only the `N`th occurrence of `expr` in `str`.
'**`once`**'	Replace only the first occurrence of `expr` in `str`.

Option	Description
'ignorecase'	Ignore case when matching and when replacing.
'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 in the MATLAB Programming Fundamentals 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 =
    'Whose w--ds these are I think I know.'
```

```
'His house is in the vi--age though;'
'He wi-- not s-- me sto--ing here'
'To watch his w--ds fi-- up with snow.'
```

See Also

, regexp, regexpi, regexptranslate, 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. See in the MATLAB Programming Fundamentals documentation for more information.

Type	Description
'escape'	Translate all special characters (e.g., '$', '.', '?', '[') in string s1 so that they are treated as literal characters when used in the regexp and regexprep functions. The translation inserts an escape character ('\') before each special character in s1. Return the new string in s2.
'wildcard'	Translate all wildcard and '.' characters in string s1 so that they are treated as literal wildcards and periods when used in the regexp and regexprep functions. The translation replaces all instances of '*' with '.*', all instances of '?' with '.', and all instances of '.' with '\.'. Return the new string in s2.

Examples

Example 1 — Using the 'escape' Option

Because regexp interprets the sequence '\n' as a newline character, it cannot locate the two consecutive characters '\' and 'n' in this string:

```
str = 'The sequence \n generates a new line';
pat = '\n';

regexp(str, pat)
ans =
      []
```

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('escape', '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

Purpose

Associate event handler for COM object event at run time

Syntax

```
h.registerevent(eventhandler)
registerevent(h, eventhandler)
```

Description

`h.registerevent(eventhandler)` registers event handler routines with their corresponding events. The `eventhandler` argument can be either a string that specifies the name of the event handler function, or a function handle that maps to that function. Strings used in the `eventhandler` argument are not case sensitive.

`registerevent(h, eventhandler)` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Show events in the MATLAB sample control:

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

MATLAB displays all events associated with the instance of the control (output is formatted):

```
Click = void Click()
DblClick = void DblClick()
MouseDown = void MouseDown(int16 Button, int16 Shift,
  Variant x, Variant y)
Event_Args = void Event_Args(int16 typeshort,
  int32 typelong, double typedouble, string typestring,
  bool typebool)
```

Register all events with the same event handler routine, `sampev`:

```
h.registerevent('sampev');
h.eventlisteners
```

MATLAB displays:

```
ans =
    'Click'          'sampev'
    'DblClick'       'sampev'
    'MouseDown'      'sampev'
    'Event_Args'     'sampev'
```

Register individual events:

```
%Unregister existing events
h.unregisterallevents;
%Register specific events
h.registerevent({'click' 'myclick'; ...
    'dblclick' 'my2click'});
h.eventlisteners
```

MATLAB displays:

```
ans =
    'click'        'myclick'
    'dblclick'     'my2click'
```

Register events using a function handle (`@sampev`) instead of the function name:

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

See Also

`events (COM)` | `eventlisteners` | `unregisterevent` | `unregisterallevents` | `isevent`

How To

-

rehash

Purpose

Refresh function and file system path caches

Syntax

```
rehash
rehash path
rehash toolbox
rehash pathreset
rehash toolboxreset
rehash toolboxcache
```

Description

`rehash` with no arguments updates the MATLAB list of known files and classes for directories on the search path that are not in *`matlabroot`*`/toolbox`. It compares the timestamps for loaded functions 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. Use `rehash` with no arguments only when you run an M-file that updates a second M-file, and the calling file needs to reuse the updated version of the second M-file before the calling file 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. Run `rehash path` only 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 not using the most current versions of your M-files.

`rehash` **`toolbox`** performs the same updates as rehash path, except it updates the list of known files and classes for *all* directories on the search path, including those in *`matlabroot`*`/toolbox`. Run `rehash toolbox` when you change, add, or remove files in *`matlabroot`*`/toolbox` during a session. Typically, you should not make changes to files and directories in *`matlabroot`*`/toolbox`.

`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 the General Preferences dialog box.

See Also

`addpath`, `clear`, `matlabroot`, `path`, `rmpath`

and in the MATLAB Desktop Tools and Development Environment documentation

release

Purpose

Release COM interface

Syntax

```
h.release
release(h)
```

Description

`h.release` releases the interface and all resources used by the interface. You must release the handle when you are done with the interface. A released interface is no longer valid. MATLAB generates an error if you try to use an object that represents that interface.

`release(h)` is an alternate syntax.

Releasing the interface does not delete the control itself (see the `delete` function), since other interfaces on that object might still be active.

COM functions are available on Microsoft Windows systems only.

Examples

1 Create an instance of a Microsoft Calendar control. Get a `TitleFont` interface and use it to change the appearance of the calendar title font:

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

TFont = cal.TitleFont;
TFont.Name = 'Viva BoldExtraExtended';
TFont.Bold = 0;
```

2 After working with the title font, release the `TitleFont` interface:

```
TFont.release;
```

3 Delete the `cal` object and the figure window:

```
cal.delete;
delete(f);
clear f;
```

See Also

`delete (COM)` | `actxcontrol` | `actxserver`

How To

- Releasing Interfaces

relationaloperators (handle)

Purpose

Equality and sorting of handle objects

Syntax

```
TF = eq(H1,H2)
TF = ne(H1,H2)
TF = lt(H1,H2)
TF = le(H1,H2)
TF = gt(H1,H2)
TF = ge(H1,H2)
```

Description

`TF = eq(H1,H2)`

`TF = ne(H1,H2)`

`TF = lt(H1,H2)`

`TF = le(H1,H2)`

`TF = gt(H1,H2)`

`TF = ge(H1,H2)`

For each pair of input arrays (`H1` and `H2`), a logical array of the same size is returned in which each element is an element-wise equality or comparison test result. These methods perform scalar expansion in the same way as the MATLAB built-in functions. See `relationaloperators` for more information.

You can make the following assumptions about the result of a handle comparison:

- The same two handles always compare as equal and the repeated comparison of any two handles always yields the same result in the same MATLAB session.
- Different handles are always not-equal.
- The order of handle values is purely arbitrary and has no connection to the state of the handle objects being compared.
- If the input arrays belong to different classes (including the case where one input array belongs to a non-handle class such as `double`) then the comparison is always false.

- If a comparison is made between a handle object and an object of a dominant class, the method of the dominant class is invoked. You should generally test only like objects because a dominant class might not define one of these methods.
- An error occurs if the input arrays are not the same size and neither is scalar.

See Also

`handle`, `meta.class`

Purpose Remainder after division

Syntax

```
R = rem(X,Y)
```

Description `R = rem(X,Y)` if `Y ~= 0`, returns `X - n.*Y` where `n = fix(X./Y)`. If `Y` is not an integer and the quotient `X./Y` is within roundoff error of an integer, then `n` is that integer. The inputs `X` and `Y` must be real arrays of the same size, or real scalars.

The following are true by convention:

- `rem(X,0)` is `NaN`
- `rem(X,X)` for `X~=0` is `0`
- `rem(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `X`.

Remarks `mod(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `Y`.

`rem(X,Y)` and `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 `sign(X)*abs(Y)`. If `Y` is zero, `rem` returns `NaN`.

See Also `mod`

Purpose

Remove key-value pairs from `containers.Map`

Syntax

`remove(M, keys)`

Description

`remove(M, keys)` erases all specified keys, and the values associated with them, from `Map` object `M.keys` can be a scalar key or a cell array of keys.

Using `remove` changes the count of the elements in the map.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

Examples

Create a `Map` object containing the names of several US states and the capital city of each:

```
US_Capitals = containers.Map( ...
{'Arizona', 'Nebraska', 'Nevada', 'New York', ...
 'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...
{'Phoenix', 'Lincoln', 'Carson City', 'Albany', ...
 'Atlanta', 'Juneau', 'Montpelier', 'Salem'});
```

After checking how many keys there are in the `US_Capitals` map, remove the key-value pair with key name `Oregon` from it:

```
US_Capitals.Count
ans =
    8

remove(US_Capitals, 'Oregon');

US_Capitals.Count
ans =
    7
```

Remove three more key-value pairs from the map:

```
remove(US_Capitals, {'Nebraska', 'Nevada', 'New York'});
US_Capitals.Count
ans =
    4
```

See Also

containers.Map, keys(Map), values(Map), size(Map), length(Map)isKey(Map),handle

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
```

The response is

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:
```

```
      acceleration
      speed
```

The members of `tsc` are listed by name at the bottom: `acceleration` and `speed`. These are the `Name` properties of `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
```

The response is

```
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 `NaN`s.

See Also

`bsxfun`, `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)
```

The display in the MATLAB Command Window is

```
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])
```

The resampled time series displays as follows:

```
Time Series Object: speed

Time vector characteristics

      Length                5
      Start time            1 seconds
      End time              4.900000e+000 seconds
```

```
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).

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`

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`, the MATLAB software 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 1-105 for related functions

Purpose Reset random stream

Class `@RandStream`

Syntax

```
reset(s)
reset(s,seed)
```

Description

`reset(s)` resets the generator for the random stream `s` to its initial internal state. This is similar to clearing `s` and recreating it using `RandStream('type',...)`, except `reset` does not set the stream's `RandnAlg`, `Antithetic`, and `FullPrecision` properties to their original values.

`reset(s,seed)` resets the generator for the random stream `s` to the initial internal state corresponding to the seed `seed`. Resetting a stream's seed can invalidate independence with other streams.

Note Resetting a stream should be used primarily for reproducing results.

Examples

1 Create a random stream object.

```
s=RandStream('mt19937ar')
```

2 Make it the default stream.

```
RandStream.setDefaultStream(s)
```

3 Reset the stream object you just created and generate 5 uniform random values using the `rand` method.

```
rand(s,1,5)

ans =

    0.3631    0.4048    0.1490    0.9438    0.1247
```

4 Reset the stream.

```
reset(s)
```

5 Generate the same 5 random values from the default stream.

```
rand(s,1,5)

ans =

    0.3631    0.4048    0.1490    0.9438    0.1247
```

See Also

`@RandStream`

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 =
```

```
    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`, `circshift`, `permute`

The colon operator `:`

Purpose

Convert between partial fraction expansion and polynomial coefficients

Syntax

```
[r,p,k] = residue(b,a)
[b,a] = residue(r,p,k)
```

Description

The `residue` function converts a quotient of polynomials to pole-residue representation, and back again.

`[r,p,k] = residue(b,a)` finds the residues, poles, and direct term of a partial fraction expansion of the ratio of two polynomials, $b(s)$ and $a(s)$, of the form

$$\frac{b(s)}{a(s)} = \frac{b_1 s^m + b_2 s^{m-1} + b_3 s^{m-2} + \ldots + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + a_3 s^{n-2} + \ldots + a_{n+1}}$$

where b_j and a_j are the `j`th elements of the input vectors `b` and `a`.

`[b,a] = residue(r,p,k)` converts the partial fraction expansion back to the polynomials with coefficients in `b` and `a`.

Definition

If there are no multiple roots, then

$$\frac{b(s)}{a(s)} = \frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} + \ldots + \frac{r_n}{s-p_n} + k(s)$$

The number of poles `n` is

```
n = length(a)-1 = length(r) = length(p)
```

The direct term coefficient vector is empty if `length(b) < length(a)`; otherwise

```
length(k) = length(b)-length(a)+1
```

If `p(j) = ...  = p(j+m-1)` is a pole of multiplicity `m`, then the expansion includes terms of the form

$$\frac{r_j}{s-p_j} + \frac{r_{j+1}}{(s-p_j)^2} + \ldots + \frac{r_{j+m-1}}{(s-p_j)^m}$$

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, `resi2` computes the residues at the repeated root locations.

Limitations

Numerically, the partial fraction expansion of a ratio of polynomials represents an ill-posed problem. If the denominator polynomial, $a(s)$, is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can make arbitrarily large changes in the resulting poles and residues. Problem formulations making use of state-space or zero-pole representations are preferable.

Examples

If the ratio of two polynomials is expressed as

$$\frac{b(s)}{a(s)} = \frac{5s^3 + 3s^2 - 2s + 7}{-4s^3 + 8s + 3}$$

then

```
b = [ 5 3 -2 7]
a = [-4 0 8 3]
```

and you can calculate the partial fraction expansion as

```
[r, p, k] = residue(b,a)

r =
   -1.4167
   -0.6653
    1.3320

p =
    1.5737
   -1.1644
   -0.4093

k =
   -1.2500
```

Now, convert the partial fraction expansion back to polynomial coefficients.

```
[b,a] = residue(r,p,k)

b =
   -1.2500   -0.7500    0.5000   -1.7500

a =
    1.0000   -0.0000   -2.0000   -0.7500
```

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. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975, p. 56.

Purpose

Restore default search path

GUI Alternatives

As an alternative to the `restoredefaultpath` function, use the Set Path dialog box.

Syntax

```
restoredefaultpath
restoredefaultpath; matlabrc
```

Description

`restoredefaultpath` sets the search path to include only folders for installed products from The MathWorks. Use `restoredefaultpath` when you are having problems with the search path.

`restoredefaultpath; matlabrc` sets the search path to include only folders for installed products from The MathWorks and corrects search path problems encountered during startup.

See Also

`addpath`, `genpath`, `matlabrc`, `rmpath`, `savepath`

Topics in the User Guide:

-
-

rethrow

Purpose

Reissue error

Note As of version 7.5, MATLAB supports error handling that is based on the `MException` class. Calling `rethrow` with a structure argument, as described on this page, is now replaced by calling rethrow with an `MException` object, as described on the reference page for `rethrow(MException)`. `rethrow` called with a structure input will be removed in a future version.

Syntax

```
rethrow(errorStruct)
```

Description

`rethrow(errorStruct)` reissues the error specified by `errorStruct`. The currently running M-file terminates and control returns to the keyboard (or to any enclosing `catch` block). The `errorStruct` argument must be a MATLAB structure containing at least the `message` and `identifier` 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.

Remarks

The `errorStruct` 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(previous_error)
end
```

See Also

`rethrow(MException)`, `throw(MException)`, `throwAsCaller(MException)`, `try`, `catch`, `error`, `assert`, `dbstop`

rethrow (MException)

Purpose

Reissue existing exception

Syntax

`rethrow(errRecord)`

Description

`rethrow(errRecord)` forces an exception (i.e., error report) to be reissued by MATLAB after the error reporting process has been temporarily suspended to diagnose or remedy the problem. MATLAB typically responds to errors by terminating the currently running program. Errors reported within a try-catch statement, however, bypass this mechanism and transfer control of the program to error handling code in the `catch` block instead. This enables you to write your own error handling procedures for parts of your program that require them.

The *errRecord* argument is a data structure derived from the `MException` class that contains information about the cause and location of the error.

The code segment below shows the format of a typical try-catch statement.

```
try                               try block
   program-code                       |
   program-code                       |
       :                              V
catch errRecord                   catch block
   error-handling code                |
       :                              |
   rethrow(errRecord)                 V
end
```

An error detected within the `try` block causes MATLAB to enter the corresponding `catch` block. The error record constructed by MATLAB in the process of reporting this error passes to the `catch` command in the statement

```
catch errRecord
```

Error handling code within the `catch` block uses the information in the error record to address the problem in some predefined manner. The `catch` block shown here ends with a `rethrow` statement which passes the error record to the caller of this function and then terminates the function:

```
rethrow(errRecord)
```

The most significant difference between `rethrow` and other MATLAB functions that throw exceptions is in how `rethrow` handles a piece of the exception record called the *stack*. The stack keeps a record of where the error occurred and what functions were called in the process. It is a struct array composed of the following fields, where each element of the array represents one record in what is often a chain of thrown exceptions:

Fields of the Exception Stack	Description
`line`	Line number from which the exception was thrown.
`name`	Name of the function being executed at the time.
`file`	Name of the M-file containing that function.

Functions such as `error`, `assert`, or `throw`, purposely overwrite the stack with the location from which one of those commands was executed. Calling `rethrow`, however, preserves information on the stack, keeping it as it was when the exception was first thrown. In doing so, `rethrow` maintains the location of the original exception, enabling you to retrace the path taken to the source of the error.

Remarks

There are four ways to throw an exception in MATLAB. Use the first of these when testing the outcome of some action for failure and reporting the failure to MATLAB:

- Test the result of some action taken by your program. If the result is found to be incorrect or unexpected, compose an appropriate message and message identifier, and pass these to MATLAB using the `error` or `assert` function.

Use one of the remaining three techniques to resume an exception that is already in progress but has been temporarily suspended in a try-catch statement:

- Reissue the original exception by returning the initial error record unmodified. Use the `MException` `rethrow` method to do this.
- Collect additional information on the cause of the error, store it in a new or modified error record, and issue a new exception based on that record. Use the `MException` `addCause` and `throw` methods to do this.
- Set or modify the stack field of a new or existing error record to make it appear that the error originated in the caller of the currently running function. Use the `MException` `throwAsCaller` method to do this.

You should always either rethrow or throw an exception when exiting the catch block if the faulty condition still exists. Otherwise your function can complete with normal status, even though function may have failed.

`rethrow` can only issue a previously caught exception. If an exception that was not previously thrown is passed to `rethrow`, the MATLAB software generates a new exception.

Examples

This example shows the difference between using `throw` and `rethrow` at the end of a catch block. The `combineArrays` function vertically concatenates arrays `A` and `B`. When the two arrays have rows of unequal length, the function throws an error.

The first time you run the function, comment out the `rethrow` command at the end of the catch block so that the function calls `throw` instead:

```
function C = combineArrays(A, B)
```

```
try
    catAlongDim1(A, B);                 % Line 3
catch errRecord
    fprintf('** ERROR: Dim 2 is %d for A, %d for B **\n', ...
        size(A,2), size(B,2))
    throw(errRecord)                    % Line 7
    % rethrow(errRecord)                % Line 8
end

function catAlongDim1(V1, V2)
    C = cat(1, V1, V2);                 % Line 12
```

When MATLAB throws the exception, it reports an error on line 7 which is the line that calls `throw`. In some cases, that might be what you want but, in this case, it does not show the true source of the error.

```
A = 4:3:19;     B = 3:4:19;
combineArrays(A, B)
** ERROR: Incompatible array sizes 6 and 5 **
??? Error using ==> combineArrays at 7
CAT arguments dimensions are not consistent.
```

Make the following changes to `combineArrays.m` so that you use `rethrow` instead:

```
% throw(errRecord)                      % Line 7
rethrow(errRecord)                      % Line 8
```

Run the function again. This time, line 12 is the first line reported which is where the MATLAB concatenation function `cat` was called and the exception originated. The next error reported is on line 3 which is where the call to `catAlongDim1` was called:

```
** ERROR: Incompatible array sizes 6 and 5 **
??? Error using ==> cat
CAT arguments dimensions are not consistent.
```

```
Error in ==> combineArrays>catAlongDim1 at 12
    C = cat(1, V1, V2);
Error in ==> combineArrays at 3
    catAlongDim1(A, B);
```

See Also

try, catch, error, assert, MException, throw(MException), throwAsCaller(MException), addCause(MException), getReport(MException), last(MException)

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`

Tiff.rewriteDirectory

Purpose

Write modified metadata to existing IFD

Syntax

```
tiffobj.rewriteDirectory()
```

Description

`tiffobj.rewriteDirectory()` writes modified metadata (tag) data to an existing directory. Use this tag when you want to change the value of a tag in an existing image file directory.

Examples

Open a `Tiff` object for modification and modify the value of a tag. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r');
%  Modify the value of a tag.
t.setTag('Software','MATLAB');
t.rewriteDirectory();
```

References

This method corresponds to the `TIFFRewriteDirectory` function in the LibTIFF C API. To use this method, you must be familiar with LibTIFF version 3.7.1, as well as the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.writeDirectory`

Tutorials

-
-

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 1-103 for related functions

rgb2ind

Purpose

Convert RGB image to indexed image

Syntax

```
[X,map] = rgb2ind(RGB, n)
X = rgb2ind(RGB, map)
[X,map] = rgb2ind(RGB, tol)
[...] = rgb2ind(..., dither_option)
```

Description

`rgb2ind` converts RGB images to indexed images using one of these methods:

- Uniform quantization
- Minimum variance quantization
- Colormap approximation

For all these methods, `rgb2ind` also dithers the image unless you specify `'nodither'` for `dither_option`.

`[X,map] = rgb2ind(RGB, n)` converts the RGB image to an indexed image `X` using minimum variance quantization. `map` contains at most `n` colors. `n` must be less than or equal to 65,536.

`X = rgb2ind(RGB, map)` converts the RGB image to an indexed image `X` with colormap `map` by matching colors in `RGB` with the nearest color in the colormap `map`. `size(map,1)` must be less than or equal to 65,536.

`[X,map] = rgb2ind(RGB, tol)` converts the RGB image to an indexed image `X` using uniform quantization. `map` contains at most `(floor(1/tol)+1)^3` colors. `tol` must be between 0.0 and 1.0.

`[...]  = rgb2ind(..., dither_option)` enables or disables dithering. `dither_option` is a string that can have one of these values.

`'dither'` (default)	dithers, if necessary, to achieve better color resolution at the expense of spatial resolution.
`'nodither'`	maps each color in the original image to the closest color in the new map. No dithering is performed.

Note The values in the resultant image `X` are indexes into the colormap `map` and cannot be used in mathematical processing, such as filtering operations.

Class Support

The input image can be of class `uint8`, `uint16`, `single`, or `double`. If the length of `map` is less than or equal to 256, the output image is of class `uint8`. Otherwise, the output image is of class `uint16`.

Remarks

If you specify `tol`, `rgb2ind` uses uniform quantization to convert the image. This method involves cutting the RGB color cube into smaller cubes of length `tol`. For example, if you specify a `tol` of 0.1, the edges of the cubes are one-tenth the length of the RGB cube. The total number of small cubes is:

```
n = (floor(1/tol)+1)^3
```

Each cube represents a single color in the output image. Therefore, the maximum length of the colormap is `n`. `rgb2ind` removes any colors that don't appear in the input image, so the actual colormap can be much smaller than `n`.

If you specify `n`, `rgb2ind` uses minimum variance quantization. This method involves cutting the RGB color cube into smaller boxes (not necessarily cubes) of different sizes, depending on how the colors are distributed in the image. If the input image actually uses fewer colors than the number you specify, the output colormap is also smaller.

If you specify map, rgb2ind uses colormap mapping, which involves finding the colors in map that best match the colors in the RGB image.

Examples

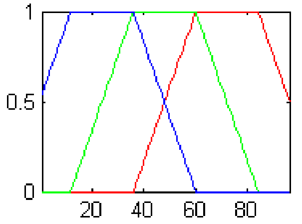
```
RGB = imread('peppers.png');
[X,map] = rgb2ind(RGB,128);
figure, imshow(X,map)
```

See Also

cmunique, dither, imapprox, ind2rgb

Purpose

Plot colormap



Syntax

```
rgbplot(cmap)
```

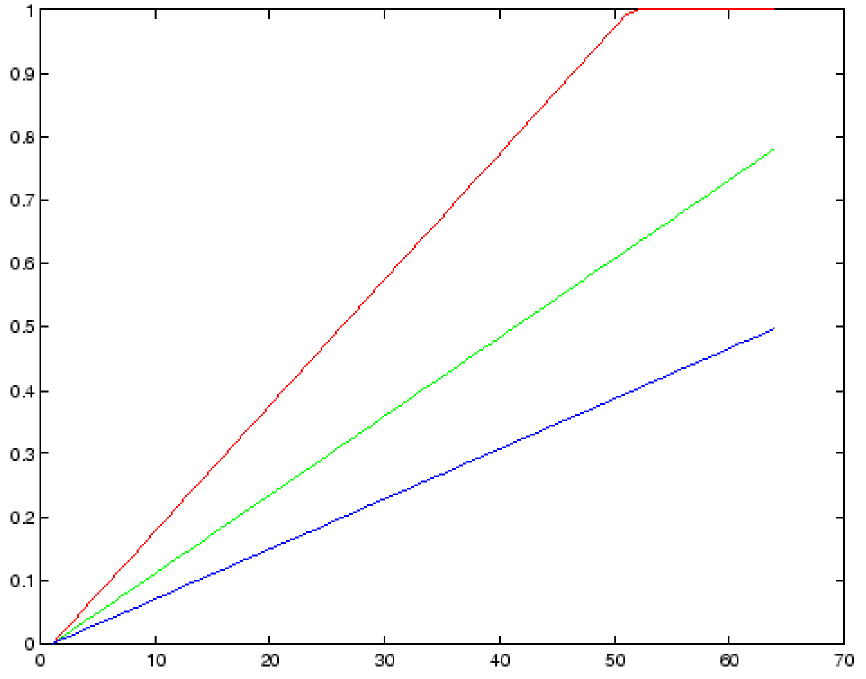
Description

`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 1-103 for related functions

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 undulating three-dimensional ribbons of uniform width using `X = 1:size(Y,1)`. Ribbons advance along the *x*-axis centered on tick marks at unit intervals, three-quarters of a unit in width. Ribbons are assigned colors from the current colormap in sequence from minimum `X` to maximum `X` (the axes `colororder` property, used by `plot` and `plot3`, does not apply to ribbon or other surface plots).

`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)` is the same as `plot(X,Y)` except that the columns of `Y` are plotted as separated ribbons in 3-D. The *y* and *z*-axes of `ribbon(X,Y)` correspond to the *x* and *y*-axes of `plot(X,Y)`.

`ribbon(X,Y,width)` specifies the width of the ribbons. The default is `0.75`. If `width = 1`, the ribbons touch, leaving no space between them when viewed down the *z*-axis. If `width > 1`, ribbons overlap and can intersect.

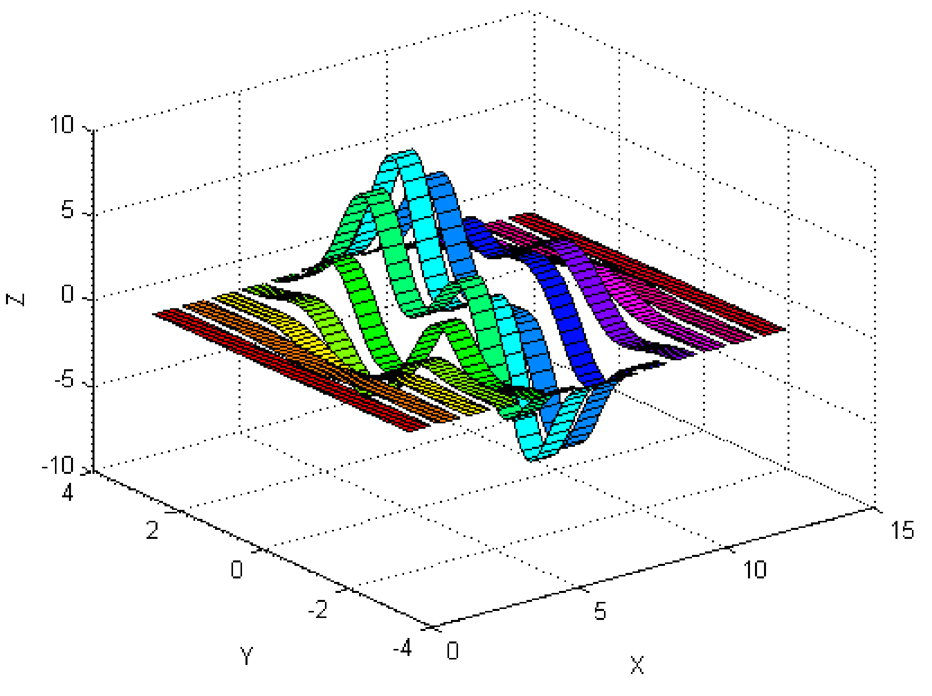
`ribbon(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`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)
xlabel('X')
ylabel('Y')
zlabel('Z')
colormap hsv
```



See Also

`plot`, `plot3`, `surface`, `waterfall`

"Polygons and Surfaces" on page 1-95 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`.

Remarks Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property.

See Also `getappdata`, `isappdata`, `setappdata`

Purpose

Remove folder

Graphical Interface

As an alternative to the `rmdir` function, use the delete feature in the Current Folder browser.

Syntax

```
rmdir('folderName')
rmdir('folderName','s')
[status, message, messageid] = rmdir('folderName','s')
```

Description

`rmdir('folderName')` removes the folder `folderName` from the current folder, where `folderName` is empty. If `folderName` is not in the current folder, specify the relative path or the full path for `folderName`.

`rmdir('folderName','s')` removes the folder `folderName` and its contents from the current folder. With the '**s**' option, `rmdir` attempts to removes all subfolders and files in `folderName` regardless of their write permissions. The result for read-only files follows the practices of the operating system.

`[status, message, messageid] = rmdir('folderName','s')` removes the folder `folderName` and its contents from the current folder, returning the status, a message, and the MATLAB message ID. Here, `status` is `1` for success and is `0` for error. `message`, `messageid`, and the **s** input argument are optional.

Remarks

When attempting to remove multiple folders, either by including a wildcard in the folder name or by specifying the `'s'` flag in `rmdir`, MATLAB produces an error if it is unable to remove all folders as expected. The error message lists the folder and files that MATLAB could not remove.

Examples

Remove Empty Folder

Remove `myfiles` from the current folder, where `myfiles` is empty:

```
rmdir('myfiles')
```

If the current folder is `matlab/work`, and `myfiles` is in `d:/matlab/work/project/`, use the relative path to remove `myfiles`:

```
rmdir('project/myfiles')
```

If the current folder is `matlab/work`, and `myfiles` is in `d:/matlab/work/project/`, use the full path to remove `myfiles`:

```
rmdir('d:/matlab/work/project/myfiles')
```

Remove Folder and All Contents

Remove `myfiles`, its subfolders, and all files in the folders, assuming `myfiles` is in the current folder:

```
rmdir('myfiles','s')
```

Remove Folder and Return Results

Remove `myfiles` from the current folder, where `myfiles` is not empty, and return the results:

```
[stat, mess, id]=rmdir('myfiles')
```

MATLAB returns:

```
stat =
     0

mess =

The directory is not empty.

id =

MATLAB:RMDIR:OSError
```

Remove `myfiles` and its contents using the `s` option, which is required for non-empty folders, and return the results:

```
[stat, mess]=rmdir('myfiles','s')
```

MATLAB returns:

```
stat =
     1

mess =

     ''
```

See Also

`catch`, `cd`, `copyfile`, `delete`, `dir`, `fileattrib`, `filebrowser`, `MException`, `mkdir`, `movefile`, `try`

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`, dynamic field names

Purpose

Remove folders from search path

GUI Alternatives

As an alternative to the rmpath function, use the Set Path dialog box.

Syntax

```
rmpath('folderName')
rmpath folderName
```

Description

rmpath('folderName') removes the specified folder from the search path . Use the full path for folderName.

rmpath folderName is the command form of the syntax.

Examples

Remove /usr/local/matlab/mytools from the search path:

```
rmpath /usr/local/matlab/mytools
```

See Also

addpath, cd, dir, genpath, matlabroot, path, pathsep, pathtool, rehash, restoredefaultpath, savepath, userpath, what

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`

Purpose

Root

Description

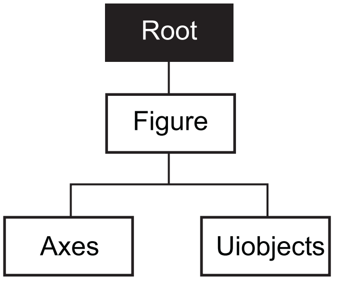
The root is a graphics object that corresponds to the computer screen. There is only one root object and it has no parent. The children of the root object are figures.

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

Root properties

Modifying Properties

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

- The 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 .

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.

`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.

`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.

`MonitorPositions`
`[x y width height;x y width height]`

Width and height of primary and secondary monitors, in pixels. This property contains the width and height of each monitor connnected 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 `MonitorPositions` on a multiheaded system returns the position for each monitor on a separate line.

```
v = get(0,'MonitorPositions')
v =
 x y width height % Primary monitor
 x y width height % Secondary monitor
```

The value of the ScreenSize property is inconsistent when using multiple monitors. If you want specific and consistent values, use the `MonitorPositions` property.

`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

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 Microsoft 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'`.

`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

```
r = roots(c)
```

Description

`r = roots(c)` returns a column vector whose elements are the roots of the polynomial `c`.

Row vector `c` contains the coefficients of a polynomial, ordered in descending powers. If `c` has `n+1` components, the polynomial it represents is $c_1 s^n + \ldots + c_n s + c_{n+1}$.

Remarks

Note the relationship of this function to `p = poly(r)`, which returns a row vector whose elements are the coefficients of the polynomial. For vectors, `roots` and `poly` are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples

The polynomial $s^3 - 6s^2 - 72s - 27$ is represented in MATLAB software as

```
p = [1 -6 -72 -27]
```

The roots of this polynomial are returned in a column vector by

```
r = roots(p)

r =
    12.1229
    -5.7345
    -0.3884
```

Algorithm

The algorithm simply involves computing the eigenvalues of the companion matrix:

```
A = diag(ones(n-1,1),-1);
A(1,:) = -c(2:n+1)./c(1);
eig(A)
```

It is possible to prove that the results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix `A`, but this does not mean that they are the exact roots of a polynomial with coefficients within roundoff error of those in `c`.

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(theta)
rose(theta,x)
rose(theta,nbins)
rose(axes_handle,...)
h = rose(...)
[tout,rout] = rose(...)
```

Description

`rose(theta)` creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range, showing the distribution of `theta` in 20 angle bins or less. The vector `theta`, expressed in radians, determines the angle of each bin from the origin. The length of each bin reflects the number of elements in `theta` that fall within a group, which ranges from 0 to the greatest number of elements deposited in any one bin.

`rose(theta,x)` uses the vector `x` to specify the number and the locations of bins. `length(x)` is the number of bins and the values of `x` specify the center angle of each bin. For example, if `x` is a five-element vector, `rose` distributes the elements of `theta` in five bins centered at the specified `x` values.

`rose(theta,nbins)` plots `nbins` equally spaced bins in the range `[0, 2*pi]`. The default is `20`.

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

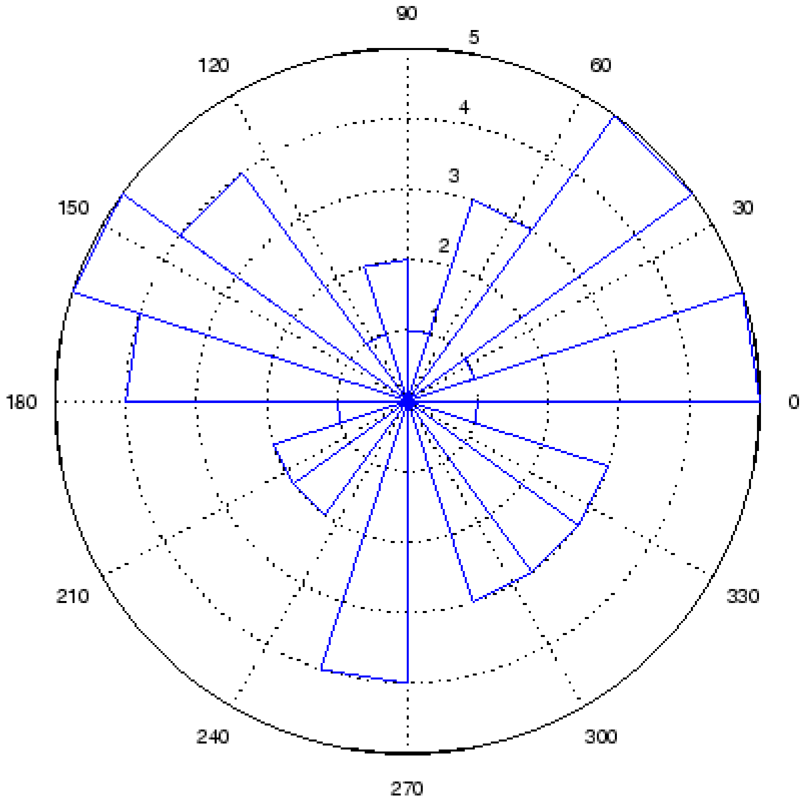
`h = rose(...)` returns the handles of the line objects used to create the graph.

`[tout,rout] = rose(...)` returns the vectors `tout` and `rout` so `polar(tout,rout)` generates the histogram for the data. This syntax does not generate a plot.

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 1-95 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 software 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 = rot90(A)
B = rot90(A,k)
```

Description

`B = rot90(A)` rotates matrix `A` counterclockwise by 90 degrees.

`B = rot90(A,k)` rotates matrix `A` counterclockwise by `k*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.

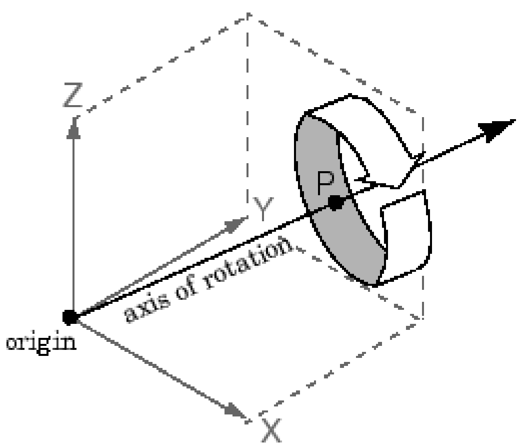
`rotate(h,direction,alpha)` rotates the graphics object `h` by `alpha` degrees. `direction` is a two- or three-element vector that describes the axis of rotation in conjunction with the origin.

`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.

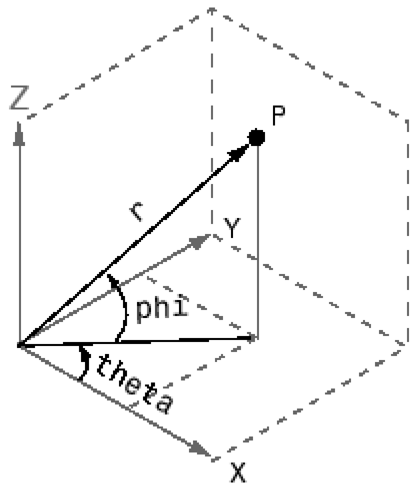
Remarks

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 `[theta phi]`. `theta` is the angle in the *x*-*y* plane counterclockwise from the positive *x*-axis. `phi` is the elevation of the direction vector from the *x*-*y* plane.



The three-element form for `direction` specifies the axis direction using Cartesian coordinates. The direction vector is the vector from the origin to (X,Y,Z).

Examples

Rotate a graphics object 180º about the *x*-axis.

```
h = surf(peaks(20));
rotate(h,[1 0 0],180)
```

Rotate a surface graphics object 45º about its center in the *z* direction.

```
h = surf(peaks(20));
zdir = [0 0 1];
center = [10 10 0];
rotate(h,zdir,45,center)
```

Remarks

`rotate` changes the `Xdata`, `Ydata`, and `Zdata` properties of the appropriate graphics object.

See Also

`rotate3d`, `sph2cart`, `view`

The axes `CameraPosition`, `CameraTarget`, `CameraUpVector`, `CameraViewAngle`

"Object Manipulation" on page 1-105 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 in the MATLAB Graphics documentation.

Syntax

```
rotate3d on
rotate3d off
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` 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 rotate mode objects via `get` and modify some of them using `set`.

- `FigureHandle <handle>` — The associated figure handle, a read-only property that cannot be set
- *`Enable`* `'on'|'off'` — Specifies whether this figure mode is currently enabled on the figure
- *`RotateStyle`* `'orbit'|'box'` — Sets the method of rotation

'orbit' rotates the entire axes; 'box' rotates a plot-box outline of the axes.

Rotate3D Mode Callbacks

You can program the following callbacks for rotate3d mode operations.

- ButtonDownFilter <function_handle> — Function to intercept ButtonDown events

  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 data object (empty in this release)
  % res [output] logical flag to determine whether the rotate
                 operation should take place or the 'ButtonDownFcn'
                 property of the object should take precedence
  ```

- ActionPreCallback <function_handle> — Function to execute before rotating

  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 Graphics object callbacks):

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

  The event data has the following field:

Axes	The handle of the axes that is being panned

- `ActionPostCallback <function_handle>` — Function to execute after rotating

  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 Graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked on
  % event_obj   object containing struct of event data (same as the
  %             event data of the 'ActionPreCallback' callback)
  ```

Rotate3D Mode Utility Functions

The following functions in pan mode query and set certain of its properties.

- `flags = isAllowAxesRotate(h,axes)` — Function querying permission to rotate 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)` — Function to set permission to pan axes

  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,'RotateStyle','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. This differs from the `camorbit` function in that while the `rotate3d` tool modifies the `View` property of the axes, the `cameraorbit` function fixes the aspect ratio and modifies the `CameraTarget`, `CameraPosition` and `CameraUpVector` properties of the axes. See for more information.

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.

Note **Do not change figure callbacks within an interactive mode.** While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a GUI that updates a figure's callbacks, the GUI should some keep track of which interactive mode is active, if any, before attempting to do this.

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.

See Also

`camorbit`, `pan`, `rotate`, `view`, `zoom`

Object Manipulation for related functions

for related properties

Purpose Round to nearest integer

Syntax `Y = round(X)`

Description `Y = round(X)` rounds the elements of `X` to the nearest integers. For complex `X`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000          -0.2000          3.4000          5.6000
  Columns 5 through 6
   7.0000           2.4000 + 3.6000i

round(a)

ans =
  Columns 1 through 4
  -2.0000                0          3.0000          6.0000
  Columns 5 through 6
   7.0000           2.0000 + 4.0000i
```

See Also `ceil`, `fix`, `floor`

Purpose

Reduced row echelon form

Syntax

```
R = 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
```

```
0     0     1    -3
0     0     0     0
```

See Also

`inv`, `lu`, `rank`

Purpose

Convert real Schur form to complex Schur form

Syntax

```
[U,T] = 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 = eye(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

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 folder to be the one in which the script file resides, executes the script, and sets the current folder 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 content
save filename options
save filename content options
save('filename', 'var1', 'var2', ...)
```

Description

`save` stores all variables from the current MATLAB workspace in a MATLAB formatted file (MAT-file) named `matlab.mat` that resides in the current working directory. Use the `load` function to retrieve data stored in MAT-files. By default, MAT-files are double-precision, binary files. You can create a MAT-file on one machine and then load it on another machine using a different floating-point format, and retaining as much accuracy and range as the different formats allow. MAT-files can also be manipulated by other programs external to MATLAB.

`save filename` stores all variables in the current workspace in the 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 a full pathname for the `filename`.

`save filename` *`content`* stores only those variables specified by *`content`* in file `filename`. If `filename` is not specified, MATLAB stores the data in a file called `matlab.mat`. See the following table.

Values for *content*	Description
`varlist`	Save only those variables that are in `varlist`. You can use the `*` wildcard to save only those variables that match the specified pattern. For example, `save('A*')` saves all variables that start with `A`.

Values for *content*	Description
-regexp exprlist	Save those variables that match any of the regular expressions in `exprlist`. See the Remarks section below.
-struct s	Save as individual variables all fields of the scalar structure `s`.
-struct s fieldlist	Save as individual variables only the specified fields of structure `s`.

In this table, the terms `varlist`, `exprlist`, and `fieldlist` refer to one or more variable names, regular expressions, or structure field names separated by either spaces or commas, depending on whether you are using the MATLAB command or function format. See the examples below:

Command format:

```
save myfile.mat firstname lastname
```

Function format:

```
save('myfile.mat', 'firstname', 'lastname')
```

`save filename` *`options`* stores all variables from the MATLAB workspace in file `filename` according to one or more of the following options. If `filename` is not specified, MATLAB stores the data in a file called `matlab.mat`.

Values for *options*	Description
-append	Add new variables to those already stored in an existing MAT-file.

Values for *options*	Description
-format	Save using the specified binary or ASCII format. See the section on, "MAT-File Format Options" on page 2-3197, below.
-version	Save in a format that can be loaded into an earlier version of MATLAB. See the section on "Version Compatibility Options" on page 2-3197, below.

`save filename` *`content options`* stores only those variables specified by *`content`* in file `filename`, also applying the specified *`options`*. If `filename` is not specified, MATLAB stores the data in a file called `matlab.mat`.

`save('filename', 'var1', 'var2', ...)` is the function form of the syntax.

MAT-File Format Options

The following table lists the valid *`MAT-file format`* options.

MAT-file format Options	How Data Is Stored
`-ascii`	Save data in 8-digit ASCII format.
`-ascii -tabs`	Save data in 8-digit ASCII format delimited with tabs.
`-ascii -double`	Save data in 16-digit ASCII format.
`-ascii -double -tabs`	Save data in 16-digit ASCII format delimited with tabs.
`-mat`	Binary MAT-file form (default).

Version Compatibility Options

The following table lists version compatibility options. These options enable you to save your workspace data to a MAT-file that can then be loaded into an earlier version of MATLAB software. The resulting

MAT-file supports only those data items and features that were available in this earlier version of MATLAB. (See the second table below for what is supported in each version.)

version Option	Use When Running ...	To Save a MAT-File That You Can Load In ...
`-v7.3`	Version 7.3 or later	Version 7.3 or later
`-v7`	Version 7.3 or later	Versions 7.0 through 7.2 (or later)
`-v6`	Version 7 or later	Versions 5 and 6 (or later)
`-v4`	Version 5 or later	Versions 1 through 4 (or later)

The default version option is the value specified in the **Preferences** dialog box. Select **File > Preferences** in the Command Window, click **General**, and then **MAT-Files** to view or change the default.

The next table shows what data items and features are supported in different versions of MATLAB. You can use this information to determine which of the version compatibility options shown above to use.

MATLAB Versions	Data Items or Features Supported
4 and earlier	Support for 2D double, character, and sparse
5 and 6	Version 4 capability plus support for ND arrays, structs, and cells
7.0 through 7.2	Version 6 capability plus support for data compression and Unicode character encoding
7.3 and later	Version 7.2 capability plus support for data items greater than or equal to 2GB

Remarks

When using the `-regexp` switch, `save` considers all variables in the argument list, with the exception of the optional filename and structure name variables, to be regular expressions. The filename, if specified, is always the first argument in the argument list, provided that this argument is a variable name. The structure name, if specified, is always the first argument following the `-struct` keyword, provided that the argument list includes that keyword.

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` option to do this:

```
save -v7.3 myfile v1 v2
```

If you are running MATLAB on a 64-bit computer system and you attempt to save a variable that is too large for a version 7 (or earlier) MAT-file, that is, you save without using the `-v7.3` option, MATLAB skips that variable during the save operation and issues a warning message to that effect.

If you are running MATLAB on a 32-bit computer system and attempt to load a variable from a `-v7.3` MAT-file that is too large to fit in 32–bit address space, MATLAB skips that variable and issues a warning message to that effect.

MAT-files saved with compression and Unicode encoding cannot be loaded into versions of MATLAB prior to MATLAB Version 7.0. If you save data to a MAT-file that you intend to load using MATLAB Version 6 or earlier, you must specify the `-v6` option when saving. This disables compression and Unicode encoding for that particular `save` operation.

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 Version 4 are not saved to the MAT-file. For example, ND arrays, structs, cells, etc. cannot be saved to a MATLAB Version 4 MAT-file. Also, variables with names that are longer than 19 characters cannot be saved to a MATLAB Version 4 MAT-file.

For information on any of the following topics related to saving to MAT-files, see in the MATLAB Data Import and Export 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 values of variables `vol` and `temp` in ASCII format to a file named `june10`:

```
save('d:\mymfiles\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 -v6 c1 x y     % Save without compression
save -v7 c2 x y     % Save with compression

d1 = dir('c1.mat');
d2 = dir('c2.mat');

d1.bytes
ans =
    45000240      % Size of the uncompressed data in bytes.
d2.bytes
ans =
    11985283      % Size of the compressed data in bytes.

d2.bytes/d1.bytes
ans =
    0.2663        % Ratio of compressed to uncompressed
```

See Also

`load`, `clear`, `diary`, `fileformats`, `fprintf`, `fwrite`, `genvarname`, `who`, `whos`, `workspace`, `regexp`

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.

Remarks

COM functions are available on Microsoft Windows systems only.

Examples

Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.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
```

MATLAB displays the original values:

```
ans =
     Label: 'Label'
    Radius: 20
```

See Also

`load (COM)`, `actxcontrol`, `actxserver`, `release`, `delete (COM)`

Purpose

Save serial port objects and variables to MAT-file

Syntax

```
save filename
save filename obj1 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` on a Windows platform

```
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.

Example

This example illustrates how to use the command and functional form of `save` on a Windows platform.

```
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 in the MATLAB Graphics documentation. 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
pdf	Portable Document Format
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 Setup** 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.

Note Whenever you specify a format for saving a figure with the **Save As** menu item , that file format is used again the next time you save that figure or a new one. If you do not want to save in the previously-used format, use **Save As** and be sure to set the **Save as type** drop-down menu to the kind of file you want to write. However, saving a figure with the `saveas` function and a format does not change the **Save as type** setting in the GUI.

If you want to control the size or resolution of figures saved in image (bit-mapped) 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_prey` using the MATLAB `fig` format. This allows you to open the file `pred_prey.fig` at a later time and continue editing it with the Plot Editor.

```
saveas(gcf,'pred_prey.fig')
```

Example 2: Specify File Format but No Extension

Save the current figure, using Adobe Illustrator format, to the file `logo`. Use the `ai` extension from the above table to specify the format. The file created is `logo.ai`.

```
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`

"Printing" on page 1-97 for related functions

Simulink users, see also `save_system`

Purpose

Modify save process for object

Syntax

`b = saveobj(a)`

Description

`b = saveobj(a)` is called by the `save` function if the class of `a` defines a `saveobj` method. `save` writes the returned value, `b`, to the MAT-file.

Define a `loadobj` method to take the appropriate action when loading the object.

If `A` is an array of objects, MATLAB invokes `saveobj` separately for each object saved.

Examples

Call the superclass `saveobj` method from the subclass implementation of `saveobj` with the following syntax:

```
classdef mySub < super
   methods
      function sobj = saveobj(obj)
         % Call superclass saveobj method
         sobj = saveobj@super(obj);
         % Perform subclass save operations
         ...
      end
   ...
   end
...
end
```

See .

Update object when saved:

```
function b = saveobj(a)
   % If the object does not have an account number,
   % call method to add account number to AccountNumber property
   if isempty(a.AccountNumber)
      a.AccountNumber = getAccountNumber(a);
```

```
      end
      b = a;
   end
```

See .

See Also

`save` | `load` | `loadobj`

Tutorials

-

Purpose

Save current search path

GUI Alternatives

As an alternative to the `savepath` function, use the Set Path dialog box.

Syntax

```
savepath
savepath folderName/pathdef.m
status = savepath...
```

Description

`savepath` saves the current MATLAB search path for use in a future session. `savepath` saves the search path to the `pathdef.m` file that MATLAB located at startup, or to the current folder if a `pathdef.m` file exists there. To save the search path programmatically each time you exit MATLAB, use `savepath` in a `finish.m` file.

`savepath folderName/`**`pathdef.m`** saves the current search path to `pathdef.m` located in `folderName`. Use this form of the syntax if you do not have write access to the current `pathdef.m`. If you do not specify `folderName`, MATLAB saves `pathdef.m` in the current folder. `folderName` can be a relative or absolute path. To use the saved search path automatically in a future session, make `folderName` be the startup folder for MATLAB.

`status = savepath...` returns `0` when `savepath` was successful and `1` when `savepath` failed.

Examples

Save the current search path to `pathdef.m`, located in `I:/my_matlab_files`:

```
savepath I:/my_matlab_files/pathdef.m
```

See Also

`addpath`, `cd`, `dir`, `finish`, `genpath`, `matlabroot`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `rmpath`, `startup`, `userpath`, `what`

Topics in the User Guide:

- .

•

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 1-by-3 matrix, it specifies the colors of the markers as RGB values. If you have 3 points in the scatter plot and wish to have the colors be indices into the colormap, `C` should be a 3-by-1 matrix. `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 `LineSpec` 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

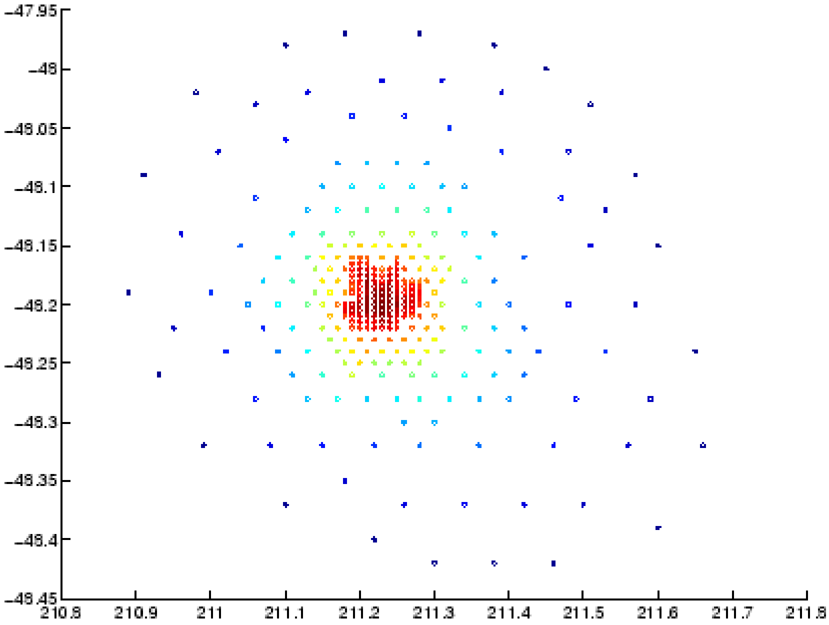
`hpatch = scatter('v6',...)` returns the handles to the patch objects created by `scatter` (see `Patch` Properties for a list of properties you can specify using the object handles and `set`).

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Example

```
load seamount
scatter(x,y,5,z)
```



See Also

`scatter3`, `plot3`

"Scatter/Bubble Plots" on page 1-96 for related functions

See `Scattergroup Properties` for property descriptions.

scatter3

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')
scatter3(...,'PropertyName',propertyvalue)
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 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 1-by-3 matrix, it specifies the colors of the markers as RGB values. If you have 3 points in the scatter plot and wish to have the colors be indices into the colormap, `C` should be a 3-by-1 matrix. `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(...,markertype)` uses the marker type specified instead of 'o' (see `LineSpec` for a list of marker specifiers).

`scatter3(...,'filled')` fills the markers.

`scatter3(...,'`*`PropertyName`*`',propertyvalue)` creates the scatter graph, applying the specified property settings. See `scattergroup` properties for a description of properties.

`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`).

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

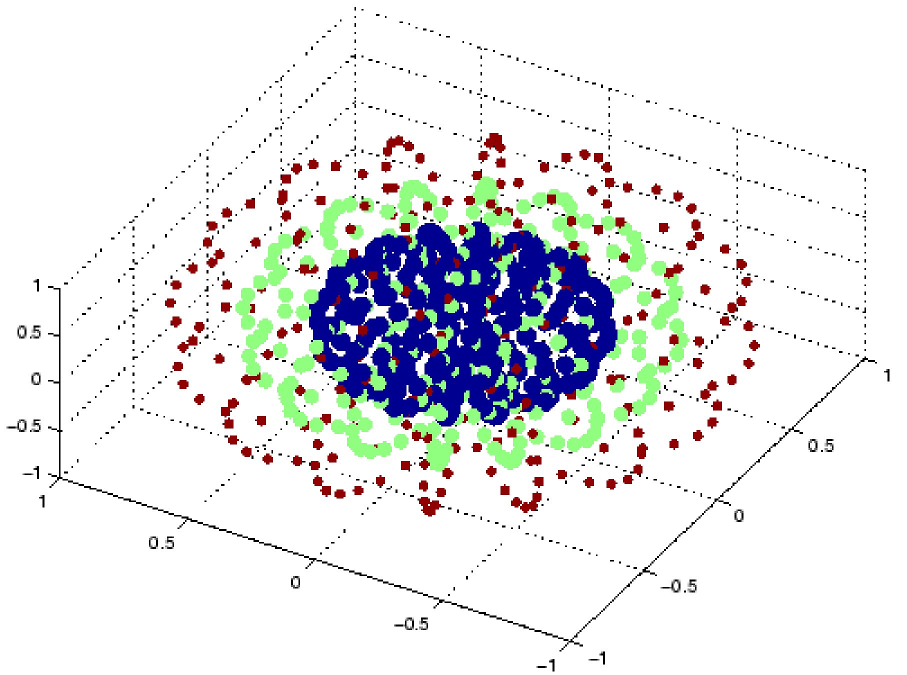
See Plot Objects and Backward Compatibility for more information.

Remarks

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 1-96 for related functions

Purpose

Define 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.

`Annotation`

`hg.Annotation` object Read Only

Control the display of scattergroup objects in legends. The `Annotation` property enables you to specify whether this scattergroup object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the scattergroup object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the scattergroup object in a legend as one entry, but not its children objects

IconDisplayStyle Value	Purpose
`off`	Do not include the scattergroup or its children in a legend (default)
`children`	Include only the children of the scattergroup as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See for more information and examples.

`BeingDeleted`

`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

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

`BusyAction`
    `cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

`ButtonDownFcn`
    string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

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

This property can be

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

- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See 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.

`CDataSource`
string (MATLAB variable)

Link CData 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 this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

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

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`CreateFcn`

string or function handle

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

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

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

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

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

`DeleteFcn`

string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

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

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

See the `BeingDeleted` property for related information.

`DisplayName`

string (default is empty string)

String used by legend for this scattergroup object. The `legend` function uses the string defined by the `DisplayName` property to label this scattergroup object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this scattergroup object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See for more examples.

`EraseMode`

`{normal} | none | xor | background`

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

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

- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

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

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

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

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

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

`HitTest`
`{on} | off`

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

`HitTestArea`
`on | {off}`

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click th eobject's lines or markers (excluding the baseline, if any) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`
character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

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 `CData` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

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

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`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`.

`SizeDataSource`
string (MATLAB variable)

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

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

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.

`Tag`

string

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

For example, you might create an areaseries object and set the `Tag` property.

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

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

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

`Type`

string (read only)

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

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

`UserData`
array

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

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
array

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 is evaluated in the base workspace to generate the `XData`.

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

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

See the `refreshdata` reference page for more information.

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

`YData`

scalar, vector, or matrix

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)

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

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

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

See the `refreshdata` reference page for more information.

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

`ZData`

vector of coordinates

Z-coordinates. A vector defining the *z*-coordinates for the graph. `XData` and `YData` must be the same length and have the same number of rows.

`ZDataSource`

string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

Purpose

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'`	`T` is triangular and is complex if `A` has complex eigenvalues.
`'real'`	`T` 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)

Matrix A	Routine
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

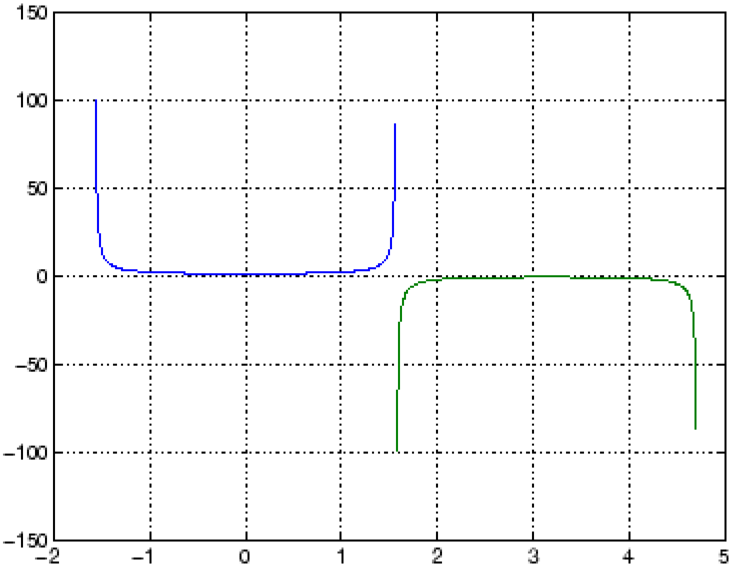
The `sec` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`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 `sec(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

$$\sec(z) = \frac{1}{\cos(z)}$$

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 = secd(X)
```

Description `Y = secd(X)` is the secant of the elements of `X`, expressed in degrees. For odd integers `n`, `secd(n*90)` is infinite, whereas `sec(n*pi/2)` is large but finite, reflecting the accuracy of the floating point value of `pi`.

See Also `sec`, `sech`, `asec`, `asecd`, `asech`

Purpose

Hyperbolic secant

Syntax

```
Y = sech(X)
```

Description

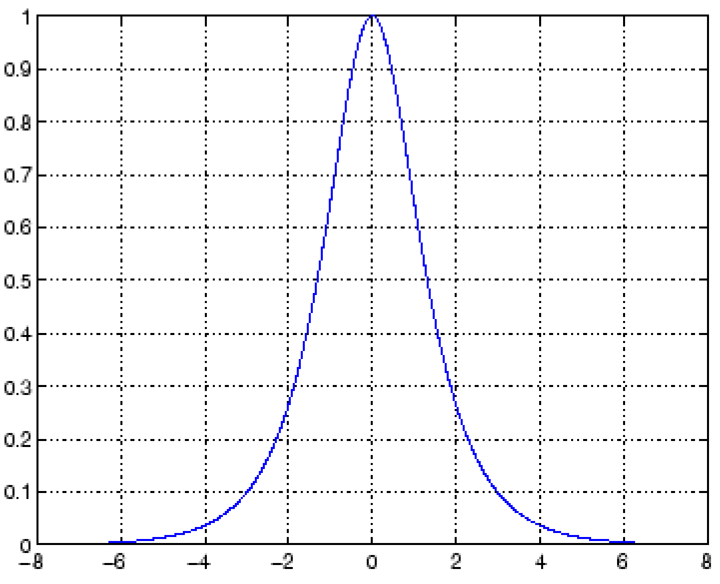
The `sech` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = 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 $-2\pi \le x \le 2\pi$.

```
x = -2*pi:0.01:2*pi;
plot(x,sech(x)), grid on
```



Algorithm

sech uses this algorithm.

$$\text{sech}(z) = \frac{1}{\cosh(z)}$$

Definition

The secant can be defined as

$$\text{sech}(z) = \frac{1}{\cosh(z)}$$

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

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,LineSpec,...)
semilogx(...,'PropertyName',PropertyValue,...)
h = semilogx(...)
h = semilogy(...)
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 `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `semilogx` plots the vector argument versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`semilogx(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples. `LineSpec` determines line style, marker symbol, and color of the plotted lines.

`semilogx(...,'PropertyName',PropertyValue,...)` sets property values for all `lineseries` graphics objects created by `semilogx`.

`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.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

If you do not specify a color when plotting more than one line, `semilogx` and `semilogy` automatically cycle through the colors and line styles in the order specified by the current axes `ColorOrder` and `LineStyleOrder` properties.

You can mix `Xn,Yn` pairs with `Xn,Yn,LineSpec` triples; for example,

```
semilogx(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

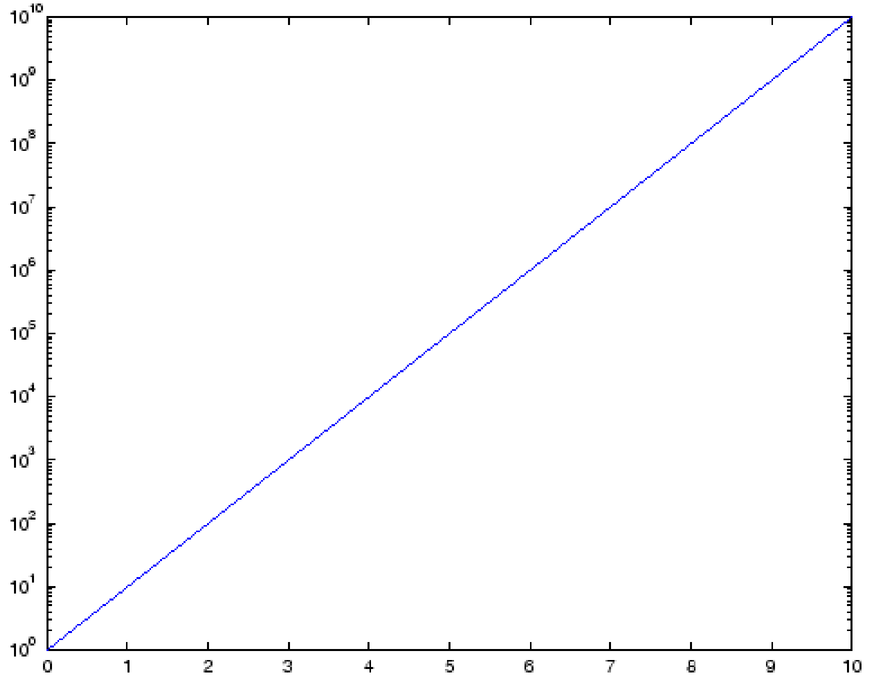
If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode will remain as it is and the new data will plot as linear.

Examples

Create a simple `semilogy` plot.

```
x = 0:.1:10;
```

```
semilogy(x,10.^x)
```



See Also

`line`, `LineSpec`, `loglog`, `plot`

"Basic Plots and Graphs" on page 1-91 for related functions

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 2-3254. 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. The MATLAB software 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

How To

•

Purpose

Create serial port object

Syntax

```
obj = serial('port')
obj = serial('port','PropertyName',PropertyValue,...)
```

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.

`Port` object name will depend upon the platform that the serial port is on. `insthwinfo` ('serial') provides a list of available serial ports. This list is an example of serial constructors on different platforms:

Platform	Serial Port Constructor
Linux and Linux 64	`serial('/dev/ttyS0');`
Mac OS X and Mac OS X 64	`serial('/dev/tty.KeySerial1');`
Solaris 64	`serial('/dev/term/a');`
Windows 32 and Windows 64	`serial('com1');`

`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 on a Windows platform.

```
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 on a Windows platform.

```
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)
```

Description `serialbreak(obj)` sends a break of 10 milliseconds to the device connected to the serial port object, `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 Handle Graphics object properties

Syntax

```
set(H,'PropertyName',PropertyValue,...)
set(H,a)
set(H,pn,pv,...)
set(H,pn,MxN_pv)
a = set(h)
pv = set(h,'PropertyName')
```

Description

Note Do not use the `set` function on Java objects as it will cause a memory leak. For more information, see

`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,MxN_pv)` 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, the MATLAB software displays the information on the screen. `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 a statement indicating that `PropertyName` does not have a fixed set of property values. 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 the MATLAB software can correctly interpret the specified `FontSize`. The same applies to figure and axes uints — 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.

```
axes;
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 the 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`

"Graphics Object Identification" on page 1-98 for related functions

Purpose

Set object or interface property to specified value

Syntax

```
h.set('pname', value)
h.set('pname1', value1, 'pname2', value2, ...)
set(h, ...)
```

Description

`h.set('pname', value)` sets the property specified in the string `pname` to the given `value`.

`h.set('pname1', value1, 'pname2', value2, ...)` sets each property specified in the `pname` strings to the given `value`.

`set(h, ...)` is an alternate syntax for the same operation.

See in the External Interfaces documentation for information on how MATLAB converts workspace matrices to COM data types.

Remarks

COM functions are available on Microsoft Windows systems only.

Examples

Create an `mwsamp` control and use `set` to change the `Label` and `Radius` properties:

```
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');
```

Here is another way to do the same thing, only without `set` and `invoke`:

```
h.Label = 'Click to fire event';
h.Radius = 40;
h.Redraw;
```

See Also

`get (COM)`, `inspect`, `isprop`, `addproperty`, `deleteproperty`

set (hgsetget)

Purpose

Assign property values to handle objects derived from `hgsetget` class

Syntax

```
set(H,'PropertyName',value,...)
set(H,pn,pv)
set(H,S)
pv = set(h,'PropertyName')
S = set(h)
```

Description

`set(H,'PropertyName',value,...)` sets the named property to the specified value for the objects in the handle array `H`.

`set(H,pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv` for all objects specified in `H`. The cell array `pn` must be `1`-by-`n`, but the cell array `pv` can be `m`-by-`n` where `m` is equal to `length(H)`. `set` updates each object with a different set of values for the list of property names contained in `pn`.

`set(H,S)` sets the properties identified by each field name of `struct` `S` with the values contained in `S`. `S` is a `struct` whose field names are object property names.

`pv = set(h,'`*`PropertyName`*`')` returns the possible values for the named property.

`S = set(h)` returns the user-settable properties and possible values for the handle object `h`. `S` is a `struct` whose field names are the object's property names and whose values are cell arrays containing the possible values of the corresponding properties. The cell array is empty for properties that do not have finite possible values.

You can use property/value string pairs, `structs`, and property/value cell array pairs in the same call to `set`.

Override the `hgsetget` class `setdisp` method to change how MATLAB displays this information.

See Also

See

`handle`, `hgsetget`, `set`, `get (hgsetget)`

Purpose

Set random stream property

Class

`@RandStream`

Syntax

```
set(S,'PropertyName',Value)
set(S,'Property1',Value1,'Property2',Value2,...)
set(S,A)
A=set(S,'Property')
set(S,'Property')
A=set(S)
set(S)
```

Description

`set(S,'PropertyName',Value)` sets the property `'PropertyName'` of the random stream `S` to the value `Value`.

`set(S,'Property1',Value1,'Property2',Value2,...)` sets multiple random stream property values with a single statement.

`set(S,A)` where `A` is a structure whose field names are property names of the random stream `S` sets the properties of `S` named by each field with the values contained in those fields.

`A=set(S,'Property')` or `set(S,'Property')` displays possible values for the specified property of `S`.

`A=set(S)` or `set(S)` displays or returns all properties of `S` and their possible values.

See Also

`@RandStream`, `get (RandStream)`, `rand`, `randn`, `randi`

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)
```

Description

`set(obj)` displays all configurable properties values for the serial port object, `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
```

Examples

This example illustrates some of the ways you can use `set` to configure or return property values for the serial port object `s`, on a Windows platform.

```
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`

set (timer)

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(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', 'callbk', 'Period', 10)
```

Use a cell array to specify the names of the properties you want to set and another cell array to specify the values of these properties.

```
set(t, {'StartDelay', 'Period'}, {30, 30})
```

See Also

`timer`, `get(timer)`

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`

setabstime (tscollection)

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`

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.

Remarks Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property.

See Also `getappdata`, `isappdata`, `rmappdata`

setDefaultStream (RandStream)

Purpose Set default random number stream

Syntax

```
prevstream = RandStream.setDefaultStream(stream)
```

Description `prevstream = RandStream.setDefaultStream(stream)` returns the current default random number stream, and designates the random number stream `stream` as the new default to be used by the `rand`, `randi`, and `randn` functions.

`rand`, `randi`, and `randn` all rely on the same stream of uniform pseudorandom numbers, known as the default stream. `randi` uses one uniform value from the default stream to generate each integer value. `randn` uses one or more uniform values from the default stream to generate each normal value. Note that there are also `rand`, `randi`, and `randn` methods for which you specify a specific random stream from which to draw values.

See Also `getDefaultStream (RandStream)`, `@RandStream`, `rand (RandStream)`, `randn (RandStream)`, `randperm (RandStream)`

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`

Tiff.setDirectory

Purpose Make specified IFD current IFD

Syntax `tiffobj.setDirectory(dirNum)`

Description `tiffobj.setDirectory(dirNum)` sets the image file directory (IFD) specified by `dirNum` as the current IFD. `Tiff` object methods operate on the current IFD. The directory index number is one-based.

Examples Open a TIFF file and move to an IFD in the file by specifying its index number. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path. The TIFF file should contain multiple images.

```
t = Tiff('myfile.tif','r');
t.setDirectory(2);
```

References This method corresponds to the `TIFFSetDirectory` function in the LibTIFF C API. To use this method, you must be familiar with LibTIFF version 3.7.1 as well as the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also `Tiff.currentDirectory` | `Tiff.nextDirectory`

Tutorials

-
-

Purpose

Override to change command window display

Syntax

`setdisp(H)`

Description

`setdisp(H)` called by `set` when `set` is called with no output arguments and a single input argument that is a handle array. Override this `hgsetget` class method in a subclass to change how property information is displayed in the command window.

See Also

See

`hgsetget`, `set (hgsetget)`

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`. On the Microsoft Windows platform, this is equivalent to undefining the variable. On most UNIX[17] 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:

17. UNIX is a registered trademark of The Open Group in the United States and other countries.

```
setenv('PATH', [getenv('PATH') ';D:\Perl\bin']);
```

See Also

getenv, system, unix, dos, !

setfield

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`, the MATLAB software 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 for help on 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

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`, dynamic field names

setinterpmethod

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);
```

```
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

setpixelposition

Purpose

Set component position in pixels

Syntax

```
setpixelposition(handle,position)
setpixelposition(handle,position,recursive)
```

Description

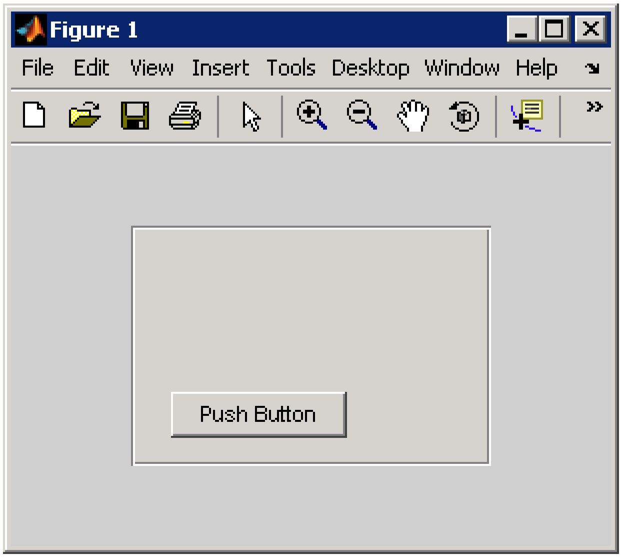
`setpixelposition(handle,position)` sets the position of the component specified by `handle`, to the specified position relative to its parent. `position` is a four-element vector that specifies the location and size of the component: [pixels from left, pixels from bottom, pixels across, pixels high].

`setpixelposition(handle,position,recursive)` sets the position as above. If Boolean `recursive` is true, the position is set relative to the parent figure of `handle`.

Example

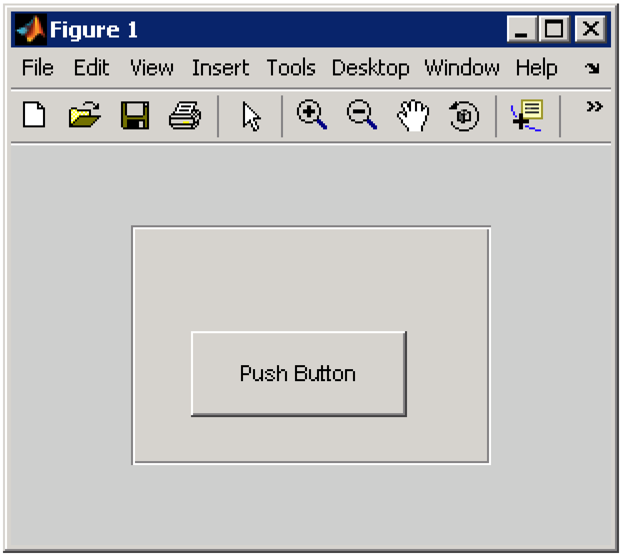
This example first creates a push button within a panel.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6]);
h1 = uicontrol(p,'Style','PushButton','Units','normalized',...
               'String','Push Button','Position',[.1 .1 .5 .2]);
```



The example then retrieves the position of the push button and changes its position with respect to the panel.

```
pos1 = getpixelposition(h1);
setpixelposition(h1,pos1 + [10 10 25 25]);
```



See Also

getpixelposition, uicontrol, uipanel

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`

Purpose Set string flag

Description This MATLAB 4 function has been renamed `char` in MATLAB 5.

Purpose Make subIFD specified by byte offset current IFD

Syntax `tiffobj.setSubDirectory(offset)`

Description `tiffobj.setSubDirectory(offset)` sets the subimage file directory (subIFD) specified by `offset` the current IFD. The offset value is given in bytes. Use this method when you want to access subIFDs linked through the `SubIFD` tag.

Examples Open a TIFF file and read the value of the `SubIFD` tag in the current IFD. The `SubIFD` tag contains byte offsets that specify the location of subIFDs in the IFD. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path. The TIFF file should contain subIFDs.

```
t = Tiff('myfile.tif','r');
%
% Read the value of the SubIFD tag to get subdirectory offsets.
offsets = t.getTag('SubIFD');
%
% Set one of the subdirectories (if more than one) as the current d
t.setSubDirectory(offsets(1));
```

References This method corresponds to the `TIFFSetSubDirectory` function in the LibTIFF C API. To use this method, you must be familiar with LibTIFF version 3.7.1, as well as the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also `Tiff.setDirectory`

Tutorials

-
-

Tiff.setTag

Purpose Set value of tag

Syntax

```
tiffobj.setTag(tagId,tagValue)
tiffobj.setTag(tagStruct)
```

Description

`tiffobj.setTag(tagId,tagValue)` sets the value of the TIFF tag specified by `tagId` to the value specified by `tagValue`. You can specify `tagId` as a character string (`'ImageWidth'`) or using the numeric tag identifier defined by the TIFF specification (256). To see a list of all the tags with their numeric identifiers, view the value of the Tiff object `TagID` property. Use the `TagID` property to specify the value of a tag. For example, `Tiff.TagID.ImageWidth` is equivalent to the tag's numeric identifier.

`tiffobj.setTag(tagStruct)` sets the values of all of the tags with name/value fields in `tagStruct`. The names of fields in `tagstruct` must be the name of TIFF tags.

Examples

Create a structure with fields named after TIFF tags and assign values to the fields. Pass this structure to the `setTag` method to set the values of these tags. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r+');

tagStruct.ImageWidth = 1600;
tagStruct.ImageLength = 3200;
tagStruct.Photometric = Tiff.Photometric.RGB;
tagStruct.BitPerSample = 8;
tagStruct.SamplesPerPixel = 3;
tagStruct.TileWidth = 160;
tagStruct.TileLength = 320;
tagStruct.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
tagStruct.Software = 'MATLAB';
t.setTag(tagStruct);
```

References

This method corresponds to the `TIFFSetField` function in the LibTIFF C API. To use this method, you must be familiar with LibTIFF version 3.7.1, as well as the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.getTag`

Tutorials

•

•

Purpose

Change name of `timeseries` object in `tscollection`

Syntax

```
tsc = settimeseriesnames(tsc,old,new)
```

Description

`tsc = settimeseriesnames(tsc,old,new)` replaces the `old` name of `timeseries` object with the new name in tsc.

See Also

`tscollection`

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`

shading

Purpose

Set color shading properties

Syntax

```
shading flat
shading faceted
shading interp
shading(axes_handle,...)
```

Description

The `shading` function controls the color shading of surface and patch graphics objects.

`shading flat` each mesh line segment and face has a constant color determined by the color value at the endpoint of the segment or the corner of the face that has the smallest index or indices.

`shading faceted` flat shading with superimposed black mesh lines. This is the default shading mode.

`shading interp` varies the color in each line segment and face by interpolating the colormap index or true color value across the line or face.

`shading(axes_handle,...)` applies the shading type to the objects in the axes specified by `axes_handle`, instead of the current axes.

Examples

Compare a flat, faceted, and interpolated-shaded sphere.

```
subplot(3,1,1)
sphere(16)
axis square
shading flat
title('Flat Shading')

subplot(3,1,2)
sphere(16)
axis square
shading faceted
title('Faceted Shading')

subplot(3,1,3)
```

```
sphere(16)
axis square
shading interp
title('Interpolated Shading')
```

Algorithm

`shading` sets the `EdgeColor` and `FaceColor` properties of all surface and patch graphics objects in the current axes. `shading` sets the appropriate values, depending on whether the surface or patch objects represent meshes or solid surfaces.

See Also

`fill`, `fill3`, `hidden`, `light`, `lighting`, `mesh`, `patch`, `pcolor`, `surf`

The `EdgeColor` and `FaceColor` properties for patch and surface graphics objects.

"Color Operations" on page 1-103 for related functions

shg

Purpose Show most recent graph window

Syntax `shg`

Description `shg` makes the current figure visible and raises it above all other figures on the screen. This is identical to using the command `figure(gca)`.

See Also `figure`, `gca`, `gcf`

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.
```

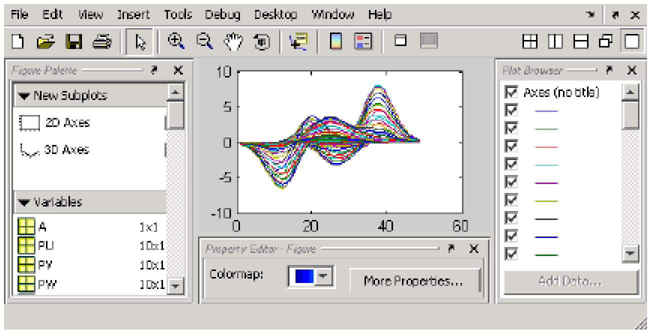
See Also

`circshift`, `reshape`, `squeeze`, `permute`, `ipermute`

showplottool

Purpose

Show or hide figure plot tool



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 in the MATLAB Graphics documentation.

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.

Note When you dock, undock, resize, or reposition a plotting tool and then close it, it will still be configured as you left it the next time you open it. There is no command to reset plotting tools to their original, default locations.

See Also

`figurepalette`, `plotbrowser`, `plottools`, `propertyeditor`

shrinkfaces

Purpose

Reduce size of patch faces

Syntax

```
shrinkfaces(p,sf)
nfv = shrinkfaces(p,sf)
nfv = shrinkfaces(fv,sf)
shrinkfaces(p)
nfv = shrinkfaces(f,v,sf)
[nf,nv] = shrinkfaces(...)
```

Description

`shrinkfaces(p,sf)` shrinks the area of the faces in patch `p` to shrink factor `sf`. A shrink factor of 0.6 shrinks each face to 60% of its original area. If the patch contains shared vertices, the MATLAB software creates nonshared vertices before performing the face-area reduction.

`nfv = shrinkfaces(p,sf)` returns the face and vertex data in the struct `nfv`, but does not set the `Faces` and `Vertices` properties of patch `p`.

`nfv = shrinkfaces(fv,sf)` uses the face and vertex data from the struct `fv`.

`shrinkfaces(p)` and `shrinkfaces(fv)` (without specifying a shrink factor) assume a shrink factor of 0.3.

`nfv = shrinkfaces(f,v,sf)` uses the face and vertex data from the arrays `f` and `v`.

`[nf,nv] = shrinkfaces(...)` returns the face and vertex data in two separate arrays instead of a struct.

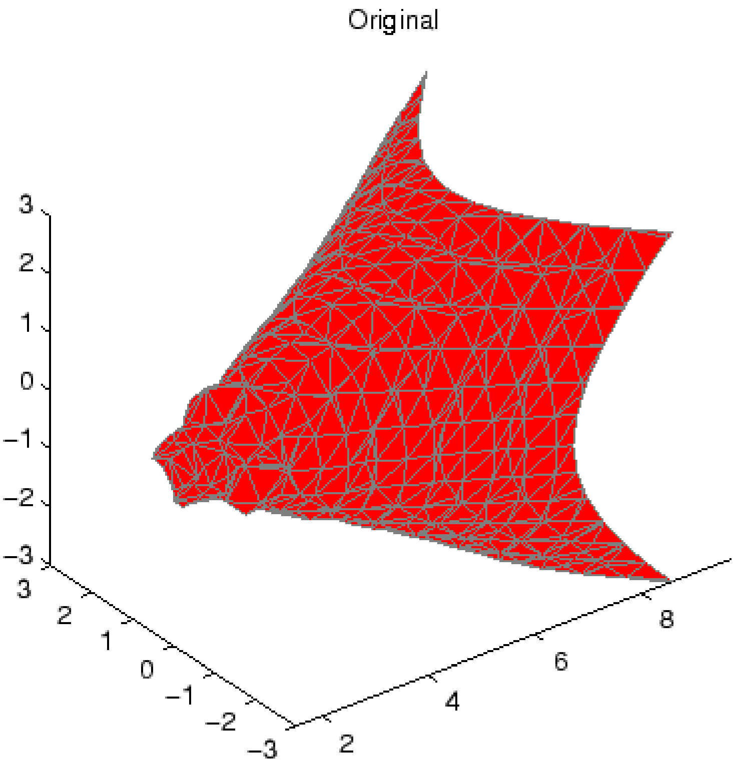
Examples

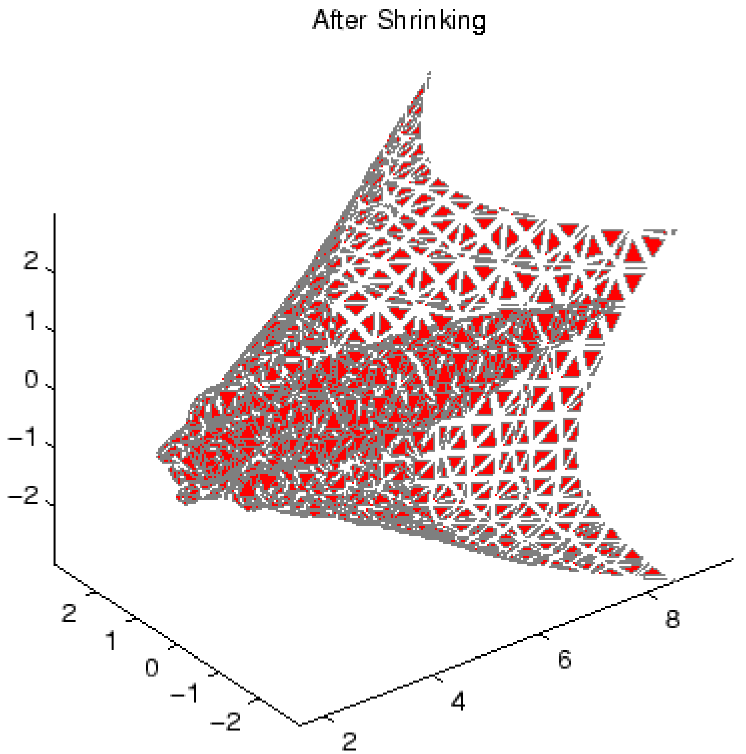
This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type `help flow` for more information). Two isosurfaces provide a before and after view of the effects of shrinking the face size.

- First `reducevolume` samples the flow data at every other point and then `isosurface` generates the faces and vertices data.

- The patch command accepts the face/vertex struct and draws the first (p1) isosurface.
- Use the daspect, view, and axis commands to set up the view and then add a title.
- The shrinkfaces command modifies the face/vertex data and passes it directly to patch.

```
[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')
```


Original
3
2
1
0
−1
−2
−3
3
2
1
0
−1
−2
−3
2
4
6
8



See Also

`isosurface`, `patch`, `reducevolume`, `daspect`, `view`, `axis`

"Volume Visualization" on page 1-106 for related functions

sign

Purpose Signum function

Syntax

```
Y = sign(X)
```

Description `Y = 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`, `sign(X) = X./abs(X)`.

See Also `abs`, `conj`, `imag`, `real`

Purpose Sine of argument in radians

Syntax

```
Y = sin(X)
```

Description `Y = sin(X)` returns the circular sine of the elements of `X`. The `sin` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Definitions The sine of an angle is:

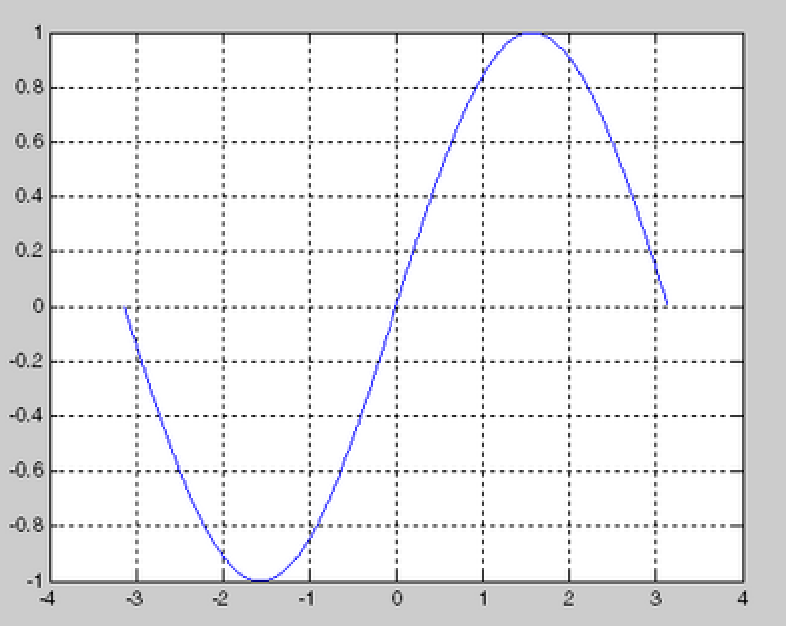
$$\sin(x) = \frac{e^{ix} - e^{-ix}}{2i}$$

For complex x:

$$\sin(x + iy) = \sin(x)\cosh(y) + i\cos(x)\sinh(y)$$

Examples Graph the sine function over the domain $-\pi \le x \le \pi$.

```
x = -pi:0.01:pi;
plot(x,sin(x)), grid on
```



References

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

Purpose Sine of argument in degrees

Syntax

```
Y = sind(X)
```

Description `Y = sind(X)` is the sine of the elements of `X`, expressed in degrees.

Examples For integers `n`, `sind(n*180)` is exactly zero, whereas `sin(n*pi)` reflects the accuracy of the floating point value of `pi`.

```
isequal(sind(180),sin(pi))
```

See Also `sin`

single

Purpose

Convert to single precision

Syntax

```
B = single(A)
```

Description

`B = single(A)` converts the matrix `A` to single precision, returning that value in `B`. `A` can be any numeric object (such as a `double`). If `A` is already single precision, `single` has no effect. Single-precision quantities require less storage than double-precision quantities, but have less precision and a smaller range.

The `single` class is primarily meant to be used to store single-precision values. Hence most operations that manipulate arrays without changing their elements are defined. Examples are `reshape`, `size`, the relational operators, subscripted assignment, and subscripted reference.

You can define your own methods for the `single` class by placing the appropriately named method in an `@single` directory within a directory on your path.

Examples

```
a = magic(4);
b = single(a);

whos
  Name      Size         Bytes  Class

  a         4x4            128  double array
  b         4x4             64  single array
```

See Also

`double`

Purpose

Hyperbolic sine of argument in radians

Syntax

```
Y = sinh(X)
```

Description

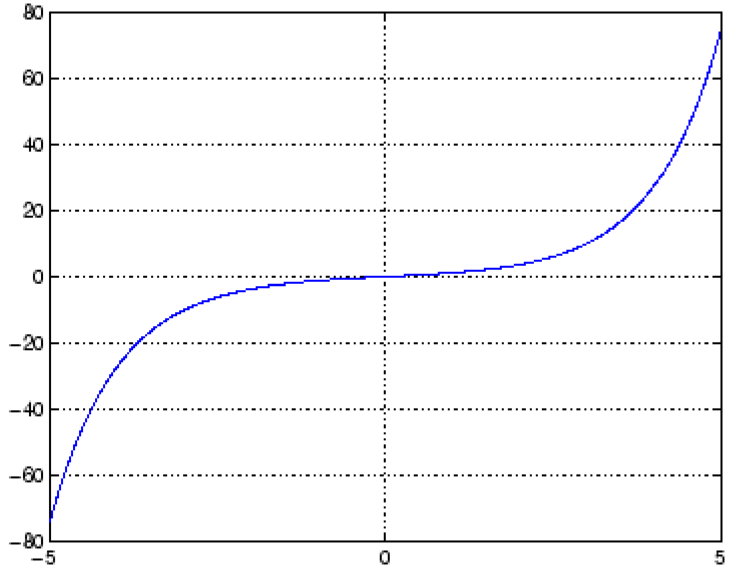
The `sinh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = sinh(X)` returns the hyperbolic sine of the elements of `X`.

Examples

Graph the hyperbolic sine function over the domain $-5 \le x \le 5$.

```
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`

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 software 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 < ndims(X)`	`di` equals the size of the `i`th dimension of `X` for $1 \le i < n$, but `dn` equals the product of the sizes of the remaining dimensions of `X`, that is, dimensions `n` through `ndims(X)`.
`n > 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 = 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 `ndims(X)`:

```
[d1,d2] = size(X)

d1 =        d2 =
     3           20
```

The “extra” dimensions are collapsed into a single product.

If `n > 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`

size (Map)

Purpose

size of `containers.Map` object

Syntax

```
d = size(M)
d = size(M, dim)
[d1, d2, ..., dn] = size(M)
```

Description

`d = size(M)` returns the number of key-value pairs in dimensions 1 and 2 of map `M`. Output `d` is a two-element row vector `[n,1]`, where `n` is the number of key-value pairs.

`d = size(M, dim)` returns the number of key-value pairs if `dim` is 1, and otherwise returns 1.

`[d1, d2, ..., dn] = size(M)` returns `[n, 1, ..., 1]` where `n` is the number of key-value pairs in map `M`.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

Examples

Create a `Map` object containing the names of several US states and the capital city of each:

```
US_Capitals = containers.Map( ...
{'Arizona', 'Nebraska', 'Nevada', 'New York', ...
 'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...
{'Phoenix', 'Lincoln', 'Carson City', 'Albany', ...
 'Atlanta', 'Juneau', 'Montpelier', 'Salem'})
```

Get the dimensions of the `Map` object array:

```
size(US_Capitals)
ans =
     8     1
```

Use the map to find the capital of one of these states:

```
state = 'Georgia';
sprintf('   The capital of %s is %s', ...
        state, US_Capitals(state))
```

```
ans =
   The capital of Georgia is Atlanta
```

See Also

containers.Map, keys(Map), values(Map), length(Map), isKey(Map), remove(Map), handle

size (serial)

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)
```

Description

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in the serial port object, `obj`.

`[m,n] = size(obj)` returns the number of rows, `m` and columns, `n` 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`

Purpose

Size of `timeseries` object

Syntax

```
size(ts)
```

Description

`size(ts)` returns `[n 1]`, where `n` is the length of the time vector for `timeseries` object `ts`.

Remarks

If you want the size of the whole data set, use the following syntax:

```
size(ts.data)
```

If you want the size of each data sample, use the following syntax:

```
getdatasamplesize(ts)
```

See Also

`getdatasamplesize`, `isempty (timeseries)`, `length (timeseries)`

TriRep.size

Purpose Size of triangulation matrix

Syntax `size(TR)`

Description `size(TR)` provides size information for a triangulation matrix. The matrix is of size `mtri`-by-`nv`, where `mtri` is the number of simplices and `nv` is the number of vertices per simplex (triangle/tetrahedron, etc).

Inputs

`TR`	Triangulation matrix

Definitions A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

See Also `size`

Purpose Size of `tscollection` object

Syntax `size(tsc)`

Description `size(tsc)` returns `[n m]`, where `n` is the length of the time vector and `m` is the number of `tscollection` members.

See Also `length (tscollection)`, `isempty (tscollection)`, `tscollection`

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 the handle `axes_handle` instead of into 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`.

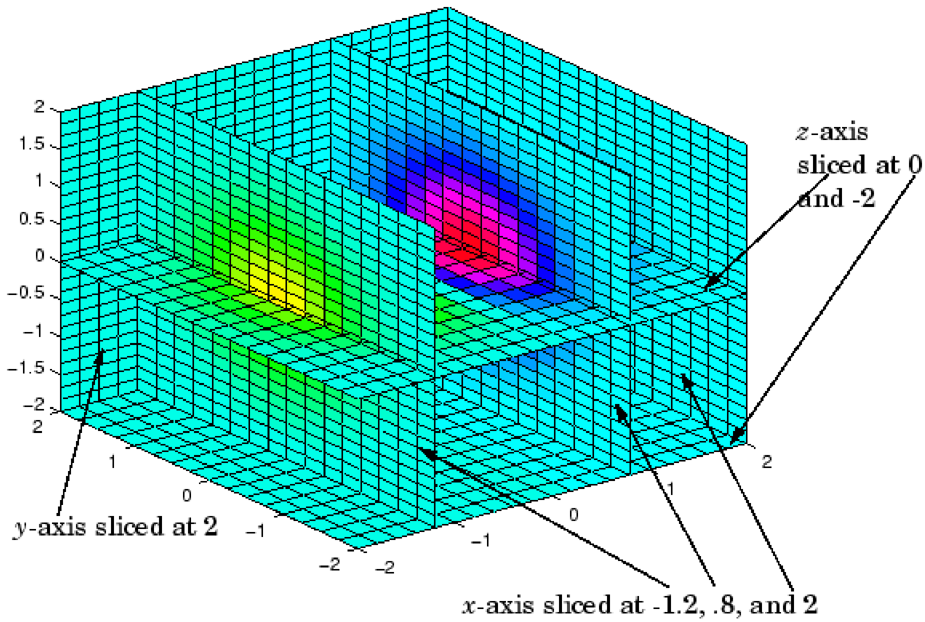
Examples

Visualize the function

$$v = xe^{(-x^2 - y^2 - z^2)}$$

over the range $-2 \le x \le 2$, $-2 \le y \le 2$, $-2 \le z \le 2$:

```
[x,y,z] = meshgrid(-2:.2:2,-2:.25:2,-2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2);
xslice = [-1.2,.8,2]; yslice = 2; zslice = [-2,0];
slice(x,y,z,v,xslice,yslice,zslice)
colormap hsv
```



Slicing At Arbitrary Angles

You can also create slices that are oriented in arbitrary planes. To do this,

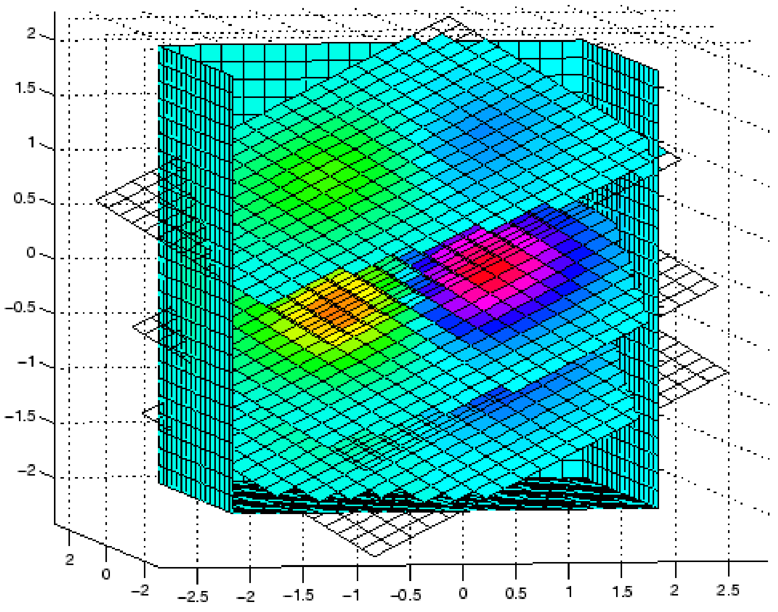
- Create a slice surface in the domain of the volume (`surf`, `linspace`).
- Orient this surface with respect to the axes (`rotate`).
- Get the `XData`, `YData`, and `ZData` of the surface (`get`).
- Use this data to draw the slice plane within the volume.

For example, these statements slice the volume in the first example with a rotated plane. Placing these commands within a `for` loop "passes" the plane through the volume along the *z*-axis.

```
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.

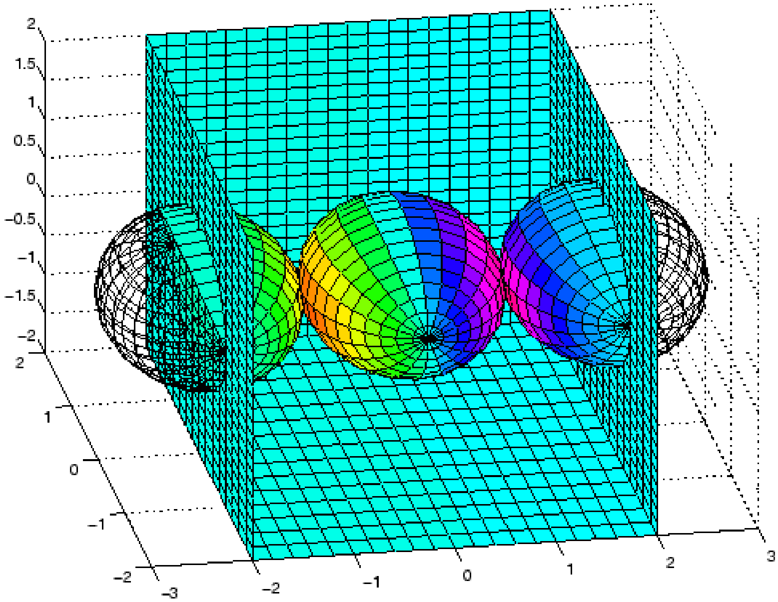


Slicing with a Nonplanar Surface

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 1-106 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)

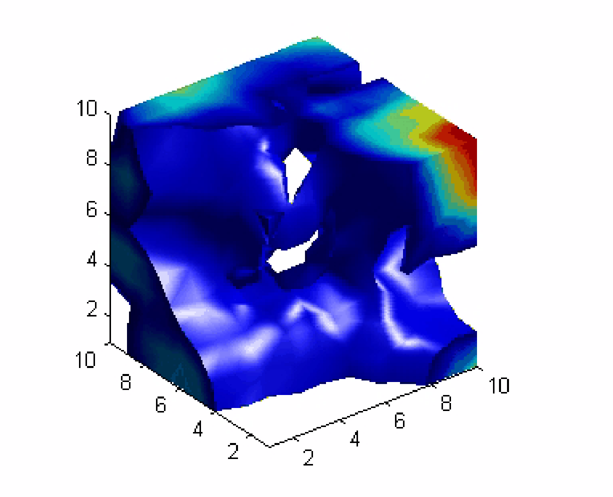
`W = smooth3(V,'filter',size)` sets the size of the convolution kernel (default is `[3 3 3]`). If `size` is scalar, then `size` is interpreted as [`size`, `size`, `size`].

`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 1-106 for related functions

See Displaying an Isosurface for another example.

snapnow

Purpose

Force snapshot of image for inclusion in published document

GUI Alternative

As an alternative to snapnow, open an M-File and select **Cell > Insert Text Markup > Force Snapshot** to insert the snapnow command into the M-File.

Syntax

```
snapnow
```

Description

The snapnow command forces a snapshot of the image or plot that the code has most recently generated for presentation in a published document. The output appears in the published document at the end of the cell that contains the snapnow command. When used outside the context of publishing an M-File, snapnow has the same behavior as drawnow. That is, if you run a file that contains the snapnow command, the MATLAB software interprets it as though it were a drawnow command.

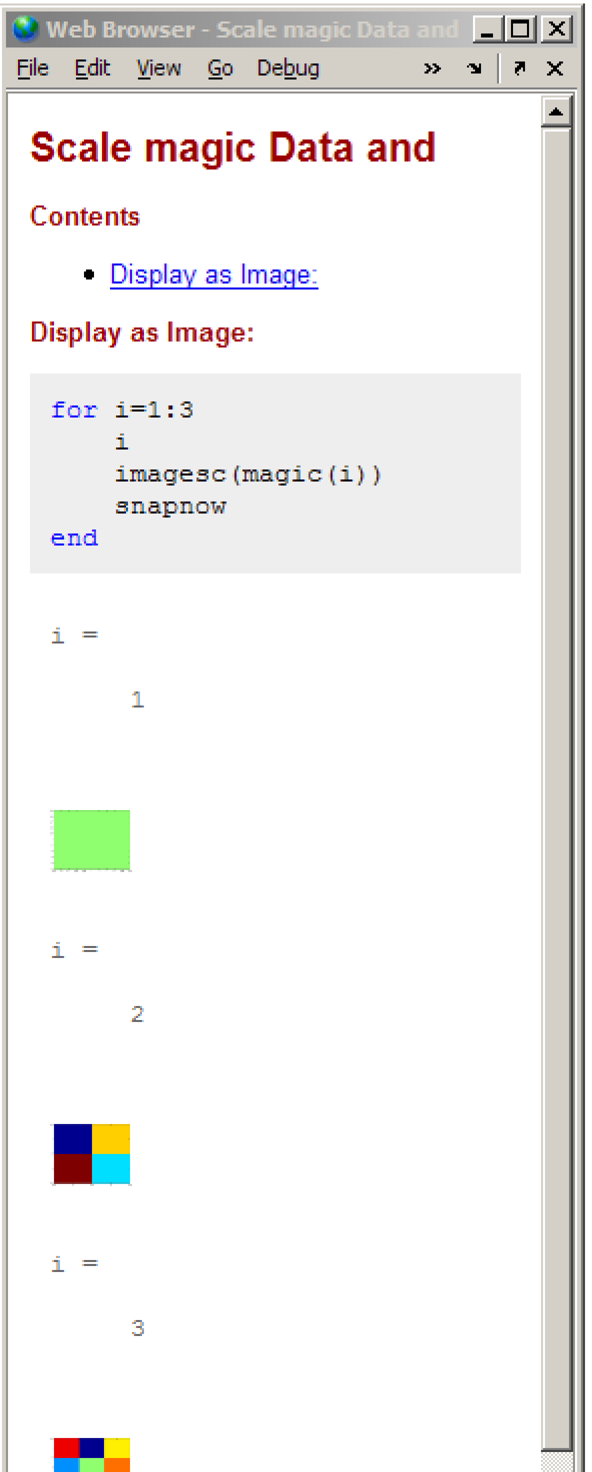
Example

This example demonstrates the difference between publishing code that contains the snapnow command and running that code. The first image shows the results of publishing the code and the second image shows the results of running the code.

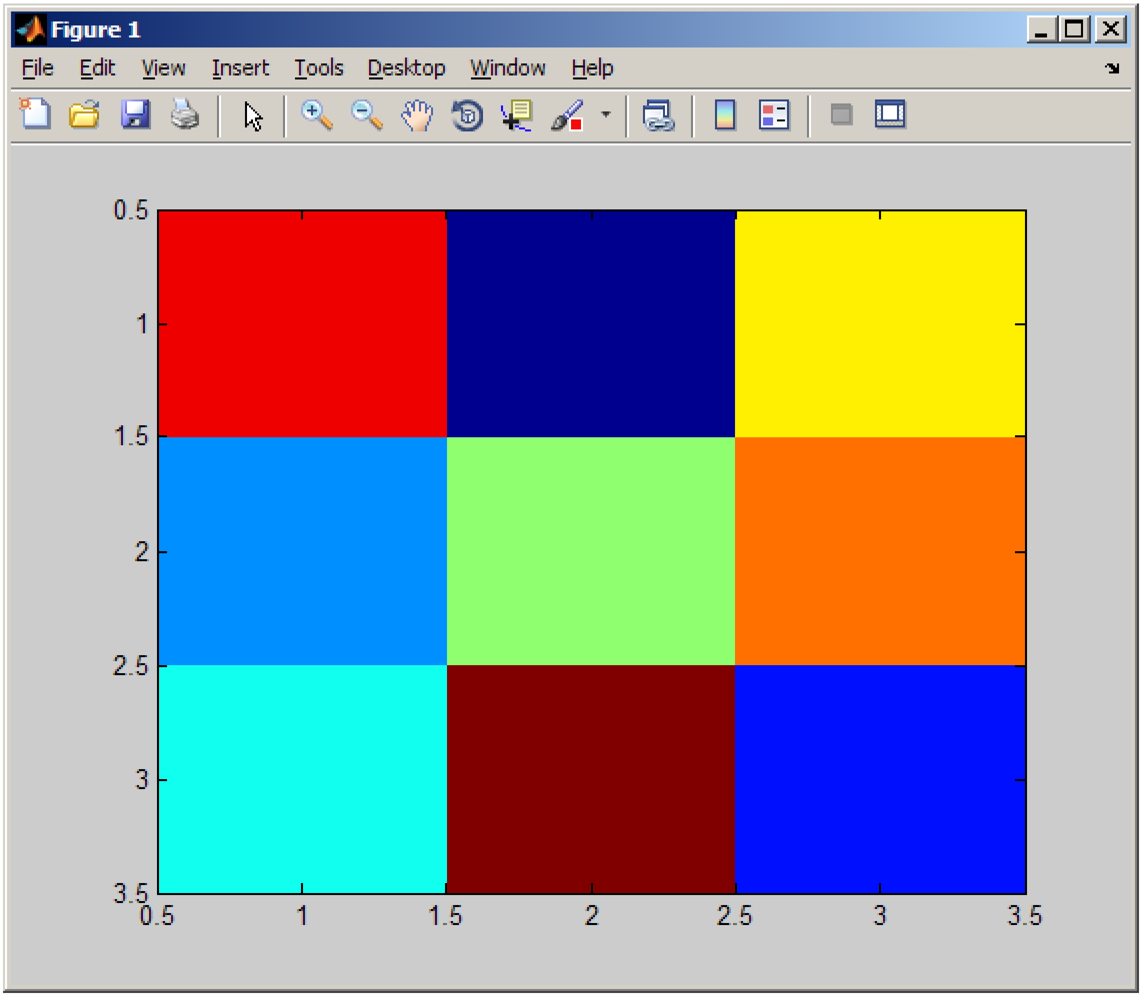
Suppose you have an M-file that contains the following code:

```
%% Scale magic Data and
%% Display as Image:

for i=1:3
    i
    imagesc(magic(i))
    snapnow
end
```

When you publish the code to HTML, the published document contains a title, a table of contents, the commented text, the code, and each of the three images produced by the for loop, along with a display of the value of `i` corresponding to each image. (In the published document shown, the size of the images have been reduced.)


Web Browser - Scale magic Data and
File Edit View Go Debug
Scale magic Data and
Contents
Display as Image:
Display as Image:
for i=1:3
i
imagesc(magic(i))
snapnow
end
i =
1
i =
2
i =
3

When you run the code, a single Figure window opens and MATLAB updates the image within this window as it evaluates each iteration of the `for` loop. (Concurrently, the Command Window displays the value of `i`.) Each successive image replaces the one that preceded it, so that the Figure window appears as follows when the code evaluation completes.



See Also drawnow

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 ascending ASCII dictionary order. You cannot use the `dim` or `mode` options with a cell array.

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., `abs(A)`, and where magnitudes are equal, further sorted by phase angle, i.e., `angle(A)`, on the interval $[-\pi, \pi]$. If `A` includes any `NaN` elements, `sort` places these at the high end.

`B = sort(A,dim)` sorts the elements along the dimension of `A` specified by a scalar `dim`.

`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:

- abs(r) < abs(s)
- abs(r) = abs(s) and angle(r)<angle(s)

  where $-\pi < angle(r) \leq \pi$

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
```

Note `sort` uses a different rule for ordering complex numbers than do the relational operators. See the Relational Operators reference page for more information. For more information about how MATLAB software treats complex numbers, see in the .

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`, `unique`

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, the MATLAB software 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 an arbitrary matrix, `A`:

```
A=floor(gallery('uniformdata',[6 7],0)*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):

```
B = sortrows(A)
B =
    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 the specified column, (e.g., `A(2:4,:)`, if sorting matrix `A` by column 2) remain in their original order:

```
C = sortrows(A,2)
C =
    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
    95    45    92    41    13     1    84
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

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:

```
D = sortrows(A,[1 7])
D =
    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
```

```
    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:

```
E = sortrows(A, -4)
E =
    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`

sound

Purpose

Convert vector into sound

Syntax

```
sound(y,Fs)
sound(y)
sound(y,Fs,bits)
```

Description

`sound(y,Fs)` sends the signal in vector `y` (with sample frequency `Fs`, in hertz) to the speaker on Microsoft Windows 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 in the range of approximately 5 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 software supports all sound devices compatible with Windows operating systems. Additional sound acquisition and generation capability is available in Data Acquisition Toolbox software. The toolbox functionality includes the ability to buffer the acquisition so that you can analyze the data as it is being acquired. See the examples on MATLAB sound acquisition and sound generation.

See Also

`auread`, `auwrite`, `soundsc`, `audioplayer`, `wavread`, `wavwrite`

Purpose

Scale data and play as sound

Syntax

```
soundsc(y,Fs)
soundsc(y)
soundsc(y,Fs,bits)
soundsc(y,...,slim)
```

Description

`soundsc(y,Fs)` sends the signal in vector `y` (with sample frequency `Fs`) to the speaker on PC and most UNIX platforms. The signal `y` is scaled to the range $-1.0 \le y \le 1.0$ 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 software supports all Windows-compatible sound devices.

See Also

`auread`, `auwrite`, `sound`, `wavread`, `wavwrite`

spalloc

Purpose Allocate space for sparse matrix

Syntax

```
S = spalloc(m,n,nzmax)
```

Description `S = spalloc(m,n,nzmax)` creates an all zero sparse matrix `S` of size `m`-by-`n` with room to hold `nzmax` nonzeros. The matrix can then be generated column by column without requiring repeated storage allocation as the number of nonzeros grows.

`spalloc(m,n,nzmax)` is shorthand for

```
sparse([],[],[],m,n,nzmax)
```

Examples To generate efficiently a sparse matrix that has an average of at most three nonzero elements per column

```
S = spalloc(n,n,3*n);
for j = 1:n
    S(:,j) = [zeros(n-3,1)' round(rand(3,1))']';end
```

Purpose

Create sparse matrix

Syntax

```
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

Purpose

Form least squares augmented system

Syntax

```
S = spaugment(A,c)
S = spaugment(A)
```

Description

`S = spaugment(A,c)` creates the sparse, square, symmetric indefinite matrix `S = [c*I A; A' 0]`. The matrix `S` is related to the least squares problem

```
min norm(b - A*x)
```

by

```
r = b - A*x
S * [r/c; x] = [b; 0]
```

The optimum value of the residual scaling factor `c`, involves `min(svd(A))` and `norm(r)`, which are usually too expensive to compute.

`S = spaugment(A)` without a specified value of `c`, uses `max(max(abs(A)))/1000`.

Note In previous versions of MATLAB product, the augmented matrix was used by sparse linear equation solvers, \ and /, for nonsquare problems. Now, MATLAB software performs a least squares solve using the `qr` factorization of `A` instead.

See Also

`spparms`

spconvert

Purpose

Import matrix from sparse matrix external format

Syntax

```
S = spconvert(D)
```

Description

`spconvert` is used to create sparse matrices from a simple sparse format easily produced by non-MATLAB sparse programs. `spconvert` is the second step in the process:

1. Load an ASCII data file containing `[i,j,v]` or `[i,j,re,im]` as rows into a MATLAB variable.

2. Convert that variable into a MATLAB sparse matrix.

`S = spconvert(D)` converts a matrix `D` with rows containing `[i,j,s]` or `[i,j,r,s]` to the corresponding sparse matrix. `D` must have an `nnz` or `nnz+1` row and three or four columns. Three elements per row generate a real matrix and four elements per row generate a complex matrix. A row of the form `[m n 0]` or `[m n 0 0]` anywhere in `D` can be used to specify `size(S)`. If `D` is already sparse, no conversion is done, so `spconvert` can be used after `D` is loaded from either a MAT-file or an ASCII file.

Examples

Suppose the ASCII file `uphill.dat` contains

```
1    1    1.000000000000000
1    2    0.500000000000000
2    2    0.333333333333333
1    3    0.333333333333333
2    3    0.250000000000000
3    3    0.200000000000000
1    4    0.250000000000000
2    4    0.200000000000000
3    4    0.166666666666667
4    4    0.142857142857143
4    4    0.000000000000000
```

Then the statements

```
load uphill.dat
H = spconvert(uphill)

H =
   (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.

spdiags

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 >= 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 2-3356 and “Example 5B” on page 2-3358, 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`.

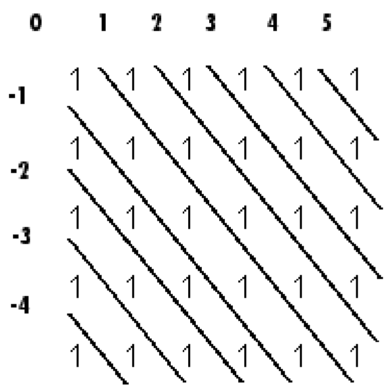
Roughly, `A`, `B`, and `d` are related by

```
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-1diagonals. 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

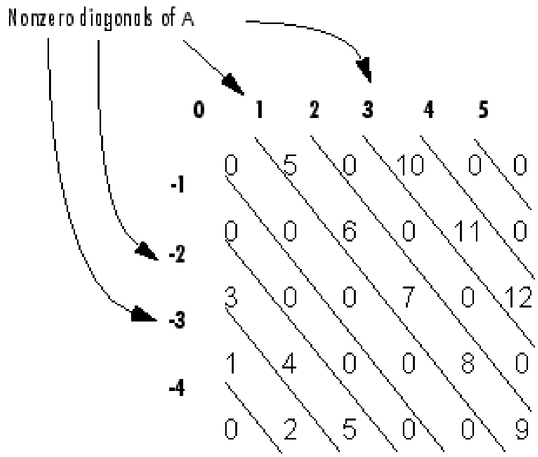
```
[B, d]  =spdiags(A)
```

returns

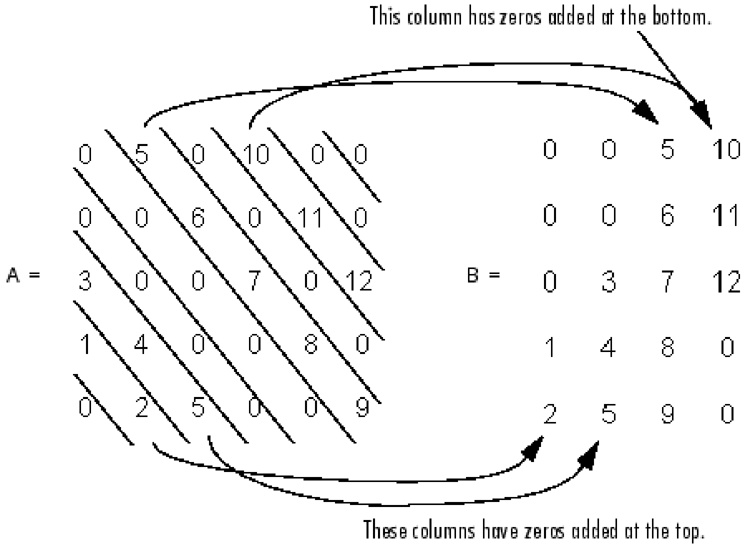
```
B =

     0     0     5    10
     0     0     6    11
     0     3     7    12
     1     4     8     0
     2     5     9     0


d =


    -3
    -2
     1
```

```
     3
```

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 2-3351.



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
```

```
 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
```

```
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`, `speye`

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)`.

Purpose Sparse identity matrix

Syntax

```
S = speye(m,n)
S = speye(n)
```

Description `S = speye(m,n)` forms an `m`-by-`n` sparse matrix with `1`s on the main diagonal.

`S = speye(n)` abbreviates `speye(n,n)`.

Examples `I = speye(1000)` forms the sparse representation of the 1000-by-1000 identity matrix, which requires only about 16 kilobytes of storage. This is the same final result as `I = sparse(eye(1000,1000))`, but the latter requires eight megabytes for temporary storage for the full representation.

See Also `spalloc`, `spones`, `spdiags`, `sprand`, `sprandn`

Purpose

Apply function to nonzero sparse matrix elements

Syntax

```
f = 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 in the MATLAB Programming documentation for more information.

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 (@)

sph2cart

Purpose

Transform spherical coordinates to Cartesian

Syntax

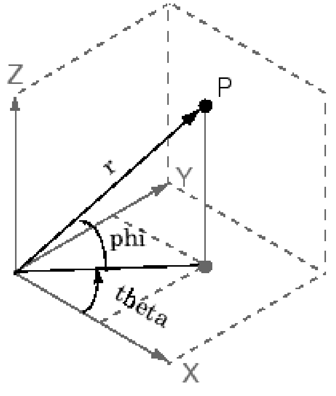
```
[x,y,z] = sph2cart(THETA,PHI,R)
```

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 (or any of them can be scalar). `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




```
x = r .* cos(phi) .* cos(theta)
y = r .* cos(phi) .* sin(theta)
      z = r .* sin(phi)
```

See Also

`cart2pol`, `cart2sph`, `pol2cart`

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.

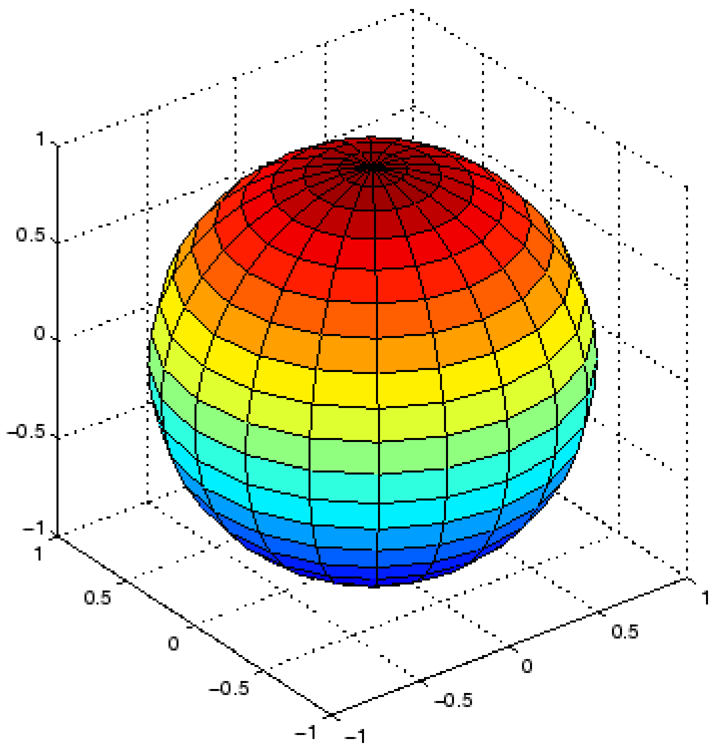
`sphere(n)` draws a `surf` plot of an `n`-by-`n` sphere in the current figure.

`[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 1-95 for related functions

Purpose

Spin colormap

Syntax

```
spinmap
spinmap(t)
spinmap(t,inc)
spinmap('inf')
```

Description

The `spinmap` function shifts the colormap RGB values by some incremental value. For example, if the increment equals 1, color 1 becomes color 2, color 2 becomes color 3, etc.

`spinmap` cyclically rotates the colormap for approximately five seconds using an incremental value of 2.

`spinmap(t)` rotates the colormap for approximately `10*t` seconds. The amount of time specified by `t` depends on your hardware configuration (e.g., if you are running MATLAB software 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 1-103 for related functions

spline

Purpose Cubic spline data interpolation

Syntax

```
yy = spline(x,Y,xx)
pp = spline(x,Y)
```

Description

`yy = spline(x,Y,xx)` uses a cubic spline interpolation to find `yy`, the values of the underlying function `Y` at the values of the interpolant `xx`. For the interpolation, the independent variable is assumed to be the final dimension of `Y` with the breakpoints defined by `x`.

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]`.

`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 x and Y are vectors of the same size, the not-a-knot end conditions are used.
- If x or Y is a scalar, it is expanded to have the same length as the other and the not-a-knot end conditions are used. (See Exceptions (1) below).
- 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. (See Exceptions (2) below.)

Exceptions

1 If `Y` is a vector that contains two more values than `x` has entries, the first and last value in `Y` are used as the endslopes for the cubic spline. If `Y` is a vector, this means

- `f(x) = Y(2:end-1)`
- `df(min(x)) = Y(1)`
- `df(max(x)) = Y(end)`

2 If `Y` is a matrix or an `N`-dimensional array with `size(Y,N)` equal to `length(x)+2`, the following hold:

- `f(x(j))` matches the value `Y(:,...,:,j+1)` for `j=1:length(x)`
- `Df(min(x))` matches `Y(:,:,...:,1)`
- `Df(max(x))` matches `Y(:,:,...:,end)`

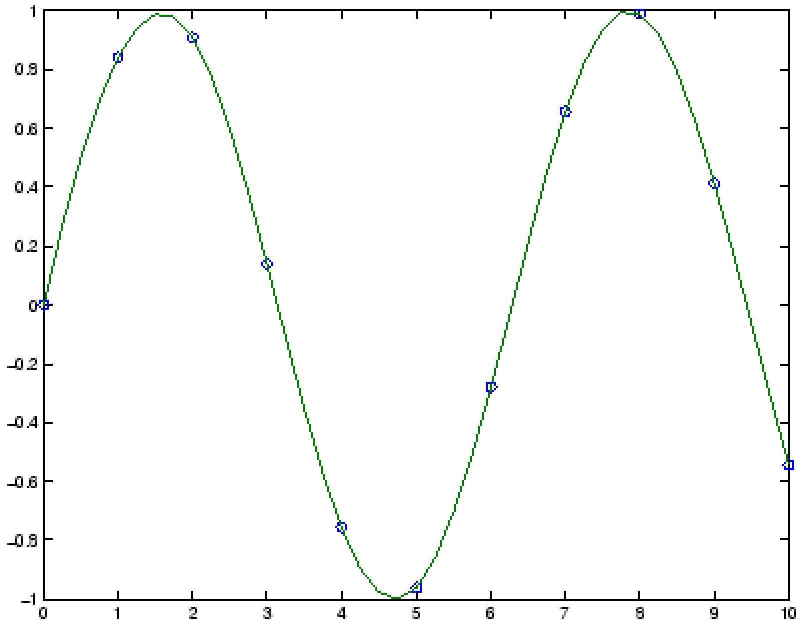
Note You can also perform spline interpolation using the `interp1` function with the command `interp1(x,y,xx,'spline')`. Note that while `spline` performs interpolation on rows of an input matrix, `interp1` performs interpolation on columns of an input matrix.

Examples

Example 1

This generates a sine curve, then samples the spline over a finer mesh.

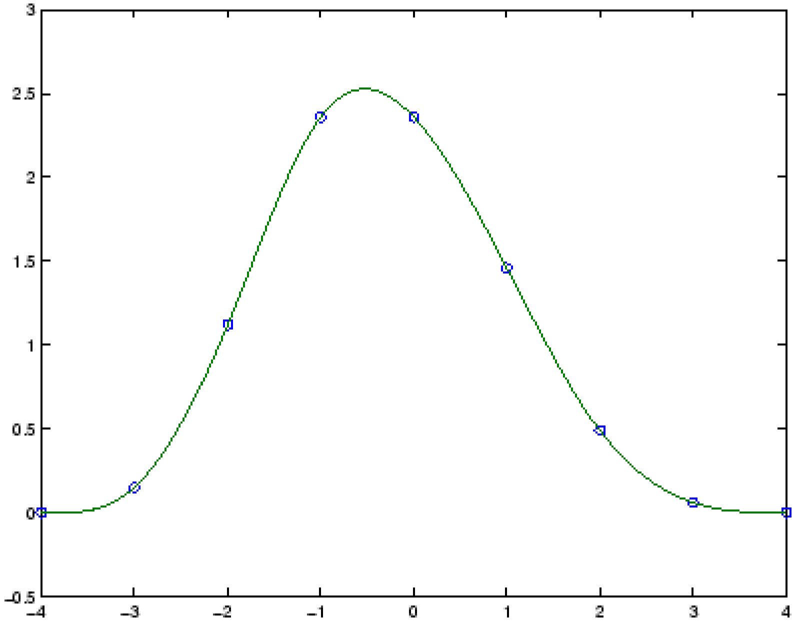
```
x = 0:10;
y = sin(x);
xx = 0:.25:10;
yy = spline(x,y,xx);
plot(x,y,'o',xx,yy)
```



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),'-');
```



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

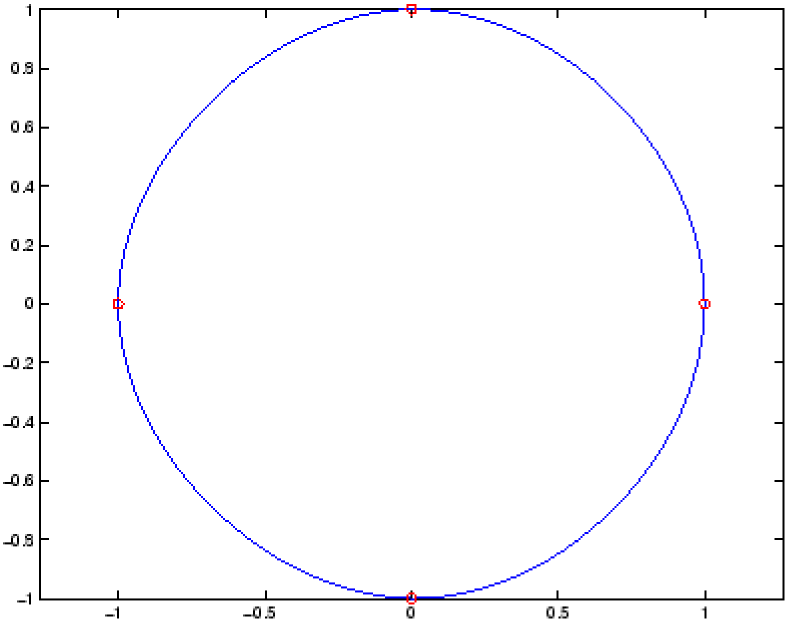
```
ans =
    270.6060
```

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
```

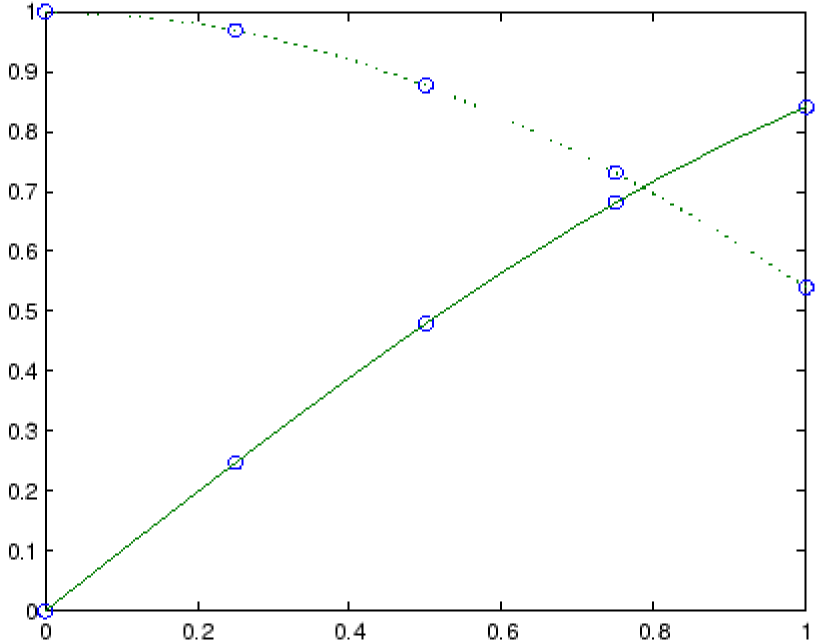
generate the plot of a circle, with the five data points `y(:,2),...,y(:,6)` marked with o's. Note that this `y` contains two more values (i.e., two more columns) than does `x`, hence `y(:,1)` and `y(:,end)` are used as endslopes.



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.

Purpose Replace nonzero sparse matrix elements with ones

Syntax `R = spones(S)`

Description `R = spones(S)` generates a matrix `R` with the same sparsity structure as `S`, but with `1`'s in the nonzero positions.

Examples `c = sum(spones(S))` is the number of nonzeros in each column.

`r = sum(spones(S'))'` is the number of nonzeros in each row.

`sum(c)` and `sum(r)` are equal, and are equal to `nnz(S)`.

See Also `nnz`, `spalloc`, `spfun`

spparms

Purpose

Set parameters for sparse matrix routines

Syntax

```
spparms('key',value)
spparms
values = spparms
[keys,values] = spparms
spparms(values)
value = spparms('key')
spparms('default')
spparms('tight')
```

Description

`spparms('key',value)` sets one or more of the *tunable* parameters used in the sparse routines. In ordinary use, you should never need to deal with this function.

The meanings of the `key` parameters are

`'spumoni'`	Sparse Monitor flag:
`0`	Produces no diagnostic output, the default
`1`	Produces information about choice of algorithm based on matrix structure, and about storage allocation
`2`	Also produces very detailed information about the sparse matrix algorithms
`'thr_rel'`, `'thr_abs'`	Minimum degree threshold is `thr_rel*mindegree + thr_abs`.
`'exact_d'`	Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.
`'supernd'`	If positive, minimum degree amalgamates the supernodes every `supernd` stages.
`'rreduce'`	If positive, minimum degree does row reduction every `rreduce` stages.

`'wh_frac'`	Rows with `density > wh_frac` are ignored in `colmmd`.
`'autommd'`	Nonzero to use minimum degree (MMD) orderings with QR-based \ and `/`.
`'autoamd'`	Nonzero to use `colamd` ordering with the UMFPACK LU-based \ and /, 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 `/`.
`'sym_tol'`	Symmetric pivot tolerance used by UMFPACK. See `lu` for more information about the role of the symmetric pivot tolerance.

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	
values(13)	'sym_tol'	0.001	

Notes

Sparse A\b on Symmetric Positive Definite A

Sparse `A\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\b on General Square A

Sparse A\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\b on Rectangular A

Sparse A\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, colamdsymamd

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.

Purpose

Sparse uniformly distributed random matrix

Syntax

```
R = sprand(S)
R = sprand(m,n,density)
R = sprand(m,n,density,rc)
```

Description

`R = sprand(S)` has the same sparsity structure as `S`, but uniformly distributed random entries.

`R = sprand(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` uniformly distributed nonzero entries (`0 <= density <= 1`).

`R = sprand(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

`sprand` uses the internal state information set with the `rand` function.

See Also

`sprandn`, `sprandsym`

sprandn

Purpose

Sparse normally distributed random matrix

Syntax

```
R = sprandn(S)
R = sprandn(m,n,density)
R = sprandn(m,n,density,rc)
```

Description

`R = sprandn(S)` has the same sparsity structure as `S`, but normally distributed random entries with mean `0` and variance `1`.

`R = sprandn(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` normally distributed nonzero entries (`(0 <= density <= 1`).

`R = sprandn(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

`sprandn` uses the internal state information set with the `randn` function.

See Also

`sprand`, `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`. For all values of `A`,

```
sprank(A) >= rank(full(A))
```

In exact arithmetic, `sprank(A) == rank(full(sprandn(A)))` with a probability of one.

Examples

```
A = [1    0    2    0
     2    0    4    0 ];

A = sparse(A);

sprank(A)

ans =
     2

rank(full(A))

ans =
     1
```

See Also

`dmperm`

Purpose

Format data into string

Syntax

```
str = sprintf(format, A, ...)
[str, errmsg] = sprintf(format, A, ...)
```

Description

`str = sprintf(format, A, ...)` applies the *format* to array *A* and any additional array arguments in column order, and returns the results to string *str*.

`[str, errmsg] = sprintf(format, A, ...)` returns an error message string when the operation is unsuccessful. Otherwise, *errmsg* is empty.

Inputs

format

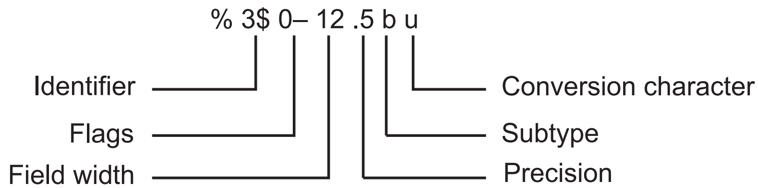
String in single quotation marks that describes the format of the output fields. Can include combinations of the following:

- Percent sign followed by a conversion character, such as `'%s'` for strings.
- Operators that describe field width, precision, and other options.
- Literal text to print.
- Escape characters, including:

`''`	Single quotation mark
`%%`	Percent character
`\\`	Backslash
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return

`\t`	Horizontal tab
`\xN`	Hexadecimal number, *N*
`\N`	Octal number, *N*

Conversion characters and optional operators appear in the following order (includes spaces for clarity):



The following table lists the available conversion characters and subtypes.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10 values
	`%ld` or `%li`	64-bit base 10 values
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a`–`f`
	`%X`	Same as `%x`, uppercase letters `A`–`F`

	`%lu`
	`%lo`
	`%lx` or `%lX`
		64-bit values, base 10, 8, or 16

Value Type	Conversion	Details
Floating-point number	%f	Fixed-point notation
	%e	Exponential notation, such as 3.141593e+00
	%E	Same as %e, but uppercase, such as 3.141593E+00
	%g	The more compact of %e or %f, with no trailing zeros
	%G	The more compact of %E or %f, with no trailing zeros
	%bx or %bX %bo %bu	Double-precision hexadecimal, octal, or decimal value Example: %bx prints pi as 400921fb54442d18
	%tx or %tX %to %tu	Single-precision hexadecimal, octal, or decimal value Example: %tx prints pi as 40490fdb
Characters	%c	Single character
	%s	String of characters

Additional operators include:

- Field width

  Minimum number of characters to print. Can be a number, or an asterisk (*) to refer to an argument in the input list. For example, the input list ('%12d', intmax) is equivalent to ('%*d', 12, intmax).

- Precision

For `%f`, `%e`, or `%E`:	Number of digits to the right of the decimal point. Example: `'%6.4f'` prints `pi` as `'3.1416'`
For `%g` or `%G`	Number of significant digits. Example: `'%6.4g'` prints `pi` as `' 3.142'`

Can be a number, or an asterisk (`*`) to refer to an argument in the input list. For example, the input list `('%6.4f', pi)` is equivalent to `('%*.*f', 6, 4, pi)`.

- Flags

Action	Flag	Example
Left-justify.	`' '`	`%-5.2f`
Print sign character (+ or).	`'+'`	`%+5.2f`
Insert a space before the value.	`' '`	`% 5.2f`
Pad with zeros.	`'0'`	`%05.2f`

- Identifier

  Order for processing inputs. Use the syntax *`n`*`$`, where *`n`* represents the position of the value in the input list.

  For example, `'%3$s %2$s %1$s %2$s'` prints inputs `'A'`, `'B'`, `'C'` as follows: `C B A B`.

The following limitations apply to conversions:

- Numeric conversions print only the real component of complex numbers.
- If you apply an integer or string conversion to a numeric value that contains a fraction, MATLAB overrides the specified conversion, and uses `%e`.

- If you apply a string conversion (`%s`) to integer values, MATLAB:
  - Issues a warning.
  - Converts values that correspond to valid character codes to characters. For example, `'%s'` converts `[65 66 67]` to `ABC`.
- Different platforms display exponential notation (such as `%e`) with a different number of digits in the exponent.

Platform	Example
Windows	`1.23e+004`
UNIX	`1.23e+04`

- Different platforms display negative zero (`-0`) differently.

	Conversion Character		
Platform	**%e or %E**	**%f**	**%g or %G**
Windows	0.000000e+000	0.000000	0
Others	-0.000000e+00	-0.000000	-0

A

Numeric or character array.

Examples

Format floating-point numbers:

```
sprintf('%0.5f',1/eps)   % 4503599627370496.00000
sprintf('%0.5g',1/eps)   % 4.5036e+15
```

Explicitly convert double-precision values to integers:

```
sprintf('%d',round(pi))   % 3
```

Combine literal text with array values:

```
sprintf('The array is %dx%d.',2,3) % The array is 2x3
```

On a Windows system, convert PC-style exponential notation (three digits in the exponent) to UNIX-style notation (two digits):

```
a = sprintf('%e', 12345.678);
if ispc
   a = strrep(a, 'e+0', 'e+');
end
```

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.

See Also

`fprintf` | `int2str` | `num2str` | `sscanf`

How To

-

Purpose

Visualize sparsity pattern

Syntax

```
spy(S)
spy(S,markersize)
spy(S,'LineSpec')
spy(S,'LineSpec',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,'LineSpec')`, where `LineSpec` is a string, uses the specified plot marker type and color.

`spy(S,'LineSpec',markersize)` uses the specified type, color, and size for the plot markers.

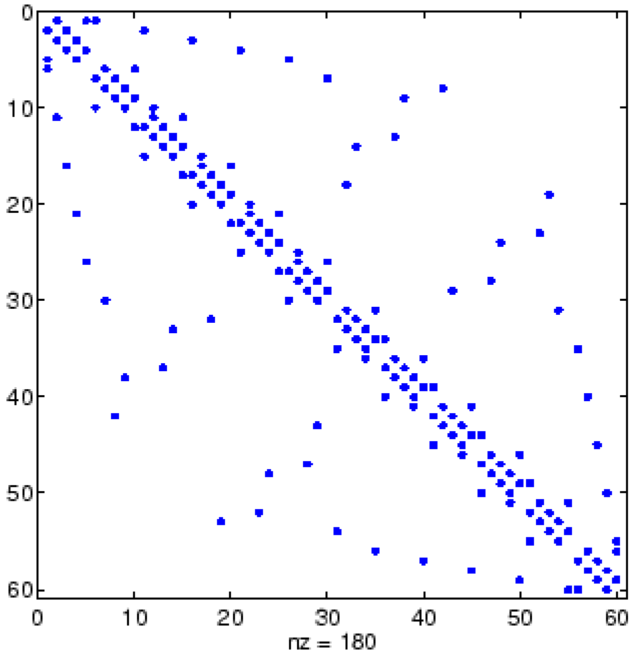
`S` is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.

Note `spy` replaces `format +`, which takes much more space to display essentially the same information.

Examples

This example plots the 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster Fuller geodesic dome. This matrix also represents the soccer ball and the carbon-60 molecule.

```
B = bucky;
spy(B)
```



See Also `find`, `gplot`, `LineSpec`, `symamd`, `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`

Purpose

Matrix square root

Syntax

```
X = sqrtm(A)
[X, resnorm] = sqrtm(A)
[X, alpha, condest] = sqrtm(A)
```

Description

`X = sqrtm(A)` is the principal square root of the matrix `A`, i.e. `X*X = 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, resnorm] = sqrtm(A)` does not print any warning, and returns the residual, `norm(A-X^2,'fro')/norm(A,'fro')`.

`[X, alpha, condest] = sqrtm(A)` returns a stability factor `alpha` and an estimate `condest` of the matrix square root condition number of `X`. The residual `norm(A-X^2,'fro')/norm(A,'fro')` is bounded approximately by `n*alpha*eps` and the Frobenius norm relative error in `X` is bounded approximately by `n*alpha*condest*eps`, where `n = max(size(A))`.

Remarks

If `X` is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.

Some matrices, like `X = [0 1; 0 0]`, 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 =
     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
```

This matrix is symmetric and positive definite. Its unique positive definite square root, Y = sqrtm(X), is a representation of the second difference operator.

```
 Y =
     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
```

Example 2

The matrix

```
X =
     7    10
    15    22
```

has four square roots. Two of them are

```
Y1 =
    1.5667    1.7408
    2.6112    4.1779
```

and

```
Y2 =
     1     2
     3     4
```

The other two are -Y1 and -Y2. All four can be obtained from the eigenvalues and vectors of X.

```
[V,D] = eig(X);
D =
    0.1386         0
         0   28.8614
```

The four square roots of the diagonal matrix D result from the four choices of sign in

```
S =
    -0.3723          0
          0    -5.3723
```

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 =

Y(:,:,1) =      Y(:,:,2) =
       0.5194          0.0346
       0.8310          0.0535

Y(:,:,3) =
       0.5297
       0.6711
```

The command `Z = squeeze(Y)` yields a 2-by-3 matrix:

```
Z =
    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 =

mat(:,:,1) =  mat(:,:,2) =

     1        1
```

```
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(str, format)
A = sscanf(str, format, sizeA)
[A, count] = sscanf(...)
[A, count, errmsg] = sscanf(...)
[A, count, errmsg, nextindex] = sscanf(...)
```

Description

`A = sscanf(str, format)` reads data from string `str`, converts it according to the `format`, and returns the results in array `A`. The `sscanf` function reapplies the `format` until either reaching the end of `str` or failing to match the `format`. If `sscanf` cannot match the `format` to the data, it reads only the portion that matches into `A` and stops processing. If `str` is a character array with more than one row, `sscanf` reads the characters in column order.

`A = sscanf(str, format, sizeA)` reads `sizeA` elements into `A`, where `sizeA` can be an integer or can have the form `[m,n]`.

`[A, count] = sscanf(...)` returns the number of elements that `sscanf` successfully reads.

`[A, count, errmsg] = sscanf(...)` returns an error message string when the operation is unsuccessful. Otherwise, `errmsg` is an empty string.

`[A, count, errmsg, nextindex] = sscanf(...)` returns one more than the number of characters scanned in `str`.

Inputs

`format`

String enclosed in single quotation marks that describes each type of element (field). Includes one or more of the following specifiers.

Field Type	Specifier	Details
Integer, signed	%d	Base 10
	%i	Base determined from the values. Defaults to base 10. If initial digits are 0x or 0X, it is base 16. If initial digit is 0, it is base 8.
Integer, unsigned	%u	Base 10
	%o	Base 8 (octal)
	%x	Base 16 (hexadecimal)
Floating-point number	%f	Floating-point fields can contain any of the following (not case sensitive): Inf, -Inf, NaN, or -NaN.
	%e	
	%g	
Character string	%s	Read series of characters, until find white space.
	%c	Read any single character, including white space. (To read multiple characters, specify field length.)
	%[...]	Read only characters in the brackets, until the first nonmatching character or white space.

Optionally:

- To skip fields, insert an asterisk (*) after the percent sign (%). For example, to skip integers, specify %*d.
- To specify the maximum width of a field, insert a number. For example, %10c reads exactly 10 characters at a time, including white space.

- To skip a specific set of characters, insert the literal characters in the *format*. For example, to read only the floating-point number from `'pi=3.14159'`, specify a *format* of `'pi=%f'`.

`sizeA`

Dimensions of the output array *A*. Specify in one of the following forms:

`inf`	Read to the end of the input string. (default)
n	Read at most *n* elements.
[*m*,*n*]	Read at most *m***n* elements in column order. *n* can be `inf`, but *m* cannot.

When the *format* includes `%s`, *A* can contain more than *n* columns. *n* refers to elements, not characters.

str

Character string.

Outputs

A

An array. If the *format* includes:

- Only numeric specifiers, *A* is numeric, of class `double`. If *sizeA* is `inf` or *n*, then *A* is a column vector. If the input contains fewer than *sizeA* elements, MATLAB pads *A* with zeros.
- Only character or string specifiers (`%c` or `%s`), *A* is a character array. If *sizeA* is `inf` or *n*, *A* is a row vector. If the input contains fewer than *sizeA* characters, MATLAB pads *A* with `char(0)`.
- A combination of numeric and character specifiers, *A* is numeric, of class `double`. MATLAB converts each character to its numeric equivalent. This conversion occurs even when the *format* explicitly skips all numeric values (for example, a *format* of `'%*d %s'`).

If MATLAB cannot match the input to the *format*, and the *format* contains both numeric and character specifiers, *A* can be numeric or character. The class of *A* depends on the values MATLAB reads before processing stops.

count

Number of elements `sscanf` reads into *A*.

errmsg

An error message string when `sscanf` cannot open the specified file. Otherwise, an empty string.

nextindex

`sscanf` counts the number of characters `sscanf` reads from *str*, and then adds one.

Examples

Read multiple floating-point values from a string:

```
s = '2.7183  3.1416';
A = sscanf(s,'%f')        % returns A = [2.7183; 3.1416]
```

Read an octal integer from a string, identified by the `'0'` prefix, using `%i` to preserve the sign:

```
sscanf('-010','%i')        % returns ans = -8
```

Read numeric values from a two-dimensional character array. By default, `sscanf` reads characters in column order. To preserve the original order of the values, read one row at a time.

```
mixed = ['abc 45 6 ghi'; 'def 7 89 jkl'];

[nrows, ncols] = size(mixed);
for k = 1:nrows
```

```
    nums(k,:) = sscanf(mixed(k,:), '%*s %d %d %*s', [1, inf]);
end;

% type the variable name to see the result
nums
```

MATLAB returns:

```
nums =
    45     6
     7    89
```

See Also

fscanf | sprintf | textscan

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(...,LineSpec)
stairs(...,'PropertyName',propertyvalue)
stairs(axes_handle,...)
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(...,LineSpec)` specifies a line style, marker symbol, and color for the graph. (See `LineSpec` for more information.)

`stairs(...,'PropertyName',propertyvalue)` creates the stairstep graph, applying the specified property settings. See `Stairseries` properties for a description of properties.

`stairs(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into 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.

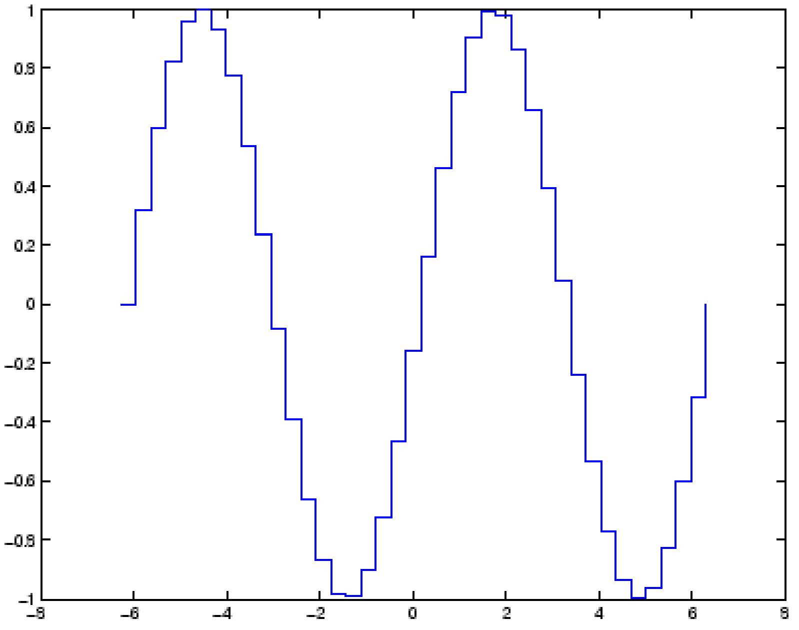
Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

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 1-94 for related functions

`Stairseries Properties` for property descriptions

Stairseries Properties

Purpose Define 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.

`Annotation`

`hg.Annotation` object Read Only

Control the display of stairseries objects in legends. The `Annotation` property enables you to specify whether this stairseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the stairseries object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the stairseries object in a legend as one entry, but not its children objects

IconDisplayStyle Value	Purpose
off	Do not include the stairseries or its children in a legend (default)
children	Include only the children of the stairseries as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the IconDisplayStyle of a graphics object with handle hobj to children, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See for more information and examples.

BeingDeleted

on | {off} Read Only

This object is being deleted. The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the BeingDeleted property to on when the object's delete function callback is called (see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

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

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

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

This property can be

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

- A function handle

  Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

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

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 this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

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

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

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

`CreateFcn`

string or function handle

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

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

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

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 for information on how to use function handles to define the callback function.

`DeleteFcn`

string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

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

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

See the `BeingDeleted` property for related information.

`DisplayName`

string (default is empty string)

String used by legend for this stairseries object. The `legend` function uses the string defined by the `DisplayName` property to label this stairseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this stairseries object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See for more examples.

`EraseMode`

`{normal} | none | xor | background`

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

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

Printing with Nonnormal Erase Modes

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

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

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

HandleVisibility
{on} | callback | off

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. HandleVisibility is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- on — Handles are always visible when HandleVisibility is on.
- callback — Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- off — Setting HandleVisibility to off makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

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

`HitTest`
`{on} | off`

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

`HitTestArea`
`on | {off}`

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click th eobject's lines or markers (excluding the baseline, if any) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineStyle`

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

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

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

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

`LineWidth`

scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`

character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the

`Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Five-pointed star (pentagram)
`h`	Six-pointed star (hexagram)
`none`	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

`MarkerSize`

size in points

Marker size. A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`

handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See for more information on parenting graphics objects.

`Selected`

`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`

`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`

string

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

For example, you might create an areaseries object and set the `Tag` property.

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

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

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

`Type`

string (read only)

Type of graphics object. This property contains a string that identifies the class of the graphics object. For 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 this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

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

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
array

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, `length(XData) == size(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 `x` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`
string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

`YData`
scalar, vector, or matrix

Stairs plot data. `YData` contains the data plotted in the stairstep graph. Each value in `YData` is represented by a marker in the stairstep graph. If `YData` is a matrix, the `stairs` function creates a line for each column in the matrix.

The input argument `y` in the `stairs` function calling syntax assigns values to `YData`.

`YDataSource`
string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that 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

`datenum`, `datestr`, `now`, `timer`, `start`, `stop`

startup

Purpose Startup file for user-defined options

Syntax

```
startup
```

Description

`startup` executes commands of your choosing when the MATLAB program starts.

Create a `startup.m` file in your MATLAB startup folder and put in the file any commands you want executed at MATLAB startup. For example, your `startup.m` file might include physical constants, defaults for Handle Graphics properties, engineering conversion factors, or anything else you want predefined in your workspace.

Algorithm

The MATLAB program executes the `matlabrc.m` file when it starts. `matlabrc.m` invokes `startup.m`, if it exists on the MATLAB search path.

You can extend this process to create additional startup files, if needed.

The MathWorks does not recommend modifying the `matlabrc.m` file, except perhaps by system administrators in network configurations.

See Also

`finish`, `matlabrc`, `matlabroot`, `path`, `quit`, `userpath`

See and Preferences in the MATLAB Desktop Tools and Development Environment documentation.

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`.

`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`

std (timeseries)

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')
   ```

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(...,LineSpec)
stem(axes_handle,...)
h = stem(...)
hlines = stem('v6',...)
```

Description

A two-dimensional stem plot displays data as lines extending from a baseline along the *x*-axis. A circle (the default) or other marker whose *y*-position represents the data value terminates each stem.

`stem(Y)` plots the data sequence `Y` as stems that extend from equally spaced and automatically generated values along the *x*-axis. When `Y` is a matrix, `stem` plots all elements in a row against the same *x* value.

`stem(X,Y)` plots `X` versus the columns of `Y`. `X` and `Y` must be vectors or matrices of the same size. Additionally, `X` can be a row or a column vector and `Y` a matrix with `length(X)` rows.

`stem(...,'fill')` specifies whether to color the circle at the end of the stem.

`stem(...,LineSpec)` specifies the line style, marker symbol, and color for the stem and top marker (the baseline is not affected). See `LineSpec` for more information.

`stem(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into 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

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

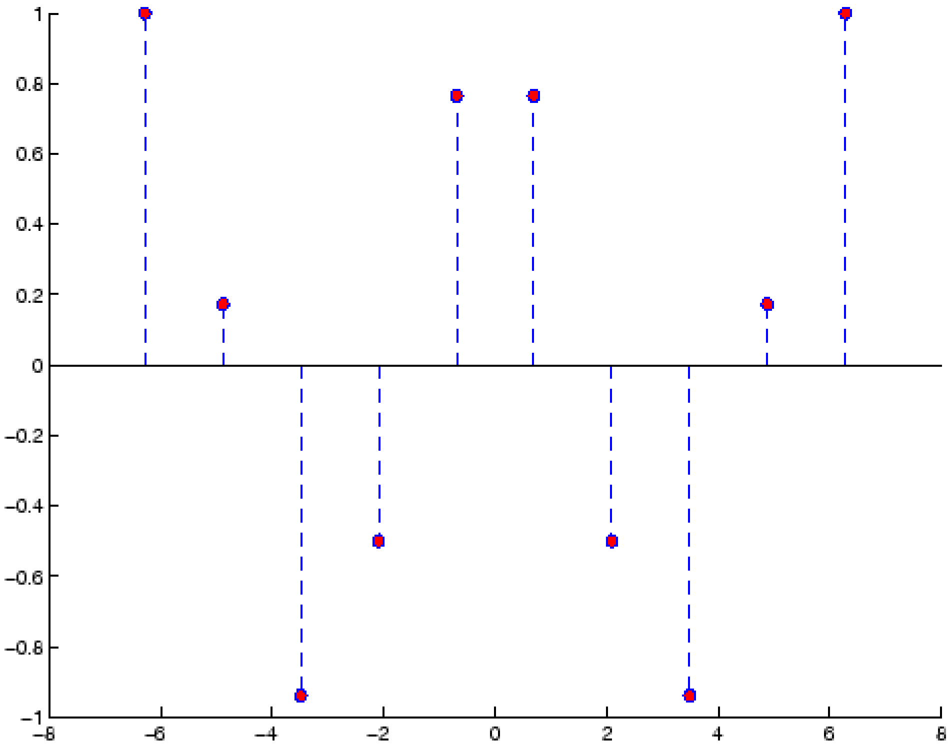
See Plot Objects and Backward Compatibility for more information.

Examples

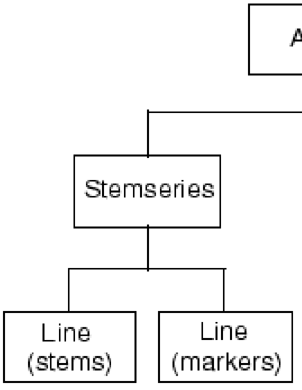
Single Series of Data

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 stemseries object contains two line objects used to draw the stem lines and the end markers. The baseline is a separate line object.



If you do not want the baseline to show, you can remove it with the following command:

```
delete(get(stem_handle,'Baseline'))
```

where `stem_handle` is the handle for the stemseries object. You can use similar code to change the color or style of the baseline, specifying any line property and value, for example,

```
set(get(stem_handle,'Baseline'),'LineWidth',3)
```

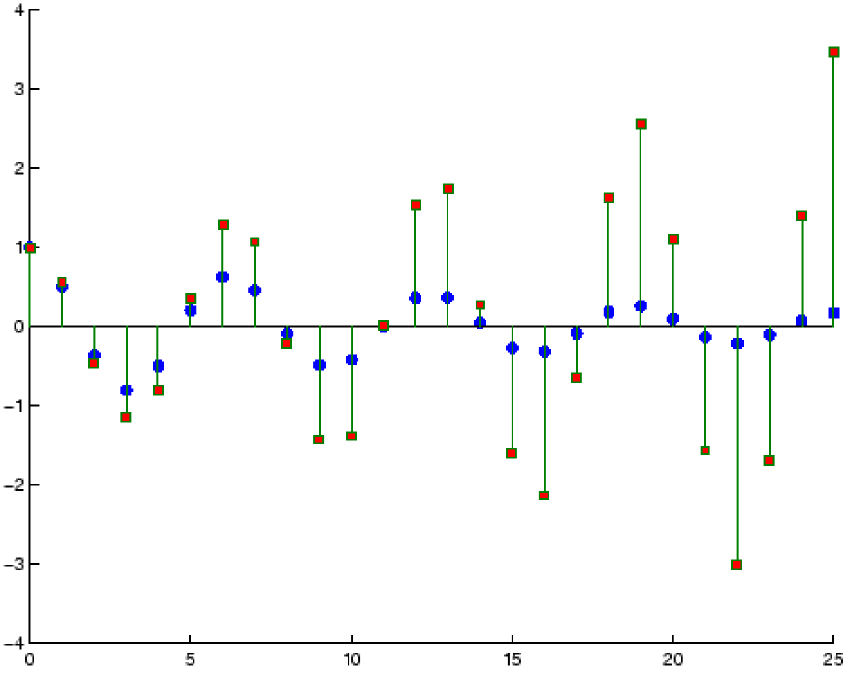
Two Series of Data on One Graph

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`.

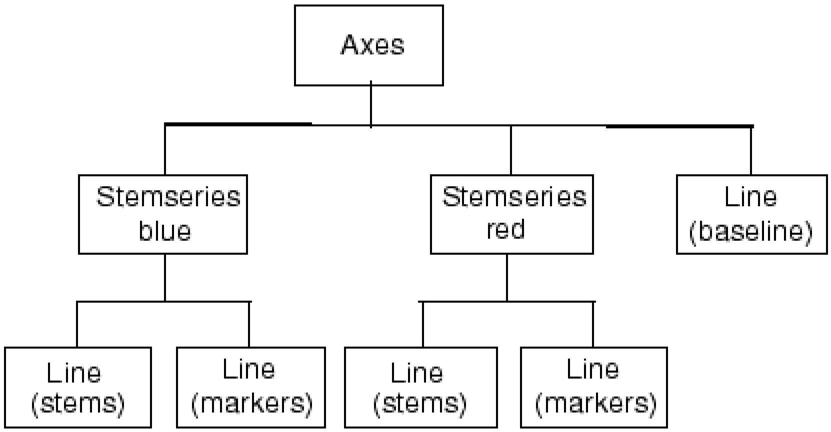
- `h(1)` is the handle to the stemseries object plotting the expression `exp(-.07*x).*cos(x)`.
- `h(2)` is the handle to the stemseries object plotting the expression `exp(.05*x).*cos(x)`.

```
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.



See Also

`bar`, `plot`, `stairs`

`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(...,LineSpec)
h = stem3(...)
hlines = stem3('v6',...)
```

Description

Three-dimensional stem plots display lines extending from the *x-y* plane. A circle (the default) or other marker symbol whose *z*-position represents the data value terminates each stem.

`stem3(Z)` plots the data sequence `Z` as stems that extend from the *x-y* plane. *x* and *y* are generated automatically. When `Z` is a row vector, `stem3` plots all elements at equally spaced *x* values against the same *y* value. When `Z` is a column vector, `stem3` plots all elements at equally spaced *y* values against the same *x* value.

`stem3(X,Y,Z)` plots the data sequence `Z` at values specified by `X` and `Y`. `X`, `Y`, and `Z` must all be vectors or matrices of the same size.

`stem3(...,'fill')` specifies whether to color the interior of the circle at the end of the stem.

`stem3(...,LineSpec)` specifies the line style, marker symbol, and color for the stems. See `LineSpec` for more information.

`h = stem3(...)` returns handles to stemseries graphics objects.

Backward-Compatible Version

`hlines = stem3('v6',...)` returns the handles of line objects instead of stemseries objects for compatibility with MATLAB 6.5 and earlier.

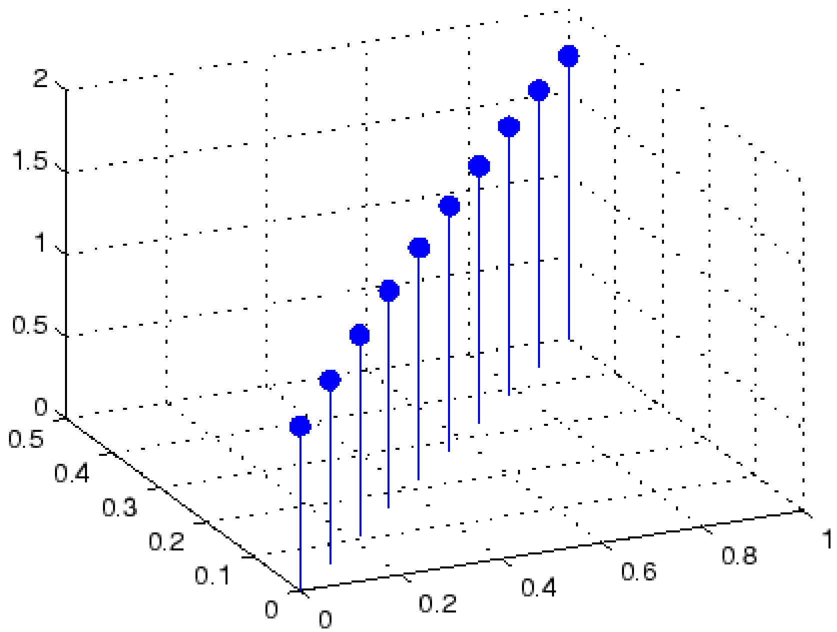
Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

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 1-94 for related functions

`Stemseries Properties` for descriptions of properties

Three-Dimensional Stem Plots for more examples

Stemseries Properties

Purpose Define 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.

`Annotation`

`hg.Annotation` object Read Only

Control the display of stemseries objects in legends. The `Annotation` property enables you to specify whether this stemseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the stemseries object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the stemseries object in a legend as one entry, but not its children objects
`off`	Do not include the stemseries or its children in a legend (default)
`children`	Include only the children of the stemseries as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See for more information and examples.

`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 the MATLAB software draws the baseline.

`BeingDeleted`

`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

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

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

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

This property can be

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

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

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

`Children`
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

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

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`
: `ColorSpec`

Color of 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 an object. You must specify the callback during the creation of the object. For example,

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

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

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

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

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

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

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

See the `BeingDeleted` property for related information.

`DisplayName`
string (default is empty string)

String used by legend for this stemseries object. The `legend` function uses the string defined by the `DisplayName` property to label this stemseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this stemseries object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object

based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.

- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See for more examples.

`EraseMode`

`{normal} | none | xor | background`

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

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

the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.

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

Printing with Nonnormal Erase Modes

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

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

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

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

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

`HitTest`
`{on} | off`

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

`HitTestArea`
`on | {off}`

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click th eobject's lines or markers (excluding the baseline, if any) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

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

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

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

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

`LineWidth`
: scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`
: character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the `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)

Marker Specifier	Description
h	Six-pointed star (hexagram)
none	No marker (default)

MarkerEdgeColor
ColorSpec | none | {auto}

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none specifies no color, which makes nonfilled markers invisible. auto sets MarkerEdgeColor to the same color as the Color property.

MarkerFaceColor
ColorSpec | {none} | auto

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the axes color, or to the figure color if the axes Color property is set to none (which is the factory default for axes objects).

MarkerSize
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the '.' symbol) at one-third the specified size.

Parent
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See for more information on parenting graphics objects.

`Selected`

`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`

`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`

string

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

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

```
t = stem(Y,'Tag','stem1')
```

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

`UserData`
array

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

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`

array

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` (by setting the `XData` property or specifying the `x` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`

string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

`YData`

scalar, vector, or matrix

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 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 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

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

```
stopasync(obj)
```

Description

`stopasync(obj)` stops any asynchronous read or write operation that is in progress for the serial port object, `obj`.

Remarks

You can write data asynchronously using the `fprintf` or `fwrite` function. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous`. In-progress asynchronous operations are indicated by the `TransferStatus` property.

If `obj` 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:

- Its `TransferStatus` property is configured to `idle`.
- Its `ReadAsyncMode` property is configured to `manual`.
- The data in its output buffer is flushed.

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 `readasync` function, or configure the `ReadAsyncMode` property to `continuous`, then the new data is appended to the existing data in the input buffer.

See Also

Functions

`fprintf`, `fwrite`, `readasync`

Properties

`ReadAsyncMode`, `TransferStatus`

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

```
str2func('str')
```

Description

`str2func('str')` constructs a function handle `fhandle` for the function named in the string `'str'`. The contents of `str` can be the name of a function M-file, or the name of an anonymous function.

You can create a function handle `fh` using any of the following four methods:

- Create a handle to a named function:

```
fh = @functionName;
fh = str2func(functionName);
```

- Create a handle to an anonymous function:

```
fh = @(x)functionDef(x);
fh = str2func('@(x)functionDef(x)');
```

You can create an array of function handles from strings by creating the handles individually with `str2func`, and then storing these handles in a cell array.

Remarks

Nested functions are not accessible to `str2func`. To construct a function handle for a nested function, you must use the function handle constructor, `@`.

Examples

Example 1

To convert the string, `'sin'`, into a handle for that function, type

```
fh = str2func('sin')
fh =
    @sin
```

Example 2

If you pass a function name string in a variable, the function that receives the variable can convert the function name to a function handle using `str2func`. The example below passes the variable, `funcname`, to function `makeHandle`, which then creates a function handle. Here is the function M-file:

```
function fh = makeHandle(funcname)
fh = str2func(funcname);
```

This is the code that calls `makdHandle` to construct the function handle:

```
makeHandle('sin')
ans =
    @sin
```

Example 3

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
```

Example 5

The `dirByType` function shown here creates an anonymous function called `dirCheck`. What the anonymous function does depends upon the value of the `dirType` argument passed in to the primary function. The example demonstrates one possible use of `str2func` with anonymous functions:

```
function dirByType(dirType)
switch(dirType)
   case 'class', leadchar = '@';
   case 'package', leadchar = '+';
   otherwise disp('ERROR: Unrecognized type'), return;
end

dirfile = @(fs)isdir(fs.name);
dirCheckStr = ['@(fs)strcmp(fs.name(1,1),''', leadchar, ''')'];
dirCheckFun = str2func(dirCheckStr);
s = dir;   filecount = length(s);

for k=1:filecount
   fstruct = s(k);
   if dirfile(fstruct) && dirCheckFun(fstruct)
       fprintf('%s directory: %s\n', dirType, fstruct.name)
   end
end
```

Generate a list of class and package directories:

```
dirByType('class')
class directory: @Point
class directory: @asset
class directory: @bond

dirByType('package')
package directory: +containers
package directory: +event
package directory: +mypkg
```

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`

Purpose

Convert string to number

Syntax

```
x = str2num('str')
[x, status] = str2num('str')
```

Description

Note `str2num` uses the `eval` function to convert the input argument. Side effects can occur if the string contains calls to functions. Using `str2double` can avoid some of these side effects.

`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 one or more numbers separated by spaces, commas, or semicolons, such as '`5`', '`10,11,12`', or '`5,10;15,20`'. In addition to numerical values and delimiters, the input string can also include a decimal point, leading `+` or `-` signs, the letter `e` or `d` preceding a power of 10 scale factor, or the letter `i` or `j` indicating a complex or imaginary number.

The following table shows several examples of valid inputs to `str2num`:

String Input	Numeric Output	Output Class
`'500'`	500	1-by-1 scalar `double`
'`500 250 125 67`'	500, 250, 125, 67	1-by-4 row vector of `double`
'`500; 250; 125; 62.5`'	500.0000 250.0000 125.0000 62.5000	4-by-1 column vector of `double`

String Input	Numeric Output	Output Class
'1 23 6 21; 53:56'	1 23 6 21 53 54 55 56	2-by-5 matrix of double
'12e-3 5.9e-3'	0.0120 0.0059	vector of double
'uint16(500)'	500	16–bit unsigned integer

If the input string does not represent a valid number or matrix, str2num(str) returns the empty matrix in x.

[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.

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.

Examples

Input a character string that contains a single number. The output is a scalar double:

```
A = str2num('500')
A =
   500

class(A)
ans =
   double
```

Repeat this operation, but this time using an unsigned 16–bit integer:

```
A = str2num('uint16(500)')
A =
    500

class(A)
```

```
ans =
   uint16
```

Try three different ways of specifying a row vector. Each returns the same answer:

```
str2num('2 4 6 8')              % Separate with spaces.
ans =
     2     4     6     8

str2num('2,4,6,8')              % Separate with commas.
ans =
     2     4     6     8

str2num('[2 4 6 8]')            % Enclose in brackets.
ans =
     2     4     6     8
```

Note that the first two of these commands do not need the MATLAB square bracket operator to create a matrix. The `str2num` function inserts the brackets for you if they are needed.

Use a column vector this time:

```
str2num('2; 4; 6; 8')
ans =
     2
     4
     6
     8
```

And now a 2-by-2 matrix:

```
str2num('2 4; 6 8')
ans =
     2     4
     6     8
```

See Also `num2str`, `str2double`, `hex2num`, `sscanf`, `sparse`, `char`, special characters

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.

`strcat` ignores trailing ASCII white space characters and omits all such characters from the output. White space characters in ASCII are space, newline, carriage return, tab, vertical tab, or form-feed characters, all of which return a `true` response from the MATLAB `isspace` function. Use the concatenation syntax `[s1 s2 s3 ...]` to preserve trailing spaces. `strcat` does not ignore inputs that are cell arrays of strings.

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 =
    'abcdejkl'    'fghimn'
```

Given the 1-by-1 cell array `c = {`Q'}`, the command `t = strcat(a,b,c)` yields

```
t =
    'abcdejklQ'    'fghimnQ'
```

See Also

`strvcat`, `cat`, `vertcat`, `horzcat`, `cellstr`, `special character []`

Purpose

Compare strings

Syntax

```
TF = strcmp('str1', 'str2')
TF = strcmp('str', C)
TF = strcmp(C1, C2)
```

Each of these syntaxes applies 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. `str1` and `str2` can be character arrays of any dimension, but `strcmp` does not return `true` unless the sizes of both arrays are equal, and the contents of the two arrays are the same.

`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` 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

Example 1

Perform a simple comparison of two strings:

```
strcmp('Yes', 'No')
ans =
     0
strcmp('Yes', 'Yes')
ans =
     1
```

Example 2

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'};
```

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
```

Example 3

Compare a string vector to a cell array of strings, a string vector to a string array, and a string array to a cell array of strings. Start by creating a cell array of strings (`cellArr`), a string array containing the same strings plus space characters for padding s(`strArr`), and a string vector containing one of the strings plus padding (`strVec`):

```
cellArr = { ...
   'There are 10 kinds of people in the world,'; ...
   'those who understand binary math,'; ...
   'and those who don''t.'};

strArr = char(cellArr);
strVec = strArr(2,:)
strVec =
    those who understand binary math,
```

Remove the space padding from the string vector and compare it to the cell array. The MATLAB software compares the string with each row of the cell array, finding a match on the second row:

```
strcmp(deblank(strVec), cellArr)
ans =
     0
     1
     0
```

Compare the string vector with the string array. Unlike the case above, MATLAB does not compare the string vector with each row of the string array. It compares the entire contents of one against the entire contents of the other:

```
strcmp(strVec, strArr)
```

```
ans =
     0
```

Lastly, compare each row of the three-row string array against the same rows of the cell array. MATLAB finds them all to be equivalent. Note that in this case you do not have to remove the space padding from the string array:

```
strcmp(strArr, cellArr)
ans =
     1
     1
     1
```

See Also

`strncmp`, `strncmpi`, `strmatch`, `strfind`, `findstr`, `regexp`, `regexpi`, `regexprep`, `regexptranslate`

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 software uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

Use the `streamline` command to plot the data returned by `stream2`.

Examples

This example draws 2-D streamlines from data representing air currents over regions of North America.

```
load wind
[sx,sy] = meshgrid(80,20:10:50);
streamline(stream2(x(:,:,5),y(:,:,5),u(:,:,5),v(:,:,5),sx,sy));
```

See Also

`coneplot`, `stream3`, `streamline`

"Volume Visualization" on page 1-106 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 software 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`.

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 1-106 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_handle,...)
h = streamline(...)
```

Description

`streamline(X,Y,Z,U,V,W,startx,starty,startz)` draws streamlines from 3-D vector data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (such as the data produced by `meshgrid`). `startx`, `starty`, `startz` define the starting positions of the streamlines. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

`streamline(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)`, where `[M,N,P] = size(U)`.

`streamline(XYZ)` assumes `XYZ` is a precomputed cell array of vertex arrays (as produced by `stream3`).

`streamline(X,Y,U,V,startx,starty)` draws streamlines from 2-D vector data `U`, `V`. The arrays `X`, `Y` define the coordinates for `U`, `V` and must be monotonic and 2-D plaid (such as the data produced by `meshgrid`). `startx` and `starty` define the starting positions of the streamlines. The output argument `h` contains a vector of line handles, one handle for each streamline.

`streamline(U,V,startx,starty)` assumes the arrays `X` and `Y` are defined as `[X,Y] = meshgrid(1:N,1:M)`, where `[M,N] = size(U)`.

`streamline(XY)` assumes `XY` is a precomputed cell array of vertex arrays (as produced by `stream2`).

`streamline(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[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_handle,...)` plots into the axes object with the handle `axes_handle` instead of the into 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 1-106 for related functions

Specifying Starting Points for Stream Plots for related information

Stream Line Plots of Vector Data for another example

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`, 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. 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 streamparticles
`EraseMode`	`xor`
`LineStyle`	`none`
`Marker`	`o`

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.

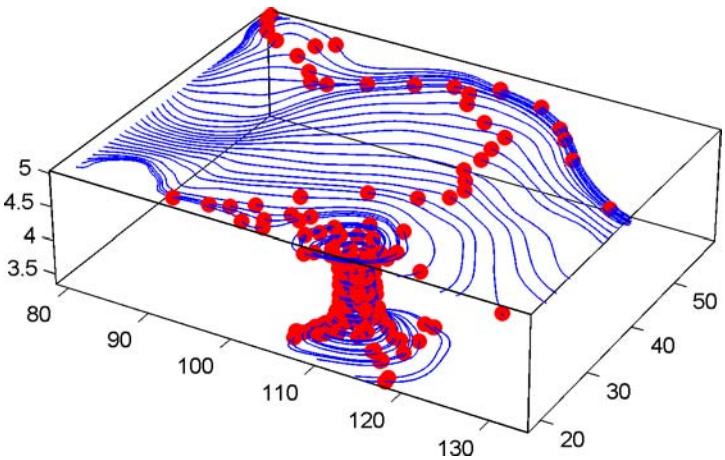
`h = streamparticles(...)` returns a vector of handles to the line objects it creates.

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 1-106 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_handle,...)
h = streamribbon(...)
```

Description

`streamribbon(X,Y,Z,U,V,W,startx,starty,startz)` draws stream ribbons from vector volume data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`). `startx`, `starty`, and `startz` define the starting positions of the stream ribbons at the center of the ribbons. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The twist of the ribbons is proportional to the curl of the vector field. The width of the ribbons is calculated automatically.

`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`.

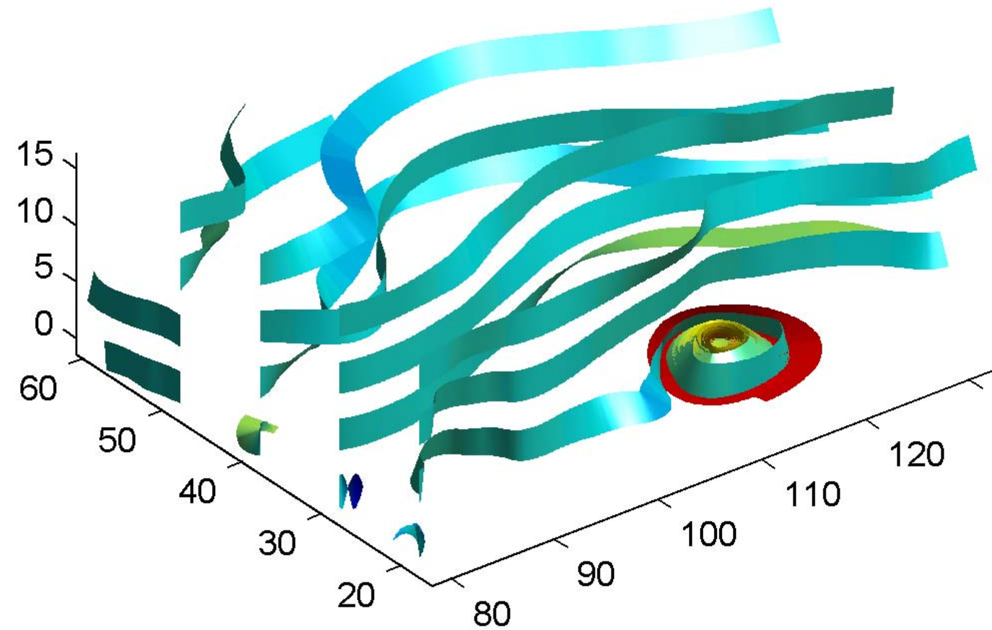
`streamribbon(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

`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.

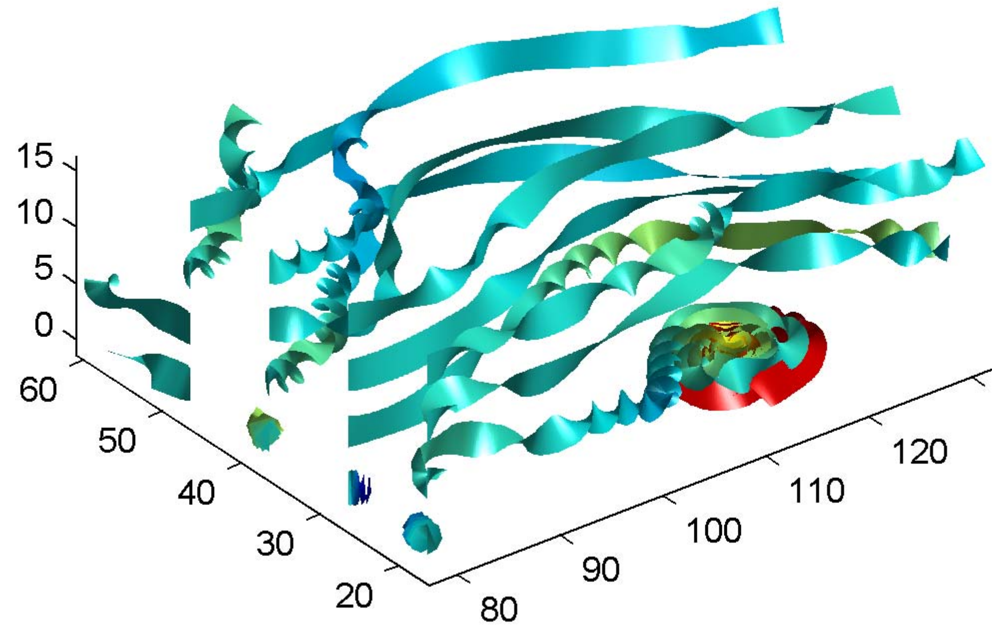
```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
% Define viewing and lighting
axis tight
shading interp;
view(3);
camlight; lighting gouraud
```



This example uses precalculated vertex data (`stream3`), curl average velocity (`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
```

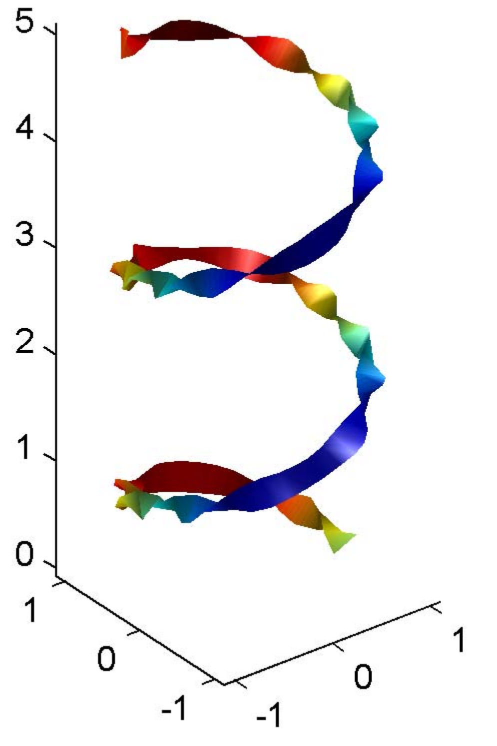
```
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
cav = curl(x,y,z,u,v,w);
spd = sqrt(u.^2 + v.^2 + w.^2).*.1;
streamribbon(verts,x,y,z,cav,spd);
% Define viewing and lighting
axis tight
shading interp
view(3)
camlight; lighting gouraud
```



This example specifies a twist angle for the stream ribbon.

```
t = 0:.15:15;
verts = {[cos(t)' sin(t)' (t/3)']};
twistangle = {cos(t)'};
streamribbon(verts,twistangle);
% Define viewing and lighting
axis tight
```

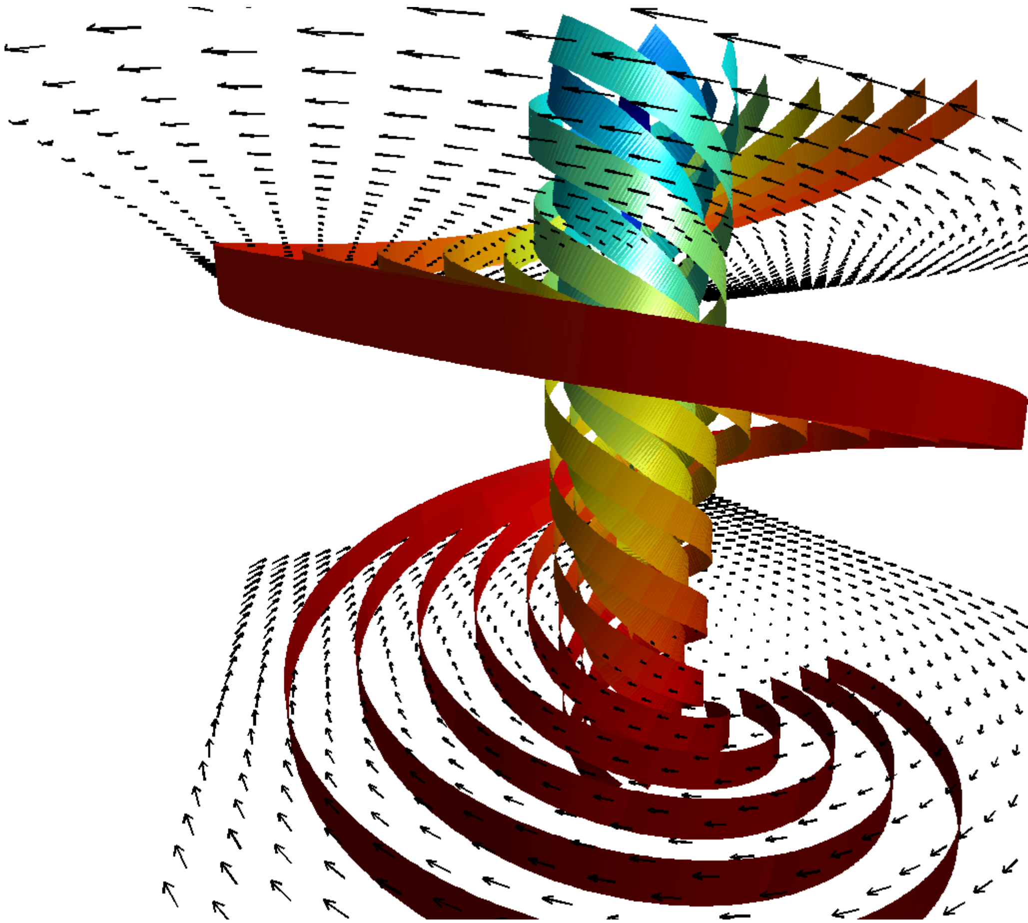
```
shading interp;
view(3);
camlight; lighting gouraud
```



This example combines cone plots (`coneplot`) and stream ribbon plots in one graph.

```
% 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;
[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 1-106 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

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_handle,...)
h = streamslice(...)
[vertices arrowvertices] = streamslice(...)
```

Description

`streamslice(X,Y,Z,U,V,W,startx,starty,startz)` draws well-spaced streamlines (with direction arrows) from vector data `U`, `V`, `W` in axis aligned *x*-, *y*-, *z*-planes starting at the points in the vectors `startx`, `starty`, `startz`. (The section Specifying Starting Points for Stream Plots provides more information on defining starting points.) The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`). `U`, `V`, `W` must be m-by-n-by-p volume arrays.

Do 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.

`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(...,'arrowsmode')` 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)
- `cubic` — Cubic interpolation
- `nearest` — Nearest-neighbor interpolation

See `interp3` for more information on interpolation methods.

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

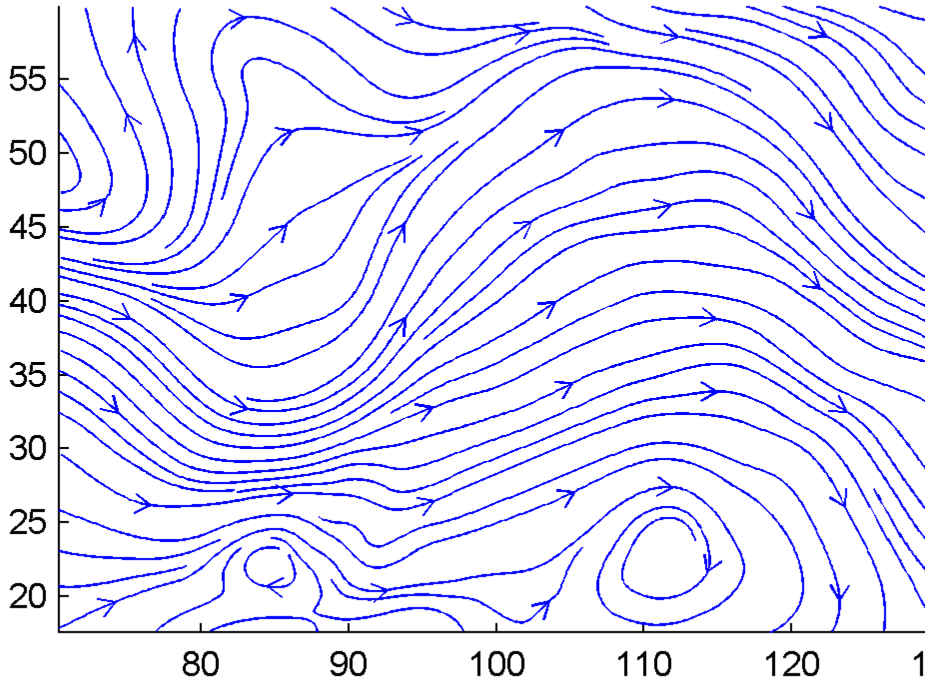
`h = streamslice(...)` returns a vector of handles to the line objects created.

`[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`.

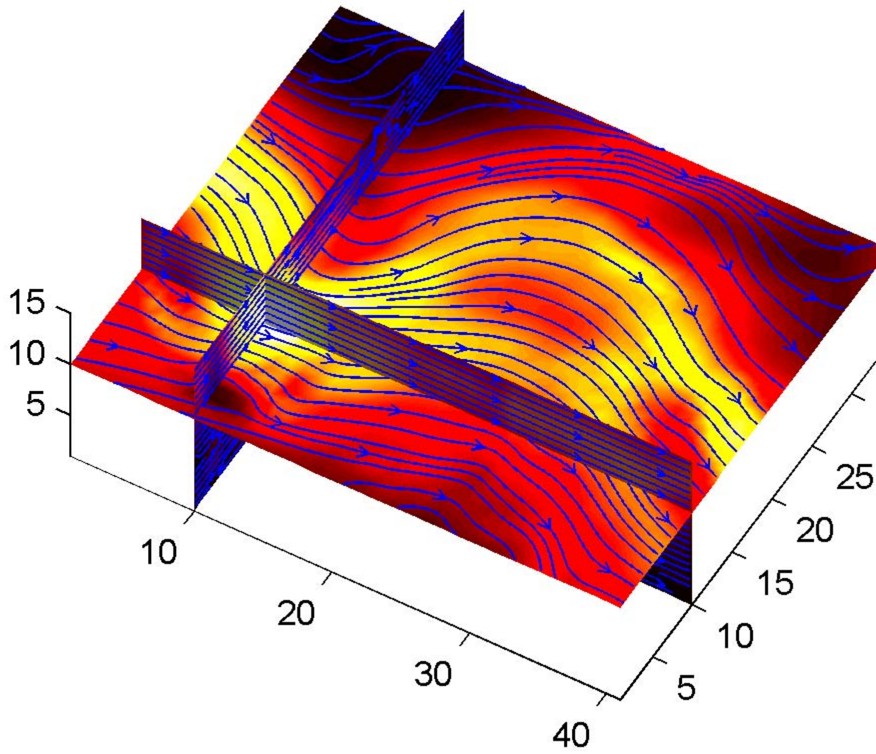
```
load wind
streamslice(x,y,z,u,v,w,[],[],[5])
axis tight
```



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
```

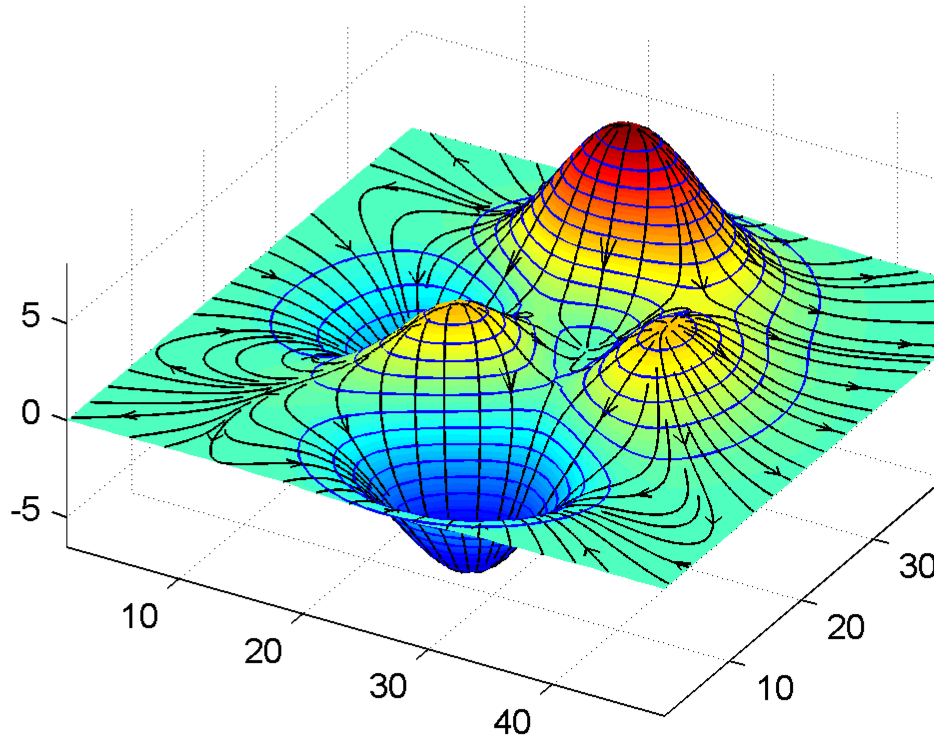
```
[verts averts] = streamslice(u,v,w,10,10,10);
streamline([verts averts])
spd = sqrt(u.^2 + v.^2 + w.^2);
hold on;
slice(spd,10,10,10);
colormap(hot)
shading interp
view(30,50); axis(volumebounds(spd));
camlight; material([.5 1 0])
```



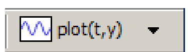
This example superimposes contour lines on a surface and then uses `streamslice` to draw lines that indicate the gradient of the surface. `interp2` is used to find the points for the lines that lie on the surface.

```
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
```



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.

See Also

`contourslice`, `slice`, `streamline`, `volumebounds`

"Volume Visualization" on page 1-106 for related functions

Specifying Starting Points for Stream Plots for related information

Purpose

Create 3-D stream tube plot

Syntax

```
streamtube(X,Y,Z,U,V,W,startx,starty,startz)
streamtube(U,V,W,startx,starty,startz)
streamtube(vertices,X,Y,Z,divergence)
streamtube(vertices,divergence)
streamtube(vertices,width)
streamtube(vertices)
streamtube(...,[scale n])
streamtube(axes_handle,...)
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.

`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.

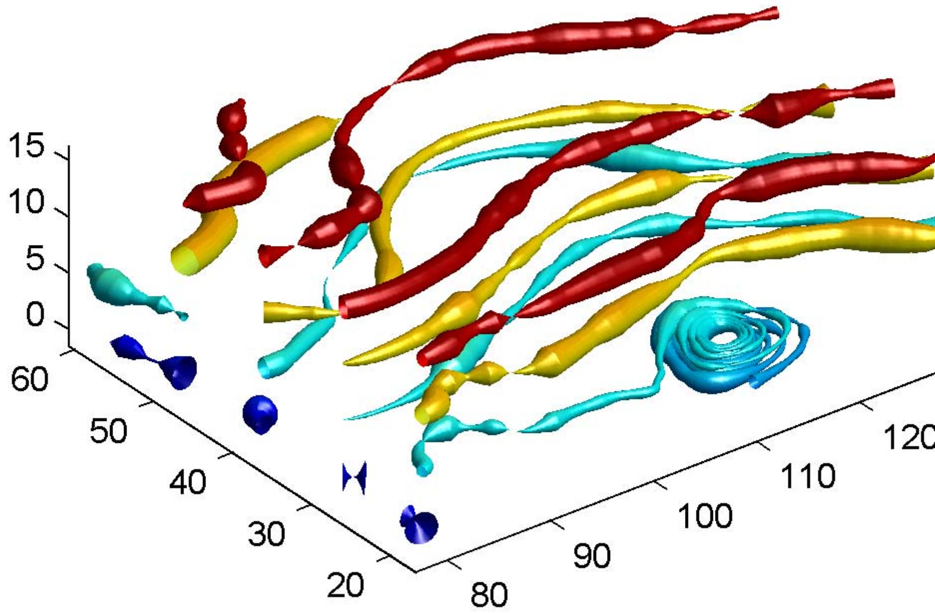
streamtube(axes_handle,...) plots into the axes object with the handle axes_handle instead of into the current axes object (gca).

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.

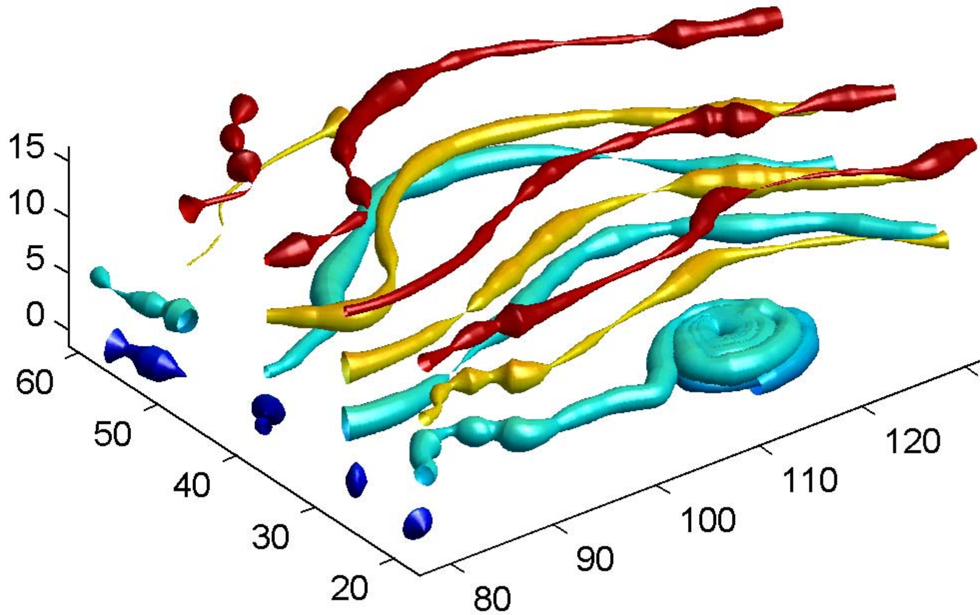
```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamtube(x,y,z,u,v,w,sx,sy,sz);
% Define viewing and lighting
view(3)
axis tight
shading interp;
camlight; lighting gouraud
```



This example uses precalculated vertex data (`stream3`) and divergence (`divergence`).

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
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
```



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.

See Also

`divergence`, `streamribbon`, `streamline`, `stream3`

"Volume Visualization" on page 1-106 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 then 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 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 . 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, strarray)
x = strmatch(str, strarray, 'exact')
```

Description

`x = strmatch(str, strarray)` looks through the rows of the character array or cell array of strings `strarray` to find strings that begin with the text contained in `str`, and returns the matching row indices. Any trailing space characters in `str` or `strarray` are ignored when matching. `strmatch` is fastest when `strarray` is a character array.

`x = strmatch(str, strarray, 'exact')` compares `str` with each row of `strarray`, looking for an exact match of the entire strings. Any trailing space characters in `str` or `strarray` are ignored when matching.

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 applies 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. str1 and str2 can be character arrays of any dimension.

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 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

Example 1

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
```

Example 2

Create two 5-by-10 string arrays `str1` and `str2` that are equal except for the element at row 4, column 3. Using linear indexing, this is element 14:

```
str1 = ['AAAAAAAAAA'; 'BBBBBBBBBB'; 'CCCCCCCCCC'; ...
        'DDDDDDDDDD'; 'EEEEEEEEEE']
str1 =
     AAAAAAAAAA
     BBBBBBBBBB
```

```
        CCCCCCCCCC
        DDDDDDDDDD
        EEEEEEEEEE

str2 = str1;
str2(4,3) = '-'
str2 =
        AAAAAAAAAA
        BBBBBBBBBB
        CCCCCCCCCC
        DD-DDDDDDD
        EEEEEEEEEE
```

Because MATLAB compares the arrays in linear order (that is, column by column rather than row by row), `strncmp` finds only the first 13 elements to be the same:

```
str1   A B C D E A B C D E A B C D E
str2   A B C D E A B C D E A B C - E
                                 |
                            element 14

strncmp(str1, str2, 13)
ans =
     1

strncmp(str1, str2, 14)
ans =
     0
```

See Also

`strcmp`, `strcmpi`, `strmatch`, `strfind`, `findstr`, `regexp`, `regexpi`, `regexprep`, `regexptranslate`

Purpose Read formatted data from string

Note `strread` is not recommended. Use `textscan` to read data from a string.

Syntax

```
A = strread('str')
[A, B, ...] = strread('str')
[A, B, ...] = strread('str', 'format')
[A, B, ...] = strread('str', 'format', N)
[A, B, ...] = strread('str', 'format', N, param, value, ...)
```

Description

`A = strread('str')` reads numeric data from input string `str` into a 1-by-`N` vector `A`, where `N` equals the number of whitespace-separated numbers in `str`. Use this form only with strings containing numeric data. See "Example 1" on page 2-3525 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 2-3526 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 2-3526 and "Example 5" on page 2-3527 below.

The table Formats for strread on page 2-3522 lists the valid format specifiers. More information on using formats is available under "Formats" on page 2-3525 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 2-3526 below.

`[A, B, ...] = strread('str', 'format', N, param, value, ...)` customizes `strread` using `param/value` pairs, as listed in the table Parameters and Values for strread on page 2-3523 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 2-3528 below.

Formats for strread

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

Formats for strread (Continued)

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 2-3528 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`	Any from the list below: `\b` Backspace `\n` New line `\r` Carriage return `\t` Horizontal tab `\\` Backslash `%%` Percent sign `''` Single quotation mark	Treats vector of characters, `*`, as white space. Default is `\b\r\n\t`.
`delimiter`	Delimiter character	Specifies delimiter character. Default is one or more whitespace characters.
`expchars`	Exponent characters	Default is `eEdD`.

Parameters and Values for strread (Continued)

param	value	Action
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 //.
emptyvalue	Value to return for empty numeric fields in delimited files	Default is NaN.

Remarks

If you terminate the input string with a newline character (\n), strread returns arrays of equal size by padding arrays of lesser size with the emptyvalue character:

```
[A,B,C] = strread(sprintf('5,7,1,9\n'),'%d%d%d', ...
             'delimiter', ',', 'emptyvalue',NaN)
A =
     5
     9
B =
     7
   NaN
C =
     1
   NaN
```

If you remove the `\n` from the input string of this example, array `A` continues to be a 2-by-1 array, but `B` and `C` are now 1-by-1.

Delimiters

If your data uses a character other than a space as a delimiter, you must use the `strread` parameter `'delimiter'` to specify the delimiter. For example, if the string `str` used a semicolon as a delimiter, you would use this command:

```
[names, types, x, y, answer] = strread(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 `strread` function continues reading `str` until the entire string is read. If there are fewer format specifiers than there are entities in `str`, `strread` reapplies the format specifiers, starting over at the beginning. See "Example 5" on page 2-3527 below.

The `format` string supports a subset of the conversion specifiers and conventions of the C language `fscanf` routine. White-space characters in the `format` string are ignored.

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
    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, sscanf

strrep

Purpose

Find and replace substring

Syntax

`modifiedStr = strrep(origStr, oldSubstr, newSubstr)`

Description

`modifiedStr = strrep(origStr, oldSubstr, newSubstr)` replaces all occurrences of the string `oldSubstr` within string `originalStr` with the string `newSubstr`.

`strrep(origStr, oldSubstr, newSubstr)`, when any of the inputs 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)` parses input string `str` from left to right, returning part or all of that string in `token`. Using the white-space character as a delimiter, the `token` output begins at the start of `str`, skipping any delimiters that might appear at the start, and includes all characters up to either the next delimiter or the end of the string. White-space characters include space (ASCII 32), tab (ASCII 9), and carriage return (ASCII 13).

The `str` argument can be a string of characters enclosed in single quotation marks, a cell array of strings each enclosed in single quotation marks, or a variable representing either of the two. If `str` is a cell array of `N` strings, then `token` is a cell array of `N` tokens, with `token{1}` derived from `str{1}`, `token{2}` from `str{2}`, and so on.

`token = strtok(str, delimiter)` is the same as the above syntax except that you specify the delimiting character(s) yourself using the `delimiter` character vector input. White-space characters are not considered to be delimiters when using this syntax unless you include them in the `delimiter` argument. If the `delimiter` input specifies more than one character, MATLAB treats each character as a separate delimiter; it does not treat the multiple characters as a delimiting string. The number and order of characters in the `delimiter` argument is unimportant.

`[token, remain] = strtok('str', ...)` returns in `remain` that part of `str`, if any, that follows `token`. 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 cell array of string remainders.

Examples

Example 1

This example uses the default white-space delimiter:

```
s = '  This is a simple example.';
[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%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`

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', {}, ...)
s = struct
s = struct([])
s = 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 `{}`, the MATLAB software 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`, `...`

`s = struct` creates a 1-by-1 structure with no fields.

`s = struct([])` creates an empty structure with no fields.

`s = struct(obj)` creates a structure `s` that is identical to the underlying structure in the input object `obj`. MATLAB does not convert obj, but rather creates s as a new structure. This structure does not retain the class information in obj.

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

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`, dynamic field names

structfun

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 Nth element of `A` is the result of applying `fun` to the Nth field of `S`, and the order of the fields is the same as that returned by a call to `fieldnames`. (`A` is returned as one or more scalar structures when the `UniformOutput` option is set to `false`. See the table below.))

`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

<table>
<tr><th>Parameter</th><th>Value</th></tr>
<tr><td><code>'UniformOutput'</code></td><td>Logical value indicating whether or not the outputs of <code>fun</code> can be returned without encapsulation in a structure. The default value is <code>true</code>.

If equal to logical 1 (<code>true</code>), <code>fun</code> 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.

If equal to logical 0 (<code>false</code>), <code>structfun</code> returns a scalar structure or multiple scalar structures having fields that are the same as the fields of the input structure <code>S</code>. The values in the output structure fields are the results of calling <code>fun</code> on the corresponding values in the input structure <code>B</code>. In this case, the outputs can be of any data type.</td></tr>
<tr><td><code>'ErrorHandler'</code></td><td>Function handle specifying the function MATLAB is to call if the call to <code>fun</code> fails. MATLAB calls the error handling function with the following input arguments:

• A structure, with the fields <code>'identifier'</code>, <code>'message'</code>, and <code>'index'</code>, 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.

The error handling function should either rethrow an error or return the same number of outputs as <code>fun</code>. These outputs are then returned as the outputs of <code>structfun</code>. If <code>'UniformOutput'</code> is <code>true</code>, the outputs of the error handler must also be scalars of the same type as the outputs of <code>fun</code>.

For example,

<code>function [A, B] = errorFunc(S, ...</code>
<code>   varargin)</code>
<code>warning(S.identifier, S.message);</code>
<code>A = NaN; B = NaN;</code></td></tr>
</table>

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`

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, vertcat, horzcat, int2str, mat2str, num2str, strings, special character []

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 vector with `ndim(A)` elements (in this case, 2), where `siz(1)` is the number of rows and `siz(2)` is the number of columns.

`IND = sub2ind(siz,I1,I2,...,In)` returns the linear index equivalent to the `n` subscripts `I1`,`I2`,...,`In` for an array of size `siz`. `siz` is an `n`-element vector that specifies the size of each array dimension.

Examples

Create a 3-by-4-by-2 array, `A`.

```
A = [17 24 1 8; 2 22 7 14; 4 6 13 20];
A(:,:,2) = A - 10

A(:,:,1) =

    17    24     1     8
     2    22     7    14
     4     6    13    20


A(:,:,2) =

     7    14    -9    -2
    -8    12    -3     4
    -6    -4     3    10
```

The value at row 2, column 1, page 2 of the array is -8.

```
A(2,1,2)
```

```
ans =

    -8
```

To convert `A(2,1,2)` into its equivalent single subscript, use `sub2ind`.

```
sub2ind(size(A),2,1,2)

ans =

    14
```

You can now access the same location in `A` using the single subscripting method.

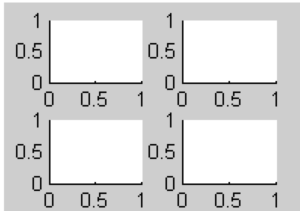
```
A(14)

ans =

    -8
```

See Also

`ind2sub`, `find`, `size`

Purpose

Create axes in tiled positions



GUI Alternatives

To add subplots to a figure, 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)
subplot(h)
subplot('Position',[left bottom width height])
subplot(..., prop1, value1, prop2, value2, ...)
h = subplot(...)
subplot(m,n,p,'v6')
```

Description

`subplot` divides the current figure into rectangular panes that are numbered rowwise. Each pane contains an axes object which you can manipulate using `Axes Properties`. 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.

subplot(m,n,p,'replace') If the specified axes object already exists, delete it and create a new axes.

subplot(m,n,P), where P is a vector, specifies an axes position that covers all the subplot positions listed in P, including those spanned by P. For example, subplot(2,3,[2 5]) creates one axes spanning positions 2 and 5 only (because there are no intervening locations in the grid), while subplot(2,3,[2 6]) creates one axes spanning positions 2, 3, 5, and 6.

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.

subplot(..., prop1, value1, prop2, value2, ...) sets the specified property-value pairs on the subplot axes object. Available property/value pairs are described more fully in Axes Properties. To add the subplot to a specific figure or uipanel, pass the handle as the value for the Parent property. You cannot specify both a Parent and a Position; that is, subplot('Position',[left bottom width height], 'Parent',h) is not a valid syntax.

h = subplot(...) returns the handle to the new axes object.

Backward-Compatible Version

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.

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.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

If a subplot specification causes a new axis to overlap a existing axis, the existing axis is deleted - unless the position of the new and existing axis are identical. For example, the statement `subplot(1,2,1)` deletes all existing axes overlapping the left side of the figure window and creates a new axis on that side—unless there is an axes there with a position that exactly matches the position of the new axes (and `'replace'` was not specified), in which case all other overlapping axes will be deleted and the matching axes will become the current axes.

You can add subplots to GUIs as well as to figures. For information about creating subplots in a GUIDE-generated GUI, see in the MATLAB Creating Graphical User Interfaces documentation.

If a `subplot` specification causes a new axes object to overlap any existing axes, `subplot` deletes the existing axes object and uicontrol objects. However, if the `subplot` specification exactly matches the position of an existing axes object, 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
```

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)

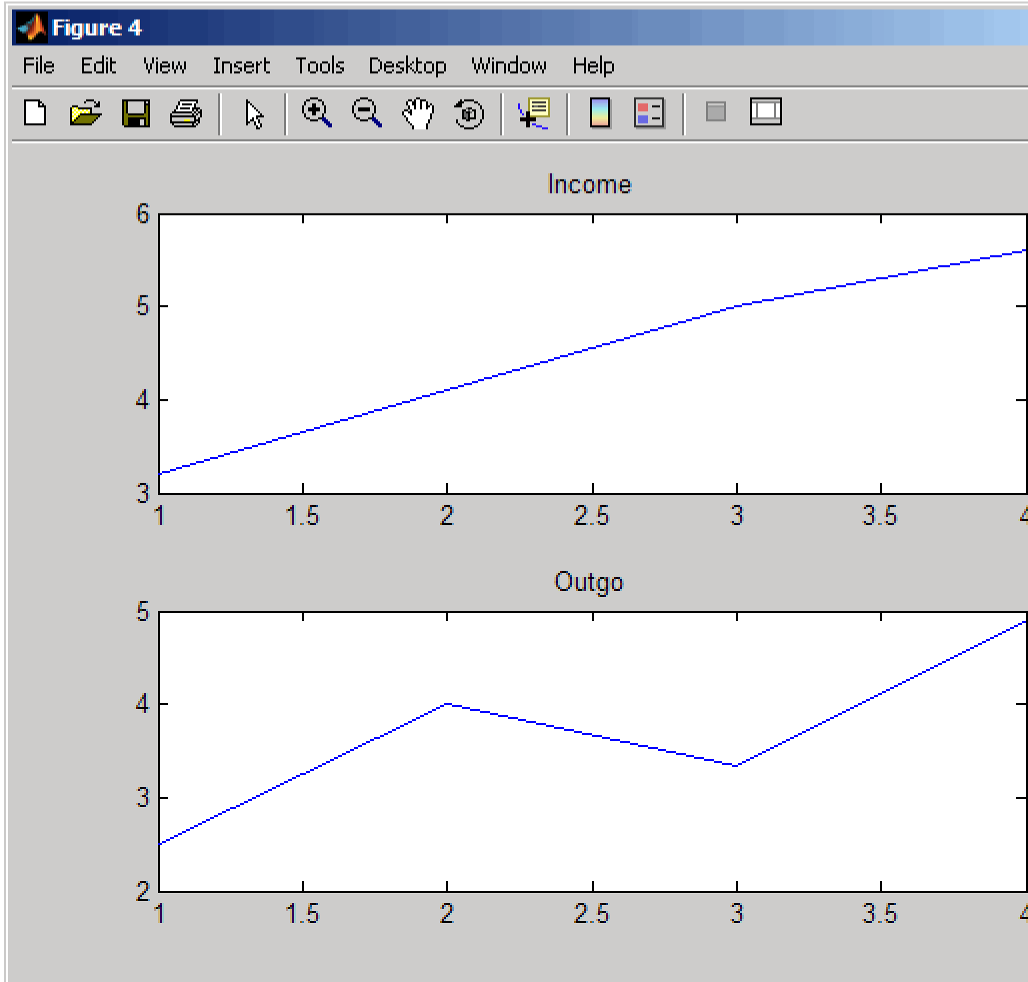
The command subplot(111) is not identical in behavior to subplot(1,1,1) and exists only for compatibility with previous releases. This syntax does not immediately create an axes object, but instead sets up the figure so that the next graphics command executes a clf reset (deleting all figure children) 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

Upper and Lower Subplots with Titles

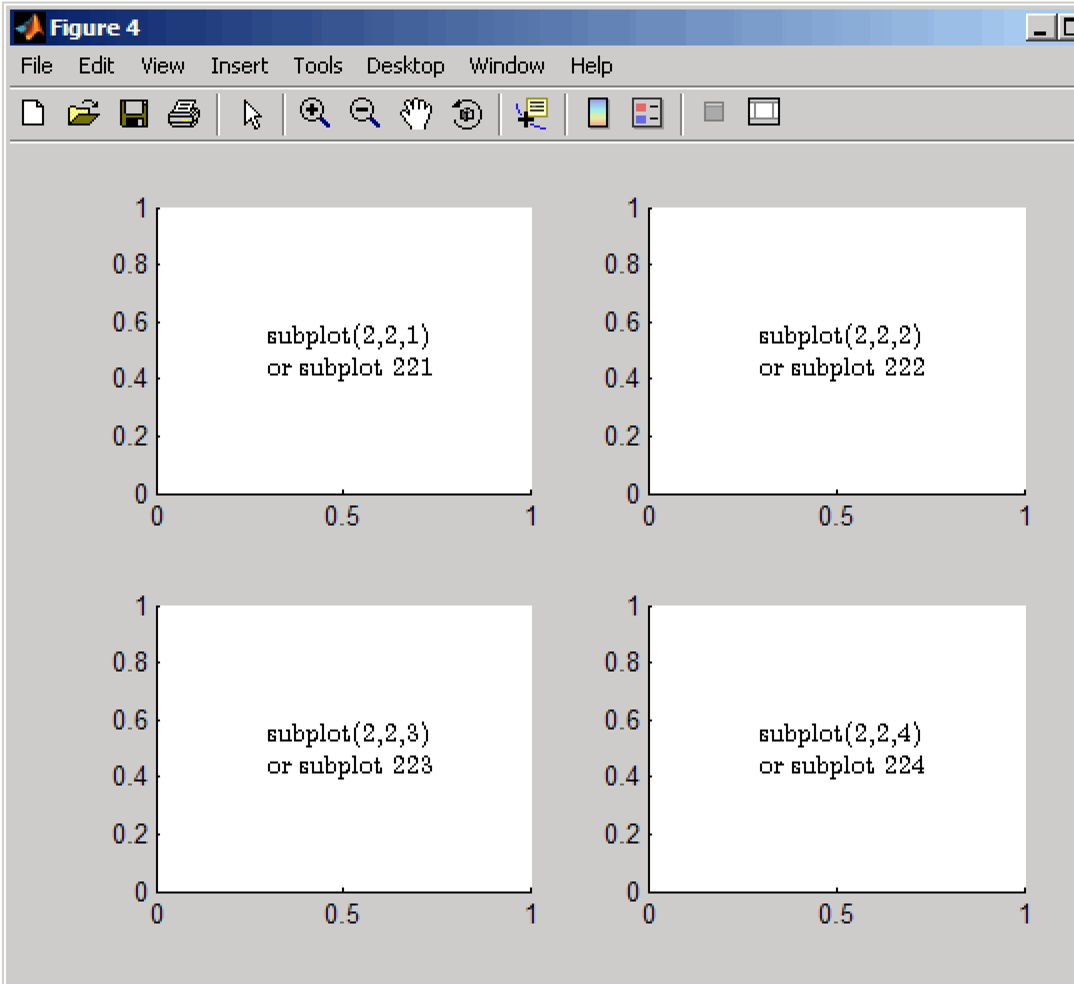
To plot income in the top half of a figure and outgo in the bottom half,

```
income = [3.2 4.1 5.0 5.6];
outgo = [2.5 4.0 3.35 4.9];
subplot(2,1,1); plot(income)
title('Income')
subplot(2,1,2); plot(outgo)
title('Outgo')
```



Subplots in Quadrants

The following illustration shows four subplot regions and indicates the command used to create each.

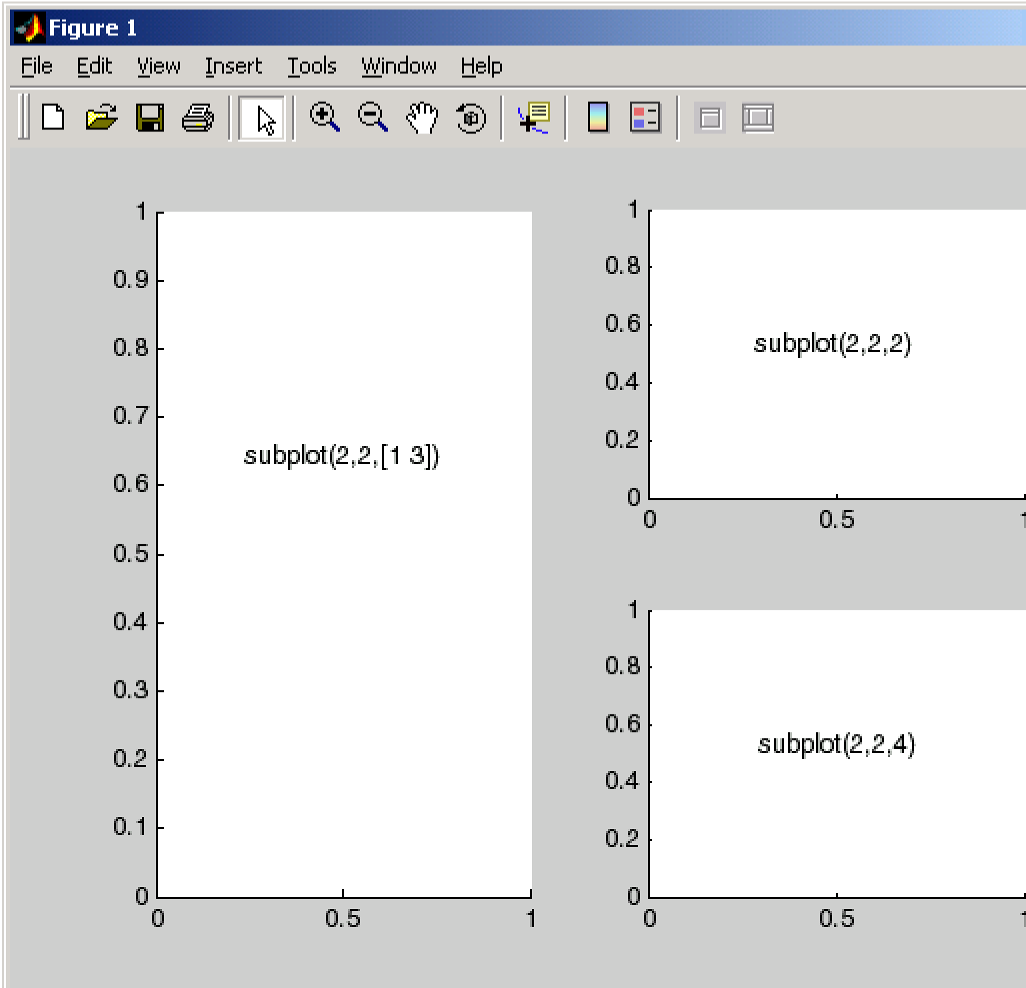


Assymetrical Subplots

The following combinations produce asymmetrical arrangements of subplots.

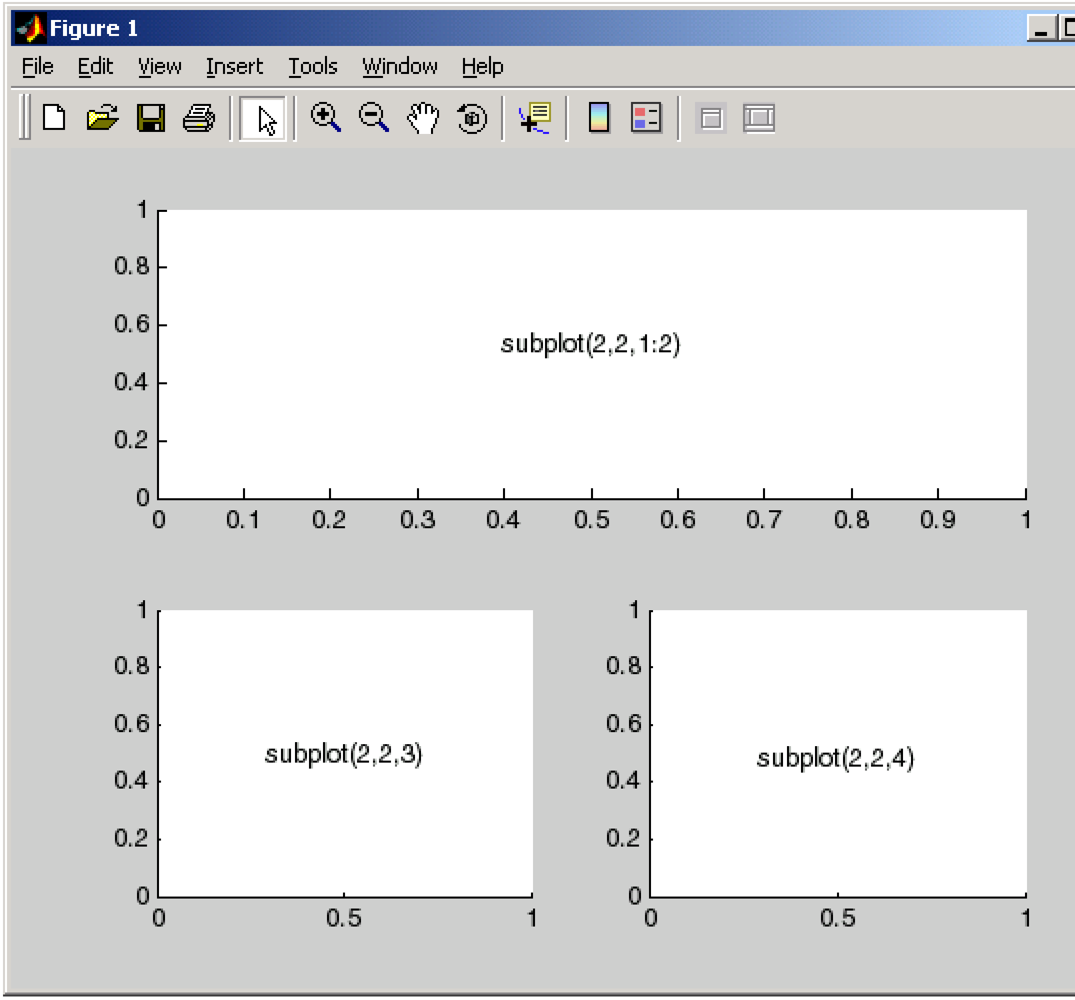
```
subplot(2,2,[1 3])
```

```
subplot(2,2,2)
subplot(2,2,4)
```



You can also use the colon operator to specify multiple locations if they are in sequence.

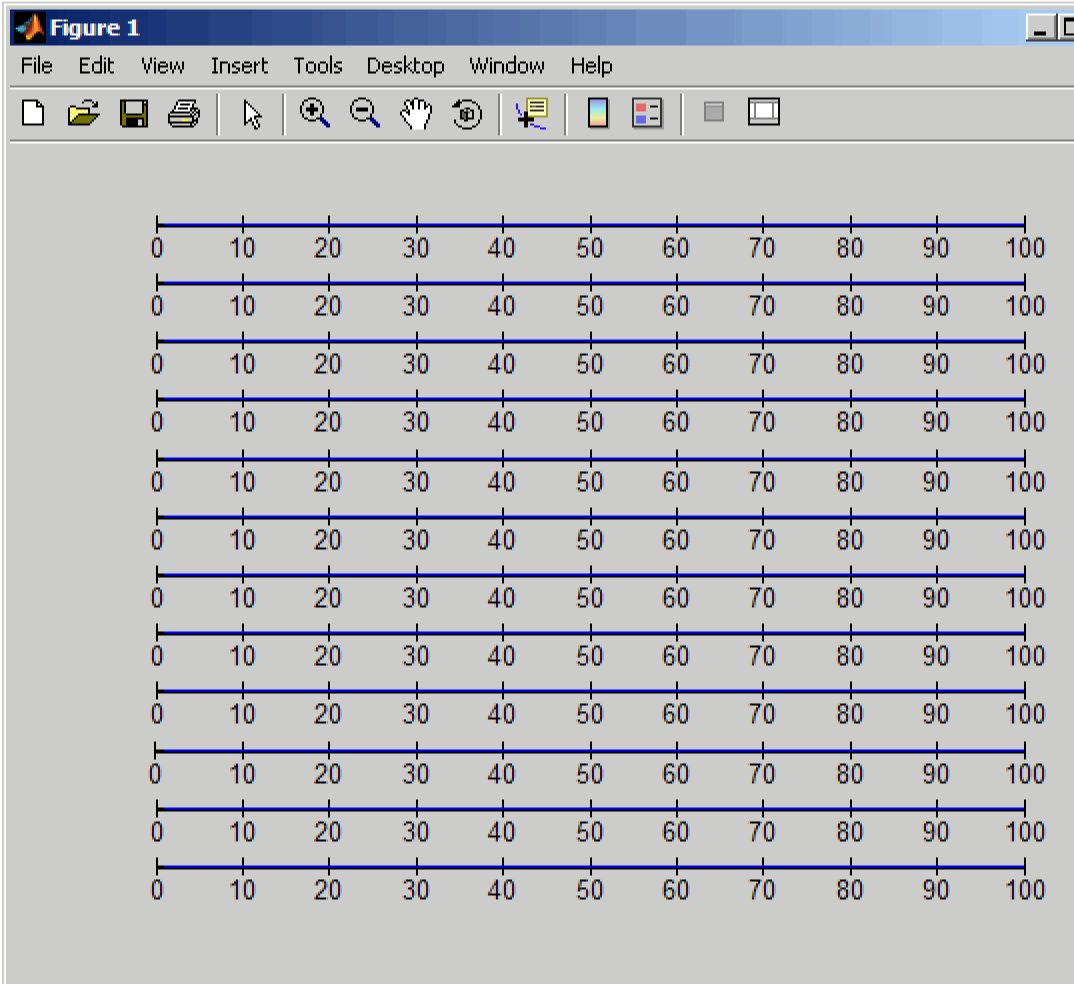
```
subplot(2,2,1:2)
subplot(2,2,3)
subplot(2,2,4)
```



Suppressing Axis Ticks

When you create many subplots in a figure, the axes tickmarks, which are shown by default, can either be obliterated or can cause axes to collapse, as the following code demonstrates:

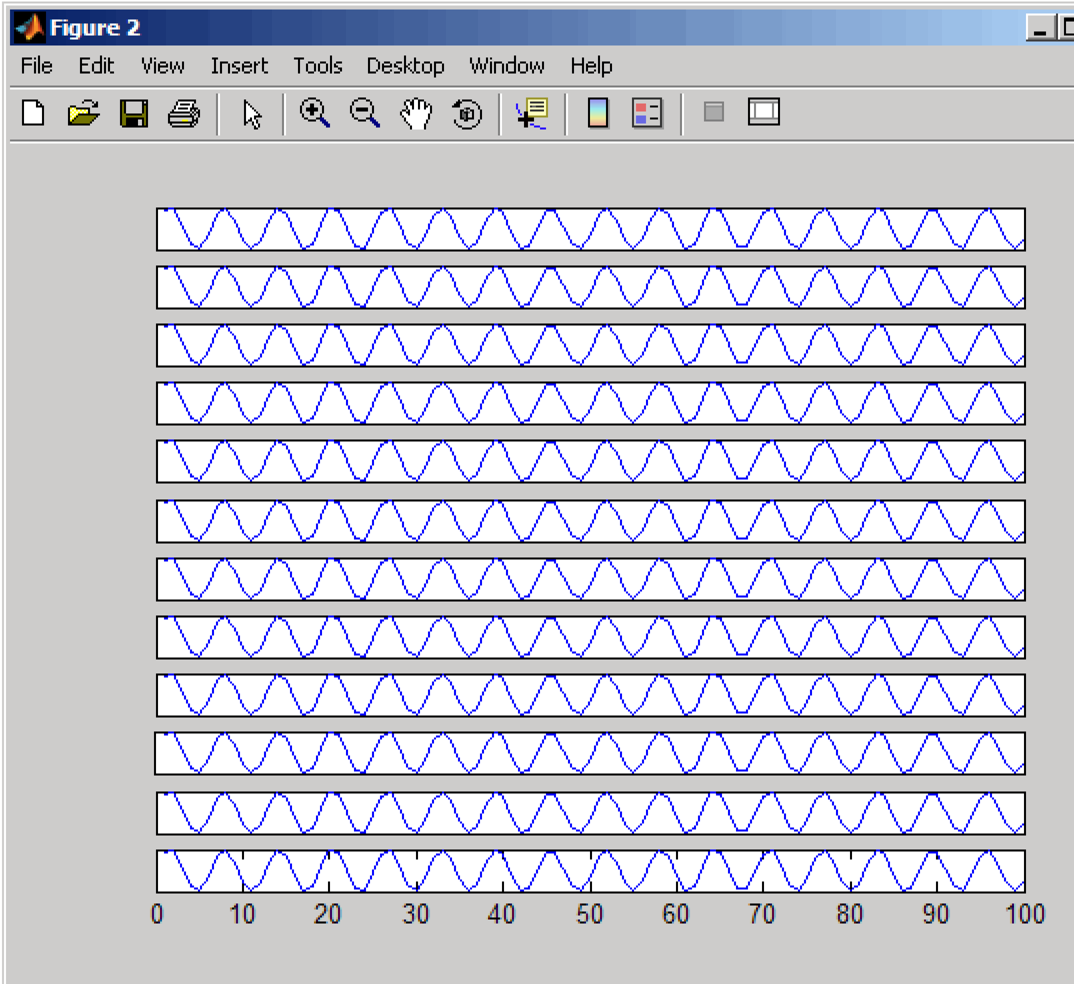
```
figure
for i=1:12
    subplot(12,1,i)
    plot (sin(1:100)*10^(i-1))
end
```



One way to get around this issue is to enlarge the figure to create enough space to properly display the tick labels.

Another approach is to eliminate the clutter by suppressing xticks and yticks for subplots as data are plotted into them. You can then label a single axes if the subplots are stacked, as follows:

```
figure
for i=1:12
    subplot(12,1,i)
    plot (sin(1:100)*10^(i-1))
    set(gca,'xtick',[],'ytick',[])
end
% Reset the bottom subplot to have xticks
set(gca,'xtickMode', 'auto')
```

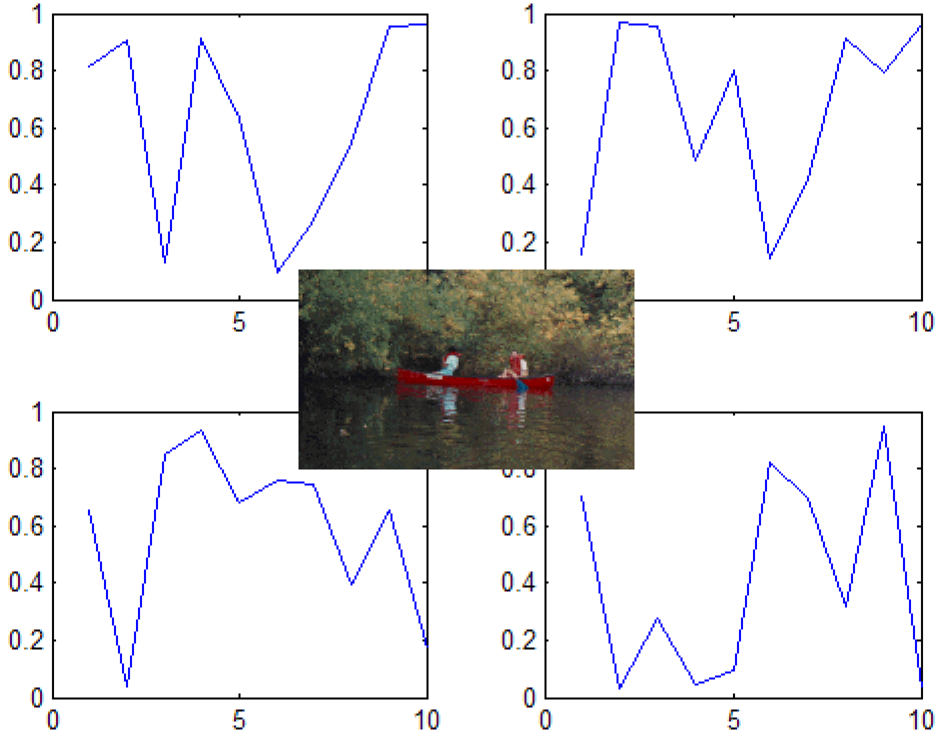


Plotting Axes Over Subplots

Place a plot in the center, on top of four other plots, using the `axes` and `subplot` functions:

```
for i = 1:4
```

```
    subplot(2, 2, i)
    plot(rand(1, 10));
end
axes('Position', [.35, .35, .3, .3]);
imshow('canoe.tif')
```



See Also

`axes`, `cla`, `clf`, `figure`, `gca`

"Basic Plots and Graphs" on page 1-91 for more information

in the MATLAB Creating Graphical User Interfaces documentation describes adding subplots to GUIs.

subsasgn

Purpose Subscripted assignment

Syntax

```
A = subsasgn(A, S, B)
```

Description

`A = subsasgn(A, S, B)` is called by MATLAB for the syntax `A(i) = B`, `A{i} = B`, or `A.i = B` when `A` is an object.

MATLAB uses the built-in `subsasgn` function to interpret indexed assignment statements. Modify the indexed assignment behavior of classes by overloading `subsasgn` in the class.

If `A` is a fundamental class (see), then an indexed reference to `A` calls the built-in `subsasgn` function. It does not call a `subsasgn` method that you have overloaded for that class. Therefore, if `A` is an array of class `double`, and there is an `@double/subsasgn` method on your MATLAB path, the statement `A(I) = B` calls the MATLAB built-in `subsasgn` function.

Inputs

`A`

Object

`S`

`struct` array with two fields, `type` and `subs`.

- `type` is a string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts, `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.
- `subs` is a cell array or string containing the actual subscripts.

`B`

Assignment value (right-hand side)

Outputs

`A`

Result of evaluating assignment.

Examples

See how MATLAB calls `subsasgn` for the expression:

```
A(1:2,:) = B;
```

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,':'}`. The string `':'` indicates a colon used as a subscript.

See how MATLAB calls `subsasgn` for the expression:

```
A{1:2} = B;
```

The syntax `A{1:2} = B` calls `A = subsasgn(A,S,B)` where `S.type = '{}'` and `S.subs = {[1 2]}`.

See how MATLAB calls `subsasgn` for the expression:

```
A.field = B;
```

The syntax `A.field = B` calls `A = subsasgn(A,S,B)` where `S.type = '.'` and `S.subs = 'field'`.

See how MATLAB calls `subsasgn` for the expression:

```
A(1,2).name(3:5)=B;
```

Simple calls combine in a straightforward way for more complicated indexing 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:

`S(1).type = '()'`	`S(2).type = '.'`	`S(3).type = '()'`
`S(1).subs = {1,2}`	`S(2).subs = 'name'`	`S(3).subs = {[3 4 5]}`

Algorithm

In the assignment `A(J,K,...)   = B(M,N,...)`, subscripts `J`, `K`, `M`, `N`, and so on, can be scalar, vector, or arrays, when all the following are true:

- The number of subscripts specified for `B`, excluding trailing subscripts equal to 1, does not exceed the value returned by `ndims(B)`.
- 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.

  See `numel` for information concerning the use of `numel` with regards to the overloaded `subsasgn` function.

See Also

`subsref` | `substruct`

Tutorials

-

Purpose

Subscript indexing with object

Syntax

`ind = subsindex(A)`

Description

`ind = subsindex(A)` called by MATLAB for the expression `X(A)` when `A` is an object, unless such an expression results in a call to an overloaded `subsref` or `subsasgn` method for `X`. `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`.) Call `subsindex` directly from an overloaded `subsref` or `subsasgn` method.

MATLAB invokes `subsindex` separately on all the subscripts in an expression, such as `X(A,B)`.

See Also

`subsasgn` | `subsasgn`

Tutorials

•

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 `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

Redefine subscripted reference for objects

Syntax

```
B = subsref(A,S)
```

Description

`B = subsref(A,S)` is called by MATLAB for the syntax `A(i)`, `A{i}`, or `A.i` when `A` is an object. `S` is a `struct` array with two fields, `type` and `subs`.

The `type` field is string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts, `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields. The subs field is a cell array or a string containing the actual subscripts.

`B` is the result of the indexed expression.

MATLAB uses the built-in `subsref` function to interpret indexed references to objects. To modify the indexed reference behavior of objects, overload `subsref` in the class.

If `A` is a fundamental class (see), then an indexed reference to `A` calls the built-in `subsref` function. It does not call a `subsref` method that you have overloaded for that class. Therefore, if `A` is an array of class `double`, and there is an `@double/subsref` method on your MATLAB path, the statement `A(I)` calls the MATLAB built-in `subsref` function.

Examples

See how MATLAB calls `subsref` for the expression:

```
A(1:2,:)
```

The syntax `A(1:2,:)` calls `B = subsref(A,S)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs={1:2,':'}`. The string `':'` indicates a colon used as a subscript.

See how MATLAB calls `subsref` for the expression:

```
A{1:2}
```

The syntax `A{1:2}` calls `B = subsref(A,S)` where `S.type='{}'` and `S.subs={[1 2]}`.

See how MATLAB calls `subsref` for the expression:

```
A.field
```

The syntax `A.field` calls `B = subsref(A,S)` where `S.type='.'` and `S.subs='field'`.

See how MATLAB calls `subsref` for the expression:

```
A(1,2).name(3:5)
```

Simple calls combine in a straightforward way for more complicated indexing expressions. In such cases, `length(S)` is the number of subscript 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 4 5]}`

See Also

`numel` | `subsasgn` | `substruct`

Tutorials

•

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).

Outputs

`S`

`struct` with these fields:

- `type`: one of `'.'`, `'()'`, or `'{}'`
- `subs`: subscript values (field name or cell array of index vectors)

Examples

Call `subsref` with arguments equivalent to the syntax:

```
B = A(3,5).field;
```

where `A` is an object of a class that implements a `subsref` method

Use `substruct` to form the input `struct`, `S`:

```
S = substruct('()',{3,5},'.','field');
```

Call the class method:

```
B = subsref(A,S);
```

The `struct` created by `substruct` in this example contains:

```
S(1)

ans =
```

```
    type: '()'
    subs: {[3]  [5]}

S(2)

ans =

    type: '.'
    subs: 'field'
```

See Also

subsasgn | subsref

Tutorials

•

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 `NaN`s 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.

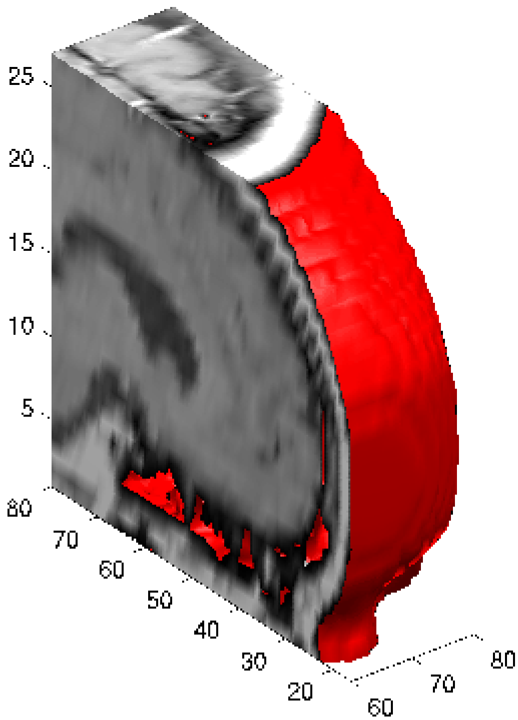
Examples

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`).

- 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 1-106 for related functions

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 or using the`dim` argument.

- Transposing

  ```
  sum(M') =
       15     15     15
  ```

- Using the `dim` argument

  ```
  sum(M,1)

  ans =

      15     15     15
  ```

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 software returns an answer of type `single`. For example,

```
sum(single([2 5 8]))

ans =

     15
```

```
class(ans)

ans =

single
```

Integer Data Types

When you apply `sum` to any of the following integer data types, MATLAB software 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)
```

```
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

Superclass names

Syntax

```
superclasses('ClassName')
superclasses(obj)
s = superclasses(...)
```

Description

`superclasses('ClassName')` displays the names of all visible superclasses of the MATLAB class with the name *`ClassName`*. Visible classes have a `Hidden` attribute value of `false` (the default).

`superclasses(obj)` `obj` is an instance of a MATLAB class. `obj` can be either a scalar object or an array of objects.

`s = superclasses(...)` returns the superclass names in a cell array of strings.

Examples

Get the name of the `hgsetget` class superclass:

```
superclasses('hgsetget')

Superclasses for class hgsetget:

    handle
```

See Also

`properties` | `methods` | `events` | `classdef`

Tutorials

•

superiorto

Purpose

Establish superior class relationship

Syntax

```
superiorto('class1', 'class2', ...)
```

Description

`superiorto('class1', 'class2', ...)` establishes that the class invoking this function in its constructor has higher precedence than the classes in the argument list.

The `superiorto` function establishes a precedence that determines which object method MATLAB calls. Use this function only from a constructor that calls the `class` function to create an object. For classes defined with `classdef` statements, see .

Examples

Show function dispatching:

`a` is an object of class `class_a`, `b` is an object of class `class_b`, and `c` is an object of class `class_c`. The constructor method for `class_c` contains the statement `superiorto('class_a')`. Then, either of the following two statements:

```
e = fun(a,c);
e = fun(c,a);
```

invokes `class_c/fun`.

If you call a function with two objects having an unspecified relationship, MATLAB considers the two objects to have equal precedence. In this case, MATLAB calls the left-most object method. So `fun(b,c)` calls `class_b/fun`, while `fun(c,b)` calls `class_c/fun`.

See Also

`inferiorto`

Purpose Open MathWorks Technical Support Web page

Syntax

```
support
```

Description `support` opens the MathWorks Technical Support Web page, `http://www.mathworks.com/support`, in the MATLAB Web browser.

This Web page contains resources including

- A search engine, including an option for solutions to common problems
- Information about installation and licensing
- A patch archive for bug fixes you can download
- Other useful resources

See Also `doc`, `web`

surf, surfc

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',...)
hsurface = surfc('v6',...)
```

Description

Use `surf` and `surfc` to view mathematical functions over a rectangular region. `surf` and `surfc` create colored parametric surfaces specified by `X`, `Y`, and `Z`, with color specified by `Z` or `C`.

`surf(Z)` creates a a three-dimensional shaded surface from the *z* components in matrix `Z`, using `x = 1:n` and `y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a geometrically rectangular grid. `Z` specifies the color data as well as surface height, so color is proportional to surface height.

`surf(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, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, the vertices of the surface faces are *(X(j), Y(i), Z(i,j))* triples. To create X and Y matrices for arbitrary domains, use the `meshgrid` function.

`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.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

`surf` and `surfc` do not accept complex inputs.

Algorithm

Abstractly, a parametric surface is parameterized by two independent variables, `i` and `j`, which vary continuously over a rectangle; for example, 1 ≤ i ≤ m and 1 ≤ j ≤ n. The three functions x(i,j), y(i,j), and z(i,j) specify the surface. When i and j are integer values, they define a rectangular grid with integer grid points. The functions x(i,j), y(i,j), and z(i,j) become three m-by-n matrices, X, Y, and Z. Surface color is a fourth function, c(i,j), denoted by matrix C.

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

```
         i-1,j
           |
i,j-1 - i,j - i,j+1
           |
         i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, [X(:) Y(:) Z(:)] returns a list of triples specifying points in 3-space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors; the four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

Surface color can be specified in two different ways: at the vertices or at the centers of each patch. In this general setting, the surface need not be a single-valued function of x and y. Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The shading function sets the shading. If the shading is interp, C must be the same size as X, Y, and Z; it specifies the colors at the vertices. The color within a surface patch is a bilinear function of the local coordinates. If the shading is faceted (the default) or flat, C(i,j) specifies the constant color in the surface patch:

```
   (i,j)    -    (i,j+1)
     |   C(i,j)  |
  (i+1,j)  -  (i+1,j+1)
```

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).

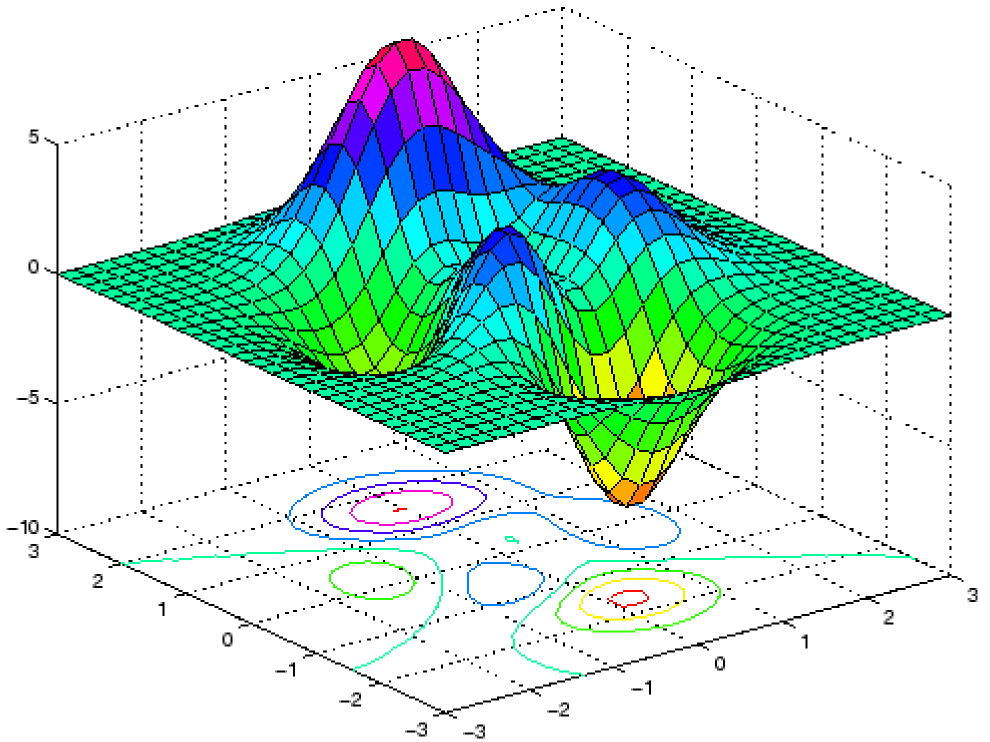
The range of X, Y, and Z or the current setting of the axes XLimMode, YLimMode, and ZLimMode properties (also set by the axis function) determines the axis labels.

The range of C or the current setting of the axes CLim and CLimMode properties (also set by the caxis function) determines the color scaling. The scaled color values are used as indices into the current colormap.

Examples

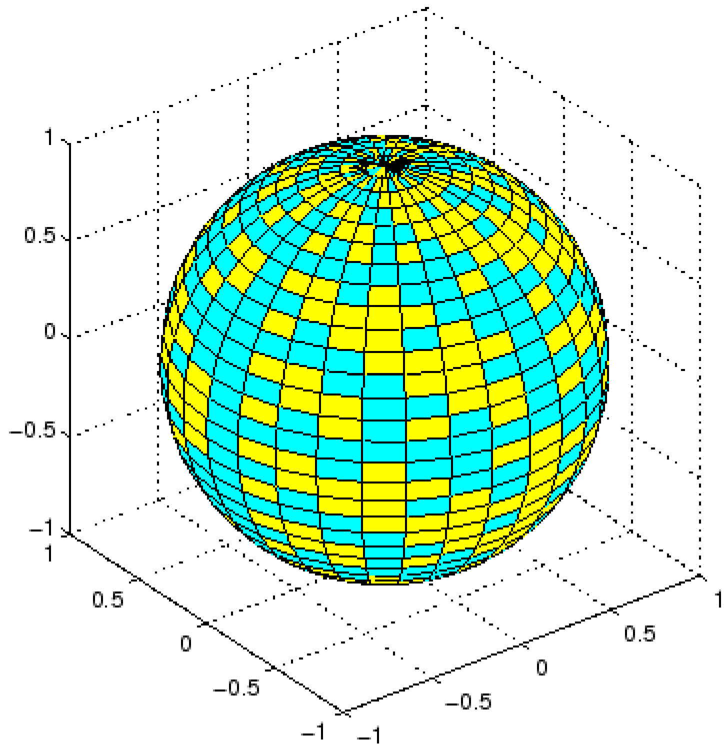
Display a surfaceplot and contour plot of the peaks surface.

```
[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`, `imagesc`, `mesh`, `meshgrid`, `pcolor`, `shading`, `trisurf`, `view`

Properties for surfaceplot graphics objects

"Surface and Mesh Creation" on page 1-102 for related functions

in the Getting Started with MATLAB documentation for background and examples.

Representing a Matrix as a Surface in the MATLAB 3-D Visualization documentation for further 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 1-106 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 `length(x) = n` and `length(y) = m` where `[m,n] = size(Z)`. In this case, the vertices of the surface facets are the triples `(x(j),y(i),Z(i,j))`. Note that `x` corresponds to the columns of `Z` and `y` corresponds to the rows of `Z`. For a complete discussion of parametric surfaces, see the `surf` function.

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

`h = surface(...)` returns a handle to the created surface object.

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

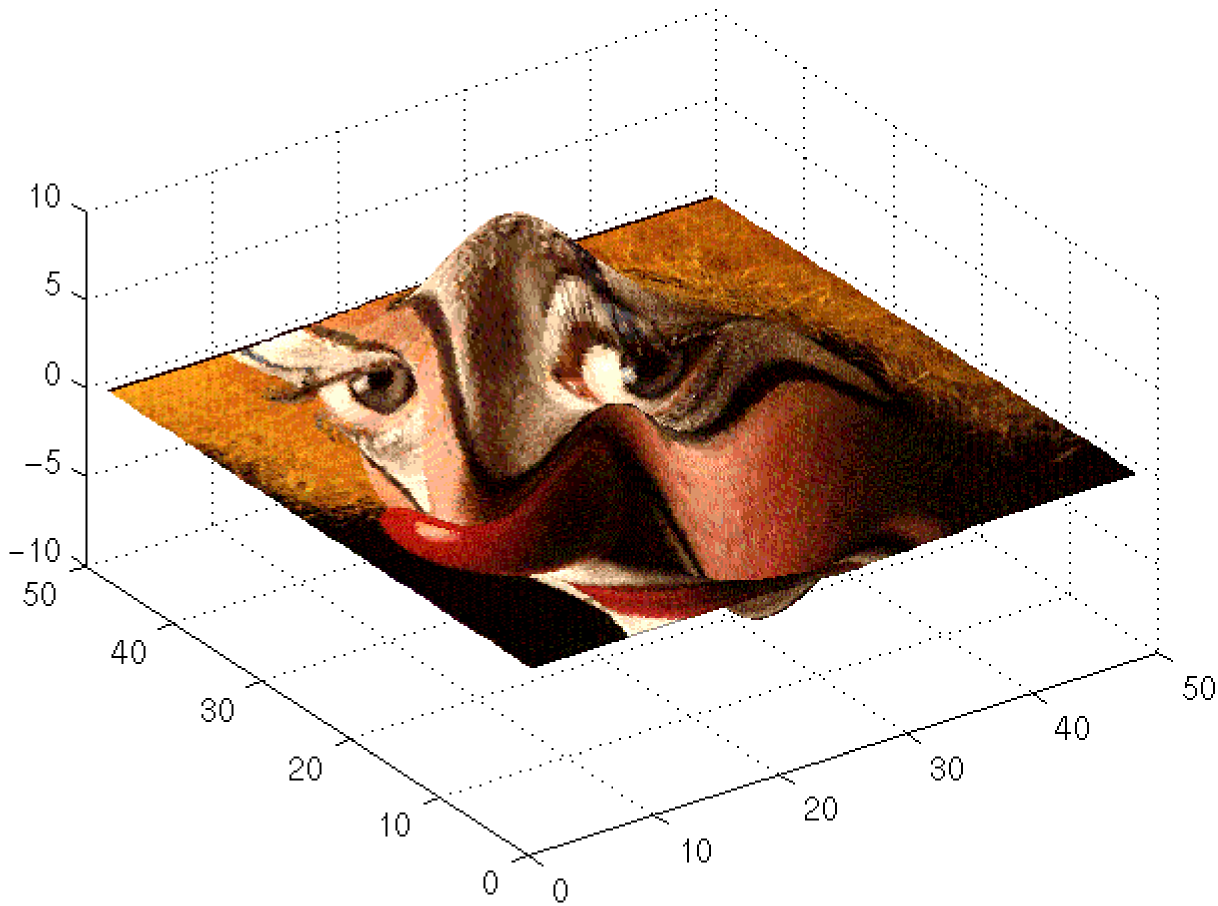
```
surface('XData',[1:size(Z,2)],...
    'YData',[1:size(Z,1)],...
    'ZData',Z,...
    'CData',Z)
```

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 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`.

Setting Default Properties

You can set default surface properties on the axes, figure, and 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

"Surface and Mesh Creation" on page 1-102 and "Object Creation" on page 1-99 for related functions

`Surface Properties` for property descriptions

Purpose

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 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 software determines the transparency in one of three ways:

- Using the elements of `AlphaData` as transparency values (`AlphaDataMapping` set to `none`)
- Using the elements of `AlphaData` as indices into the current alphamap (`AlphaDataMapping` set to `direct`)
- Scaling the elements of `AlphaData` to range between the minimum and maximum values of the axes `ALim` property (`AlphaDataMapping` set to `scaled`, the default)

`AlphaDataMapping`

`none | direct | {scaled}`

Transparency mapping method. This property determines how MATLAB interprets indexed alpha data. 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.

`Annotation`

`hg.Annotation` object Read Only

Control the display of surface objects in legends. The `Annotation` property enables you to specify whether this surface object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the surface object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this surface object in a legend (default)
`off`	Do not include this surface object in a legend
`children`	Same as `on` because surface objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the `IconDisplayStyle` property

See for more information and examples.

`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 for an example.

`BeingDeleted`

`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

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

`BusyAction`

`cancel | {queue}`

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

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

`ButtonDownFcn`

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

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is over the 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. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property). For example, the following function takes different action depending on what type of selection was made:

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   sel_typ = get(gcbf,'SelectionType')
   switch sel_typ
      case 'normal'
         disp('User clicked left-mouse button')
         set(src,'Selected','on')
      case 'extend'
         disp('User did a shift-click')
         set(src,'Selected','on')
```

```
        case 'alt'
           disp('User did a control-click')
           set(src,'Selected','on')
           set(src,'SelectionHighlight','off')
     end
  end
```

Suppose `h` is the handle of a surface object and that the `button_down` function is on your MATLAB path. The following statement assigns the function above to the `ButtonDownFcn`:

```
set(h,'ButtonDownFcn',@button_down)
```

See 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.

`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`

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

Callback function executed during object creation. This property defines a callback function 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 passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

`DeleteFcn`

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

Delete surface callback function. A callback function that executes when you delete the surface object (e.g., when you issue a `delete` command or clear the axes `cla` or figure `clf`). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
```

```
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See 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.

`DisplayName`
string (default is empty string)

String used by legend for this surface object. The `legend` function uses the string defined by the `DisplayName` property to label this surface object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this surface object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See for more examples.

`EdgeAlpha`

`{scalar = 1} | flat | interp`

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

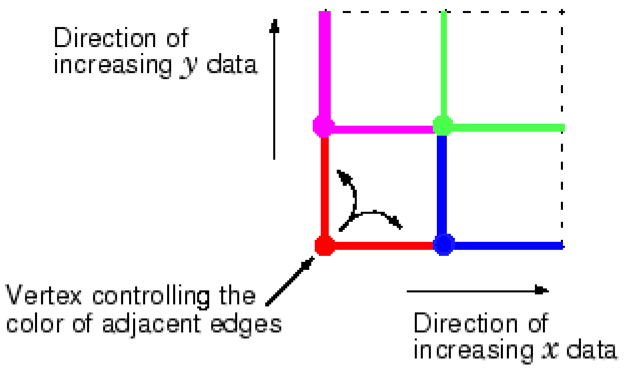
- `scalar` — A single non-Nan scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) means fully opaque and `0` means completely transparent.
- `flat` — The alpha data (`AlphaData`) value for the first vertex of the face determines the transparency of the edges.
- `interp` — Linear interpolation of the alpha data (`AlphaData`) values at each vertex determines the transparency of the edge.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

`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 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

- `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 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

Marker Specifier	Description
.	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 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 for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When this property is `on`, MATLAB displays a dashed bounding box around the surface if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing a dashed bounding box around the surface. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`SpecularColorReflectance`
scalar in the range 0 to 1

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

`SpecularExponent`
scalar >= 1

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

`SpecularStrength`
scalar >= 0 and <= 1

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

You can also set the intensity of the ambient and diffuse components of the light on the surface object. See the `AmbientStrength` and `DiffuseStrength` properties. Also see the `material` function.

`Tag`
string

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

`Type`
string (read only)

Class of the graphics object. The class of the graphics object. For surface objects, `Type` is always the string `'surface'`.

`UIContextMenu`
handle of a uicontextmenu object

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

`UserData`
matrix

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

`VertexNormals`
vector or matrix

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

`Visible`
`{on} | off`

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

`XData`
vector or matrix

X-coordinates. The *x*-position of the surface points. If you specify a row vector, `surface` replicates the row internally until it has the same number of columns as `ZData`.

`YData`
vector or matrix

Y-coordinates. The *y*-position of the surface points. If you specify a row vector, `surface` replicates the row internally until it has the same number of rows as `ZData`.

`ZData`
matrix

Z-coordinates. The *z*-position of the surfaceplot data points. See the Description section for more information.

Surfaceplot Properties

Purpose

Define 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 software determines the transparency in one of three ways:

- Using the elements of `AlphaData` as transparency values (`AlphaDataMapping` set to `none`)
- Using the elements of `AlphaData` as indices into the current alphamap (`AlphaDataMapping` set to `direct`)
- Scaling the elements of `AlphaData` to range between the minimum and maximum values of the axes `ALim` property (`AlphaDataMapping` set to `scaled`, the default)

`AlphaDataMapping`

`{none} | direct| scaled`

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

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

`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.

`Annotation`

`hg.Annotation` object Read Only

Control the display of surfaceplot objects in legends. The Annotation property enables you to specify whether this surfaceplot object is represented in a figure legend.

Querying the Annotation property returns the handle of an hg.Annotation object. The hg.Annotation object has a property called LegendInformation, which contains an hg.LegendEntry object.

Once you have obtained the hg.LegendEntry object, you can set its IconDisplayStyle property to control whether the surfaceplot object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
on	Represent this surfaceplot object in a legend (default)
off	Do not include this surfaceplot object in a legend
children	Same as on because surfaceplot objects do not have children

Setting the IconDisplayStyle property

These commands set the IconDisplayStyle of a graphics object with handle hobj to off:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the IconDisplayStyle property

See for more information and examples.

`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 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`

`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.

`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)

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 an object. You must specify the callback during the creation of the object. For example,

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

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

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

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

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

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

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

See the `BeingDeleted` property for related information.

`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 object. See the `AmbientStrength` and `SpecularStrength` properties.

`DisplayName`
string (default is empty string)

String used by legend for this surfaceplot object. The `legend` function uses the string defined by the `DisplayName` property to label this surfaceplot object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this surfaceplot object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See for more examples.

`EdgeAlpha`
`{scalar = 1} | flat | interp`

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

- `scalar` — A single non-Nan scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) means fully opaque and `0` means completely transparent.
- `flat` — The alpha data (`AlphaData`) value for the first vertex of the face determines the transparency of the edges.

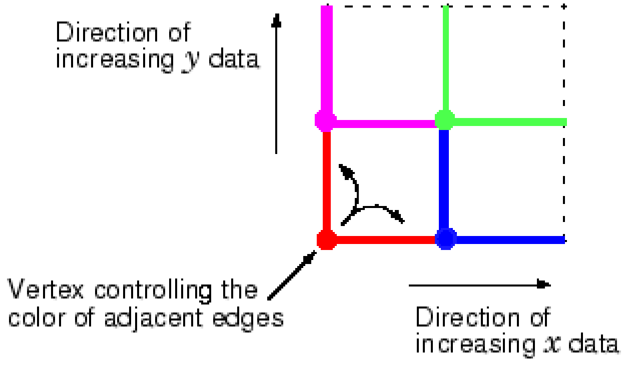
- `interp` — Linear interpolation of the alpha data (`AlphaData`) values at each vertex determines the transparency of the edge.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

`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.
- `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 objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing

with EraseMode none, you cannot print these objects because MATLAB stores no information about their former locations.

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

Printing with Nonnormal Erase Modes

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

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

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

FaceAlpha

{scalar = 1} | flat | interp | texturemap

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

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

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `FaceAlpha`.

`FaceColor`

`ColorSpec | 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 for `surface`.)

`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. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to

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

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

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

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

`HitTest`
: `{on} | off`

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

`Interruptible`
: `{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

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

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

Specifier String	Line Style
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line
none	No line

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

LineWidth
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default LineWidth is 0.5 points.

Marker
character (see table)

Marker symbol. The Marker property specifies the type of markers that are displayed at plot vertices. You can set values for the Marker property independently from the LineStyle property. Supported markers include those shown in the following table.

Marker Specifier	Description
+	Plus sign
o	Circle

Marker Specifier	Description
`*`	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 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.

`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 parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See for more information on parenting graphics objects.

`Selected`
: `on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
: `{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`SpecularColorReflectance`
: scalar in the range 0 to 1

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

`SpecularExponent`

scalar >= 1

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

`SpecularStrength`

scalar >= 0 and <= 1

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

You can also set the intensity of the ambient and diffuse components of the light on the surfaceplot object. See the `AmbientStrength` and `DiffuseStrength` properties. Also see the `material` function.

`Tag`

string

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

For example, you might create an areaseries object and set the `Tag` property.

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

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

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

`Type`
string (read only)

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

`UserData`
array

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

`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`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
: vector or matrix

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` (by setting the `XData` property or specifying the `x` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`
: string (MATLAB variable)

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

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

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

See the refreshdata reference page for more information.

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

YData
vector or matrix

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 `ZData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`.

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

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

See the `refreshdata` reference page for more information.

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

Purpose

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

```
surfl(Z)
surfl(...,'light')
surfl(...,s)
surfl(X,Y,Z,s,k)
h = surfl(...)
```

Description

The `surfl` function displays a shaded surface based on a combination of ambient, diffuse, and specular lighting models.

`surfl(Z)` and surfl(X,Y,Z) create three-dimensional shaded surfaces using the default direction for the light source and the default lighting coefficients for the shading model. `X`, `Y`, and `Z` are vectors or matrices that define the x, y, and z components of a surface.

`surfl(...,'light')` produces a colored, lighted surface using a MATLAB light object. This produces results different from the default lighting method, `surfl(...,'cdata')`, which changes the color data for the surface to be the reflectance of the surface.

`surfl(...,s)` specifies the direction of the light source. `s` is a two- or three-element vector that specifies the direction from a surface to a light source. `s = [sx sy sz]` or `s = [azimuth elevation]`. The default `s` is 45° counterclockwise from the current view direction.

`surfl(X,Y,Z,s,k)` specifies the reflectance constant. `k` is a four-element vector defining the relative contributions of ambient light, diffuse reflection, specular reflection, and the specular shine coefficient. k = [ka kd ks shine] and defaults to [.55,.6,.4,10].

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

Remarks

surfl does not accept complex inputs.

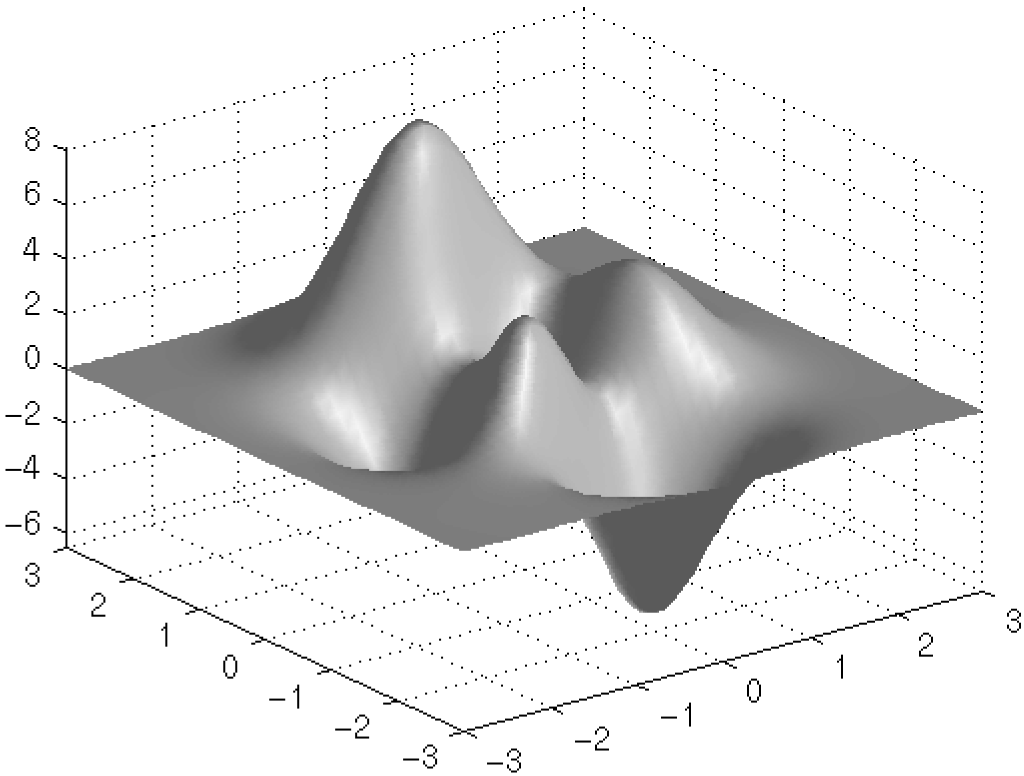
For smoother color transitions, use colormaps that have linear intensity variations (e.g., gray, copper, bone, pink).

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 surfl(X',Y',Z'). Because of the way surface normal vectors are computed, surfl requires matrices that are at least 3-by-3.

Examples

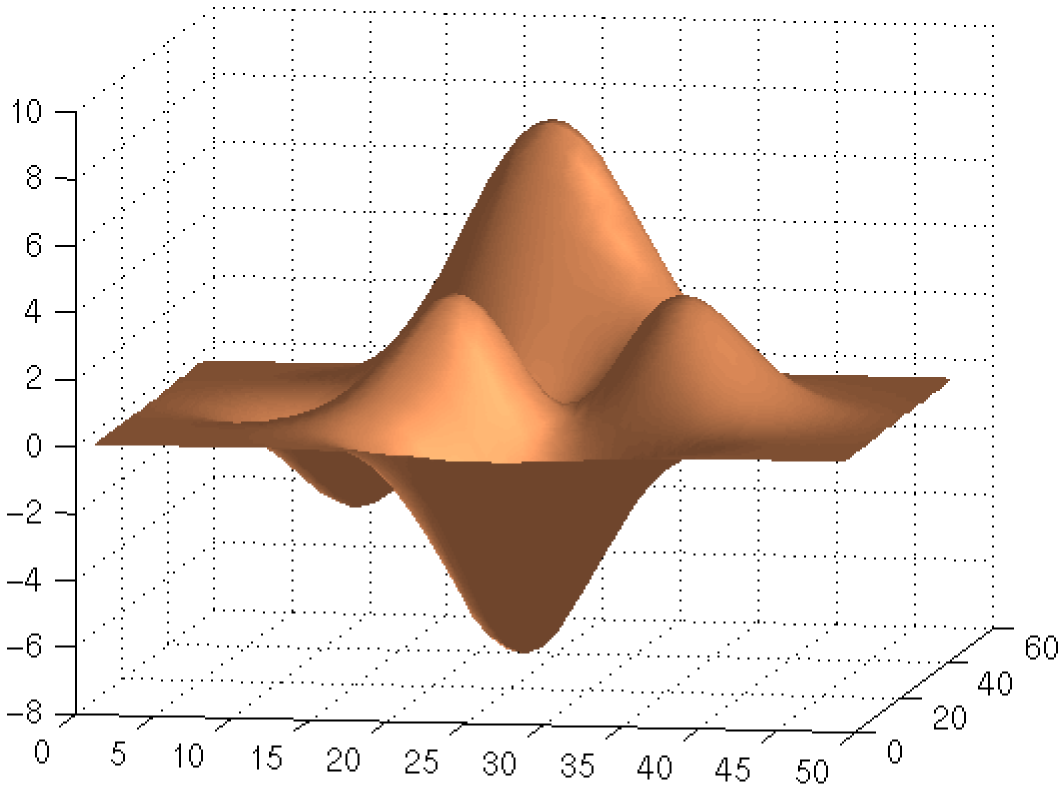
View peaks using colormap-based lighting.

```
[x,y] = meshgrid(-3:1/8:3);
z = peaks(x,y);
surfl(x,y,z);
shading interp
colormap(gray);
axis([-3  3  -3  3  -8  8])
```



To plot a lighted surface from a view direction other than the default,

```
view([10 10])
grid on
hold on
surfl(peaks)
shading interp
colormap copper
hold off
```



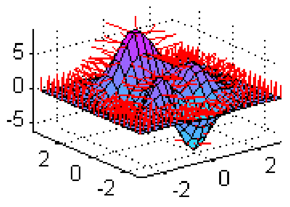
See Also colormap, shading, light

"Surface and Mesh Creation" on page 1-102 for functions related to surfaces

"Lighting" on page 1-106 for functions related to lighting

Purpose

Compute and display 3-D surface normals



Syntax

```
surfnorm(Z)
surfnorm(X,Y,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` does not accept complex inputs.

`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. Reverse the direction of the normals by calling `surfnorm` with transposed arguments:

```
surfnorm(X',Y',Z')
```

`[Nx,Ny,Nz] = surfnorm(...)` returns the components of the three-dimensional surface normals for the surface.

`surfl` uses `surfnorm` to compute surface normals when calculating the reflectance of a surface.

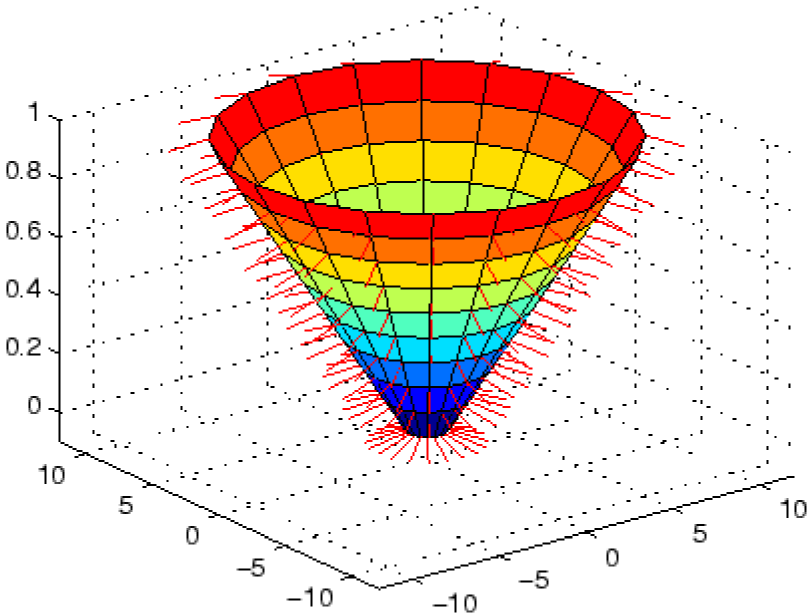
Algorithm

The surface normals are based on a bicubic fit of the data in `X`, `Y`, and `Z`. For each vertex, diagonal vectors are computed and crossed to form the normal.

Examples

Plot the normal vectors for a truncated cone.

```
[x,y,z] = cylinder(1:10);
surfnorm(x,y,z)
axis([-12 12 -12 12 -0.1 1])
```



See Also

surf, quiver3

“Color Operations” on page 1-103 for related functions

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 >= 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] = 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
`options.tol`	Convergence tolerance: `norm(AV-US,1)<=tol*norm(A,1)`	`1e-10`
`options.maxit`	Maximum number of iterations	`300`
`options.disp`	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 `norn(A*V-U*S,1) <= tol*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]`.

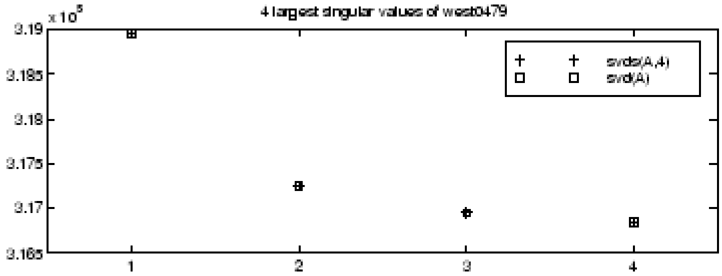
`svds(A,k,0)` uses `eigs` to find the `2k` smallest magnitude eigenvalues and corresponding eigenvectors of `B = [0 A; A' 0]`, and then selects the `k` positive eigenvalues and their eigenvectors.

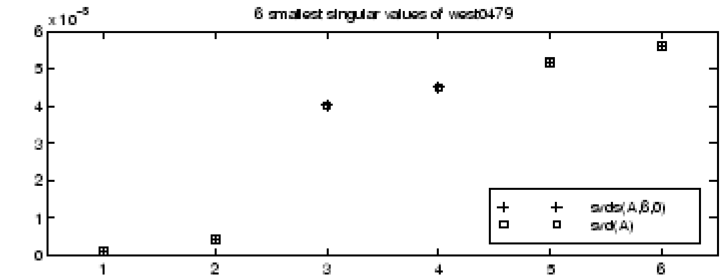
Example

`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))
sl = 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`

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

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 `case`s 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`

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.

stats(1)	Number of dense or empty rows ignored by symamd
stats(2)	Number of dense or empty columns ignored by symamd
stats(3)	Number of garbage collections performed on the internal data structure used by symamd (roughly of size 8.4*nnz(tril(S,-1)) + 9n integers)
stats(4)	0 if the matrix is valid, or 1 if invalid
stats(5)	Rightmost column index that is unsorted or contains duplicate entries, or 0 if no such column exists
stats(6)	Last seen duplicate or out-of-order row index in the column index given by stats(5), or 0 if no such row index exists
stats(7)	Number of duplicate and out-of-order row indices

Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `symamd`. For this reason, `symamd` verifies that `S` is valid:

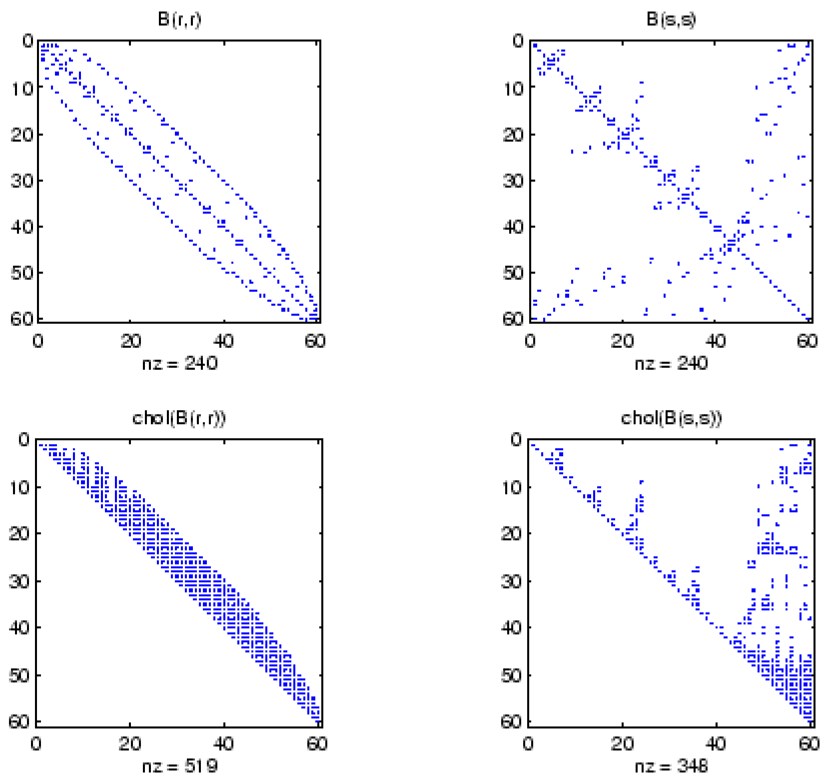
- If a row index appears two or more times in the same column, `symamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `symamd` sorts each column of its internal copy of the matrix `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.

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`, `amd`

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/`

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 of `R=chol(A'*A)`. `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)` (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.

The flop count for a subsequent Cholesky factorization is `sum(count.^2)`

Return Value	Description
`h`	Height of the elimination tree
`parent`	The elimination tree itself
`post`	Postordering of the elimination tree
`R`	0-1 matrix having the structure of `chol(A)` for the symmetric case, `chol(A'*A)` for the `'col'` case, or `chol(A*A')` for the `'row'` case.

`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.

`[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.

See Also

`chol`, `etree`, `treelayout`

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 in the MATLAB Programming documentation for more information.

, 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 `norm(b-A*x)/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 `inv(sqrt(M))*A*inv(sqrt(M))*y = inv(sqrt(M))*b` for `y` and then return `x = in(sqrt(M))*y`. If `M` is `[]` then `symmlq` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\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	`symmlq` converged to the desired tolerance `tol` within `maxit` iterations.
1	`symmlq` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`symmlq` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `symmlq` 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 `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = symmlq(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = symmlq(A,b,...)` also returns a vector of estimates of the `symmlq` residual norms at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,resveccg] = 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 software 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 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 preordering for LU or Cholesky factorization of matrices that come from long, skinny problems. The ordering works for both symmetric and nonsymmetric `S`.

For a real, symmetric sparse matrix, `S`, the eigenvalues of `S(r,r)` are the same as those of `S`, but `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

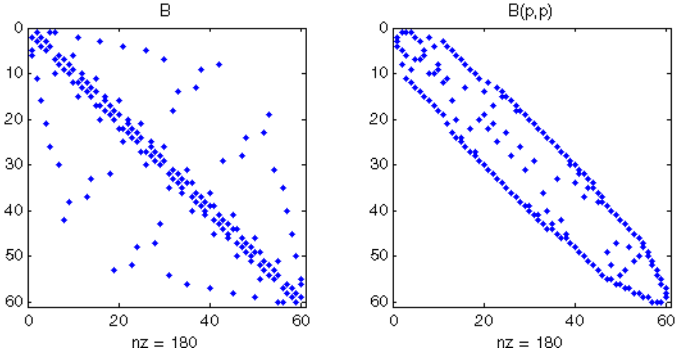
```
subplot(1,2,1), spy(B), title('B')
```

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.

- '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 2-3678.

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 2-3678.

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`

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[18] shell script, and directs the output to the MATLAB software. 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 a Microsoft Windows platform with a `command` that relies on the current folder fails when the current folder is specified using a UNC pathname because DOS does not support UNC pathnames. When this happens, MATLAB returns the 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 folder to a mapped drive prior to running `system` or a function that calls `system`.

Examples On a Windows system, display the current folder by accessing the operating system.

```
[status currdir] = system('cd')
```

18. UNIX is a registered trademark of The Open Group in the United States and other countries.

```
status =
    0
currdir =
    D:\work\matlab\test
```

See Also

`!` (exclamation point), `computer`, `dos`, `perl`, `unix`, `winopen`

in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Tangent of argument in radians

Syntax

```
Y = tan(X)
```

Description

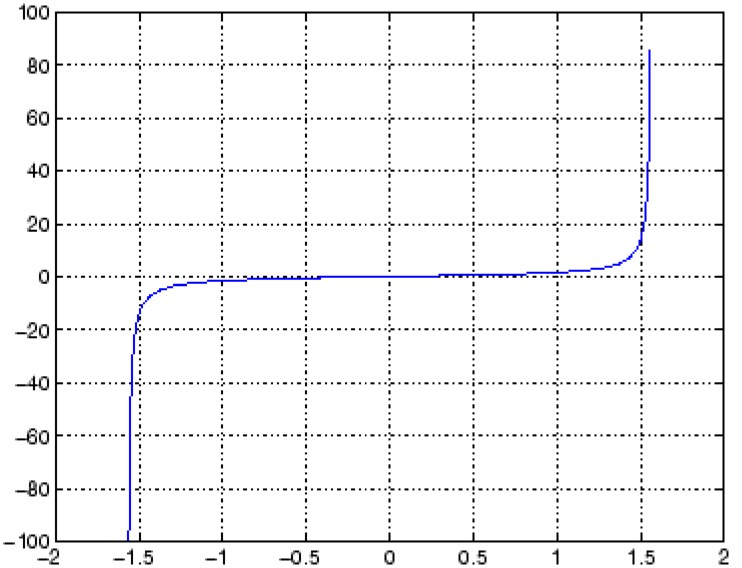
The `tan` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`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 = tand(X)
```

Description `Y = tand(X)` is the tangent of the elements of `X`, expressed in degrees. For odd integers `n`, `tand(n*90)` is infinite, whereas `tan(n*pi/2)` is large but finite, reflecting the accuracy of the floating point value of `pi`.

See Also `tan`, `tanh`, `atan`, `atan2`, `atand`, `atanh`

Purpose

Hyperbolic tangent

Syntax

```
Y = tanh(X)
```

Description

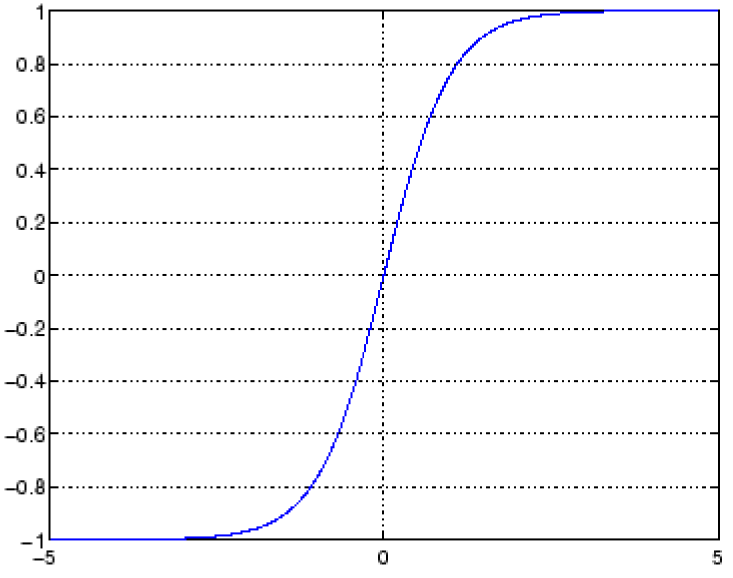
The `tanh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = tanh(X)` returns the hyperbolic tangent of each element of `X`.

Examples

Graph the hyperbolic tangent function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
plot(x,tanh(x)), grid on
```



Definition

The hyperbolic tangent can be defined as

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$$

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,rootfolder)
entrynames = tar(...)
```

Description

`tar(`*`tarfilename`*`,`*`files`*`)` creates a tar file named *`tarfilename`* from the list of files and folders specified in *`files`*. Folders recursively include all of their content. If *`files`* includes relative paths, the tar file also contains relative paths. The tar file does not include absolute paths.

`tarfilename` is a string specifying the name of the tar file. If *`tarfilename`* has no extension, MATLAB appends the `.tar` extension. 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 folders included in *`tarfilename`*.

Individual files that are on the MATLAB path can be specified as partial path names. Otherwise an individual file can be specified relative to the current folder or with an absolute path.

Folders must be specified relative to the current folder or with absolute paths. On UNIX systems, folders can also start with `~/` or `~`*`username`*`/`, which expands to the current user's home folder or the specified user's home folder, respectively. The wildcard character `*` can be used when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

`tar(`*`tarfilename`*`,`*`files`*`,`*`rootfolder`*`)` specifies the path for *`files`* relative to *`rootfolder`* rather than the current folder. Relative paths in the tar file reflect the relative paths in *`files`*, and do not include path information from *`rootfolder`*.

`entrynames`` = tar(...)` returns a string cell array of the names of the files contained in *`tarfilename`*. If *`files`* includes relative paths, *`entrynames`* also contains relative paths.

Example Tar all files in the current folder to the file `backup.tgz`.

```
tar('backup.tgz','.');
```

See Also `gzip`, `gunzip`, `untar`, `unzip`, `zip`

Purpose
Name of system's temporary folder

Syntax

```
tmp_folder = tempdir
```

Description
`tmp_folder = tempdir` returns the name of the system's temporary folder, if one exists. This function does not create a new folder.

See Also
`delete`, `recycle`, `tempname`

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 Also `tempdir`

Purpose

Tetrahedron mesh plot

Syntax

```
tetramesh(T,X,c)
tetramesh(T,X)
tetramesh(TR)
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 a Delaunay triangulation of a 3-D set of points. 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.

`tetramesh(T,X)` uses `C = 1:m` as the color for the `m` tetrahedra. Each tetrahedron has a different color (modulo the number of colors available in the current colormap).

`tetramesh(TR)` displays the tetrahedra in a Triangulation representation.

`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'`.

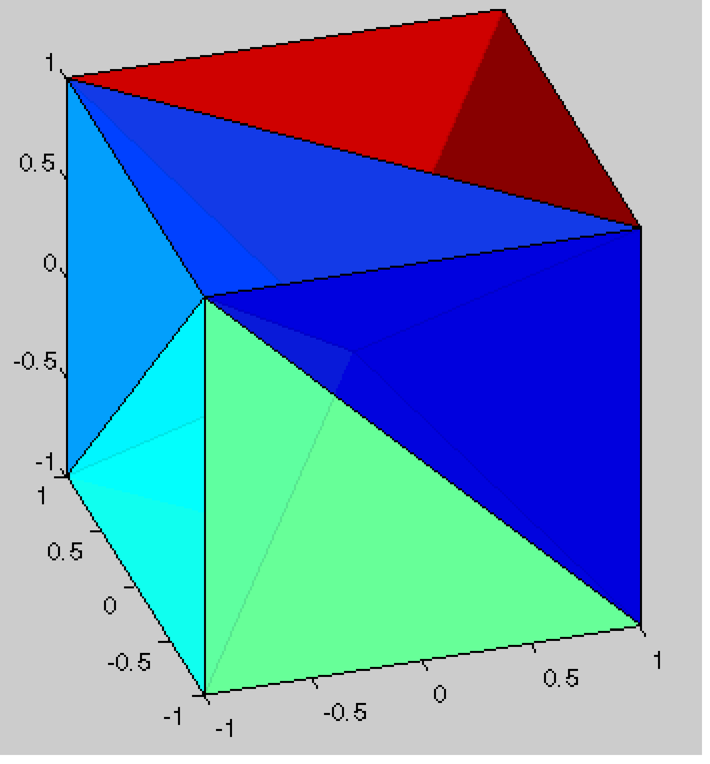
`tetramesh(...,'param','value','param','value'...)` allows additional patch property name/property value pairs to be used when displaying the tetrahedrons. For example, the default transparency parameter is set to `0.9`. You can overwrite this value by using the property name/property value pair `('FaceAlpha',value)` where `value` is a number between `0` and `1`. See `Patch Properties` for information about the available properties.

Examples

Generate a 3-D 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.
dt = DelaunayTri(x,y,z);
Tes = dt(:,:);
X = [x(:) y(:) z(:)];
tetramesh(Tes,X);
camorbit(20,0)
```



You can also plot the Delaunay triangulation directly.

```
close(gcf);
tetramesh(dt);
```

See Also

trimesh, trisurf, patch, delaunayn, TriRep, TriRep.freeBoundary

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 (~~~).

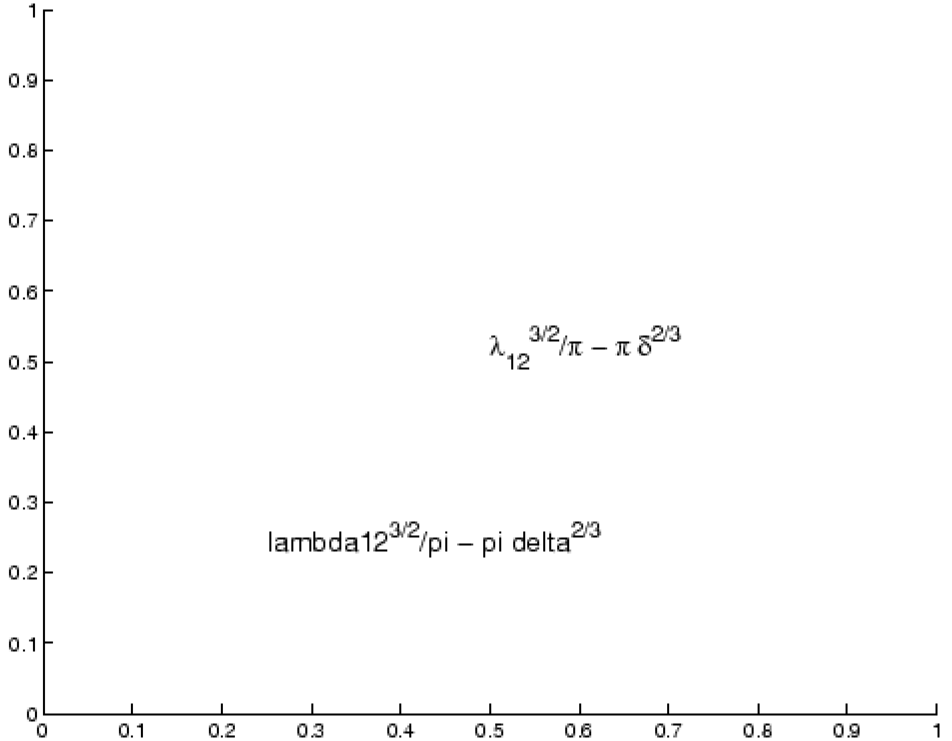
Examples

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'))
```



See Also

`text`, `title`, `xlabel`, `ylabel`, `zlabel`, the text `String` property

"Annotating Plots" on page 1-92 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)` `x` and `y` must be numbers of class `double`.

`text(x,y,z,'string')` adds the string in 3-D coordinates. `x`, `y` and `z` must be numbers of class `double`.

`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 "Examples" on page 2-3698. You can use other units to position the text by setting the text `Units` property to `normalized` or one of the nonrelative units (`pixels`, `inches`, `centimeters`, `points`, or `characters`).

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

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).

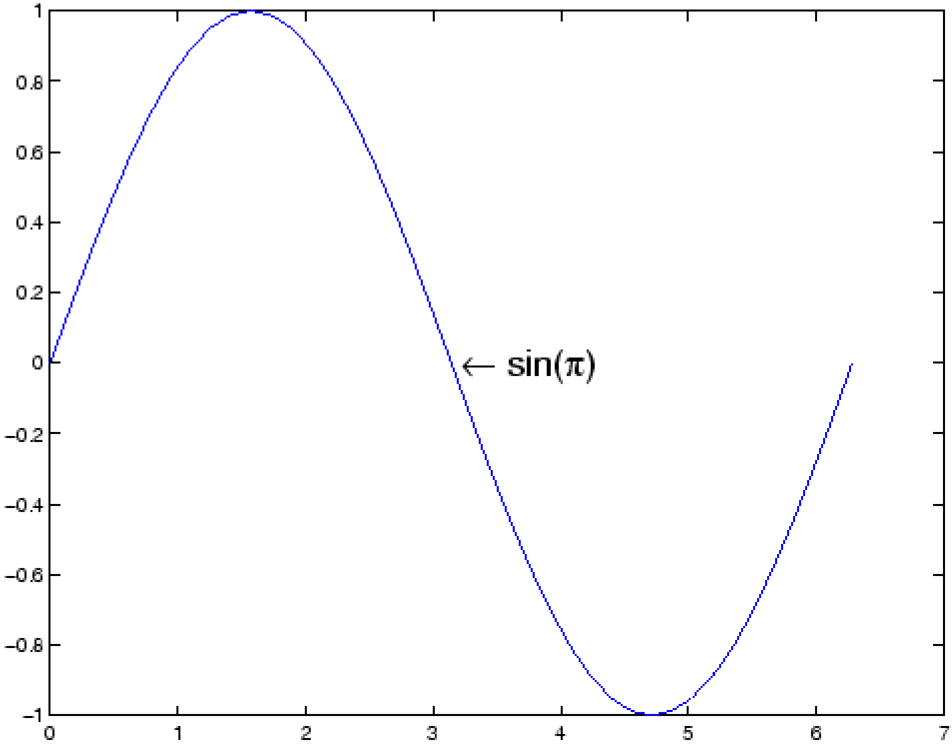
`text` does not respect the setting of the figure or axes `NextPlot` property. This allows you to add text objects to an existing axes without setting `hold` to `on`.

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(π)`



The statement

```
text(x,y,'\ite^{i\omega\tau} = cos(\omega\tau) + i sin(\omega\tau)')
```

uses embedded TeX sequences to produce

$$e^{i\omega\tau} = \cos(\omega\tau) + i \sin(\omega\tau)$$

Setting Default Properties

You can set default text properties on the axes, figure, and 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, strings, title, xlabel, ylabel, zlabel

"Object Creation" on page 1-99 for related functions

Text Properties for property descriptions

Purpose Text properties

Modifying Properties

You can set and query graphics object properties using the property editor or the `set` and `get` commands.

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

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

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

Text Property Descriptions

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

`Annotation`

`hg.Annotation` object Read Only

Control the display of text objects in legends. The `Annotation` property enables you to specify whether this text object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the text object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this text object in a legend (default)
`off`	Do not include this text object in a legend
`children`	Same as `on` because text objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

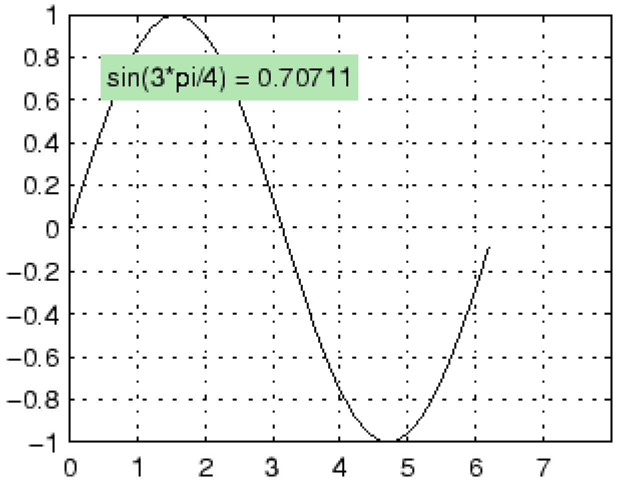
Using the `IconDisplayStyle` property

See for more information and examples.

`BackgroundColor`
`ColorSpec | {none}`

Color of text extent rectangle. This property enables you to define a color for the rectangle that encloses the text `Extent` plus the text `Margin`. 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. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

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

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

`BusyAction`

`cancel | {queue}`

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

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

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is over the 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. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property). For example, the following function takes different action depending on what type of selection was made:

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   sel_typ = get(gcbf,'SelectionType')
   switch sel_typ
      case 'normal'
         disp('User clicked left-mouse button')
         set(src,'Selected','on')
      case 'extend'
         disp('User did a shift-click')
         set(src,'Selected','on')
      case 'alt'
         disp('User did a control-click')
         set(src,'Selected','on')
         set(src,'SelectionHighlight','off')
   end
end
```

Suppose `h` is the handle of a text object and that the `button_down` function is on your MATLAB path. The following statement assigns the function above to the `ButtonDownFcn`:

```
set(h,'ButtonDownFcn',@button_down)
```

See 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 is black. See `ColorSpec` for more information on specifying color.

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

Callback function executed during object creation. A callback function that executes when MATLAB creates a 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',@text_create)
```

defines a default value on the root level that sets the figure Pointer property to crosshairs whenever you create a text object. The callback function must be on your MATLAB path when you execute the above statement.

```
function text_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
 set(gcbf,'Pointer','crosshair')
end
```

MATLAB executes this function after setting all text properties. Setting this property on an existing text object has no effect. The function must define at least two input arguments (handle of object created and an event structure, which is empty for this property).

The handle of the object whose CreateFcn is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root CallbackObject property, which you can query using gcbo.

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

DeleteFcn

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

Delete text callback function. A callback function that executes when you delete the text object (e.g., when you issue a delete command or clear the axes cla or figure clf). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
```

```
    obj_tp = get(src,'Type');
    disp([obj_tp, ' object deleted'])
    disp('Its user data is:')
    disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

`DisplayName`

string (default is empty string)

String used by legend for this text object. The `legend` function uses the string defined by the `DisplayName` property to label this text object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this text object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.

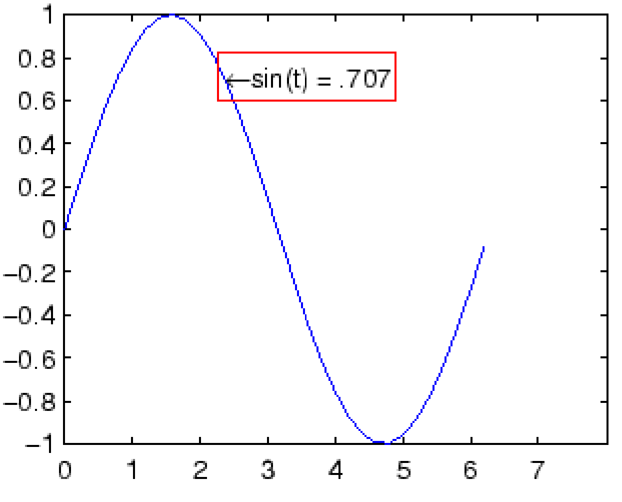
- If you specify a string for the DisplayName property and create the legend using the figure toolbar, then MATLAB uses the string defined by DisplayName.
- To add programmatically a legend that uses the DisplayName string, call legend with the toggle or show option.

See for more examples.

EdgeColor
ColorSpec | {none}

Color of edge drawn around text extent rectangle plus margin. This property enables you to specify the color of a box drawn around the text Extent plus the text Margin. 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',...
'EdgeColor','red');
```



For additional features, see the following properties:

- BackgroundColor — Color of the rectangle's interior (none by default)
- LineStyle — Style of the rectangle's edge line (first set EdgeColor)
- LineWidth — Width of the rectangle's edge line (first set EdgeColor)
- Margin — Increases the size of the rectangle by adding a margin to the area defined by the text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the EdgeColor rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the EdgeColor property and the area defined by the BackgroundColor change.

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,

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`

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 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.

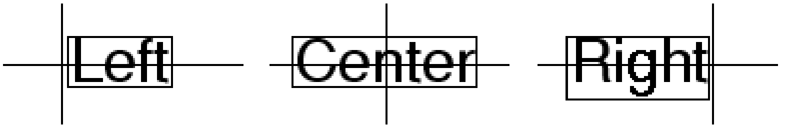
```
load earth
image(X,'ButtonDownFcn','bd_function'); colormap(map)
```

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.

`HorizontalAlignment` viewed with the `VerticalAlignment` set to middle (the default).



See the `Extent` property for related information.

`Interpreter`

`latex | {tex} | none`

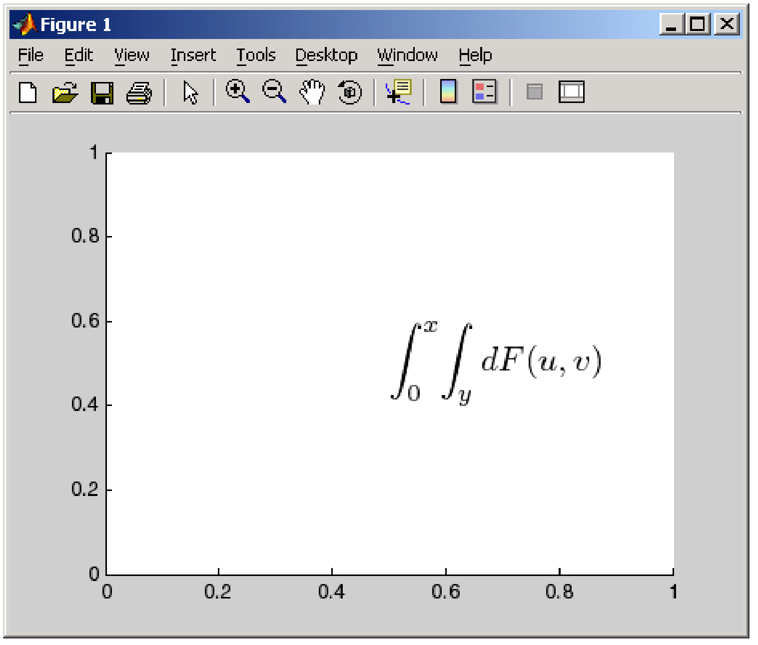
Interpret TEX instructions. This property controls whether MATLAB interprets certain characters in the `String` property as TEX instructions (default) or displays all characters literally. The options are:

- `latex` — Supports a basic subset of the LATEX markup language.
- `tex` — Supports a subset of plain TEX markup language. See the `String` property for a list of supported TEX instructions.
- `none` — Displays literal characters.

Latex Interpreter

To enable the LaTEX 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 TEX.

- Donald E. Knuth, *The TEXbook*, Addison Wesley, 1986.
- The TEX Users Group home page: http://www.tug.org

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether a text callback routine can be interrupted by subsequently invoked callback routines. Text objects have three properties that define callback routines: `ButtonDownFcn`, `CreateFcn`, and `DeleteFcn`. See the `BusyAction` property for information on how MATLAB executes callback routines.

`LineStyle`

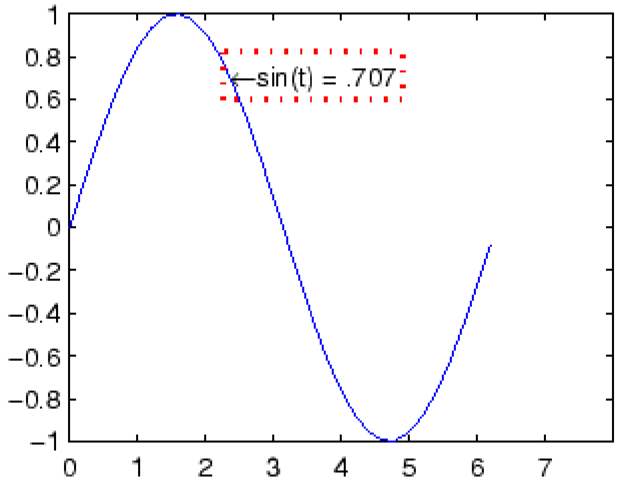
`{-} | -- | : | -. | none`

Edge line type. This property determines the line style used to draw the edges of the text `Extent`. The available line styles are shown in the following table.

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),...
      '\leftarrowsin(t) = .707',...
      'EdgeColor','red',...
      'LineWidth',2,...
      'LineStyle',':');
```



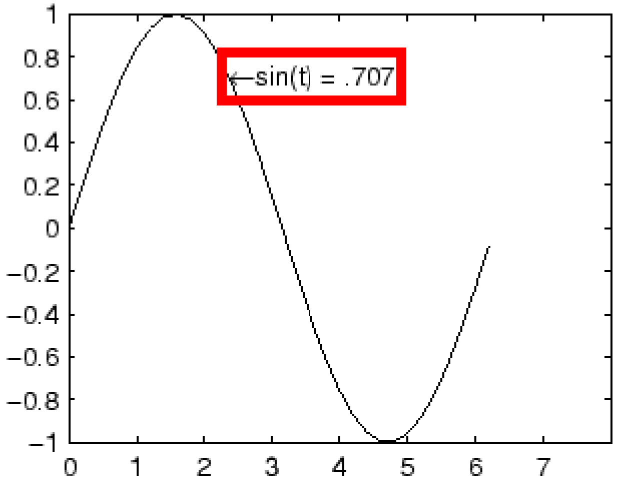
For additional features, see the following properties:

- `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. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

`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),...
'\leftarrowsin(t) = .707',...
'EdgeColor','red',...
'LineWidth',3);
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (`none` by default)
- `EdgeColor` — Color of the rectangle's edge (`none` by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increases the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background

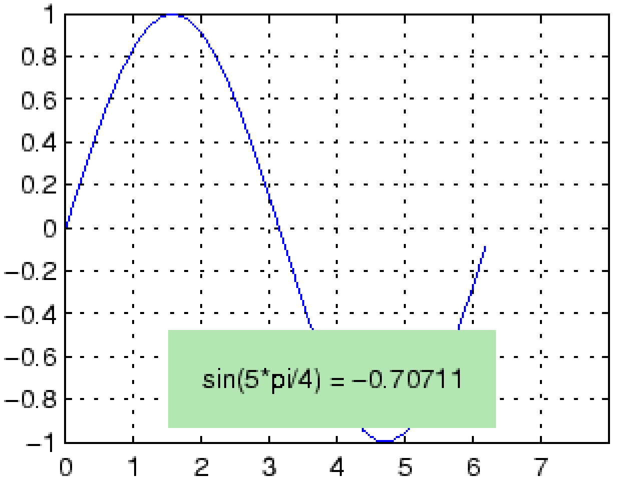
area that is enclosed by the EdgeColor rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the EdgeColor property and the area defined by the BackgroundColor change.

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`)

See how margin affects text extent properties

This example enables you to change the values of the `Margin` property and observe the effects on the `BackgroundColor` area and the `EdgeColor` rectangle.

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

Click to run example — Use your scroll wheel to vary the `Margin`.

`Parent`

handle of axes, hggroup, or hgtransform

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

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

`Position`

`[x,y,[z]]`

Location of text. A two- or three-element vector, `[x y [z]]`, that specifies the location of the text in three dimensions. If you omit the z value, it defaults to 0. All measurements are in units specified by the Units property. Initial value is [0 0 0].

Rotation
scalar (default = 0)

Text orientation. This property determines the orientation of the text string. Specify values of rotation in degrees (positive angles cause counterclockwise rotation).

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.

Note The words `default`, `factory`, and `remove` are reserved words that will not appear in a figure when quoted as a normal string. In order to display any of these words individually, type `'\reserved_word'` instead of `'reserved_word'`.

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
\alpha	α	\upsilon	υ	\sim	~
\beta	β	\phi	Φ	\leq	≤
\gamma	γ	\chi	χ	\infty	∞
\delta	δ	\psi	ψ	\clubsuit	♣
\epsilon	ε	\omega	ω	\diamondsuit	♦
\zeta	ζ	\Gamma	Γ	\heartsuit	♥
\eta	η	\Delta	Δ	\spadesuit	♠
\theta	Θ	\Theta	Θ	\leftrightarrow	↔
\vartheta		\Lambda	Λ	\leftarrow	←
\iota	ι	\Xi	Ξ	\uparrow	↑
\kappa	κ	\Pi	Π	\rightarrow	→
\lambda	λ	\Sigma	Σ	\downarrow	↓
\mu	µ	\Upsilon		\circ	º

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\nu	ν	\Phi	Φ	\pm	±
\xi	ξ	\Psi	Ψ	\geq	≥
\pi	π	\Omega	Ω	\propto	∝
\rho	ρ	\forall	∀	\partial	∂
\sigma	σ	\exists	∃	\bullet	•
\varsigma	ς	\ni	∍	\div	÷
\tau	τ	\cong	≅	\neq	≠
\equiv	≡	\approx	≈	\aleph	
\Im	ℑ	\Re	ℜ	\wp	℘
\otimes	⊗	\oplus	⊕	\oslash	∅
\cap	∩	\cup	∪	\supseteq	⊇
\supset	⊃	\subseteq	⊆	\subset	⊂
\int	∫	\in		\o	ο
\rfloor	□	\lceil	□	\nabla	∇
\lfloor	□	\cdot	·	\ldots	...
\perp	⊥	\neg	¬	\prime	′
\wedge	∧	\times	x	\0	∅
\rceil	□	\surd	√	\mid	|
\vee	∨	\varpi	ϖ	\copyright	©
\langle	〈	\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

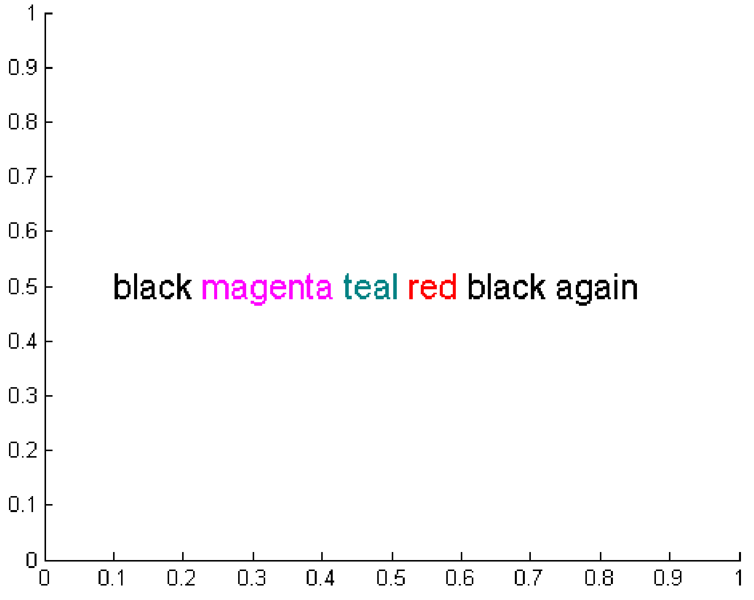
Use the `\color` modifier to change the color of characters following it from the previous color (which is `black` by default). Syntax is:

- `\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'])
```



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 2-3698 in the `text` reference page for more information.

When `Interpreter` is set to `none`, no characters in the `String` are interpreted, and all are displayed when the text is drawn.

When `Interpreter` is set to `latex`, MATLAB provides a complete LaTeX interpreter for text objects. See the `Interpreter` property for more information.

`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'`.

`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.

`Units`

`pixels | normalized | inches |`
`| characters | 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;).

- Units of `characters` are based on the size of characters in the default system font. The width of one character unit is the width of the letter x, the height of one character unit is the distance between the baselines of two lines of text.
- `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 axes 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.

`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`.

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

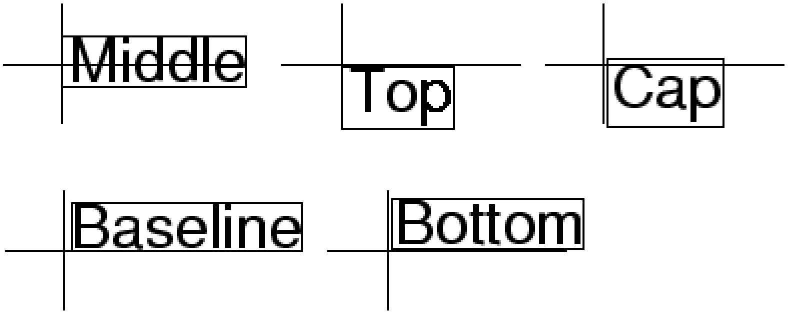
Vertical alignment of text. This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the `Position` property. The possible values mean

- 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 `textread` is not recommended. Use `textscan` to read data from a text file.

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(...,param,value,...) customizes textread using param/value pairs, as listed in the table below.

param	value	Action
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: ' ' Space \b Backspace \n Newline \r Carriage return \t Horizontal tab	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 =
  12.34000000000000
y =
    45
answer =
    'Yes'
```

Example 2 — Read as Fixed Format File, Ignoring the Floating Point Value

The first line of `mydata.dat` is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a fixed format file, ignoring the floating-point value.

```
[names, types, y, answer] = textread('mydata.dat', ...
'%9c %5s %*f %2d %3s', 1)
```

returns

```
names =
Sally
types =
    'Level1'
y =
    45
answer =
    'Yes'
```

`%*f` in the `format` string causes `textread` to ignore the floating point value, in this case, `12.34`.

Example 3 — Read Using Literal to Ignore Matching Characters

The first line of `mydata.dat` is

```
Sally    Type1 12.34 45 Yes
```

Read the first line of the file, ignoring the characters `Type` in the second field.

```
[names, typenum, x, y, answer] = textread('mydata.dat', ...
'%s Type%d %f %d %s', 1)
```

returns

```
names =
    'Sally'
typenum =
    1
x =
  12.34000000000000
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`, `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

Note 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, 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`. The `format` input is a string of conversion specifiers enclosed in single quotation marks. The number of specifiers determines the number of cells in the cell array `C`.

`C = textscan(fid, 'format', N)` reads data from the file, using the `format` `N` times, where `N` is a positive integer. You can read additional data 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. For a list of all valid parameter strings, value descriptions, and defaults, see "User Configurable Options" on page 2-3742.

`C = textscan(fid, 'format', N, param, value, ...)` reads data from the file, using the `format` `N` times, and using nondefault parameter settings specified by pairs of `param` and `value` arguments.

`C = textscan(str, ...)` reads data from string `str`. You can use the `format`, `N`, and parameter/value arguments described above with this syntax. However, for strings, repeated calls to `textscan` restart the scan from the beginning each time. (See "Example 10 — Resuming a Text Scan of a String" on page 2-3751.)

`[C, position] = textscan(...)` returns the file or string position at the end of the scan as the second output argument. For a file, this is the value that `ftell(fid)` would return after calling `textscan`. For a string, `position` indicates how many characters `textscan` read.

Remarks

When `textscan` reads a specified file or string, it attempts to match the data to the `format` string. If `textscan` fails to convert a data field, it stops reading and returns all fields read before the failure.

Basic Conversion Specifiers

The `format` input is a string of one or more conversion specifiers. The following table lists the basic specifiers.

Field Type	Specifier	Details

Integer, signed	`%d`
	`%d8`
	`%d16`
	`%d32`
	`%d64`
		32-bit
		8-bit
		16-bit
		32-bit
		64-bit

Integer, unsigned	`%u`
	`%u8`
	`%u16`
	`%u32`
	`%u64`
		32-bit
		8-bit
		16-bit
		32-bit
		64-bit

Floating-point number	`%f`
	`%f32`
	`%f64`
	`%n`
		64-bit (`double`)
		32-bit (`single`)
		64-bit (`double`)
		64-bit (`double`)

Character strings	`%s`
	`%q`
	`%c`
		String
		String, possibly double-quoted
		Any single character, including a delimiter

Field Type	Specifier	Details
Pattern-matching strings	`%[...]`	Read only characters in the brackets, until the first nonmatching character. Use `%[]...]` to include `]`. Example: `%[mus]` reads `'summer '` as `'summ'`.
	`%[^...]`	Read only characters not in the brackets, until the first matching character. Use `%[^]...]` to exclude `]`. Example: `%[^xrg]` reads `'summer '` as `'summe'`.

For each numeric conversion specifier, `textscan` returns a `K`-by-1 MATLAB numeric vector to the output cell array `C`, where `K` is the number of times that `textscan` finds a field matching the specifier. For each string conversion specifier, `textscan` returns a `K`-by-1 cell vector of strings. For each character conversion of the form `%Nc` (see "Field Length" on page 2-3740), `textscan` returns a `K`-by-`N` character array.

Field Length

You can specify the number of characters or digits to read by inserting a number between the percent character (%) and the format specifier. For floating-point numbers (`%n`, `%f`, `%f32`, `%f64`), you also can specify the number of digits read to the right of the decimal point.

Specifier	Action Taken
`%Nc`	Read `N` characters, including delimiter characters. Example: `%9c` reads `'Let's Go!'` as `'Let's Go!'`.

Specifier		Action Taken
%Ns %Nq %N[...] %N[^...]	%Nn %Nd... %Nu... %Nf...	Read N characters or 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.
%N.Dn %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 .

Skipping Fields or Parts of Fields

The textscan function reads all characters in your file in sequence unless you tell it to ignore a particular field or a portion of a field.

Use the following format specifiers to skip or read portions of fields:

Specifier	Action Taken
%*...	Skip the field. textscan does not create an output cell for any field that it skips. Example: '%s %*s %s %s %*s %*s %s' (spaces are optional) converts the string 'Blackbird singing in the dead of night' to four output cells with the strings 'Blackbird' 'in' 'the' 'night'

Specifier	Action Taken
`%*n...`	Ignore `n` characters of the field, where `n` is an integer less than or equal to the number of characters in the field. Example: `%*4s` reads `'summer '` as `'er'`.
literal	Ignore the specified characters of the field. Example: `Level%u8` reads `'Level1'` as `1`. Example: `%u8Step` reads `'2Step'` as `2`.

The `textscan` function does not include leading white-space characters in the processing of any data fields. When processing numeric data, `textscan` also ignores trailing white space.

User Configurable Options

This table shows the valid `param-value` options and their default values. Parameter names are not case sensitive.

Parameter	Value	Default
`BufSize`	Maximum string length in bytes	4095
`CollectOutput`	If `true`, `textscan` concatenates consecutive output cells with the same data type into a single array.	`0 (false)`

Parameter	Value	Default
CommentStyle	Symbol(s) designating text to ignore. Specify a single string (such as '%') to ignore characters following the string on the same line. Specify a cell array of two strings (such as {'/*', '*/'}) to ignore characters between the strings.	None
Delimiter	Field delimiter character(s)	White space
EmptyValue	Value to return for empty numeric fields in delimited files	NaN
EndOfLine	End-of-line character	Determined from the file: \n, \r, or \r\n
ExpChars	Exponent characters	'eEdD'
HeaderLines	Number of lines to skip. (Includes the remainder of the current line.)	0
MultipleDelimsAsOne	If true, textscan treats consecutive delimiters as a single delimiter. Only valid if you specify the delimiter option.	0 (false)

Parameter	Value	Default
ReturnOnError	Determines behavior when textscan fails to read or convert. If true, textscan terminates without an error and returns all fields read. If false, textscan terminates with an error and does not return an output cell array.	1 (true)
TreatAsEmpty	String(s) in the data file to treat as an empty value. Can be a single string or cell array of strings. Only applies to numeric fields.	None
Whitespace	White-space characters	' \b\t'

Field and Row Delimiters

Within each row, the default field delimiter is white space. White space can be any combination of space (' '), backspace ('\b'), or tab ('\t') characters.

If you use the default (white space) field delimiter, textscan interprets repeated white-space characters as a single delimiter. If you specify a nondefault delimiter, textscan interprets repeated delimiter characters as separate delimiters, and returns an empty value to the output cell. (See “Example 5 — Specifying Delimiter and Empty Value Conversion” on page 2-3747 and “Example 7 — Handling Repeated Delimiters” on page 2-3748.)

Rows delimiters are end-of-line (EOL) character sequences. The default end-of-line setting depends on the format of your file, and can include a newline character ('\n'), a carriage return ('\r'), or a combination of the two ('\r\n').

For more information, see “Example 9 — Using Nondefault Control Characters” on page 2-3750.

Numeric Fields

textscan converts numeric fields to the specified output type according to MATLAB rules regarding overflow, truncation, and the use of NaN, Inf, and -Inf.

For example, MATLAB represents an integer NaN as zero. If textscan finds an empty field associated with an integer format specifier (such as %d or %u), it returns the empty value as zero and not NaN. (See “Example 2 — Reading Different Types of Data” on page 2-3746 and “Example 5 — Specifying Delimiter and Empty Value Conversion” on page 2-3747.)

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 as follows:

<table>
<tr><th>Form</th><th>Example</th></tr>
<tr><td>±<real>±<imag>i|j</td><td>5.7-3.1i</td></tr>
<tr><td>±<imag>i|j</td><td>-7j</td></tr>
</table>

Do not include embedded white space in a complex number. textscan interprets embedded white space as a field delimiter.

Examples

Note The following examples include spaces between the conversion specifiers to make the format value easier to read. Spaces are not required.

Example 1 — Reading a String

Read the following string, truncating each value to one decimal digit. The specifier %*1d tells textscan to skip the remaining digit:

```
str = '0.41 8.24 3.57 6.24 9.27';
```

```
C = textscan(str, '%3.1f %*1d');
```

`textscan` returns a 1-by-1 cell array `C`:

```
C{1} = [0.4; 8.2; 3.5; 6.2; 9.2]
```

Example 2 — Reading Different Types of Data

The 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
```

Open the file, and read each column with the appropriate conversion specifier:

```
fid = fopen('scan1.dat');
C = textscan(fid, '%s %s %f32 %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 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 3 — Removing a Literal String

Remove the text `'Level'` from each field in the second column of the data from Example 2:

```
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

```
C{2} = [1; 2; 3]                        class uint8
```

Example 4 — Reading Only the First Field

Read the first column of the file in Example 2 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:

```
names{1} = {'Sally'; 'Joe'; 'Bill'}
```

Example 5 — Specifying Delimiter and Empty Value Conversion

The comma-delimited file data.csv contains

```
1,  2,  3,  4,   ,  6
7,  8,  9,   , 11, 12
```

Read the file, converting empty cells to -Inf:

```
fid = fopen('data.csv');
C = textscan(fid, '%f %f %f %f %u32 %f', 'delimiter', ',', ...
             'EmptyValue', -Inf);
fclose(fid);
```

textscan returns a 1-by-6 cell array C with the following cells:

```
C{1} = [1; 7]          class double
C{2} = [2; 8]          class double
C{3} = [3; 9]          class double
```

```
C{4} = [4; -Inf]        class double (empty converted to -Inf)
C{5} = [0; 11]          class uint32 (empty converted to 0)
C{6} = [6; 12]          class double
```

`textscan` converts the empty value in `C{4}`, associated with a floating-point format, to `-Inf`. Because MATLAB represents unsigned integer `-Inf` as `0`, `textscan` converts the empty value in `C{5}` to `0` and not `-Inf`.

Example 6 — Using Custom Empty Value Strings and Comments

The comma-delimited file `data2.csv` contains the lines

```
abc, 2, NA, 3, 4
// Comment Here
def, na, 5, 6, 7
```

Designate the input that `textscan` should treat as comments or empty values:

```
fid = fopen('data2.csv');
C = textscan(fid, '%s %n %n %n %n', 'delimiter', ',', ...
             'treatAsEmpty', {'NA', 'na'}, ...
             'commentStyle', '//');
fclose(fid);
```

`textscan` returns a 1-by-5 cell array `C` with the following cells:

```
C{1} = {'abc'; 'def'}
C{2} = [2; NaN]
C{3} = [NaN; 5]
C{4} = [3; 6]
C{5} = [4; 7]
```

Example 7 — Handling Repeated Delimiters

The file `data3.csv` contains

```
1,2,3,,4
```

```
5,6,7,,8
```

To treat the repeated commas as a single delimiter, use the `MultipleDelimsAsOne` parameter, with a value of `1`:

```
fid = fopen('data3.csv');
C = textscan(fid, '%f %f %f %f', 'delimiter', ',', ...
             'MultipleDelimsAsOne', 1);
fclose(fid);
```

`textscan` returns a 1-by-4 cell array `C` with the following cells:

```
C{1} = [1; 5]
C{2} = [2; 6]
C{3} = [3; 7]
C{4} = [4; 8]
```

Example 8 — Using the CollectOutput Switch

The file `grades.txt` contains

```
Student_ID  | Test1  | Test2  | Test3
   1          91.5     89.2     77.3
   2          88.0     67.8     91.0
   3          76.3     78.1     92.5
   4          96.4     81.2     84.6
```

The default value for the `CollectOutput` switch is 0 (`false`), and `textscan` returns each column of the numeric data in a separate array:

```
fid = fopen('grades.txt');

% read column headers
C_text = textscan(fid, '%s', 4, 'delimiter', '|');

% read numeric data
C_data0 = textscan(fid, '%d %f %f %f')

C_data0 =
```

```
    [4x1 int32]    [4x1 double]    [4x1 double]    [4x1 double]
```

Set CollectOutput to 1 (true) to collect the consecutive columns of the same class (the test scores, which are all double) into a single array:

```
frewind(fid);

C_text = textscan(fid, '%s', 4, 'delimiter', '|');

C_data1 = textscan(fid, '%d %f %f %f', ...
                        'CollectOutput', 1)

C_data1 =
    [4x1 int32]    [4x3 double]

fclose(fid);
```

Example 9 — Using Nondefault Control Characters

When you specify one of the following escape sequences for any parameter value, textscan converts that sequence to the corresponding control character:

\b	Backspace
\n	Newline
\r	Carriage return
\t	Tab
\\	Backslash (\)

If your data uses a different control character, use the sprintf function to explicitly convert the escape sequence in your call to textscan.

For example, the following string includes a form feed character, \f:

```
lyric = sprintf('Blackbird\fsinging\fin\fthe\fdead\fof\fnight');
```

To read the string using `textscan`, call the `sprintf` function to explicitly convert the form feed:

```
C = textscan(lyric, '%s', 'delimiter', sprintf('\f'));
```

`textscan` returns a 1-by-1 cell array `C`:

```
C{1} =
    {'Blackbird'; 'singing'; 'in'; 'the'; 'dead'; 'of'; 'night'}
```

Example 10 — Resuming a Text Scan of a String

If you resume a text scan of a file by calling `textscan` with the same file identifier (`fid`), `textscan` automatically resumes reading at the point where it terminated the last read.

If your input is a string rather than a file, `textscan` reads from the beginning of the string each time. To resume a scan from any other position in the string, you must use the two-output argument syntax in your initial call to `textscan`. For example, given the string

```
lyric = 'Blackbird singing in the dead of night'
```

Read the first word of the string:

```
[firstword, pos] = textscan(lyric,'%9c', 1);
```

Resume the scan:

```
lastpart = textscan(lyric(pos+1:end), '%s');
```

See Also

`load`, `type`, `importdata`, `uiimport`, `dlmread`, `xlsread`, `fscanf`, `fread`

in the MATLAB Data Import and Export documentation

Purpose

Wrapped string matrix for given uicontrol

Syntax

```
outstring = textwrap(h,instring)
outstring = textwrap(h,instring,columns)
[outstring,position] = textwrap(...)
```

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 = textwrap(h,instring,columns)` returns an `outstring` with each line wrapped at `columns` characters. Spaces are included in the character count.

`[outstring,position] = textwrap(...)` 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.

`textwrap` maintains the original line breaks in the input cell array and adds new ones. It can calculate uicontrol positions with any type of `Units`, including normalized units.

Remarks

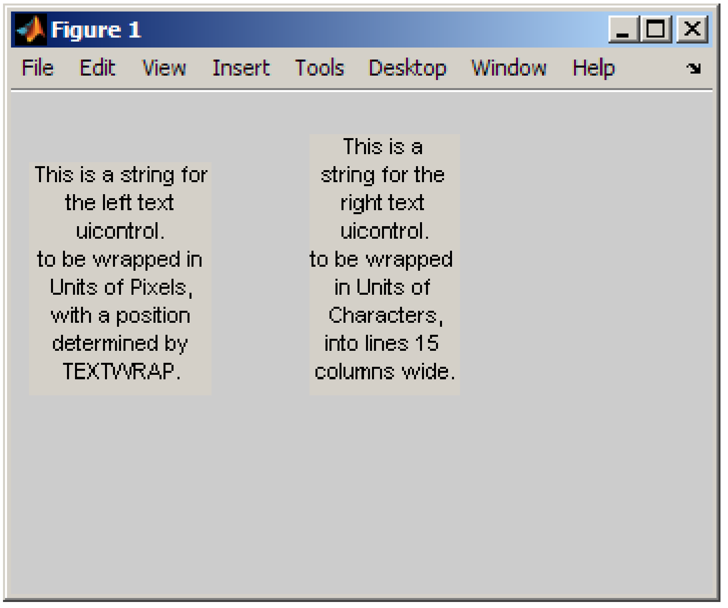
When programming a GUI, do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

Example

Place two text-wrapped strings in text uicontrols. The left one has a `Position` calculated by `textwrap` in `Units` of `pixels`; the right one's `Position` is calculated manually in `Units` of `characters`:

```
hf = figure('Position',[560 528 350 250]);
% Make a text uicontrol to wrap in Units of Pixels
% Create it in Units of Pixels, 100 wide, 10 high
pos = [10 100 100 10];
ht = uicontrol('Style','Text','Position',pos);
```

```
string = {'This is a string for the left text uicontrol.',...
          'to be wrapped in Units of Pixels,',...
          'with a position determined by TEXTWRAP.'};
% Wrap string, also returning a new position for ht
[outstring,newpos] = textwrap(ht,string);
set(ht,'String',outstring,'Position',newpos)

% Make another text uicontrol to wrap to a column width of 15
colwidth = 15;
% Create it in Units of Pixels, 100 wide, 10 high
pos1 = [150 100 100 10];
ht1 = uicontrol('Style','Text','Position',pos1);
string1 = {'This is a string for the right text uicontrol.',...
          'to be wrapped in Units of Characters,',...
          'into lines 15 columns wide.'};
outstring1 = textwrap(ht1,string1,colwidth);
% Reset Units of ht1 to Characters to use the result
set(ht1,'Units','characters')
newpos1 = get(ht1,'Position');
% Set new Position in Characters to be specified colwidth
% with height the length of the outstring1 cell array + 1.
newpos1(3) = colwidth;
newpos1(4) = length(outstring1)+1;
set(ht1,'String',outstring1,'Position',newpos1)
```



See Also

`align`, `uicontrol`

Purpose

Transpose-free quasi-minimal residual method

Syntax

```
x = tfqmr(A,b)
x = tfqmr(afun,b)
x = tfqmr(a,b,tol)
x = tfqmr(a,b,tol,maxit)
x = tfqmr(a,b,tol,maxit,m)
x = tfqmr(a,b,tol,maxit,m1,m2,x0)
[x,flag] = tfqmr(A,B,...)
[x,flag,relres] = tfqmr(A,b,...)
[x,flag,relres,y]y(A,b,...)
[x,flag,relres,iter,resvec] = tfqmr(A,b,...)
```

Description

`x = tfqmr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and the right-hand side column vector `b` must have length `n`.

`x = tfqmr(afun,b)` accepts a function handle `afun` instead of the matrix `A`. `afun(x)` accepts a vector input `x` and returns the matrix-vector product `A*x`. In all of the following syntaxes, you can replace `A` by `afun`. See in the MATLAB Programming documentation for more information., in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`.

`x = tfqmr(a,b,tol)` specifies the tolerance of the method. If `tol` is [] then `tfqmr` uses the default, 1e-6.

`x = tfqmr(a,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is [] then `tfqmr` uses the default, `min(N,20)`.

`x = tfqmr(a,b,tol,maxit,m)` and `x = tfqmr(a,b,tol,maxit,m1,m2)` use preconditioners `m` or `m=m1*m2` and effectively solve the system `A*inv(M)*x = B` for `x`. If `M` is `[]` then a preconditioner is not applied. `M` may be a function handle `mfun` such that `mfun(x)` returns `m\x`.

`x = tfqmr(a,b,tol,maxit,m1,m2,x0)` specifies the initial guess. If `x0` is `[]` then `tfqmr` uses the default, an all zero vector.

`[x,flag] = tfqmr(A,B,...)` also returns a convergence flag:

Flag	Convergence
0	tfqmr converged to the desired tolerance tol within maxit iterations.
1	tfqmr iterated maxit times but did not converge.
2	Preconditioner m was ill-conditioned.
3	tfqmr stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during tfqmr became too small or too large to continue computing.

[x,flag,relres] = tfqmr(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, then relres <= tol.

[x,flag,relres,y]y(A,b,...) also returns the iteration number at which x was computed: 0 <= iter <= maxit.

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

Examples

```
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 = tfqmr(A,b,tol,maxit,M1,M2,[]);
```

You can also use a matrix-vector product function as input:

```
function y = afun(x,n)
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);
x1 = tfqmr(@(x)afun(x,n),b,tol,maxit,M1,M2);
```

If applyOp is a function suitable for use with qmr, it may be used with tfqmr by wrapping it in an anonymous function:

```
x1 = tfqmr(@(x)applyOp(x,'notransp'),b,tol,maxit,M1,M2);
```

See Also

qmr, bicg, bicgstab, bicgstablcgs, gmres, lsqr, luinc, minres, pcg, symmlq, mldivide (\)

throw (MException)

Purpose

Issue exception and terminate function

Syntax

`throw(errRecord)`

Description

`throw(errRecord)` issues an exception based on the information contained in error record *`errRecord`*. The exception terminates the currently running function and returns control to its caller. The *`errRecord`* argument is a data structure derived from the `MException` class that contains information on the cause of the error and where it occurred. The `throw` function passes *`errRecord`* back to the caller of the currently running function. and eventually back to the Command Window when the program terminates. The error record is made available to any calling function by means of the `catch` function, and to the Command Window by means of the `MException.last` function.

Unlike `throwAsCaller` and `rethrow`, the `throw` function also sets the *stack* field of the *`errRecord`* to the location from which `throw` was called.

Remarks

There are four ways to throw an exception in MATLAB. Use the first of these when testing the outcome of some action for failure and reporting the failure to MATLAB:

- Test the result of some action taken by your program. If the result is found to be incorrect or unexpected, compose an appropriate message and message identifier, and pass these to MATLAB using the `error` or `assert` function.

Use one of the remaining three techniques to resume an exception that is already in progress but has been temporarily suspended in a try-catch statement:

- Reissue the original exception by returning the initial error record unmodified. Use the `MException` `rethrow` method to do this.
- Collect additional information on the cause of the error, store it in a new or modified error record, and issue a new exception based on that record. Use the `MException` `addCause` and `throw` methods to do this.

- Set or modify the stack field of a new or existing error record to make it appear that the error originated in the caller of the currently running function. Use the `MException` `throwAsCaller` method to do this.

Examples

Example 1

This example tests the output of M-file `evaluate_plots` and throws an exception if it is not acceptable:

```
[minval, maxval] = evaluate_plots(p24, p28, p41);
if minval < lower_bound || maxval > upper_bound
    errRecord = MException('VerifyOutput:OutOfBounds', ...
       'Results are outside the allowable limits');
    throw(errRecord);
end
```

Example 2

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

```
function data = read_it(filename);
try
   fid = fopen(filename, 'r');
   data = fread(fid);
catch errRecord1
   if strcmp(errRecord1.identifier, 'MATLAB:FileIO:InvalidFid')
      msg = sprintf('\n%s%s%s', 'Cannot open file ', filename, ...
         '. Try another location?  ');
      reply = input(msg, 's')
      if reply(1) == 'y'
          newdir = input('Enter directory name:  ', 's');
      else
          throw(errRecord1);
      end
```

```
        addpath(newdir);
        try
           fid = fopen(filename, 'r');
           data = fread(fid);
        catch errRecord2
           errRecord3 = addCause(errRecord2, errRecord1)
           throw(errRecord3);
        end
        rmpath(newdir);
     end
  end
  fclose(fid);
```

If you run this function in a `try`-`catch` block at the command line, you can look at the error record by assigning it to a variable (`errRecord`) with the `catch` command.

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

  errRecord
  errRecord =
   MException object with properties:

      identifier: 'MATLAB:FileIO:InvalidFid'
         message: 'Invalid file identifier.  Use fopen to generate a vali
           stack: [1x1 struct]
           cause: {[1x1 MException]}

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

See Also

try, catch, error, assert, MException, throwAsCaller(MException), rethrow(MException), addCause(MException), getReport(MException), last(MException)

throwAsCaller (MException)

Purpose Throw exception as if from calling function

Syntax `throwAsCaller(errRecord)`

Description `throwAsCaller(errRecord)` throws an exception from the currently running M-file based on `MException` object `errRecord`. The MATLAB software exits the currently running function and returns control to either the keyboard or an enclosing `catch` block in a calling function. Unlike the `throw` function, MATLAB omits the current stack frame from the `stack` field of the `MException`, thus making the exception look as if it is being thrown by the caller of the function.

In some cases, it is not relevant to show the person running your program the true location that generated an exception, but is better to point to the calling function where the problem really lies. You might also find `throwAsCaller` useful when you want to simplify the error display, or when you have code that you do not want made public.

Remarks There are four ways to throw an exception in MATLAB. Use the first of these when testing the outcome of some action for failure and reporting the failure to MATLAB:

- Test the result of some action taken by your program. If the result is found to be incorrect or unexpected, compose an appropriate message and message identifier, and pass these to MATLAB using the `error` or `assert` function.

Use one of the remaining three techniques to resume an exception that is already in progress but has been temporarily suspended in a try-catch statement:

- Reissue the original exception by returning the initial error record unmodified. Use the `MException` `rethrow` method to do this.
- Collect additional information on the cause of the error, store it in a new or modified error record, and issue a new exception based on that record. Use the `MException` `addCause` and `throw` methods to do this.

- Set or modify the stack field of a new or existing error record to make it appear that the error originated in the caller of the currently running function. Use the `MException` `throwAsCaller` method to do this.

Examples

The function `klein_bottle`, in this example, generates a Klein Bottle figure by revolving the figure-eight curve defined by XYKLEIN. It defines a few variables and calls the function `draw_klein`, which executes three functions in a `try-catch` block. If there is an error, the `catch` block issues an exception using either `throw` or `throwAsCaller`:

```
function klein_bottle(pq)
ab = [0 2*pi];
rtr = [2 0.5 1];
box = [-3 3 -3 3 -2 2];
vue = [55 60];
draw_klein(ab, rtr, pq, box, vue)

function draw_klein(ab, rtr, pq, box, vue)
clf
try
   tube('xyklein',ab, rtr, pq, box, vue);
   shading interp
   colormap(pink);
catch errRecord
   throw(errRecord)
%   throwAsCaller(errRecord)
end
```

Call the `klein_bottle` function, passing a vector, and the function completes normally by drawing the figure.

```
klein_bottle([40 40])
```

Call the function again, this time passing a scalar value. Because the `catch` block issues the exception using `throw`, MATLAB displays error messages for line 16 of function `draw_klein`, and for line 6 of function `klein_bottle`:

```
klein_bottle(40)
??? Error using ==> klein_bottle>draw_klein at 16
Attempted to access pq(2); index out of bounds because numel(pq)=1.

Error in ==> klein_bottle at 6
draw_klein(ab, rtr, pq, box, vue)
```

Run the function again, this time changing the `klein_bottle.m` file so that the `catch` block uses `throwAsCaller` instead of `throw`. This time, MATLAB only displays the error at line 6 of the main program:

```
klein_bottle(40)
??? Error using ==> klein_bottle at 6
Attempted to access pq(2); index out of bounds because numel(pq)=1.
```

See Also

`try`, `catch`, `error`, `assert`, `MException`, `throw(MException)`, `rethrow(MException)`, `addCause(MException)`, `getReport(MException)`, `last(MException)`

Purpose

Measure performance using stopwatch timer

Syntax

```
tic; any_statements; toc;
tic; any_statements; tElapsed=toc;
tStart=tic; any_statements; toc(tStart);
tStart=tic; any_statements; tElapsed=toc(tStart);
```

Description

`tic; any_statements; toc;` measures the time it takes the MATLAB software to execute the one or more lines of MATLAB code shown here as `any_statements`. The `tic` command starts a stopwatch timer, MATLAB executes the block of statements, and `toc` stops the timer, displaying the time elapsed in seconds.

`tic; any_statements; tElapsed=toc;` makes the same time measurement, but assigns the elapsed time output to a variable, `tElapsed`. MATLAB does not display the elapsed time unless you omit the terminating semicolon. The value returned by `toc` is a scalar double that represents the elapsed time in seconds.

`tStart=tic; any_statements; toc(tStart);` makes the same time measurement, but allows you the option of running more than one stopwatch timer concurrently. You assign the output of `tic` to a variable `tStart` and then use that same variable when calling `toc`. MATLAB measures the time elapsed between the tic and its related toc command and displays the time elapsed in seconds. This syntax enables you to time multiple concurrent operations, including the timing of nested operations.

`tStart=tic; any_statements; tElapsed=toc(tStart);` is the same as the command shown above, except that MATLAB assigns the elapsed time output to a variable, `tElapsed`. MATLAB does not display the elapsed time unless you omit the terminating semicolon. The value returned by `toc` is a scalar double that represents the elapsed time in seconds.

Remarks

Using the third syntax shown above, you can nest `tic-toc` pairs.

When using the simpler tic and toc syntax, avoid using consecutive tics as they merely overwrite the internally-recorded starting time. Consecutive tocs however, may be useful as each toc returns the increasing time that has elapsed since the most recent tic. Using this mechanism, you can take multiple measurements from a single point in time.

When using the tStart=tic and toc(tStart) syntax, it is advisable to select a unique variable for tStart. If you accidentally overwrite this variable prior to the toc for which it is needed, you will get inaccurate results for the time measurement.

tStart is a 64-bit unsigned integer, scalar value. This value is only useful as an input argument for a subsequent call to toc.

The clear function does not reset the starting time recorded by a tic command.

Examples

Measure 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)
```

Measure the minimum and average time to compute a summation of Bessel functions:

```
REPS = 1000;   minTime = Inf;   nsum = 10;
tic;

for i=1:REPS
   tStart = tic;   total = 0;
   for j=1:nsum,
```

```
        total = total + besselj(j,REPS);
    end

    tElapsed = toc(tStart);
    minTime = min(tElapsed, minTime);
end
averageTime = toc/REPS;
```

See Also

clock, cputime, etime, profile

Tiff class

Purpose

MATLAB Gateway to LibTIFF library routines

Description

The `Tiff` class represents a connection to a Tagged Image File Format (TIFF) file and provides access to many of the capabilities of the LibTIFF library. Use the methods of the `Tiff` object to call routines in the LibTIFF library. In most cases, the syntax of the `Tiff` method is similar to the syntax of the corresponding LibTIFF library function.

While you can use the `imread` and `imwrite` functions to read and write TIFF files, the `Tiff` class offers capabilities that these functions don't provide, such as reading subimages, writing tiles and strips of image data, and modifying individual TIFF tags.

Construction

`obj = Tiff(filename,mode)` creates a `Tiff` object associated with the TIFF file `filename`. `mode` specifies the type of access to the file.

A TIFF file is made up of one or more image file directories (IFDs). An IFD contains image data and associated metadata. IFDs can also contain subIFDs which also contain image data and metadata. When you open a TIFF file for reading, the Tiff object makes the first IFD in the file the *current* IFD. `Tiff` methods operate on the current IFD. You can use `Tiff` object methods to navigate among the IFDs and the subIFDs in a TIFF file.

When you open a TIFF file for writing or appending, the `Tiff` object automatically creates a IFD in the file for writing subsequent data. This IFD has all the default values specified in TIFF Revision 6.0.

When creating a new TIFF file, before writing any image to the file, you must create certain required fields (tags) in the file. These tags include `ImageWidth`, `ImageHeight`, `BitsPerSample`, `SamplesPerPixel`, `Compression`, `PlanarConfiguration`, and `Photometric`. If the image data has a stripped layout, the IFD contains the `RowsPerStrip` tag. If the image data has a tiled layout, the IFD contains the `TileWidth` and `TileHeight` tags. Use the `setTag` method to define values for these tags.

Inputs

`filename`

Text string specifying name of file.

mode

One of the following text strings specifying the type of access to the TIFF file.

Supported Values

Parameter	Description
'r'	Open file for reading
'w'	Open file for writing; discard existing contents
'a'	Open or create file for writing; append data to end of file.
'r+'	Open (do not create) file for reading and writing

Properties

Compression

Specify scheme used to compress image data

This property identifies all supported values for the Compression tag. You can use this property to specify the value of this tag when using the setTag method.

Supported Values

None
CCITTRLE (Read-only)
CCITTFax3
CCITTFax4
LZW
JPEG
CCITTRLEW (Read-only)

PackBits
Deflate
AdobeDeflate (Same as deflate

Example:

```
tiffobj.setTag('Compression', Tiff.Compression.JPEG);
```

`ExtraSamples`

Describe extra components

This property identifies all supported values for the `ExtraSamples` tag. Use this property to specify the value of this tag when using the `setTag` method.

Unspecified
AssociatedAlpha
UnassociatedAlpha

Example:

```
tiffobj.setTag('ExtraSamples', Tiff.ExtraSamples.AssociatedAlpha)
```

`InkSet`

Specify set of inks used in separated image

This property identifies all supported values for the `InkSet` tag. Use this property to specify the value of this tag when using the `setTag` method. In this context, separated refers to photometric interpretation, not the planar configuration.

Supported Values

`CMYK`	Order of components: cyan, magenta, yellow, black. Usually, a value of 0 represents 0% ink coverage and a value of 255 represents 100% ink coverage for that component, but consult the TIFF specification for `DotRange`. When you specify CMYK, do not set the `InkNames` tag.
`MultiInk`	Any ordering other than CMYK. Consult the TIFF specification for `InkNames` field for a description of the inks used.

Example:

```
tiffobj.setTag('InkSet', Tiff.InkSet.CMYK);
```

`Orientation`

Specify visual orientation of the image data.

This property identifies all supported values for the `Orientation` tag. The first row represents the top of the image, and the first column represents the left side. Use this property to specify the value of this tag when using the `setTag` method. Support for this tag is for informational purposes only, and it does not affect how MATLAB reads or writes the image data.

Supported Values

`TopLeft`
`TopRight`
`BottomRight`
`BottomLeft`
`LeftTop`
`RightTop`

RightBottom
LeftBottom

Example:

```
tiffobj.setTag('Orientation', Tiff.Orientation.TopRight);
```

`Photometric`

Specify color space of image data

This property identifies all supported values for the `Photometric` tag. Use this property to specify the value of this tag when using the `setTag` method.

Supported Values

MinIsWhite
MinIsBlack
RGB
Palette
Mask
Separated (CMYK)
YCbCr
CIELab
ICCLab
ITULab

Example:

```
tiffobj.setTag('Photometric', Tiff.Photometric.RGB);
```

`PlanarConfiguration`

Specifies how image data components are stored on disk

This property identifies all supported values for the `PlanarConfiguration` tag. Use this property to specify the value of this tag when using the `setTag` method.

Supported Values

`Chunky`	Store component values for each pixel contiguously. For example, in the case of RGB data, the first three pixels would be stored in the file as RGBRGBRGB etc. Almost all TIFF images have contiguous planar configurations.
`Separate`	Store component values for each pixel separately. For example, in the case of RGB data, the red component would be stored separately in the file from the green and blue components.

Example:

```
tiffobj.setTag('PlanarConfiguration', Tiff.PlanarConfiguration
```

`ResolutionUnit`

Specify unit of measurement used for `XResolution` and `YResolution` tags

This property identifies all supported values for the `XResolution` and `YResolution` tags. Use this property to specify the value of this tag when using the `setTag` method.

Supported Values

`None` (default)
`Inch`
`Centimeter`

Example:

```
tiffobj.setTag('YResolution', Tiff.ResolutionUnit.Inch);
```

SampleFormat

Specify how to interpret each pixel sample

This property identifies all supported values for the SampleFormat tag. Use this property to specify the value of this tag when using the setTag method.

Supported Values

Uint
Int
IEEEFP
Void
ComplexInt
ComplexIEEEFP

Example:

```
tiffobj.setTag('SampleFormat', Tiff.SampleFormat.IEEEFP);
```

SubFileType

Specify type of image

This property identifies all supported values for the SubFileType tag. SubFileType is a bitmask that indicates the type of the image. Use this property to specify the value of this tag when using the setTag method.

Supported Values

`Default`	Default value for single image file or first image.
`ReducedImage`	The current image is a thumbnail or reduced-resolution image that typically would be found in a sub-IFD.
`Page`	The image is a single image of a multi-image (or multipage) file.
`Mask`	The image is a transparency mask for another image in the file. The photometric interpretation value must be Photometric.Mask.

Example:

```
tiffobj.setTag('SubFileType', Tiff.SubFileType.Mask);
```

`TagID`

List of recognized TIFF tag names with their ID numbers

This property identifies all the supported TIFF tags with their ID numbers. Use this property to specify a tag when using the `setTag` method. For example, `Tiff.TagID.ImageWidth` returns the ID of the `ImageWidth` tag. To get a list of the names of supported tags, use the `getTagNames` method.

Example:

```
tiffobj.setTag(Tiff.TagID.ImageWidth, 300);
```

`Thresholding`

Specifies technique used to convert from gray to black and white pixels.

This property identifies all supported values for the `Thresholding` tag. Use this property to specify the value of this tag when using the `setTag` method.

Supported Values

`BiLevel` (default)
`HalfTone`
`ErrorDiffuse`

Example:

```
tiffobj.setTag('Thresholding', Tiff.Thresholding.HalfTone);
```

`YCbCrPositioning`

Specify relative positioning of chrominance samples

This property identifies all supported values for the `YCbCrPositioning` tag. This property specifies the positioning of chrominance components relative to luminance samples. Use this property to specify the value of this tag when using the `setTag` method.

Supported Values

`Centered`	Specify for compatibility with industry standards such as PostScript Level 2
`Cosited`	Specify for compatibility with most digital video standards such as CCIR Recommendation 601-1.

Example:

```
tiffobj.setTag('YCbCrPositioning', Tiff.YCbCrPositioning.Centered)
```

Methods

close	Close Tiff object
computeStrip	Index number of strip containing specified coordinate

computeTile	Index number of tile containing specified coordinates
currentDirectory	Index of current IFD
getTag	Value of specified tag
getTagNames	List of recognized TIFF tags
getVersion	LibTIFF library version
isTiled	Determine if tiled image
lastDirectory	Determine if current IFD is last in file
nextDirectory	Make next IFD current IFD
numberOfStrips	Total number of strips in image
numberOfTiles	Total number of tiles in image
read	Read entire image
readEncodedStrip	Read data from specified strip
readEncodedTile	Read data from specified tile
rewriteDirectory	Write modified metadata to existing IFD
setDirectory	Make specified IFD current IFD
setSubDirectory	Make subIFD specified by byte offset current IFD
setTag	Set value of tag
write	Write entire image
writeDirectory	Create new IFD and make it current IFD
writeEncodedStrip	Write data to specified strip
writeTile	Write data to specified tile

Examples

Create a new TIFF file using the `Tiff` object. To run this example, your directory must be writable.

```
t = Tiff('myfile.tif', 'w');
%
% Close the Tiff object
t.close();
```

See Also

`imread` | `imwrite`

Tutorials

-
-

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 2-3779 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(timer)`, `disp(timer)`, `get(timer)`, `isvalid(timer)`, `set(timer)`, `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(timer)` function. To set the value of the properties of a timer object, use the `set(timer)` function.

Property Name	Property Description	Data Types, Values, Defaults, Access	
AveragePeriod	Average time between TimerFcn executions since the timer started. Note: Value is NaN until timer executes two timer callbacks.	Data type	double
		Default	NaN
		Read only	Always
BusyMode	Action taken when a timer has to execute TimerFcn before the completion of previous execution of TimerFcn. 'drop' — Do not execute the function. 'error' — Generate an error. Requires ErrorFcn to be set. 'queue' — Execute function at next opportunity.	Data type	Enumerated string
		Values	'drop' 'error' 'queue'
		Default	'drop'
		Read only	While Running = 'on'
ErrorFcn	Function that the timer executes when an error occurs. This function executes before the StopFcn. See for more information.	Data type	Text string, function handle, or cell array
		Default	None
		Read only	Never

Property Name	Property Description	Data Types, Values, Defaults, Access	
`ExecutionMode`	Determines how the timer object schedules timer events. See for more information.	Data type	Enumerated string

		Values	`'singleShot'`
			`'fixedDelay'`
			`'fixedRate'`
			`'fixedSpacing'`

		Default	`'singleShot'`
		Read only	While `Running = 'on'`
`InstantPeriod`	The time between the last two executions of `TimerFcn`.	Data type	`double`
		Default	`NaN`
		Read only	Always
`Name`	User-supplied name.	Data type	Text string
		Default	`'timer-`*`i`*`'`, where *`i`* is a number indicating the *`i`*th timer object created this session. To reset *`i`* to 1, execute the `clear classes` command.
		Read only	Never

Property Name	Property Description	Data Types, Values, Defaults, Access	
ObjectVisibility	Provides a way for application developers to prevent end-user access to the timer objects created by their application. The timerfind function does not return an object whose ObjectVisibility property is set to 'off'. Objects that are not visible are still valid. If you have access to the object (for example, from within the M-file that created it), you can set its properties.	Data type	Enumerated string
		Values	'off' 'on'
		Default	'on'
		Read only	Never
Period	Specifies the delay, in seconds, between executions of TimerFcn.	Data type	double
		Value	Any number >= 0.001
		Default	1.0
		Read only	While Running = 'on'
Running	Indicates whether the timer is currently executing.	Data type	Enumerated string
		Values	'off' 'on'
		Default	'off'
		Read only	Always

Property Name	Property Description	Data Types, Values, Defaults, Access	
`StartDelay`	Specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`.	Data type	`double`
		Values	Any number >= 0
		Default	0
		Read only	While `Running` =`'on'`
`StartFcn`	Function the timer calls when it starts. See for more information.	Data type	Text string, function handle, or cell array
		Default	None
		Read only	Never
`StopFcn`	Function the timer calls when it stops. The timer stops when • You call the timer `stop` function • The timer finishes executing `TimerFcn`, i.e., the value of `TasksExecuted` reaches the limit set by `TasksToExecute`. • An error occurs (The `ErrorFcn` is called first, followed by the `StopFcn`.) See for more information.	Date type	Text string, function handle, or cell array
		Default	None
		Read only	Never

Property Name	Property Description	Data Types, Values, Defaults, Access	
`Tag`	User supplied label.	Data type	Text string
		Default	Empty string ('')
		Read only	Never
`TasksToExecute`	Specifies the number of times the timer should execute the function specified in the `TimerFcn` property.	Data type	`double`
		Values	Any number > 0
		Default	`Inf`
		Read only	Never
`TasksExecuted`	The number of times the timer has called `TimerFcn` since the timer was started.	Data type	`double`
		Values	Any number >= 0
		Default	0
		Read only	Always
`TimerFcn`	Timer callback function. See for more information.	Data type	Text string, function handle, or cell array
		Default	None
		Read only	Never
`Type`	Identifies the object type.	Data type	Text string
		Values	`'timer'`
		Read only	Always

Property Name	Property Description	Data Types, Values, Defaults, Access	
`UserData`	User-supplied data.	Data type	User-defined
		Default	`[]`
		Read only	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)`, `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)`, `timer`, `timerfind`

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.

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 samples is `5` and the total number of data values is `5 x 4 x 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.

Note The words `default`, `factory`, and `remove` are reserved words that will not appear in a title when quoted as a normal string. In order to display any of these words individually, type `'\reserved_word'` instead of `'reserved_word'`.

Examples

Display today's date in the current axes:

```
title(date)
```

Include a variable's value in a title:

```
f = 70;
c = (f-32)/1.8;
title(['Temperature is ',num2str(c),'C'])
```

Make a multi-colored title:

```
title(['\fontsize{16}black {\color{magenta}magenta '...
'\color[rgb]{0 .5 .5}teal \color{red}red} black again'])
```

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 1-92 for related functions

Text Properties for information on setting parameter/value pairs in titles

Adding Titles to Graphs for more information on ways to add titles

todatenum

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`

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.

For a real vector `r`, `T = toeplitz(r)` returns the symmetric Toeplitz matrix formed from vector `r`, where `r` defines the first row of the matrix. For a complex vector `r` with a real first element, `T = toeplitz(r)` returns the Hermitian Toeplitz matrix formed from `r`, where `r` defines the first row of the matrix and `r'` defines the first column. When the first element of `r` is not real, the resulting matrix is Hermitian off the main diagonal, i.e., $\mathrm{T}_{ij} = \mathrm{conj}(\mathrm{T}_{ji})$ for $i \neq j$.

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`

toolboxdir

Purpose Root folder for specified toolbox

Syntax

```
toolboxdir('tbxFolderName')
s = toolboxdir('tbxFolderName')
s = toolboxdir tbxFolderName
```

Description `toolboxdir('tbxFolderName')` returns a string that is the absolute path to the specified toolbox, `tbxFolderName`, where `tbxFolderName` is the folder name for the toolbox.

`s = toolboxdir('tbxFolderName')` returns the absolute path to the specified toolbox to the output argument, `s`.

`s = toolboxdir tbxFolderName` is the command form of the syntax.

Remarks `toolboxdir` is particularly useful for MATLAB Compiler software. The base folder of all toolboxes installed with MATLAB software is:

```
matlabroot/toolbox/tbxFolderName
```

However, in deployed mode, the base folders of the toolboxes are different. `toolboxdir` returns the correct root folder, whether running from MATLAB or from an application deployed with the MATLAB Compiler software.

Example Obtain the path for the Control System Toolbox software:

```
s = toolboxdir('control')
```

MATLAB returns:

```
s = \\myhome\r2009a\matlab\toolbox\control
```

See Also `ctfroot` (in the MATLAB Compiler product), `fullfile`, `matlabroot`, `path`,

Purpose Sum of diagonal elements

Syntax

```
b = trace(A)
```

Description `b = trace(A)` is the sum of the diagonal elements of the matrix `A`.

Algorithm `trace` is a single-statement M-file.

```
t = sum(diag(A));
```

See Also `det`, `eig`

transpose (timeseries)

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 the size for time-series data (up to three dimensions) display 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

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`

treelayout

Purpose

Lay out tree or forest

Syntax

```
[x,y] = treelayout(parent,post)
[x,y,h,s] = treelayout(parent,post)
```

Description

`[x,y] = treelayout(parent,post)` lays out a tree or a forest. `parent` is the vector of parent pointers, with `0` for a root. `post` is an optional postorder permutation on the tree nodes. If you omit `post`, `treelayout` computes it. `x` and `y` are vectors of coordinates in the unit square at which to lay out the nodes of the tree to make a nice picture.

`[x,y,h,s] = treelayout(parent,post)` also returns the height of the tree `h` and the number of vertices `s` in the top-level separator.

See Also

`etree`, `treeplot`, `etreeplot`, `symbfact`

Purpose Plot picture of tree

Syntax

```
treeplot(p)
treeplot(p,nodeSpec,edgeSpec)
```

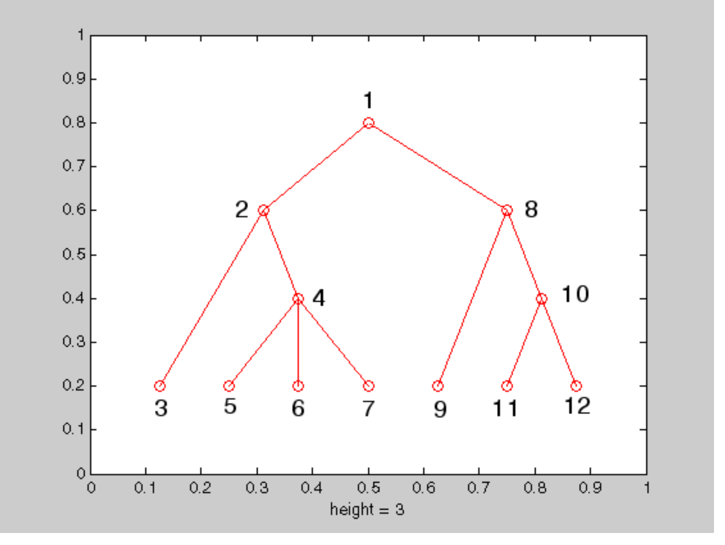
Description

`treeplot(p)` plots a picture of a tree given a vector of parent pointers, with `p(i)` = 0 for a root.

`treeplot(p,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `''` to omit one or both.

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

Purpose

Lower triangular part of matrix

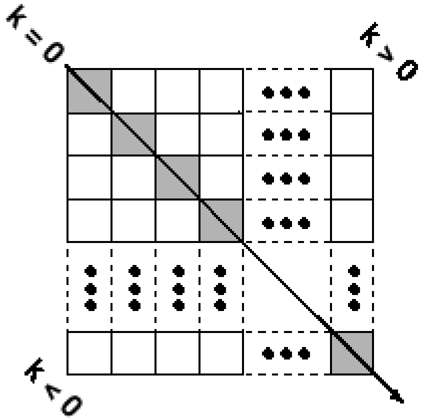
Syntax

```
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 kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

```
tril(ones(4,4),-1)

ans =

     0     0     0     0
     1     0     0     0
     1     1     0     0
     1     1     1     0
```

See Also

`diag`, `triu`

trimesh

Purpose Triangular mesh plot

Syntax

```
trimesh(Tri,X,Y,Z,C)
trimesh(Tri,X,Y,Z)
trimesh(Tri, X, Y)
trimesh(TR)
trimesh(...'PropertyName',PropertyValue...)
h = trimesh(...)
```

Description

`trimesh(Tri,X,Y,Z,C)` displays triangles defined in the m-by-3 face matrix `Tri` as a mesh. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices. The edge color is defined by the vector `C`.

`trimesh(Tri,X,Y,Z)` uses `C = Z` so color is proportional to surface height.

`trimesh(Tri, X, Y)` displays the triangles in a 2-D plot.

`trimesh(TR)` displays the triangles in a `TriRep` triangulation representation.

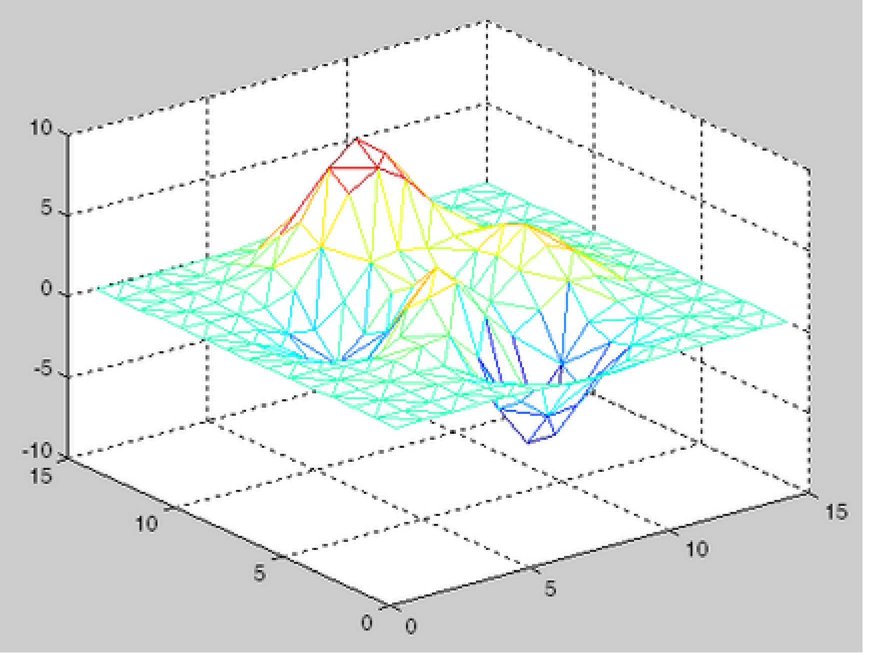
`trimesh(...'`*`PropertyName`*`',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

`h = trimesh(...)` returns a handle to the displayed triangles.

Example

Create vertex vectors and a face matrix, then create a triangular mesh plot.

```
[x,y]=meshgrid(1:15,1:15);
tri = delaunay(x,y);
z = peaks(15);
trimesh(tri,x,y,z)
```



If the surface is already a triangulation representation it may be plotted as follows:

```
tr = TriRep(tri, x(:), y(:), z(:));
trimesh(tr)
```

See Also

patch, trisurf, delaunay, DelaunayTri, TriRep

Purpose Numerically evaluate triple integral

Syntax

```
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)
```

Description

`triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)` evaluates the triple integral `fun(x,y,z)` over the three dimensional rectangular region `xmin <= x <= xmax`, `ymin <= y <= ymax`, `zmin <= z <= zmax`. `fun` is a function handle. See 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.

, 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 <= x <= pi`, `0 <= y <= 1`, `-1 <= z <= 1`. Note that the integrand can be evaluated with a vector `x` and scalars `y` and `z`.

See Also

`dblquad`, `quad2d`, `quad`, `quadgk`, `quadl`, `function handle` (`@`),

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 `LineSpec` for information about line specifications.

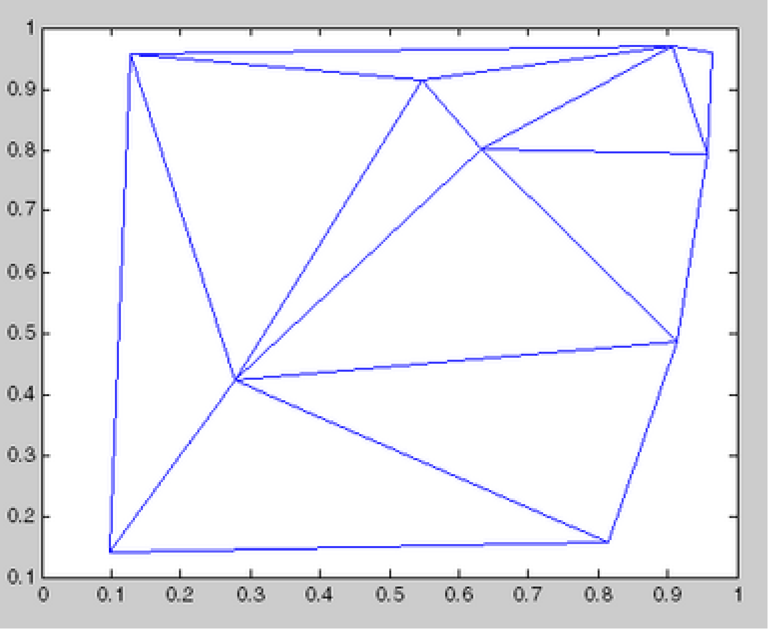
`h = triplot(...)` returns a vector of handles to the displayed triangles.

`triplot(...,'param','value','param','value'...)` allows additional line property name/property value pairs to be used when creating the plot. See `Line Properties` for information about the available properties.

Examples

Plot a Delaunay triangulation for 10 randomly generated points.

```
X = rand(10,2);
dt = DelaunayTri(X);
triplot(dt)
```



Plot the Delaunay triangulation in face-vertex format.

```
tri = dt(:,:);
triplot(tri, X(:,1), X(:,2));
```

See Also

`DelaunayTri`, `delaunay`, `trimesh`, `trisurf`

TriRep class

Purpose Triangulation representation

Description `TriRep` provides topological and geometric queries for triangulations in 2-D and 3-D space. For example, for triangular meshes you can query triangles attached to a vertex, triangles that share an edge, neighbor information, circumcenters, or other features. You can create a `TriRep` directly using existing triangulation data. Alternatively, you can create a Delaunay triangulation, via `DelaunayTri`, which provides access to the `TriRep` functionality.

Construction

TriRep	Triangulation representation

Methods

baryToCart	Converts point coordinates from barycentric to Cartesian
cartToBary	Convert point coordinates from cartesian to barycentric
circumcenters	Circumcenters of specified simplices
edgeAttachments	Simplices attached to specified edges
edges	Triangulation edges
faceNormals	Unit normals to specified triangles
featureEdges	Sharp edges of surface triangulation
freeBoundary	Facets referenced by only one simplex
incenters	Incenters of specified simplices
isEdge	Test if vertices are joined by edge

neighbors	Simplex neighbor information
size	Size of triangulation matrix
vertexAttachments	Return simplices attached to specified vertices

Properties

X	Coordinates of the points in the triangulation
Triangulation	Triangulation data structure

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Indexing

`TriRep` objects support indexing into the triamgulation using parentheses (). The syntax is the same as for arrays.

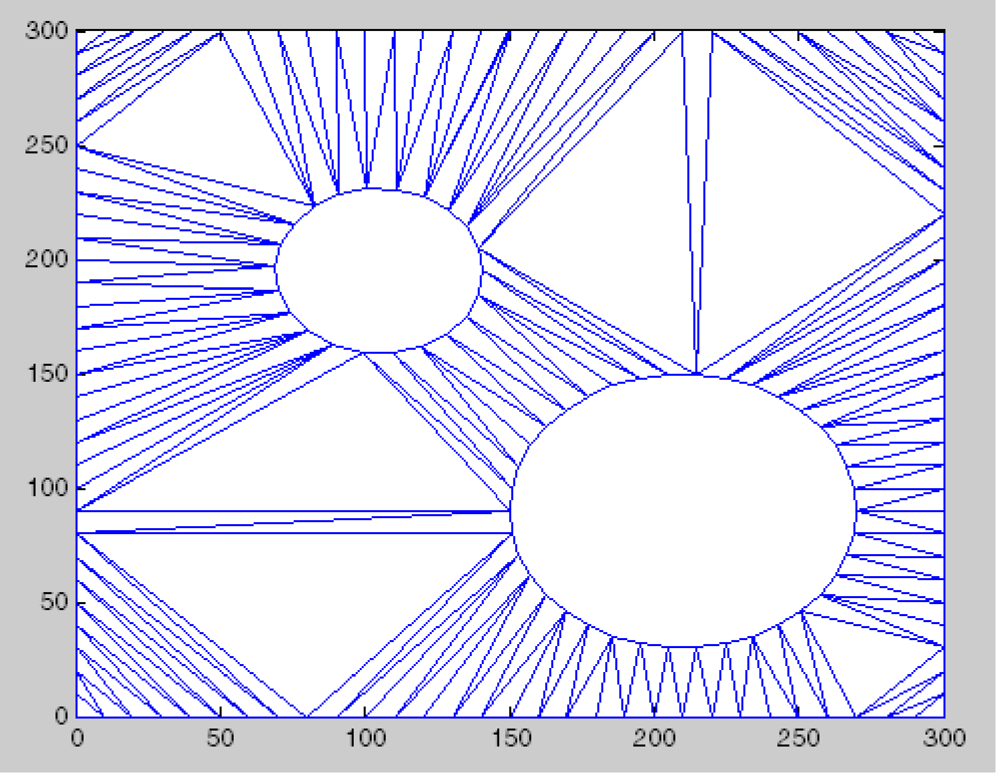
Examples

Load a 2-D triangulation and use the `TriRep` constructor to build an array of the free boundary edges:

```
load trimesh2d
```

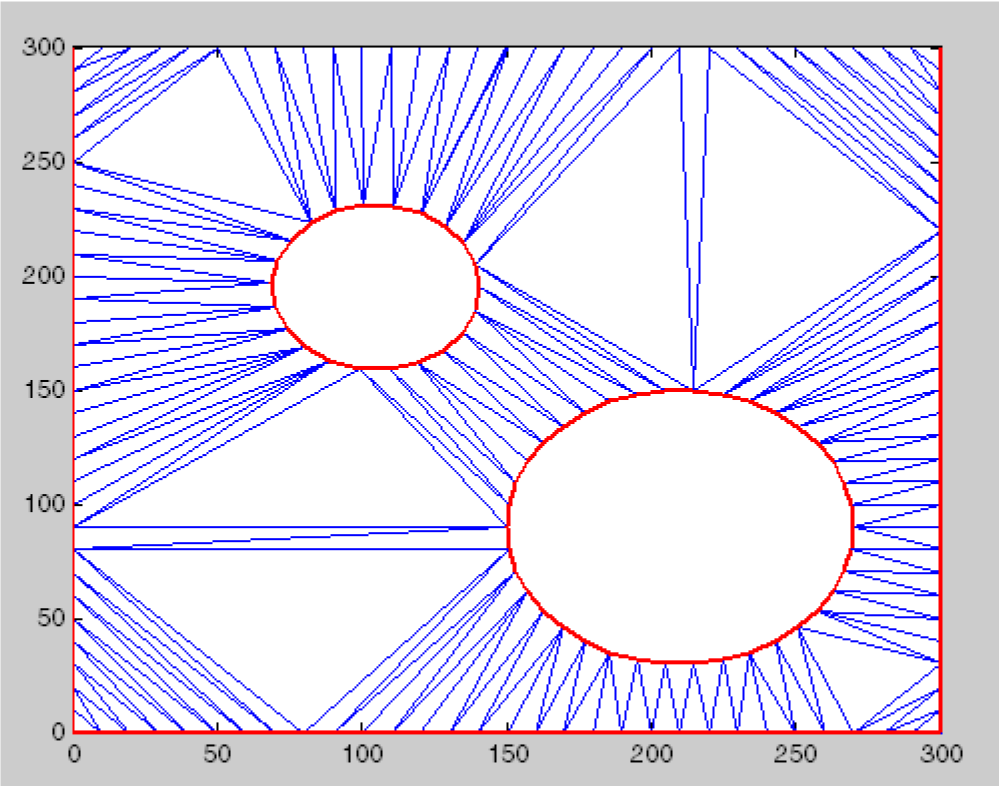
This loads triangulation `tri` and vertex coordinates `x`, `y`:

```
trep = TriRep(tri, x,y);
fe = freeBoundary(trep)';
triplot(trep);
```



You can add the free edges `fe` to the plot:

```
hold on;
plot(x(fe), y(fe), 'r','LineWidth',2);
hold off;
axis([-50 350 -50 350]);
axis equal;
```



See Also

`DelaunayTri` class
`TriScatteredInterp` class

Purpose

Triangulation representation

Syntax

```
TR = TriRep(TRI, X, Y)
TR = TriRep(TRI, X, Y, Z)
TR = TriRep(TRI, X)
```

Description

`TR = TriRep(TRI, X, Y)` creates a 2-D triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `(X, Y)`. `TRI` is an `m`-by-3 matrix that defines the triangulation in face-vertex format, where `m` is the number of triangles. Each row of `TRI` is a triangle defined by indices into the column vector of vertex coordinates `(X, Y)`.

`TR = TriRep(TRI, X, Y, Z)` creates a 3-D triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `(X, Y, Z)`. `TRI` is an `m`-by-3 or `m`-by-4 matrix that defines the triangulation in simplex-vertex format, where where m is the number of simplices; triangles or tetrahedra in this case. Each row of `TRI` is a simplex defined by indices into the column vector of vertex coordinates `(X, Y, Z)`.

`TR = TriRep(TRI, X)` creates a triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `X`. `TRI` is an `m`-by-`n` matrix that defines the triangulation in simplex-vertex format, where `m` is the number of simplices and `n` is the number of vertices per simplex. Each row of `TRI` is a simplex defined by indices into the array of vertex coordinates `X`. `X` is an`mpts`-by-`ndim` matrix where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, where $2 \le \text{ndim} \le 3$.

Examples

Load a 3-D tetrahedral triangulation compute the free boundary. First, load triangulation `tet` and vertex coordinates `X`.

```
load tetmesh
```

Create the triangulation representation and compute the free boundary.

```
trep = TriRep(tet, X);
[tri, Xb] = freeBoundary(trep);
```

See Also

`TriScatteredInterp`
— A guide to MATLAB's object-oriented and functional capabilities for computational geometry.

TriScatteredInterp class

Purpose

Interpolate scattered data

Description

A scattered data set defined by locations `X` and corresponding values `V` can be interpolated using a Delaunay triangulation of `X`. This produces a surface of the form `V = F(X)`. The surface can be evaluated at any query location `QX`, using `QV = F(QX)`, where `QX` lies within the convex hull of `X`. The interpolant `F` always goes through the data points specified by the sample.

Definitions

The *Delaunay triangulation* of a set of points is a triangulation such that the unique circle circumscribed about each triangle contains no other points in the set. The *convex hull* of a set of points is the smallest convex set containing all points of the original set. These definitions extend naturally to higher dimensions.

Construction

TriScatteredInterp	Interpolate scattered data

Properties

`X`	Defines locations of scattered data points in 2-D or 3-D space.	
`V`	Defines value associated with each data point.	
`Method`	Defines method used to interpolate the data .	
	`natural`	Natural neighbor interpolation
	`linear`	Linear interpolation (default)
	`nearest`	Nearest neighbor interpolation

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Examples

Create a data set:

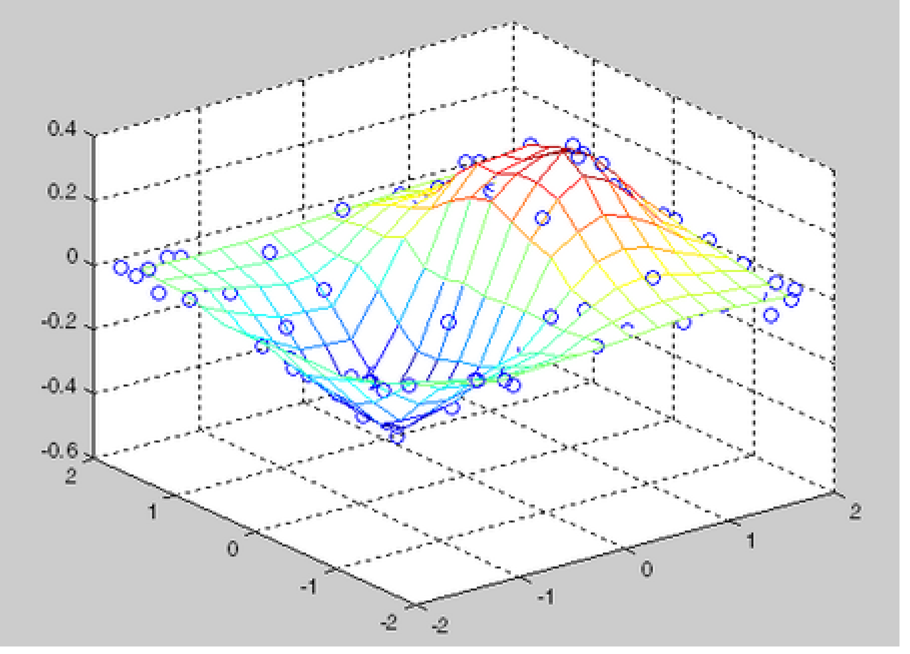
```
x = rand(100,1)*4-2;
y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

Construct the interpolant:

```
F = TriScatteredInterp(x,y,z);
```

Evaluate the interpolant at the locations (qx, qy). The corresponding value at these locations is qz:

```
ti = -2:.25:2;
[qx,qy] = meshgrid(ti,ti);
qz = F(qx,qy);
mesh(qx,qy,qz);
hold on;
plot3(x,y,z,'o');
```



See Also

`DelaunayTri`
`interp1`
`interp2`
`interp3`
`meshgrid`

Purpose Interpolate scattered data

Syntax

```
F = TriScatteredInterp()
F = TriScatteredInterp(X, V)
F = TriScatteredInterp(X, Y, V)
F = TriScatteredInterp(X, Y, Z, V)
F = TriScatteredInterp(DT, V)
F = TriScatteredInterp(..., method)
```

Description

`F = TriScatteredInterp()` creates an empty scattered data interpolant. This can subsequently be initialized with sample data points and values (`Xdata`, `Vdata`) via `F.X = Xdata` and `F.V = Vdata`.

`F = TriScatteredInterp(X, V)` creates an interpolant that fits a surface of the form `V = F(X)` to the scattered data in (`X`, `V`). `X` is a matrix of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, `ndim >= 2`. The column vector `V` defines the values at `X`, where the length of `V` equals `mpts`.

`F = TriScatteredInterp(X, Y, V)` and `F = TriScatteredInterp(X, Y, Z, V)` allow the data point locations to be specified in alternative column vector format when working in 2-D and 3-D.

`F = TriScatteredInterp(DT, V)` uses the specified `DelaunayTri` object `DT` as a basis for computing the interpolant. The matrix `DT.X` is of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, `2 <= ndim <= 3`. `V` is a column vector that defines the values at `DT.X`, where the length of `V` equals `mpts`.

`F = TriScatteredInterp(..., method)` allows selection of the technique `method` used to interpolate the data.

Inputs

X	Matrix of size mpts-by-ndim, where mpts is the number of points and ndim is the dimension of the space where the points reside.	
V	Column vector that defines the values at X, where the length of V equals mpts.	
DT	Delaunay triangulation of the scattered data locations	
method	natural	Natural neighbor interpolation
	linear	Linear interpolation (default)
	nearest	Nearest-neighbor interpolation

Outputs

F	Creates an interpolant that fits a surface of the form V = F(X) to the scattered data.

Evaluation

To evaluate the interpolant, express the statement in Monge's form V=F(x), V=F(x,y), or V=F(x,y,z).

Definitions

The *Delaunay triangulation* of a set of points is a triangulation such that the unique circle circumscribed about each triangle contains no other points in the set.

Examples

Create a data set:

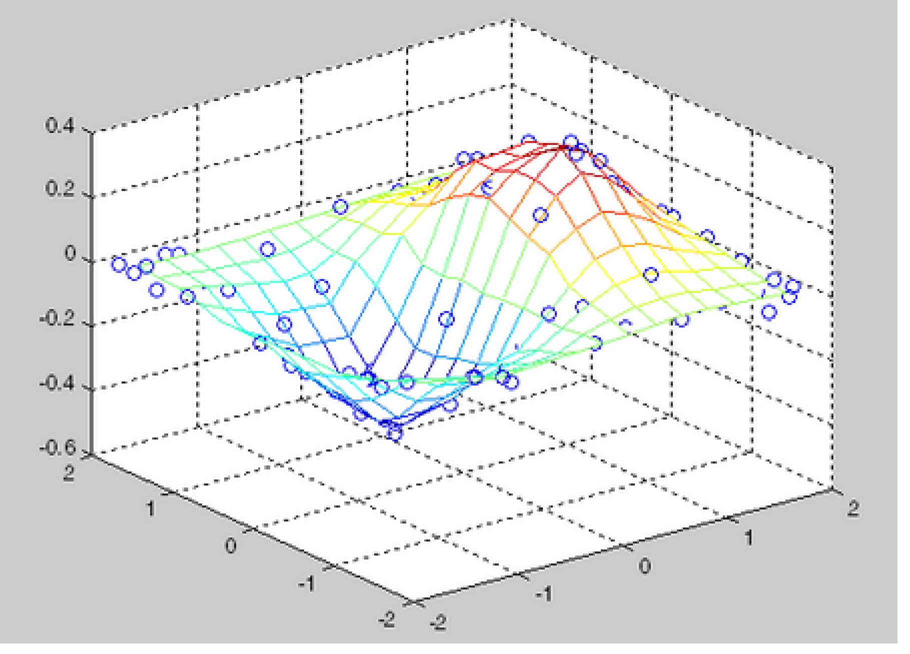
```
x = rand(100,1)*4-2;
y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

Construct the interpolant:

```
F = TriScatteredInterp(x,y,z);
```

Evaluate the interpolant at the locations (qx, qy). The corresponding value at these locations is qz .

```
ti = -2:.25:2;
[qx,qy] = meshgrid(ti,ti);
qz = F(qx,qy);
mesh(qx,qy,qz);
hold on;
plot3(x,y,z,'o');
```



See Also

DelaunayTri
interp1
interp2
interp3
meshgrid

trisurf

Purpose

Triangular surface plot

Syntax

```
trisurf(Tri,X,Y,Z,C)
trisurf(Tri,X,Y,Z)
trisurf(tr)
trisurf(...'PropertyName',PropertyValue...)
h = trisurf(...)
```

Description

`trisurf(Tri,X,Y,Z,C)` 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. The color is defined by the vector `C`.

`trisurf(Tri,X,Y,Z)` uses `C=Z` so color is proportional to surface height.

`trisurf(tr)` displays the triangles in a `TriRep` triangulation representation. It uses `C = TR.X(:,3)` so surface color is proportional to height.

`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.

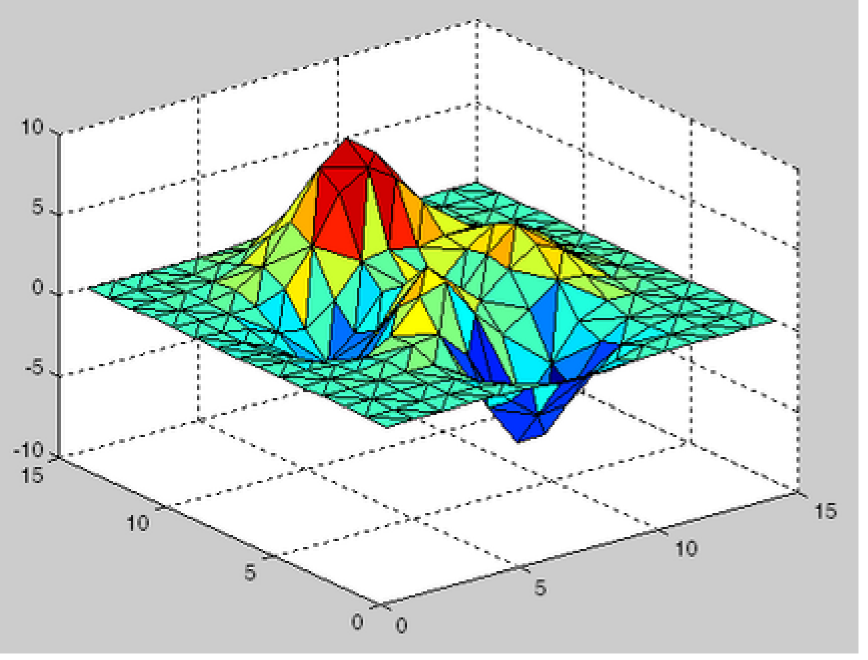
Example

Create vertex vectors and a face matrix, then create a triangular surface plot.

```
[x,y]=meshgrid(1:15,1:15);
tri = delaunay(x,y);
z = peaks(15);
trisurf(tri,x,y,z)
```

If the surface is in the form of a `TriRep` triangulation representation, plot it as follows:

```
tr = TriRep(tri, x(:), y(:), z(:));
trisurf(tr)
```



See Also

`patch`, `surf`, `tetramesh`, `trimesh`, `triplot`, `delaunay`, `TriRep`, `DelaunayTri`

"Surface and Mesh Creation" on page 1-102 for related functions

Purpose Upper triangular part of matrix

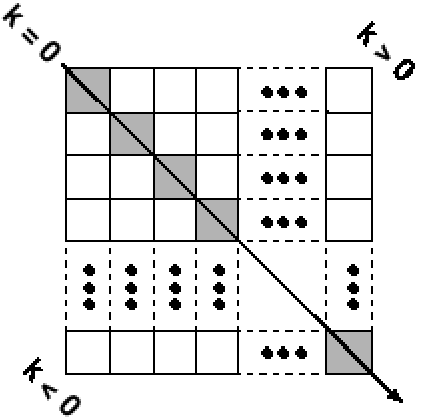
Syntax

```
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 kth diagonal of X. k = 0 is the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the main diagonal.



Examples

```
triu(ones(4,4),-1)

ans =

     1     1     1     1
     1     1     1     1
     0     1     1     1
     0     0     1     1
```

See Also diag, tril

Purpose

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 Execute statements and catch resulting errors

Syntax

```
try
```

Description

`try` marks the beginning of a try-catch statement, a two-part sequence of commands used in detecting and handling errors. The try-catch enables you to bypass default error handling for selected segments of your program code and use your own procedures instead. The two parts of a try-catch statement are a `try` block and a `catch` block (see the figure below). The `try` block begins with the `try` command and ends just before to the `catch` command:

```
try                              try block
   program-code                      |
   program-code                      |
       :                             V
catch exception                 catch block
   error-handling code               |
       :                             |
   rethrow(exception)                V
end
```

The `try` block contains one or more commands for which special error handling is required by your program. Any error detected while executing statements in the `try` block immediately turns program control over to the `catch` block. Code in the `catch` block provides error handling that specifically addresses errors that might originate from statements in the preceding `try` block.

Both the `try` and `catch` blocks may contain additional try-catch statements nested within them.

See in the Programming Fundamentals documentation for more information.

Remarks Specifying the `try`, `catch`, and `end` commands, as well as the commands that make up the `try` and `catch` blocks, on separate lines is recommended. If you combine any of these components on the same line, separate them with commas.

Examples

Example 1

The first part of this example attempts to vertically concatenate two matrices that have an unequal number of columns:

```
A = rand(5,3);   B = rand(5,4);
C = [A; B];
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

Using a try-catch statement, you can provide more information about what went wrong:

```
try
   C = [A; B];
catch exception
   if strcmp(exception.identifier, ...
      'MATLAB:catenate:dimensionMismatch')
      [~, colA] = size(A);   [~, colB] = size(B);
      disp(exception.message);
      fprintf('Matrix A has %d columns while matrix B has %d\n', ..
          colA, colB);
   end
end
```

Running the program displays the following message:

```
CAT arguments dimensions are not consistent.
Matrix A has 3 columns while matrix B has 4
```

Example 2

The `catch` block in this example checks to see if the specified file could not be found. If this is the case, the program allows for the possibility that a common variation of the filename extension (e.g., `jpeg` instead of `jpg`) was used by retrying the operation with a modified extension.

This is done using a `try-catch` statement that is nested within the original `try-catch`.

```
function d_in = read_image(filename)
[path name ext] = fileparts(filename);
try
   fid = fopen(filename, 'r');
   d_in = fread(fid);
catch exception1
   % Get last segment of the error message identifier.
   idSegLast = regexp(exception1.identifier, ...
                     '(?<=:)\w+$', 'match');

   % Did the read fail because the file could not be found?
   if strcmp(idSegLast, 'InvalidFid') && ...
      ~exist(filename, 'file')

      % Yes. Try modifying the filename extension.
      switch ext
      case '.jpg'    % Change jpg to jpeg
          filename = strrep(filename, '.jpg', '.jpeg')
      case '.jpeg'   % Change jpeg to jpg
          filename = strrep(filename, '.jpeg', '.jpg')
      case '.tif'    % Change tif to tiff
          filename = strrep(filename, '.tif', '.tiff')
      case '.tiff'   % Change tiff to tif
          filename = strrep(filename, '.tiff', '.tif')
      otherwise
         fprintf('File %s not found\n', filename);
         rethrow(exception1);
      end

      % Try again, with modifed filenames.
      try
         fid = fopen(filename, 'r');
         d_in = fread(fid);
      catch exception2
```

```
            fprintf('Unable to access file %s\n', filename);
            exception2 = addCause(exception2, exception1);
            rethrow(exception2)
         end
      end
   end
```

See Also

catch, error, assert, MException, throw(MException), rethrow(MException), throwAsCaller(MException), addCause(MException), getReport(MException), last(MException), 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 parameter:

- `Name` — String that specifies the name of this `tscollection` object

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	tscollection name as a string. This can differ from the tscollection name in the MATLAB workspace.
Time	When TimeInfo.StartDate is empty, values are measured relative to 0 . When TimeInfo.StartDate is defined, values represent date strings measured relative to the StartDate. The length of Time must be the same as the first or the last dimension of Data for each collection .
TimeInfo	Contains fields for contextual information about Time: • Units — Time units with any of the following values: 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', 'nanoseconds' • Start — Start time • End — End time (read only) • Increment — Interval between subsequent time values. NaN when times are not uniformly sampled. • Length — Length of the time vector (read only) • Format — String defining the date string display format. See datestr. • StartDate — Date string defining the reference date. See setabstime (tscollection). • UserData — Any additional user-defined information

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
- `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

tsearch will be removed in a future release. Use DelaunayTri/pointLocation instead.

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

DelaunayTri, delaunay, delaunayn, tsearchn

Purpose

N-D closest simplex search

Syntax

```
t = tsearchn(X,TES,XI)
[t,P] = tsearchn(X,TES,XI)
```

Description

`t = tsearchn(X,TES,XI)` returns the indices `t` of the enclosing simplex of the Delaunay tessellation `TES` for each point in `XI`. `X` is an `m`-by-`n` matrix, representing `m` points in N-dimensional space. `XI` is a `p`-by-`n` matrix, representing `p` points in N-dimensional space. `tsearchn` returns `NaN` for all points outside the convex hull of `X`. `tsearchn` requires a tessellation `TES` of the points `X` obtained from `delaunayn`.

`[t,P] = tsearchn(X,TES,XI)` also returns the barycentric coordinate `P` of `XI` in the simplex `TES`. `P` is a `p`-by-`n+1` matrix. Each row of `P` is the Barycentric coordinate of the corresponding point in `XI`. It is useful for interpolation.

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

`DelaunayTri`, `tsearch`

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.

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`	Time-series data, where each data sample corresponds to a specific time. 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`. By default, `NaN`s are used to represent missing or unspecified data. Set the `TreatNaNasMissing` property to determine how missing data is treated in calculations.
`DataInfo`	Contains fields for storing contextual information about `Data`: • `Unit` — String that specifies data units • `Interpolation` — A `tsdata.interpolation` object that specifies the interpolation method for this time series. Fields of the `tsdata.interpolation` object include: - `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

Time Series Object Properties (Continued)

Property	Description
`Events`	An array of `tsdata.event` objects that stores event information for this time series. You add events by using the `addevent` method. Fields of the `tsdata.event` object include the following: • `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. `StartDate` is empty when you have a numerical (non-date-string) time vector.

Time Series Object Properties (Continued)

Property	Description
IsTimeFirst	Logical value (true or false) specifies 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 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. When you set this property to: • true — The first dimension of the data array is aligned with the time vector. For example: `ts=timeseries(rand(3,3),1:3, 'IsTimeFirst',true);` • false — The last dimension of the data array is aligned with the time vector. For example: `ts=timeseries(rand(3,3),1:3, 'IsTimeFirst',false);` After a time series is created, this property is read only.
Name	Time-series name entered as a string. This name can differ from the name of the time-series variable in the MATLAB workspace.
Quality	An integer vector or array containing values -128 to 127 that specifies the quality in terms of codes defined by QualityInfo.Code. 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. 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.

Time Series Object Properties (Continued)

Property	Description
`QualityInfo`	Provides a lookup table that converts numerical `Quality` codes to readable descriptions. `QualityInfo` fields include the following: • `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 Lengths of `Code` and `Description` must match.
`Time`	Array of time values. 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. The length of `Time` must be the same as either the first or the last dimension of `Data`.

Time Series Object Properties (Continued)

Property	Description
TimeInfo	Uses the following fields for storing contextual information about Time: • 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 datestr function reference page for more information. • StartDate — Date string defining the reference date. See the MATLAB setabstime (timeseries) function reference page for more information. • UserData — Stores any additional user-defined information
TreatNaNasMissing	Logical value that specifies how to treat NaN values in Data: • 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 path.

If you do not specify a file 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`, `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*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])
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`

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, uipanels, and uibuttongroups. It cannot contain ActiveX controls.

`uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2,...)` creates a visible container component in the current figure window. This component manages exclusive selection behavior for uicontrols of style `radiobutton` and `togglebutton`.

`handle = uibuttongroup(...)` creates a uibuttongroup object and returns a handle to it in `handle`.

A `uibuttongroup` object can have `axes`, `uicontrol`, `uipanel`, and `uibuttongroup` objects as children. However, only uicontrols of style `radiobutton` and `togglebutton` are managed by the component.

When programming a button group, you do not code callbacks for the individual buttons; instead, use its `SelectionChangeFcn` callback to manage responses to selections. The following example illustrates how you use `uibuttongroup` event data to do this.

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.

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.

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.

Remarks

If you set the `Visible` property of a `uibuttongroup` object to `'off'`, any child objects it contains (buttons, button groups, etc.) become invisible along with the `uibuttongroup` panel itself. However, doing this does *not* affect the settings of the `Visible` property of any of its child objects, even though all of them remain invisible until the button group's visibility is set to `'on'`. `uipanel` components also behave in this manner.

Examples

This example creates a uibuttongroup with three radiobuttons. It manages the radiobuttons with the `SelectionChangeFcn` callback, `selcbk`.

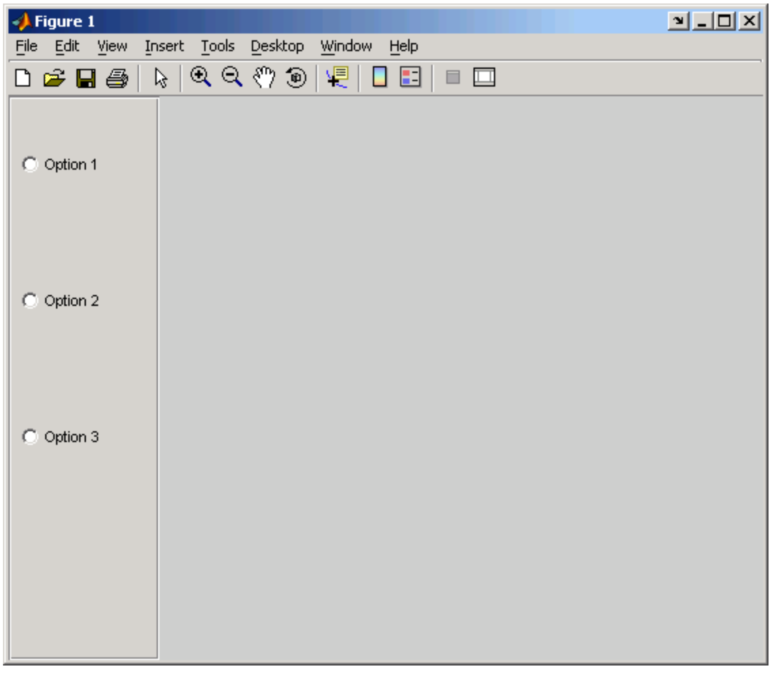
When you select a new radio button, `selcbk` displays the uibuttongroup handle on one line, the `EventName`, `OldValue`, and `NewValue` fields of the event data structure on a second line, and the value of the `SelectedObject` property on a third line.

```
% 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');
```

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`

Purpose

Describe button group properties

Modifying Properties

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

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

Uibuttongroup takes its default property values from uipanel. To set a uibuttongroup default property value, set the default for the corresponding `uipanel` property. Note that you can set no default values for the uibuttongroup `SelectedObject` and `SelectionChangeFcn` properties.

For more information about changing the default value of a property see . 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
`BackgroundColor`	Color of the button group background
`BeingDeleted`	This object is being deleted
`BorderType`	Type of border around the button group
`BorderWidth`	Width of the button group border in pixels
`BusyAction`	Interruption of other callback routines
`ButtonDownFcn`	Button-press callback routine
`Children`	All children of the button group

Property Name	Description
Clipping	Clipping of child axes, panels, and button groups to the button group. Does not affect child user interface controls (uicontrol)
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
Interruptible	Callback routine interruption mode
Parent	uibuttongroup object's parent
Position	Button group position relative to parent figure, panel, or button group
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
SelectionHighlight	Object highlighted when selected

Property Name	Description
`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
`Type`	Object class
`UIContextMenu`	Associate context menu with the button group
`Units`	Units used to interpret the position vector
`UserData`	User-specified data
`Visible`	Button group visibility **Note** Controls the visibility of a uibuttongroup and of its child axes, uibuttongroups. uipanels, and child uicontrols. Setting it does not change their `Visible` property.

`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.

`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.

`BorderType`
`none | {etchedin} | etchedout | beveledin | beveledout | line`

Border of the uibuttongroup area. Used to define the button group area graphically. Etched and beveled borders provide a 3-D look. Use the `HighlightColor` and `ShadowColor` properties to specify the border color of etched and beveled borders. A line border is 2-D. Use the `ForegroundColor` property to specify its color.

`BorderWidth`
integer

Width of the button group border. The width of the button group borders in pixels. The default border width is 1 pixel. 3-D borders wider than 3 may not appear correctly at the corners.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.

- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`ButtonDownFcn`

string or function handle

Button-press callback routine. A callback routine that executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uibuttongroup. This is useful for implementing actions to interactively modify object properties, such as size and position, when they are clicked on (using the `selectmoveresize` function, for example).

If you define this routine as a string, the string can be a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

`Children`

vector of handles

Children of the uibuttongroup. A vector containing the handles of all children of the uibuttongroup. Although a uibuttongroup manages only uicontrols of style `radiobutton` and `togglebutton`, its children can be axes, uipanels, uibuttongroups, and other uicontrols. You can use this property to reorder the children.

`Clipping`
`{on} | off`

Clipping mode. By default, MATLAB clips a uibuttongroup's child axes, uipanels, and uibuttongroups to the uibuttongroup rectangle. If you set `Clipping` to `off`, the axis, uipanel, or uibuttongroup is displayed outside the button group rectangle. This property does not affect child uicontrols which, by default, can display outside the button group rectangle.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uibuttongroup` object. MATLAB sets all property values for the uibuttongroup before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uibuttongroup being created.

Setting this property on an existing uibuttongroup object has no effect.

To define a default `CreateFcn` callback for all new uibuttongroups you must define the same default for all uipanels. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uibuttongroup`. For example, the code

```
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.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See 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`

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.

`Interruptible`
`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the waiting callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine 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');
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 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

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.

Note For GUIDE GUIs, `hObject` contains the handle of the selected radio button or toggle button. See for more information.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`ShadowColor`
`ColorSpec`

3-D frame shadow color. `ShadowColor` is a three-element RGB vector or one of the MATLAB predefined names, specifying the shadow color. See the `ColorSpec` reference page for more information on specifying color.

`Tag`
string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the `Tag` value `'FormatTb'`.

```
h = findobj(figurehandles,'Tag','FormatTb')
```

`Title`
string

Title string. The text displayed in the button group title. You can position the title using the `TitlePosition` property.

If the string value is specified as a cell array of strings or padded string matrix, only the first string 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 . 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 ofthe title. This property determines the location of the title string, in relation to the uibuttongroup.

`Type`
`string` (read-only)

Object class. This property identifies the kind of graphics object. For uibuttongroup objects, `Type` is always the string `'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 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, as are all child objects of the button group. When a button group is hidden in this manner, you can still query and set its properties.

Note The value of a uibuttongroup's `Visible` property determines whether its child components, such as axes, buttons, uipanels, and other uibuttongroups, are visible. However, changing the `Visible` property of a button group does *not* change the settings of the `Visible` property of its child components even though hiding the button group causes them to be hidden.

Purpose

Create context menu

Syntax

`handle = uicontextmenu('PropertyName',PropertyValue,...)`

Description

`handle = uicontextmenu('PropertyName',PropertyValue,...)` 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.

You create context menu items using the `uimenu` function. Menu items appear in the order the `uimenu` statements appear. You associate a context menu with an object using the `UIContextMenu` property for the object and specifying the context menu's handle as the property value.

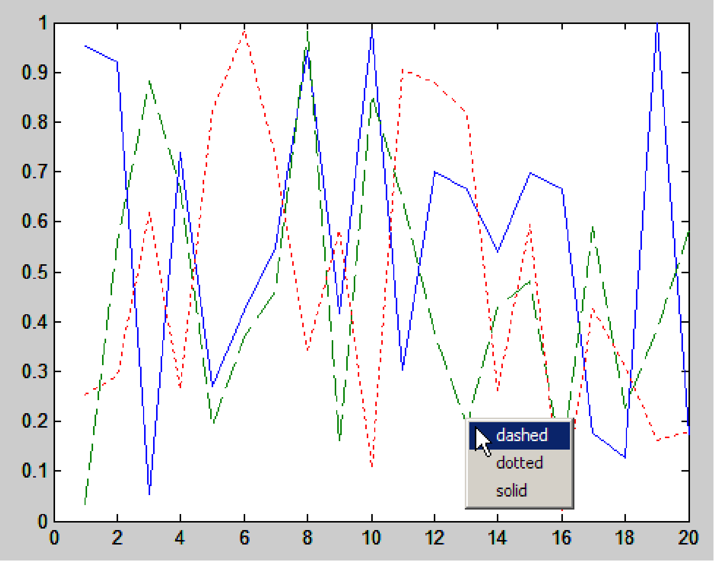
Example

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.

```
% Create axes and save handle
hax = axes;
% Plot three lines
plot(rand(20,3));
% Define a context menu; it is not attached to anything
hcmenu = uicontextmenu;
% Define callbacks for context menu items that change linestyle
hcb1 = ['set(gco, ''LineStyle'', ''--'')'];
hcb2 = ['set(gco, ''LineStyle'', '':'')'];
hcb3 = ['set(gco, ''LineStyle'', ''-'')'];
% Define the context menu items and install their callbacks
item1 = uimenu(hcmenu, 'Label', 'dashed', 'Callback', hcb1);
item2 = uimenu(hcmenu, 'Label', 'dotted', 'Callback', hcb2);
item3 = uimenu(hcmenu, 'Label', 'solid',  'Callback', hcb3);
% Locate line objects
hlines = findall(hax,'Type','line');
% Attach the context menu to each line
for line = 1:length(hlines)
    set(hlines(line),'uicontextmenu',hcmenu)
```

```
end
```

When the user right clicks or presses **Alt+click** on the line, the context menu appears, as shown in this figure:



Generally, you need to attach context menus to lines at the time they are plotted in order to be sure that the menus are available to users at once. Therefore, code such as the above could be placed in or called from the callbacks that perform plotting for the GUI.

You should only define callbacks as strings if they need to perform simple actions. For example, if you wanted to add check marks to menu items (using the `Checked` uimenu property) to indicate the current style for a each line, you should define the menu item callbacks as function handles and place the code for them in the GUI's M-file rather than placing callback strings in the figure.

See Also

in the MATLAB Creating Graphical User Interfaces documentation `uibuttongroup`, `uicontrol`, `uimenu`, `uipanel`

Uicontextmenu Properties

Purpose

Describe context menu 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.

For more information about changing the default value of a property see . For an example, see the `CreateFcn` property.

Uicontext-menu 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
`BeingDeleted`	This object is being deleted
`BusyAction`	Callback routine interruption
`Callback`	Control action
`Children`	The uimenus defined for the uicontextmenu
`CreateFcn`	Callback routine executed during object creation
`DeleteFcn`	Callback routine executed during object deletion
`HandleVisibility`	Whether handle is accessible from command line and GUIs
`Interruptible`	Callback routine interruption mode
`Parent`	Uicontextmenu object's parent

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

`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. 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 `gcbo` to get the handle of the uicontextmenu being created.

Setting this property on an existing uicontextmenu object has no effect.

You can define a default `CreateFcn` callback for all new uicontextmenus. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uicontextmenu`. For example, the code

```
set(0,'DefaultUicontextmenuCreateFcn','set(gcbo,...
    ''Visible'',''on'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new context menu. It sets the default `Visible` property of a context menu.

To override this default and create a context menu whose `Visible` property is set to a different value, call `uicontextmenu` with code similar to

```
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`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See 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 for information on how to use function handles to define a callback function.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from

the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

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

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

`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, which must be a figure.

`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 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

```
handle = uicontrol('PropertyName',PropertyValue,...)
handle = uicontrol(parent,'PropertyName',PropertyValue,...)
handle = uicontrol
uicontrol(uich)
```

Description

`uicontrol` creates a uicontrol graphics objects (user interface controls), which you use to implement graphical user interfaces.

`handle = uicontrol('PropertyName',PropertyValue,...)` creates a uicontrol and assigns the specified properties and values to it. It assigns the default values to any properties you do not specify. The default uicontrol style is a pushbutton. The default parent is the current figure. See the Uicontrol Properties reference page for more information.

`handle = uicontrol(parent,'PropertyName',PropertyValue,...)` creates a uicontrol in the object specified by the handle, `parent`. If you also specify a different value for the `Parent` property, the value of the `Parent` property takes precedence. `parent` can be the handle of a figure, uipanel, or uibuttongroup.

`handle = uicontrol` creates a pushbutton in the current figure. The `uicontrol` function assigns all properties their default values.

`uicontrol(uich)` gives focus to the uicontrol specified by the handle, `uich`.

When selected, most uicontrol objects perform a predefined action. MATLAB software supports numerous styles of uicontrols, each suited for a different purpose:

- 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 and enable users to select one or more items. The `Min` and `Max` properties control the selection mode:

  If `Max-Min>1`, then multiple selection is allowed.

  If `Max-Min<=1`, then only single selection is allowed.

  The `Value` property indicates selected entries and contains the indices into the list of strings; a vector value indicates multiple selections. MATLAB evaluates the list box's callback routine after any mouse button up event that changes the `Value` property. Therefore, you may need to add a "Done" button to delay action caused by multiple clicks on list items.

  List boxes whose `Enable` property is `on` differentiate between single and double left clicks and set the figure `SelectionType` property to `normal` or `open` accordingly before evaluating the list box's `Callback` property. For such list boxes, **Ctrl**-left click and **Shift**-left click also set the figure `SelectionType` property to `normal` or `open` to indicate a single or double click.

- `'popupmenu'` – Pop-up menus (also known as drop-down menus or combo boxes) open to display a list of choices 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.

- '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 mutually exclusive behavior for 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

- Adding a uicontrol to a figure removes the figure toolbar when the figure's `Toolbar` property is set to `'auto'` (which is the default). To prevent this from happening, set the `Toolbar` property to `'figure'`. The user can restore the toolbar by selecting **Figure Toolbar** from the **View** menu regardless of this property setting.
- 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)
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`

Uicontrol Properties

Purpose

Describe user interface control (`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 . 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
`BackgroundColor`	Object background color
`BeingDeleted`	This object is being deleted
`BusyAction`	Callback routine interruption
`ButtonDownFcn`	Button-press callback routine

Property	Purpose
`Callback`	Control action
`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
`Extent`	position rectangle (read only)
`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)

Property	Purpose
`Parent`	Uicontrol object's parent
`Position`	Size and location of uicontrol object
`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.

Note On Solaris 2 systems, setting the background color of a slider has no effect.

`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. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`ButtonDownFcn`

string or function handle (GUIDE sets this property)

Button-press callback routine. A callback routine that 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.

For examples of `Callback` callbacks for each style of component:

- For GUIDE GUIs, see .
- For programmatically created GUIs, see .

Callback routines defined for static text do not execute because no action is associated with these objects.

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**.
- For a multiline editable text box, press **Ctl+Enter**.

`CData`
matrix

Truecolor image displayed on control. A three-dimensional matrix of RGB values that defines a truecolor image displayed on a control, which must be a **push button** or **toggle button**. Each value must be between `0.0` and `1.0`. Setting `CData` on a **radio button** or **checkbox** will replace the default `CData` on these controls. The control will continue to work as expected, but its state is not reflected by its appearance when clicked.

For **push buttons** and **toggle buttons**, `CData` overlaps the `String`. In the case of **radio buttons** and **checkboxes**, `CData` takes precedence over `String` and, depending on its size, it can displace the text.

Setting `CData` to `[]` restores the default `CData` for **radio buttons** and **checkboxes**.

`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`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See 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`.

See 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 `SelectionType` property.
2. Executes the uicontrol `Callback` routine, if any. (Static text components do not use callbacks.)

3. Does *not* set the figure `CurrentPoint` property and does *not* execute either the uicontrol `ButtonDownFcn` or the figure `WindowButtonDownFcn` callback.

Single-clicking or double-clicking an enabled uicontrol with the left mouse button sets the figure `SelectionType` property to `normal`, unless the uicontrol `Style` is `listbox`. For list boxes, double-clicking sets the figure `SelectionType` property to `open` on the second of the two clicks, enabling the list box callback to detect a set of multiple choices.

When you left-click on a uicontrol whose `Enable` property is `off` or `inactive`, or when you right-click a uicontrol whose `Enable` property has any value, MATLAB performs these actions in this order:

1. Sets the figure `SelectionType` property.
2. Sets the figure `CurrentPoint` property.
3. Executes the figure `WindowButtonDownFcn` callback, if provided.
4. Executes the uicontrol `ButtonDownFcn` callback, if provided.

`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 `height` element of the `Extent` property returned always indicates the height of a single line, and its `width` element always indicates the width of the longest line, even if the string wraps when displayed on the control. Edit boxes are considered multiline if `Max` - `Min` > 1.

`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.

Note MATLAB GUIs do not support the `Marlett` and `Symbol` font families.

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.

Tip To determine what fonts exist on your system (which can differ from the GUI user's system), use the `uisetfont` GUI to select a font and return its name and other characteristics in a MATLAB structure.

`FontSize`

size in `FontUnits`

Font size. A number specifying the size of the font in which to display the String, in units determined by the `FontUnits` property. The default point size is system dependent.

`FontUnits`

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

Font size units. This property determines the units used by the `FontSize` property. `Normalized` units interpret `FontSize` as a fraction of the height of the uicontrol. When you resize the uicontrol, MATLAB modifies the screen `FontSize` accordingly.

`pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = $^1/_{72}$ inch).

`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 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.

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.

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'

Event Data Structure Field	Description	Examples:			
		a	=	Shift	Shift/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 for information on how to use function handles to define the callback function.

ListboxTop
scalar

Index of top-most string displayed in list box. This property applies only to the listbox style of uicontrol. It specifies which string appears in the top-most position in a list box that is not large enough to display all list entries. ListboxTop is an index into the array of strings defined by the String property and must have a value between 1 and the number of strings. Noninteger values are fixed to the next lowest integer.

Max
scalar

Maximum value. This property specifies the largest value allowed for the Value property. Different styles of uicontrols interpret Max differently:

- Check boxes – Max is the setting of the Value property while the check box is selected.
- Editable text – The Value property does not apply. If Max - Min > 1, then editable text boxes accept multiline input. If Max - Min <= 1, then editable text boxes accept only single line input. The absolute values of Max and Min have no effect on the number of

lines an edit box can contain; a multiline edit box can contain any number of lines.

- List boxes – If `Max - Min` > 1, then list boxes allow multiple item selection. If `Max - Min` <= 1, then list boxes do not allow multiple item selection. When they do, `Value` can be a vector of indices.
- 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 – The `Value` property does not apply. If `Max - Min` > 1, then editable text boxes accept multiline input. If `Max - Min` <= 1, then editable text boxes accept only single line input. The absolute values of `Max` and `Min` have no effect on the number of lines an edit box can contain; a multiline edit box can contain any number of lines.
- List boxes – If `Max - Min` > 1, then list boxes allow multiple item selection. If `Max - Min` <= 1, then list boxes allow only single item selection. When they do, `Value` can be a vector of indices.
- 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 horizontally, If `height` is greater than `width`, then the slider is oriented vertically.

Note The height of a pop-up menu is determined by the font size. The height you set in the position vector is ignored. The height element of the position vector is not changed.

On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the position vector exceeds this constraint, the displayed height of the slider is the maximum allowed. The height element of the position vector is not changed.

`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]`, and `min_step` should be less than `max_step`.

Numbers outside [0 1] can cause the slider not to render or produce unexpected results. 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. and both should be positive numbers less then 1.

For example, if you create the following slider,

```
uicontrol('Style','slider','Min',1,'Max',7,...
          'Value',2,'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 `num2str`, before assigning it to the `String` property.

For uicontrol objects that display only one line of text (check box, push button, radio button, toggle button), if the string value is specified as a cell array of strings or padded string matrix, only the first string of a cell array or of a padded string matrix is displayed; the rest are ignored. Vertical slash ('`|`') characters are not interpreted as line breaks and instead show up in the text displayed in the uicontrol.

For multiple line editable text or static text controls, line breaks occur between each row of the string matrix, 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 the items in any of the formats shown in the following table.

String Property Format	Example
Cell array of strings	`{'one' 'two' 'three'}`
Padded string matrix	`['one ';'two ';'three']`
String vector separated by vertical slash (`|`) characters	`['one|two|three']`

If you specify a component width that is too small to accommodate one or more of the specified strings, MATLAB truncates those strings with an ellipsis. 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 **push buttons** and **toggle buttons**, `CData` overlaps the `String`. In the case of **radio buttons** and **checkboxes**, `CData` takes precedence over `String` and, depending on its size, can displace the text.

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.

- `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 directory

Syntax

```
uigetdir
directory_name = uigetdir
directory_name = uigetdir(start_path)
directory_name = uigetdir(start_path,dialog_title)
```

Description

`uigetdir` displays a modal dialog box enabling the user to browse through the directory structure and select a directory or type the name of a directory. If the directory exists, `uigetdir` returns the selected path when the user clicks **OK**. For Microsoft Windows platforms, `uigetdir` opens a dialog box in the base directory (the Windows desktop) with the current directory selected. See "Remarks" on page 2-3919 for information about UNIX and Mac platforms.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in the MATLAB Figure Properties.

`directory_name = uigetdir` returns the path to the selected directory when the user clicks **OK**. If the user clicks **Cancel** or closes the dialog window, `directory_name` is set to`0`.

`directory_name = uigetdir(start_path)` opens a dialog box with the directory specified by `start_path` selected. If `start_path` is 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 directory. If `start_path` is not a valid directory path, the dialog box opens in the base directory. For Windows, this is the Windows desktop. See "Remarks" on page 2-3919 for information about UNIX and Mac platforms.

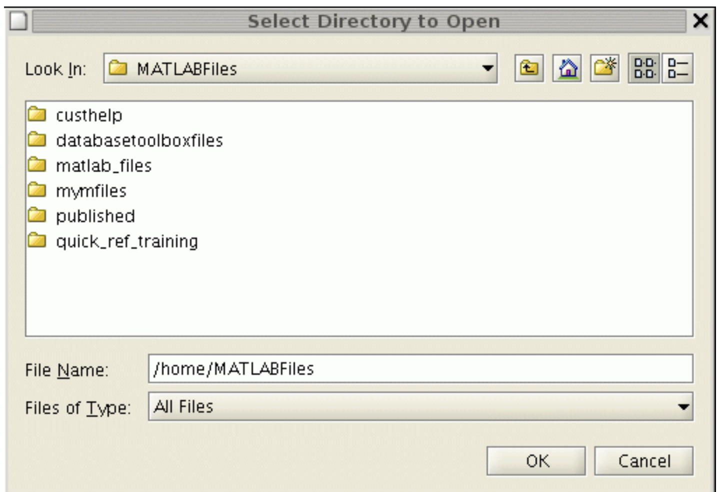
`directory_name = uigetdir(start_path,dialog_title)` opens a dialog box with the specified title. On Windows platforms, the string replaces the default caption inside the dialog box for specifying instructions to the user. The default `dialog_title` is`Select Directory to Open`. See "Remarks" on page 2-3919 for information about UNIX and Mac platforms.

Note On Windows platforms, 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.

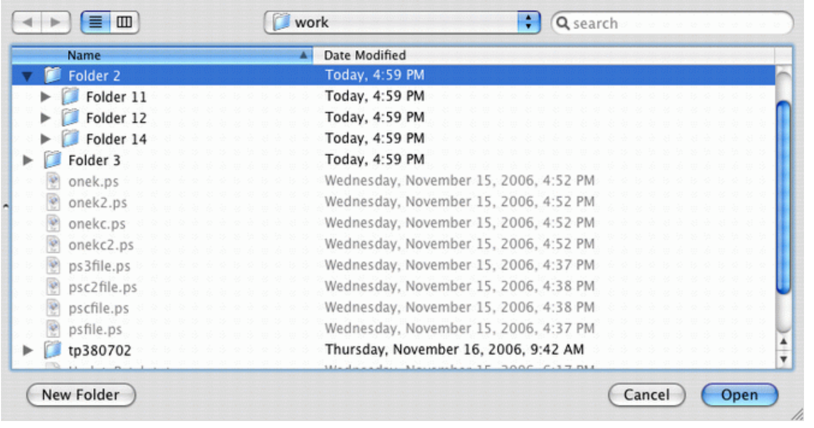
Remarks

For Windows platforms, the dialog box is similar to those shown in the "Examples" on page 2-3920 below.

For UNIX platforms, `uigetdir` opens a dialog box in the base directory (the directory from which MATLAB is started) with the current directory selected. The `dialog_title` string replaces the default title of the dialog box. The dialog box is similar to the one shown in the following figure.



For Mac platforms,`uigetdir` opens a dialog box in the base directory (the current directory) with the current directory open. The `dialog_title` string, if any, is ignored. The dialog box is similar to the one shown in the following figure.



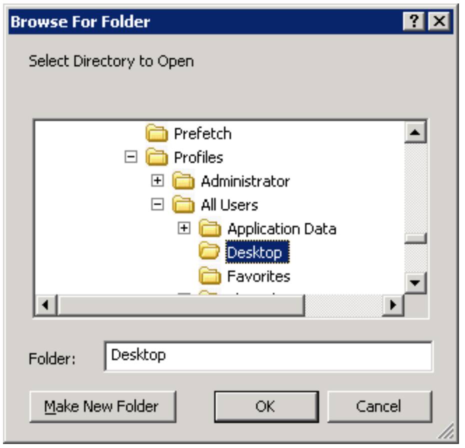
Examples

Example 1

The following statement displays directories on the `C:` drive.

```
dname = uigetdir('C:\');
```

The dialog box is shown in the following figure.



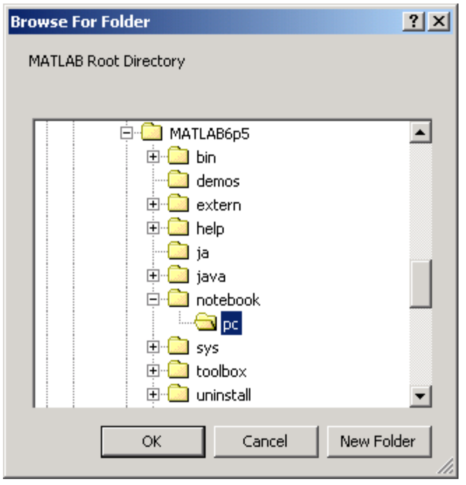
If the user selects the directory `Desktop`, as shown in the figure, and clicks **OK**, `uigetdir` returns

```
dname =
C:\WINNT\Profiles\All Users\Desktop
```

Example 2

The following statement uses the `matlabroot` command to display the MATLAB root directory in the dialog box:

```
uigetdir(matlabroot,'MATLAB Root Directory')
```



If the user selects the directory `MATLAB6.5/notebook/pc`, as shown in the figure, `uigetdir` returns a string like

```
C:\MATLAB6.5\notebook\pc
```

assuming that MATLAB is installed on drive `C:\`.

See Also

`uigetfile`, `uiputfile`

Purpose

Open standard dialog box for retrieving files

Syntax

```
uigetfile
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec)
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,
   DialogTitle)
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,
   DialogTitle,DefaultName)
[FileName,PathName,FilterIndex] = uigetfile(...,'MultiSelect',
   selectmode)
```

Description

`uigetfile` displays a modal dialog box that lists files in the current directory and enables the user to select or type the name of a file to be opened. If the filename is valid and if the file exists, `uigetfile` returns the filename when the user clicks **Open**. Otherwise `uigetfile` displays an appropriate error message from which control returns to the dialog box. The user can then enter another filename or click **Cancel**. If the user clicks **Cancel** or closes the dialog window, `uigetfile` returns `0`.

Successful execution of `uigetfile` does not open a file; it only returns the name of an existing file that the user designates.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution, use the `uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in the MATLAB Figure Properties.

`[FileName,PathName,FilterIndex] = uigetfile(FilterSpec)` displays only those files with extensions that match `FilterSpec`. The `uigetfile` function appends `'All Files'` to the list of file types.`FilterSpec` can be a string or a cell array of strings, and can include the `*` wildcard.

- If `FilterSpec` is a string that contains a filename, the filename is displayed and selected in the **File name** field and the file's extension is used as the default filter.
- If `FilterSpec` is a string, it can include a path. That path can contain `'.'`,`'..'`, or `'/'`. For example, `'../*.m'` lists all M-files in the directory above the current directory.
- If `FilterSpec` is a cell array of strings, the first column contains a list of file extensions. The optional second column contains a corresponding list of descriptions. These descriptions replace standard descriptions in the **Files of type** field. A description cannot be an empty string. "Example 2" on page 2-3928 and "Example 3" on page 2-3929 illustrate use of a cell array as `FilterSpec`.

If `FilterSpec` is not specified, `uigetfile` uses the default list of file types (i.e., all MATLAB files).

After the user clicks **Open** and if the filename exists,`uigetfile` returns the name of the file in `FileName` and its path in `PathName`. If the user clicks **Cancel** or closes the dialog window, `FileName` and `PathName` are set to `0`.

`FilterIndex` is the index of the filter selected in the dialog box. Indexing starts at 1. If the user clicks **Cancel** or closes the dialog window, `FilterIndex` is set to `0`.

`[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle)` displays a dialog box that has the title `DialogTitle`. To use the default file types and specify a dialog title, enter

```
uigetfile('',DialogTitle)
```

`[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle,DefaultName)` displays a dialog box in which the filename specified by `DefaultName` appears in the **File name** field. `DefaultName` can also be a path or a path/filename. In this case, `uigetfile` opens the dialog box in the directory specified by the path. See "Example 6" on page 2-3932 . Note that you can use `'.'`,`'..'`, or `'/'` in the `DefaultName` argument.

If the specified path does not exist, `uigetfile` opens the dialog box in the current directory.

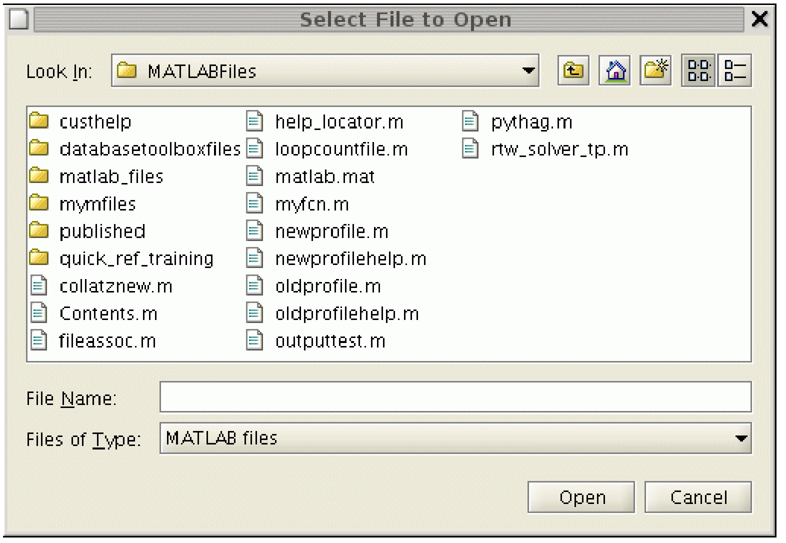
`[FileName,PathName,FilterIndex] = uigetfile(...,'MultiSelect',`*`selectmode`*`)` sets the multiselect mode to specify if multiple file selection is enabled for the `uigetfile` dialog. Valid values for *`selectmode`* are `'on'` and `'off'` (default). If `'MultiSelect'` is `'on'` and the user selects 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. Filenames in the cell array are in the sort order native to your platform. Because multiple selections are always in the same directory, `PathName` is always a string that represents a single directory.

Remarks

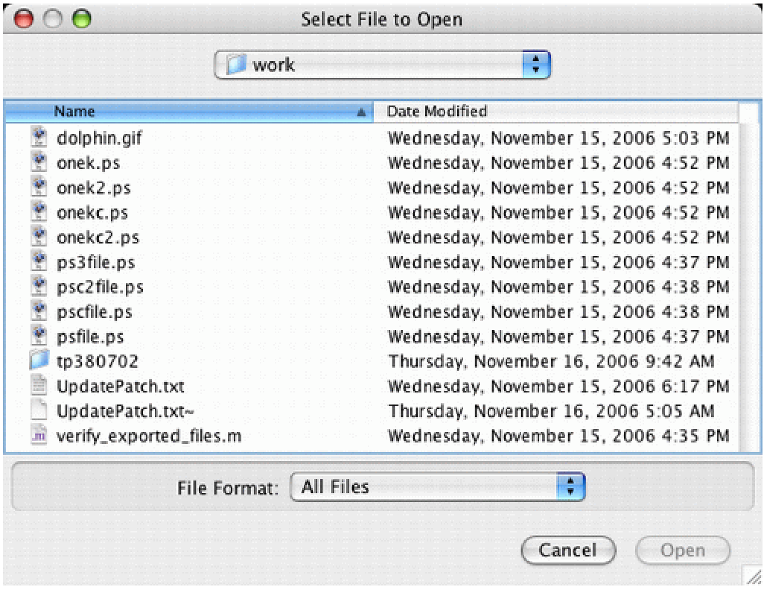
If the user includes either of the "wildcard" characters `'*'` or `'?'` in a file name, `uigetfile` does not respond to clicking **Open**. The dialog box remains open until the user cancels it or removes the wildcard characters. This restriction applies to all platforms, even to file systems that permit these characters in file names.

For Microsoft Windows platforms, the dialog box is the Windows dialog box native to your platform. Because of this, it may differ from those shown in "Examples" on page 2-3927 below.

For UNIX platforms, the dialog box is similar to the one shown in the following figure.



For Mac platforms, the dialog box is similar to the one shown in the following figure.



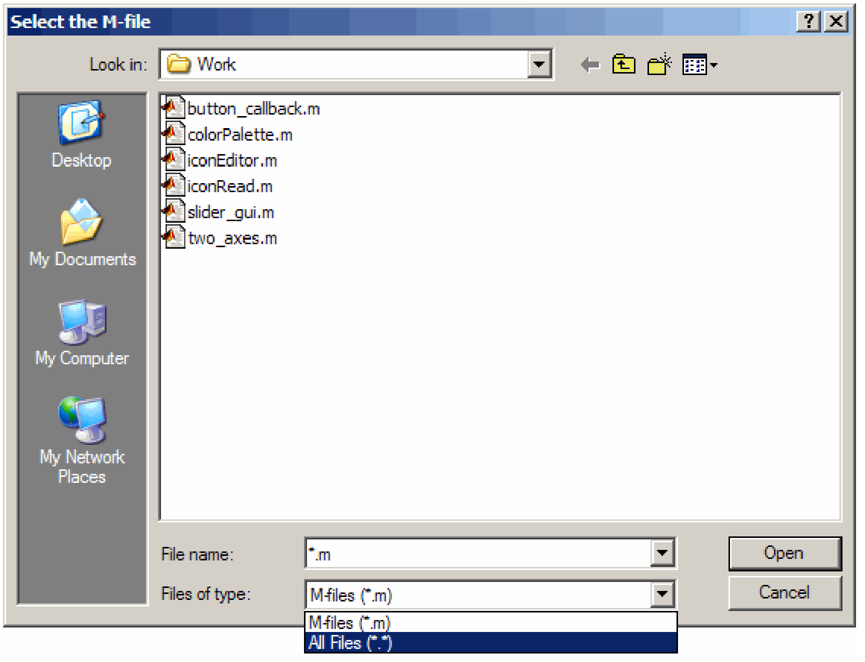
Examples

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');
```

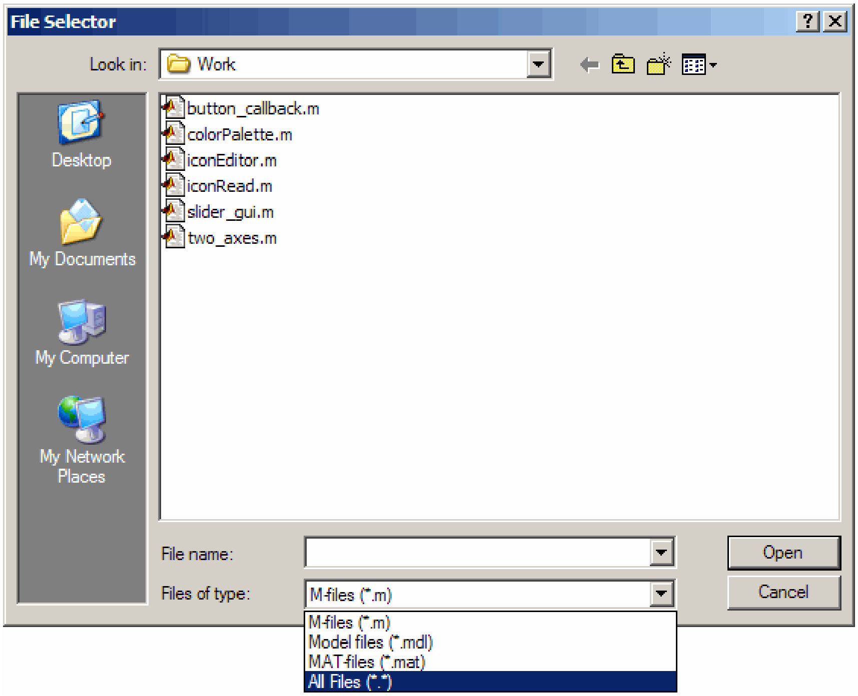
The dialog box is shown in the following figure.



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');
```

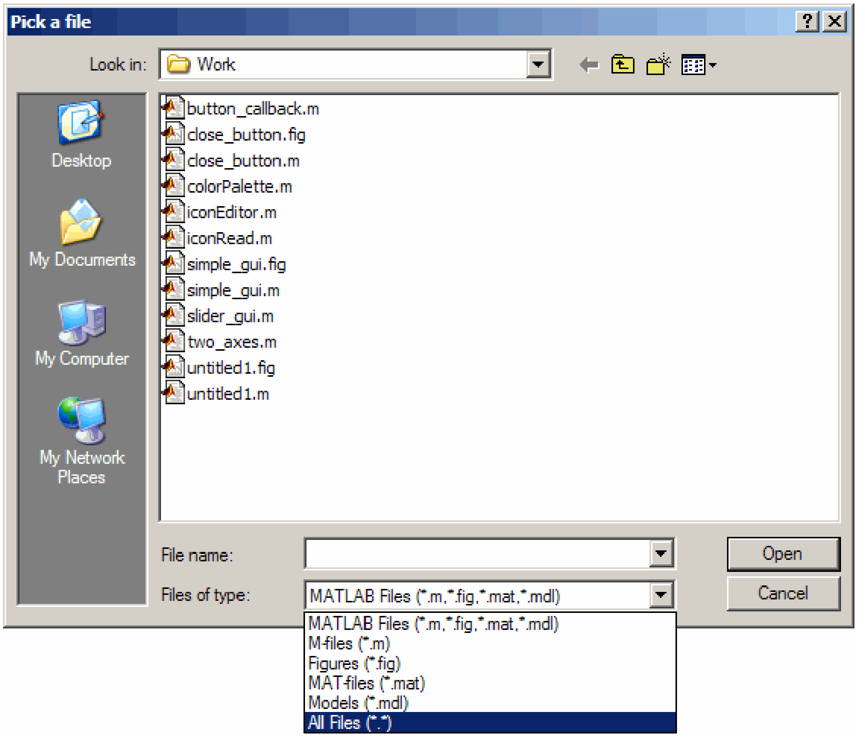


Example 3

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.

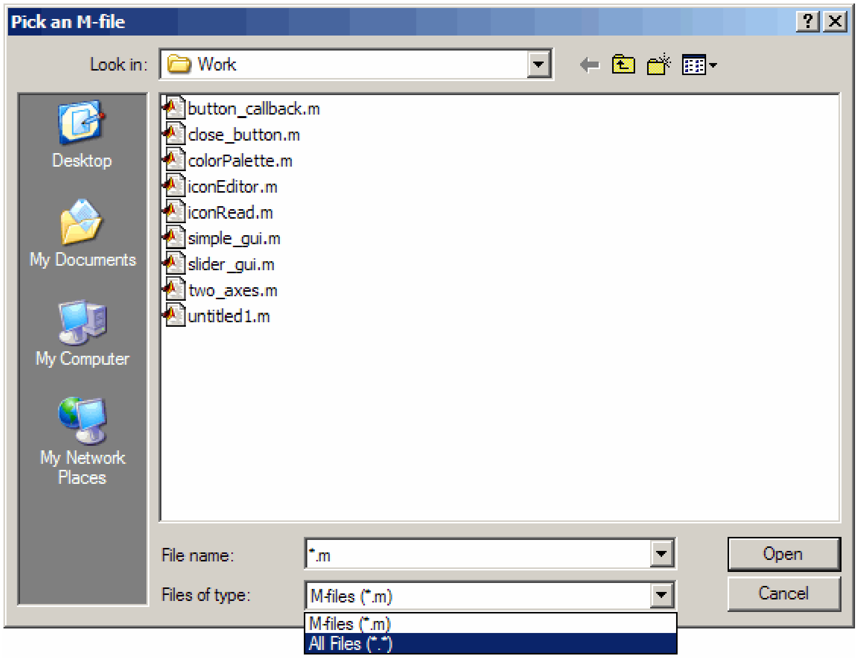


Example 4

The following code checks for the existence of the file and displays a message about the result of the open operation.

```
[filename, pathname] = uigetfile('*.m', 'Pick an M-file');
```

```
if isequal(filename,0)
   disp('User selected Cancel')
else
   disp(['User selected', fullfile(pathname, filename)])
end
```

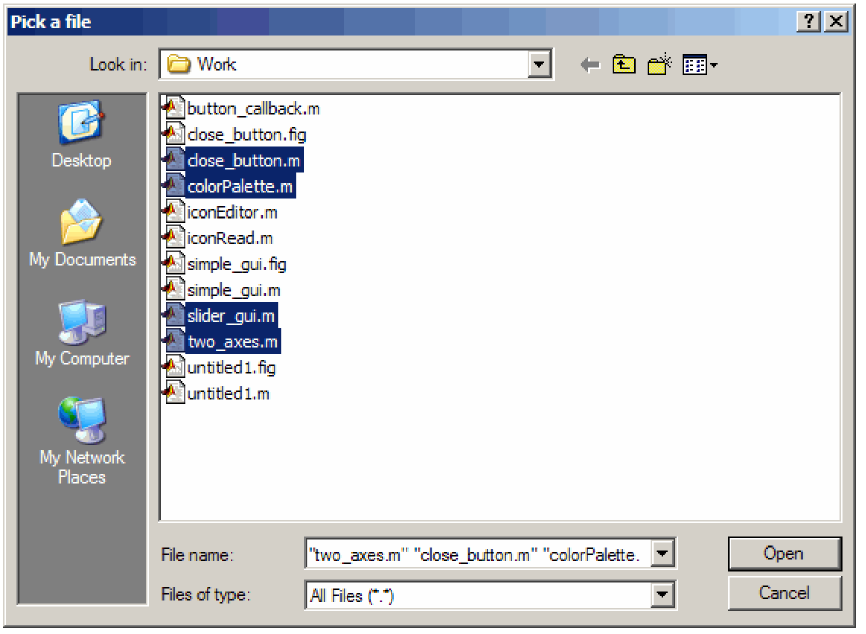


Example 5

This example creates a list of file types and gives them descriptions that are different from the defaults, then enables multiple file selection. The user can select multiple files by holding down the **Shift** or **Ctrl** key and clicking on a file.

```
[filename, pathname, filterindex] = uigetfile( ...
{  '*.mat','MAT-files (*.mat)'; ...
   '*.mdl','Models (*.mdl)'; ...
   '*.*',  'All Files (*.*)'}, ...
```

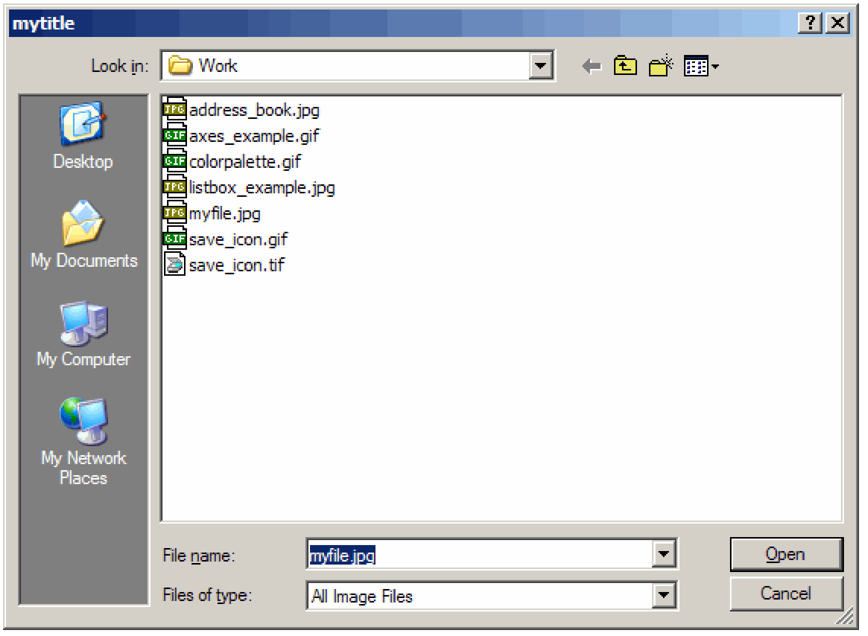
```
'Pick a file', ...
'MultiSelect', 'on');
```



Example 6

This example uses the `DefaultName` argument to specify a start path and a default filename for the dialog box.

```
uigetfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...
          '*.*','All Files' },'mytitle',...
          'C:\Work\myfile.jpg')
```



See Also

`uigetdir`, `uiputfile`

uigetpref

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.

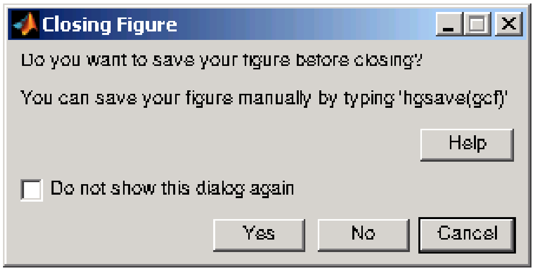
(...'ExtraOptions',eo)creates extra buttons which are not mapped to any preference settings. eo can be a string or a cell array of strings. By default it is {} and no extra buttons are created. If the user chooses one of these buttons, the dialog is closed and the string is returned in value.

(...'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?'
```

```
 ''
 '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

uiimport

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`, `importdata`, `clipboard`, `fileformats`

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 software 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.

Uimenus 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 uimenus on the menu bar (MATLAB controls the built-in menus and their handles are not available to the user). When `MenuBar` is `none`, uimenus are the only items on the menu bar (that is, the built-in menus do not appear).

You can set and query property values after creating the menu using `set` and `get`.

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`, `gcbo`, `set`, `get`, `figure`

Purpose

Describe menu 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
`BeingDeleted`	This object is being deleted
`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:

- 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, `''`.

`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. 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
    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 uimenus are other uimenus, which function as submenus. You can use this property to reorder the menus.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uimenu` object. MATLAB sets all property values for the uimenu before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uimenu being created.

Setting this property on an existing uimenu object has no effect.

You can define a default `CreateFcn` callback for all new uimenus. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uimenu`. For example, the code

```
set(0,'DefaultUimenuCreateFcn','set(gcbo,...
    ''Visible'',''on'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new menu. It sets the default `Visible` property of a `uimenu` object.

To override this default and create a menu whose `Visible` property is set to a different value, call `uimenu` with code similar to

```
hpt = uimenu(...,'CreateFcn','set(gcbo,...
''Visible'',''off'')')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uimenu` call. In the example above, if instead of redefining the `CreateFcn` property for this uimenu, you had explicitly set `Visible` to `off`, the default `CreateFcn` callback would have set `Visible` back to the default, i.e., `on`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See for information on how to use function handles to define a callback function.

`DeleteFcn`

string or function handle

Delete uimenu callback routine. A callback routine that executes when you delete the uimenu object (e.g., when you issue a `delete` command or cause the figure containing the uimenu to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root `CallbackObject` property, which is more simply queried using gcbo.

See 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`, `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.

`Tag`
string

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

`Type`
string (read only)

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

`UserData`
matrix

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

`Visible`
`{on} | off`

Uimenu visibility. By default, all uimenus are visible. When set to `off`, the uimenu is not visible, but still exists and you can query and set its properties.

Purpose

Convert to unsigned integer

Syntax

```
I = uint8(X)
I = uint16(X)
I = uint32(X)
I = uint64(X)
```

Description

I = uint*(X) converts the elements of array X into unsigned integers. X can be any numeric object (such as a double). The results of a uint* operation are shown in the next table.

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.

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.

Note Only the lower order integer data types support math operations. Math operations are not supported for int64 and uint64.

A particularly efficient way to initialize a large array is by specifying the data type (i.e., class name) for the array in the zeros, ones, or eye function. For example, to create a 100-by-100 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 modal 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 A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

Note Only the form `uiopen('LOAD')` can be compiled into a standalone application. You can create a file selection dialog that can be compiled using `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).

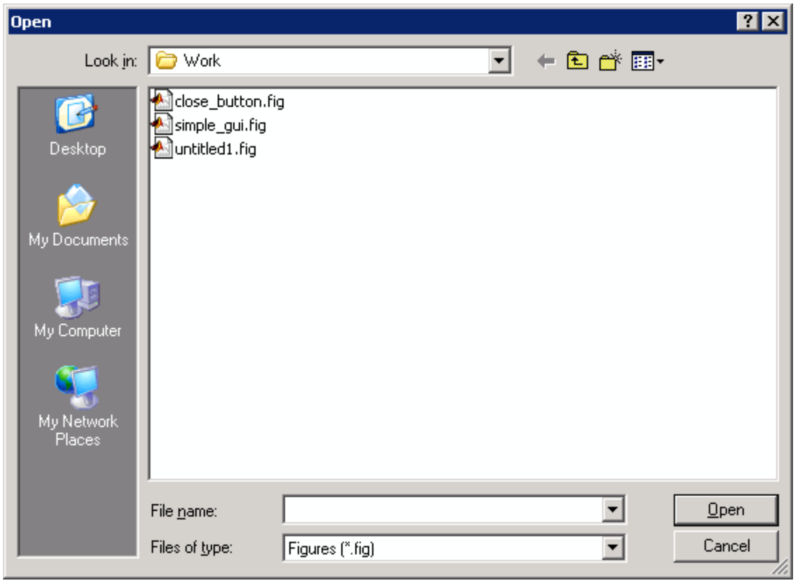
`uiopen('FIGURE')` displays the dialog with the file filter set to figure files (*.fig).

`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.

Examples

Typing `uiopen('figure')` sets the **Files of type** field to `Figures (*.fig)`:



See Also

`uigetfile`, `uiputfile`, `uisave`

Purpose Create panel container object

Syntax

```
h = uipanel('PropertyName1',value1,'PropertyName2',value2,
   ...)
h = uipanel(parent,'PropertyName1',value1,'PropertyName2',
   value2,...)
```

Description A uipanel groups components. It can contain user interface controls with which the user interacts directly. It can also contain axes, other uipanels, and uibuttongroups. It cannot contain ActiveX controls.

`h = uipanel('PropertyName1',value1,'PropertyName2',value2,...)` 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.

`h = uipanel(parent,'PropertyName1',value1,'PropertyName2',value2,...)` 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`.

A `uipanel` object can have `axes`, `uicontrol`, `uipanel`, and `uibuttongroup` objects as children. For the children of a uipanel, the `Position` property is interpreted relative to the uipanel. If you move the panel, the children automatically move with it and maintain their positions relative to the panel.

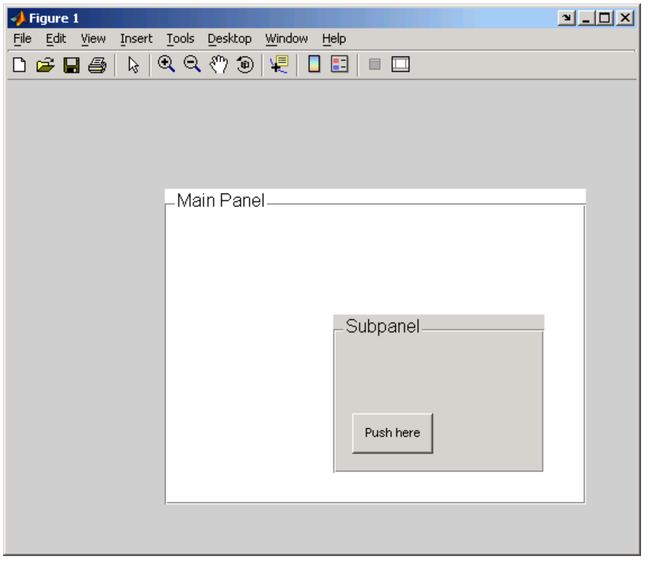
After creating a `uipanel` object, you can set and query its property values using `set` and `get`.

Remarks If you set the `Visible` property of a uipanel object to `'off'`, any child objects it contains (buttons, button groups, axes, etc.) become invisible along with the panel itself. However, doing this does *not* affect the settings of the `Visible` property of any of its child objects, even though all of them remain invisible until the uipanel's visibility is set to `'on'`. `uibuttongroup` components also behave in this manner.

Examples

This example creates a uipanel in a figure, then creates a subpanel in the first panel. Finally, it adds a pushbutton to the subpanel. Both panels use the default `Units` property value, `normalized`. Note that default `Units` for the `uicontrol` pushbutton is pixels.

```
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

Describe panel 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 uipanels also apply to uibuttongroups.

For more information about changing the default value of a property see . 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
`BackgroundColor`	Color of the uipanel background
`BeingDeleted`	This object is being deleted
`BorderType`	Type of border around the uipanel area.

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, uipanels, 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
Interruptible	Callback routine interruption mode
Parent	Uipanel object's parent
Position	Panel position relative to parent figure or uipanel
ResizeFcn	User-specified resize routine
Selected	Whether object is selected

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 panel
Type	Object class
UIContextMenu	Associates uicontextmenu with the uipanel
Units	Units used to interpret the position vector
UserData	User-specified data
Visible	Uipanel visibility. **Note** Controls the visibility of a uipanel and of its child axes, uibuttongroups. uipanels, and child uicontrols. Setting it does not change their `Visible` property.

`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.

`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.

`BorderType`
`none | {etchedin} | etchedout | beveledin | beveledout | line`

Border of the uipanel area. Used to define the panel area graphically. Etched and beveled borders provide a 3-D look. Use the `HighlightColor` and `ShadowColor` properties to specify the border color of etched and beveled borders. A line border is 2-D. Use the `ForegroundColor` property to specify its color.

`BorderWidth`
`integer`

Width of the panel border. The width of the panel borders in pixels. The default border width is 1 pixel. 3-D borders wider than 3 may not appear correctly at the corners.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.

- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`ButtonDownFcn`
string or function handle

Button-press callback routine. A callback routine that executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uipanel. This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using the selectmoveresize function, for example).

If you define this routine as a string, the string can be a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

`Children`
vector of handles

Children of the uipanel. A vector containing the handles of all children of the uipanel. A `uipanel` object's children are axes, uipanels, uibuttongroups, and uicontrols. You can use this property to reorder the children.

`Clipping`
`{on} | off`

Clipping mode. By default, MATLAB clips a uipanel's child axes, uipanels, and uibuttongroups to the uipanel rectangle. If you set `Clipping` to `off`, the axis, uipanel, or uibuttongroup is displayed outside the panel rectangle. This property does not affect child uicontrols which, by default, can display outside the panel rectangle.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uipanel` object. MATLAB sets all property values for the uipanel before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uipanel being created.

Setting this property on an existing uipanel object has no effect.

You can define a default `CreateFcn` callback for all new uipanels. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uipanel`. For example, the code

```
set(0,'DefaultUipanelCreateFcn','set(gcbo,...
    ''FontName'',''arial'',''FontSize'',12)')
```

creates a default `CreateFcn` callback that runs whenever you create a new panel. It sets the default font name and font size of the uipanel title.

Note Uibuttongroup takes its default property values from uipanel. Defining a default property for all uipanels defines the same default property for all uibuttongroups.

To override this default and create a panel whose `FontName and` `FontSize` properties are set to different values, call `uipanel` with code similar to

```
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.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See for information on how to use function handles to define a callback function.

`DeleteFcn`
string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the uipanel object (e.g., when you issue a delete command or clear the figure containing the uipanel). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine. The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

`FontAngle`
`{normal} | italic | oblique`

Character slant used in the Title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

`FontName`
string

Font family used in the Title. The name of the font in which to display the `Title`. To display and print properly, this must be a font that your system supports. The default font is system dependent. To eliminate the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan), set `FontName` to the string `FixedWidth` (this string value is case insensitive).

```
set(uicontrol_handle,'FontName','FixedWidth')
```

This then uses the value of the root `FixedWidthFontName` property which can be set to the appropriate value for a locale from `startup.m` in the end user's environment. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font

`FontSize`
integer

Title font size. A number specifying the size of the font in which to display the Title, in units determined by the `FontUnits` property. The default size is system dependent.

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

Title font size units. `Normalized` units interpret `FontSize` as a fraction of the height of the uipanel. When you resize the uipanel, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`FontWeight`

`light | {normal} | demi | bold`

Weight of characters in the title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to bold causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor`

`ColorSpec`

Color used for title font and 2-D border line. A three-element RGB vector or one of the MATLAB predefined names, specifying the font or line color. See the `ColorSpec` reference page for more information on specifying color.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.

- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

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

`HighlightColor`
`ColorSpec`

3-D frame highlight color. A three-element RGB vector or one of the MATLAB predefined names, specifying the highlight color. See the `ColorSpec` reference page for more information on specifying color.

`Interruptible`
`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing

- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

`Parent`

handle

Uipanel parent. The handle of the uipanel's parent figure, uipanel, or uibuttongroup. You can move a uipanel object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

`Position`

position rectangle

Size and location of uipanel relative to parent. The rectangle defined by this property specifies the size and location of the panel within the parent figure window, uipanel, or uibuttongroup. Specify `Position` as

```
[left bottom width height]
```

`left` and `bottom` are the distance from the lower-left corner of the parent object to the lower-left corner of the `uipanel` object. `width` and `height` are the dimensions of the uipanel rectangle, including the title. All measurements are in units specified by the `Units` property.

`ResizeFcn`

string or function handle

Resize callback routine. MATLAB executes this callback routine whenever a user resizes the uipanel and the figure `Resize` property is set to `on`, or in GUIDE, the Resize behavior option is set to `Other`. You can query the uipanel `Position` property to determine its new size and position. During execution of the callback routine, the handle to the figure being resized is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

All axes, uipanel, uitable and uicontrol objects that have their `Units` set to `normalized` automatically resize proportionally to the figure. You can define individual resize functions for any such object as needed. For example, 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 a uipanel ResizeFcn callback; however, the ResizeFcn is not called again as a result.

A figure's uipanels resize before the figure itself does. Nested uipanels resize from inner to outer, with child ResizeFcns being called before parent ResizeFcns.

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 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

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 . 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.

`Type`

`string` (read-only)

Object class. This property identifies the kind of graphics object. For uipanel objects, `Type` is always the string `'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`.

`Visible`
`{on} | off`

Uipanel visibility. By default, a `uipanel` object is visible. When set to `'off'`, the uipanel is not visible, as are all child objects of the panel. When a panel is hidden in this manner, you can still query and set its properties.

Note The value of a uipanel's `Visible` property determines whether its child components, such as axes, buttons, uibuttongroups, and other uipanels, are visible. However, changing the `Visible` property of a panel does *not* change the settings of the `Visible` property of its child components even though hiding the panel causes them to be hidden.

Purpose

Create push button on toolbar

Syntax

```
hpt = uipushtool
hpt = uipushtool('PropertyName1',value1,'PropertyName2',
   value2,...)
hpt = uipushtool(ht,...)
```

Description

`hpt = uipushtool` creates a push button on the uitoolbar at the top of the current figure window, sets all its properties to default values, and returns a handle to the tool. If no uitoolbar exists, one is created. The uitoolbar is the parent of the uipushtool. Use the returned handle `hpt` to set properties of the tool. The `ClickedCallback` passes the handle as its first argument. The button has no icon, but its border highlights when you hover over it with the mouse cursor. Add an icon by setting `CData` for the tool.

`hpt = uipushtool('`*`PropertyName1`*`',value1,'`*`PropertyName2`*`',value2,...)`, creates a uipushtool and returns a handle to it. `uipushtool` assigns the specified property values, and assigns default values to the remaining properties. You can change the property values at a later time using the `set` function. You can specify properties as parameter name/value pairs, cell arrays containing parameter names and values, or structures with fields containing parameter names and values as input arguments. For a complete list, see Uipushtool Properties. Type `get(hpt)` to see a list of uipushtool object properties and their current values. Type `set(hpt)` to see a list of uipushtool object properties that you can set and their legal property values.

`hpt = uipushtool(ht,...)` creates a button with `ht` as a parent. `ht` must be a uitoolbar handle.

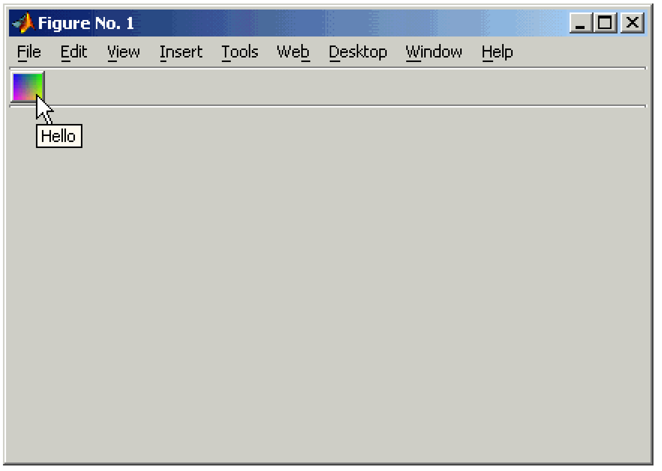
Uipushtools appear in figures whose `Window Style` is `'normal'` or `'docked'`. Push tools do not appear in figures with `'modal'` `WindowStyle`. If you change the `WindowStyle` of a figure containing a uitoolbar and its uipushtool children to `'modal'`, the uipushtools continue to exist as `Children` of the uitoolbar. However, they do not display until you change the figure WindowStyle to 'normal' or 'docked'.

Unlike push buttons, uipushtools have no way to indicate that you have double-clicked them. That is, a double click does not set the figure SelectionType property to 'open'. Double-clicking a uipushtool simply executes its ClickedCallback twice in succession. Also, uipushtools cannot have context menus.

Examples

Create a uitoolbar object and places a uipushtool object on it. Generate an icon for the tool by reading a GIF file containing a MATLAB icon. Convert the indexed image to a truecolor image before specifying it as CData.

```
h  = figure('ToolBar','none');
ht = uitoolbar(h);
% Use a MATLAB icon for the tool
[X map] = imread(fullfile(...
    matlabroot,'toolbox','matlab','icons','matlabicon.gif'));
% Convert indexed image and colormap to truecolor
icon = ind2rgb(X,map);
% Create a uipushtool in the toolbar
hpt = uipushtool(ht,'CData',icon,...
                 'TooltipString','uipushtool',...
                 'ClickedCallback','disp(''Hello World!'')')
```



Alternatives

You can also create toolbars with push tools using GUIDE.

See Also

`get` | `set` | `uicontrol` | `uitoggletool` | `uitoolbar` | Uipushtool Properties

Tutorials

-

How To

-

Uipushtool Properties

Purpose

Describe push tool 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 uitoolbar handle, or a uipushtool handle. *`PropertyName`* is the name of the `Uipushtool` property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see Setting Default Property Values.

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.

Property	Purpose
Enable	Enable or disable the uipushtool.
HandleVisibility	Control access to object's handle.
HitTest	Whether selectable by mouse click
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.
UIContextMenu	Uicontextmenu object associated with the uipushtool
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 software sets the BeingDeleted property to on when the object's delete function callback is called (see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's BeingDeleted property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`CData`
3-dimensional array

Truecolor image displayed on control. An *n*-by-*m*-by-3 array of RGB values that defines a truecolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0. 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.

`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`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See 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 for information on how to use function handles to define a callback function.

`Enable`
`{on} | off`

Enable or disable the uipushtool. This property controls how uipushtools respond to mouse button clicks, including which callback routines execute.

- `on` – The uipushtool is operational (the default).
- `off` – The uipushtool is not operational and its image (set by the `Cdata` property) is grayed out.

When you left-click a uipushtool whose `Enable` property is `on`, MATLAB performs these actions in this order:

1. Executes the push tool's `ClickedCallback` routine.
2. Does *not* set the figure `CurrentPoint` property and does not execute the figure's `WindowButtonDownFcn` callback.
3. Does *not* set the figure `SelectionType` property.

When you left-click a uipushtool whose `Enable` property is `off`, or when you right-click a uipushtool whose `Enable` property has any value, no action is reported, no callback executes, and neither the `SelectionType` nor `CurrentPoint` figure properties are modified.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property.

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

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

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

`HitTest`
`{on} | off`

Selectable by mouse click. This property has no effect on uipushtool objects.

`Interruptible`
`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- 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

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'`.

`UIContextMenu`
handle

Associate a context menu with uicontrol. This property has no effect on uipushtool objects.

`UserData`
array

User specified data. You can specify `UserData` as any array you want to associate with the uipushtool object. The object does not use this data, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Uipushtool visibility. By default, all uipushtools are visible. When set to `off`, the uipushtool is not visible, but still exists and you can query and set its properties.

Purpose

Open standard dialog box for saving files

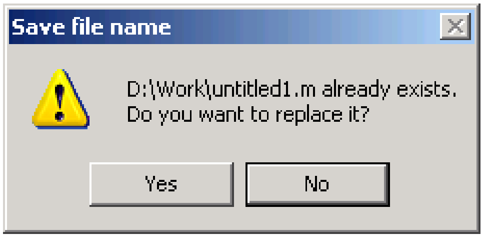
Syntax

```
uiputfile
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec)
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,
   DialogTitle)
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,
   DialogTitle,DefaultName)
```

Description

`uiputfile` displays a modal dialog box used to select or specify a file for saving. The dialog box lists the files and directories in the current directory. If the selected or specified filename is valid, it is returned in `ans`.

If an existing filename is selected or specified, the following warning dialog box is displayed.



The user can select **Yes** to replace the existing file or **No** to return to the dialog to select another filename. If the user selects **Yes**, `uiputfile` returns the name of the file. If the user selects **No**, `uiputfile` returns `0`.

Successful execution of `uiputfile` does not create a file; it only returns the name of a new or existing file that the user designates.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in the MATLAB Figure Properties.

[FileName,PathName,FilterIndex] = uiputfile(FilterSpec) displays only those files with extensions that match FilterSpec. The uiputfile function appends 'All Files' to the list of file types.FilterSpec can be a string or a cell array of strings, and can include the * wildcard. For example, '*.m' lists all the MATLAB M-files.

- If FilterSpec is a string that contains a filename, the filename is displayed and selected in the **File name** field and the file's extension is used as the default filter.
- If FilterSpec is a string, it can include a path. That path can contain '.','..', or '/'. For example, '../*.m' lists all M-files in the directory above the current directory.
- If FilterSpec is a cell array of strings, the first column contains a list of file extensions. The optional second column contains a corresponding list of descriptions. These descriptions replace standard descriptions in the **Save as type** field. A description cannot be an empty string. "Example 3" on page 2-3996 and "Example 4" on page 2-3997 illustrate use of a cell array as FilterSpec.

If FilterSpec is not specified, uiputfile uses the default list of file types (i.e., all MATLAB files).

After the user clicks **Save** and if the filename is valid,uiputfile returns the name of the selected file in FileName and its path in PathName. If the user clicks the **Cancel** button, closes the dialog window, or if the filename is not valid, FileName and PathName are set to 0.

FilterIndex is the index of the filter selected in the dialog box. Indexing starts at 1. If the user clicks the **Cancel** button, closes the dialog window, or if the file does not exist, FilterIndex is set to 0.

If no output arguments are specified, the filename is returned in ans.

[FileName,PathName,FilterIndex] = 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)
```

`[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,DialogTitle,DefaultName)` displays a dialog box in which the filename specified by `DefaultName` appears in the **File name** field. `DefaultName` can also be a path or a path/filename. In this case, `uiputfile` opens the dialog box in the directory specified by the path. See "Example 6" on page 2-3999. Note that you can use `'.'`,`'..'`, or `'/'` in the `DefaultName` argument.

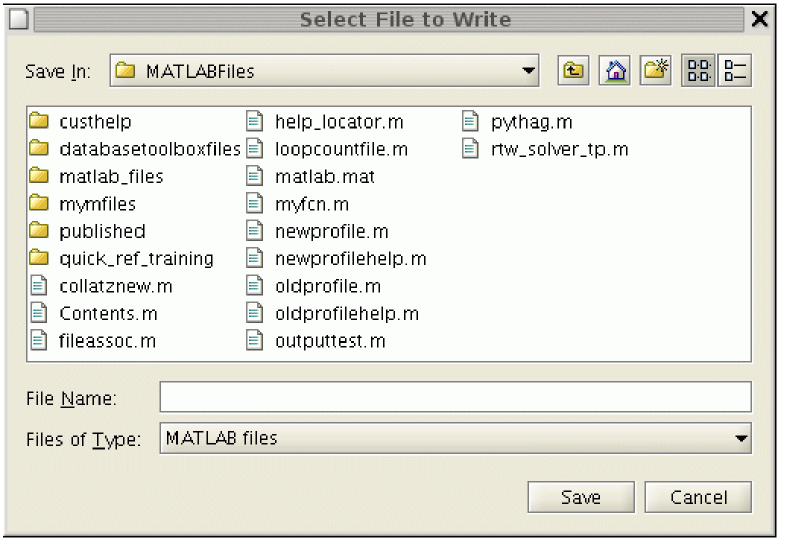
If the specified path does not exist, `uiputfile` opens the dialog box in the current directory.

Remarks

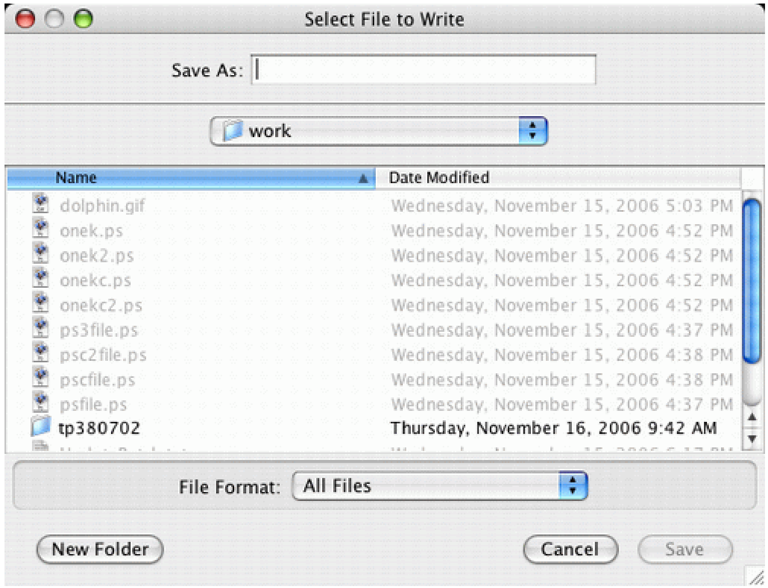
If the user includes either of the "wildcard" characters `'*'` or `'?'` in a file name, `uiputfile` does not respond to clicking **Save**. The dialog box remains open until the user cancels it or removes the wildcard characters. This restriction applies to all platforms, even to file systems that permit these characters in file names.

For Microsoft Windows platforms, the dialog box is the Windows dialog box native to your platform. Because of this, it may differ from those shown in the examples below.

For UNIX platforms, the dialog box is similar to the one shown in the following figure.



For Mac platforms, the dialog box is similar to the one shown in the following figure.

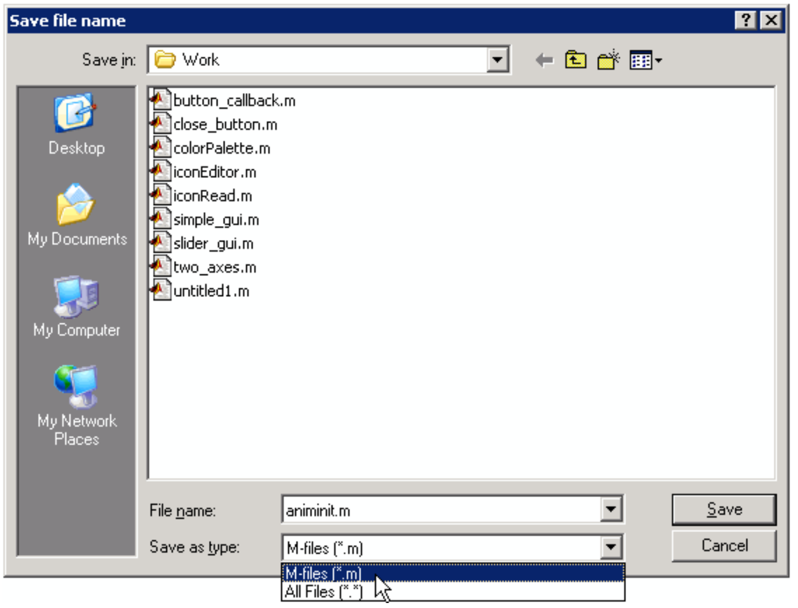


Examples

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 (*.*)`

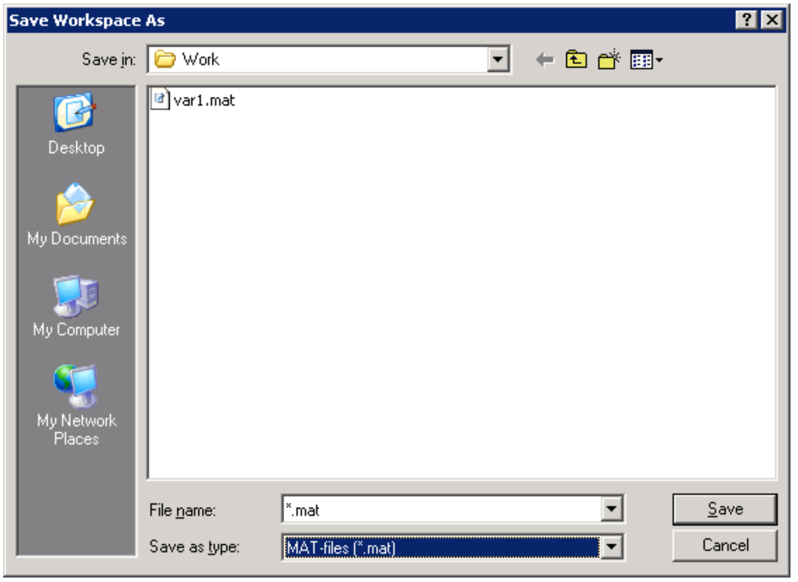
```
[file,path] = uiputfile('animinit.m','Save file name');
```



Example 2

The following statement displays a dialog box titled `'Save Workspace As'` with the filter specifier set to MAT-files.

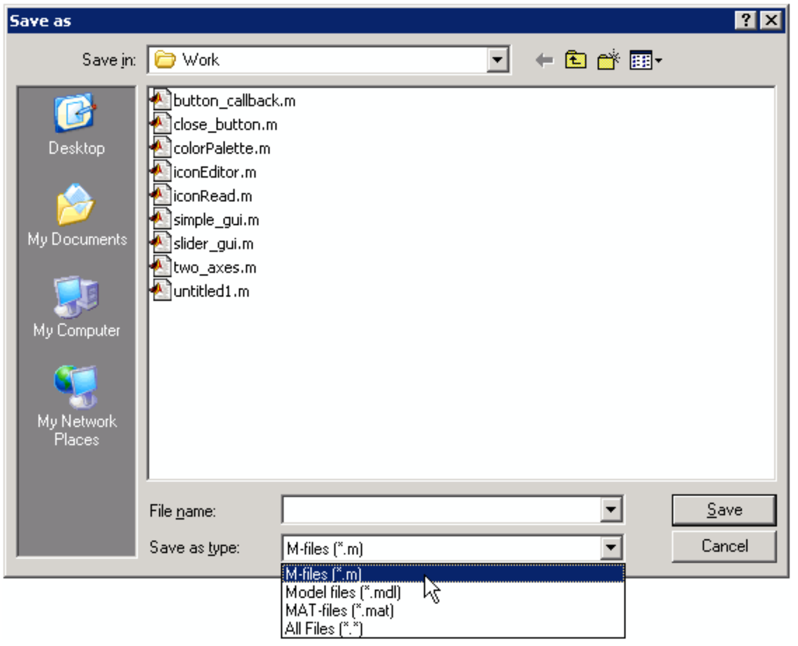
```
[file,path] = uiputfile('*.mat','Save Workspace As');
```



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');
```

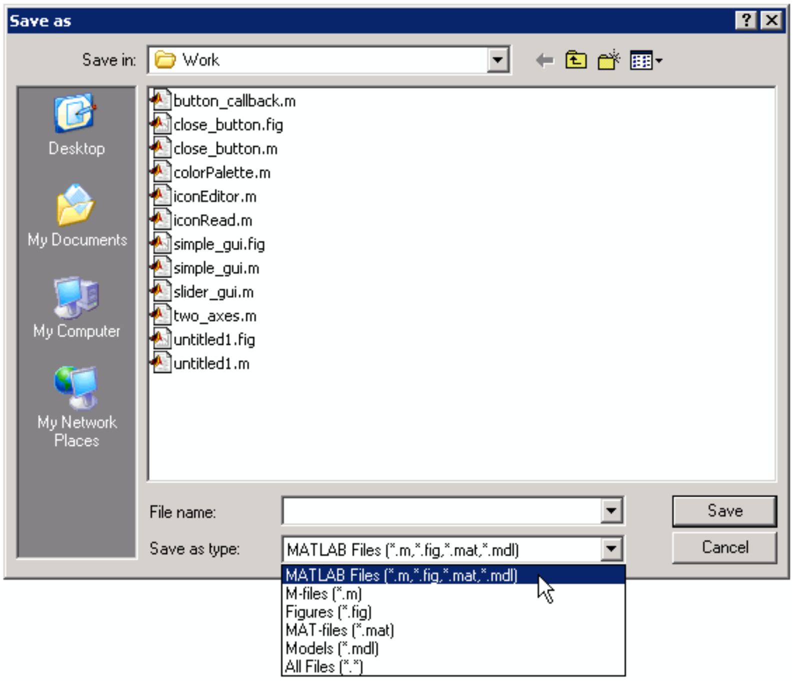


Example 4

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

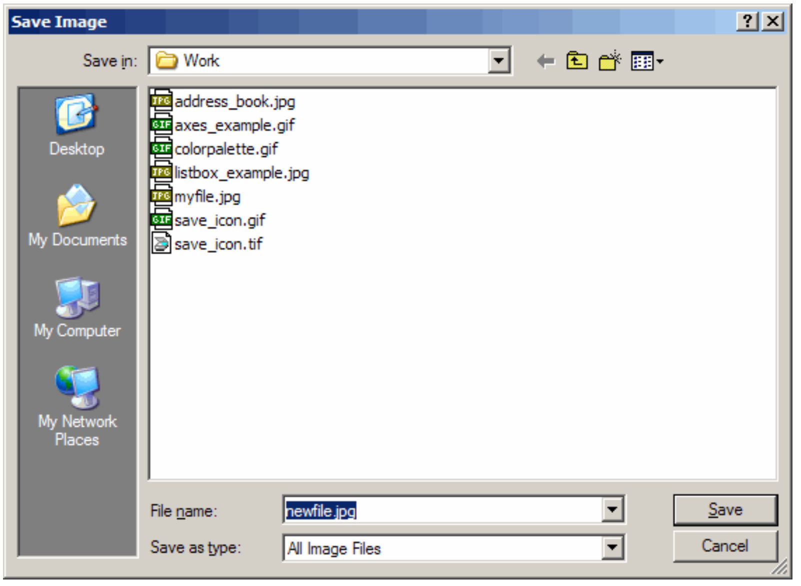
The following code checks for the existence of the file and displays a message about the result of the open operation.

```
[filename, pathname] = uiputfile('*.m','Pick an M-file');
if isequal(filename,0) || isequal(pathname,0)
   disp('User selected Cancel')
else
```

```
    disp(['User selected',fullfile(pathname,filename)])
end
```

Example 6

```
uiputfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...
          '*.*','All Files' },'Save Image',...
          'C:\Work\newfile.jpg')
```



See Also

`uigetdir`, `uigetfile`

Purpose

Resume execution of blocked M-file

Syntax

`uiresume(h)`

Description

`uiresume(h)` resumes the M-file execution that `uiwait` suspended.

Remarks

The `uiwait` and `uiresume` functions block and resume MATLAB program execution. 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.

When used in conjunction with a modal dialog, `uiresume` can resume the execution of the M-file that `uiwait` suspended while presenting a dialog box.

Example

This example creates a GUI with a **Continue** push button. The example calls `uiwait` to block MATLAB execution until `uiresume` is called. This happens when the user clicks the **Continue** push button because the push button's `Callback` callback, which responds to the click, calls `uiresume`.

```
f = figure;
h = uicontrol('Position',[20 20 200 40],'String','Continue',...
              'Callback','uiresume(gcbf)');
disp('This will print immediately');
uiwait(gcf);
disp('This will print after you click Continue');
close(f);
```

`gcbf` is the handle of the figure that contains the object whose callback is executing.

is a more complex example for a GUIDE GUI. See for an example for a programmatically created GUI.

See Also

`dialog`, `figure`, `uicontrol`, `uimenu`, `uiwait`, `waitfor`

Purpose

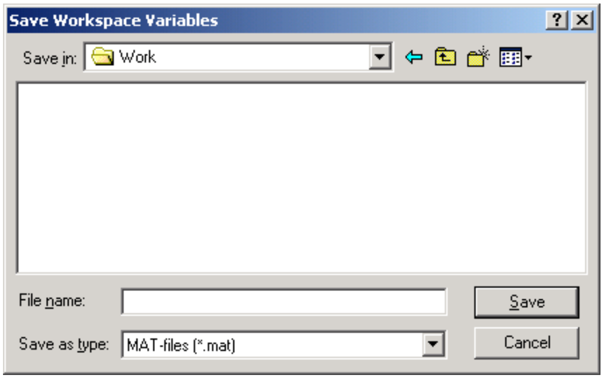
Open standard dialog box for saving workspace variables

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. By default, the dialog box opens in your current directory.



Note The `uisave` dialog box is modal. A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

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 file `my_vars.mat`. The default filename is `matlab.mat`.

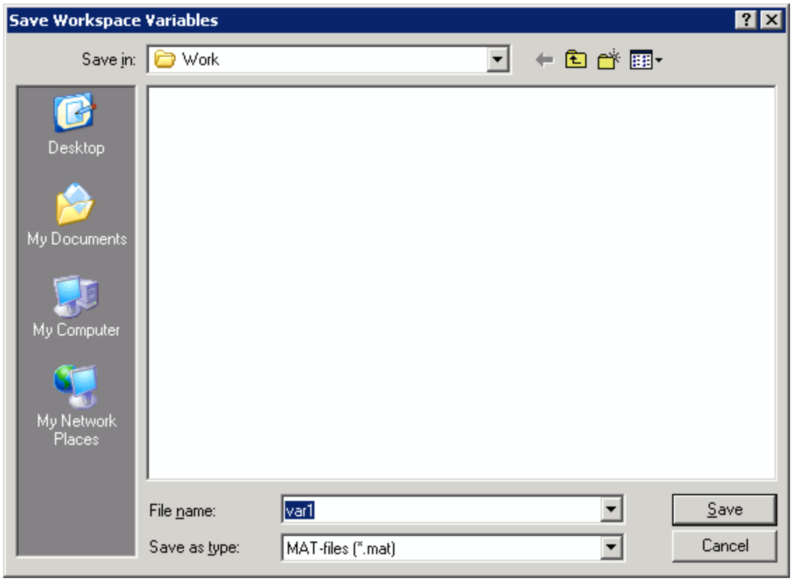
`uisave(variables)` saves only the variables listed in `variables`. For a single variable, `variables` can be a string. For more than one variable, `variables` must be a cell array of strings.

`uisave(variables,filename)` uses the specified `filename` as the default **File name** in the Save Workspace Variables dialog box.

Example

This example creates workspace variables `h` and `g`, and then displays the Save Workspace Variables dialog box in the current directory with the default **File name** set to `var1`.

```
h = 365;
g = 52;
uisave({'h','g'},'var1');
```



Clicking **Save** stores the workspace variables `h` and `g` in the file `var1.mat` in the displayed directory.

See Also

`uigetfile`, `uiputfile`, `uiopen`

Purpose Open standard dialog box for setting object's `ColorSpec`

Syntax

```
c = uisetcolor
c = uisetcolor([r g b])
c = uisetcolor(h)
c = uisetcolor(...,'dialogTitle')
```

Description

`c = uisetcolor` displays a modal color selection dialog appropriate to the platform, and returns the color selected by the user. The dialog box is initialized to `white`.

`c = uisetcolor([r g b])` displays a dialog box 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 = uisetcolor(h)` displays a dialog box 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 = uisetcolor(...,'dialogTitle')` displays a dialog box 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.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

See Also `ColorSpec`

uisetfont

Purpose

Open standard dialog box for setting object's font characteristics

Syntax

```
uisetfont
uisetfont(h)
uisetfont(S)
uisetfont(...,'DialogTitle')
S = uisetfont(...)
```

Description

`uisetfont` enables you to change font properties (`FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle`) for a `text`, `axes`, or `uicontrol` object. The function returns a structure consisting of font properties and values. You can specify an alternate title for the dialog box.

`uisetfont` displays a modal dialog box and returns the selected font properties.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`uisetfont(h)` displays a modal dialog box, initializing the font property values with the values of those properties for the object whose handle is `h`. Selected font property values are applied to the current object. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont(S)` displays a modal dialog box, initializing the font property values with the values defined for the specified structure (`S`). `S` must define legal values for one or more of these properties: `FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle` and the field names must match the property names exactly. If other properties are defined, they are ignored. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont(...,'DialogTitle')` displays a modal dialog box with the title `DialogTitle` and returns the values of the font properties selected in the dialog box.

`S = uisetfont(...)` returns the properties `FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle` as fields in a structure. If the user presses **Cancel** from the dialog box or if an error occurs, the output value is set to 0.

Example

These statements create a text object, then display a dialog box (labeled Update Font) that enables you to change the font characteristics:

```
h = text(.5,.5,'Figure Annotation');
uisetfont(h,'Update Font')
```

These statements create two push buttons, then set the font properties of one based on the values set for the other:

```
% Create push button with string ABC
c1 = uicontrol('Style', 'pushbutton', ...
     'Position', [10 10 100 20], 'String', 'ABC');
% Create push button with string XYZ
c2 = uicontrol('Style', 'pushbutton', ...
     'Position', [10 50 100 20], 'String', 'XYZ');
% Display set font dialog box for c1, make selections,
& and save to d
d = uisetfont(c1);
% Apply those settings to c2
set(c2, d)
```

See Also

`axes`, `text`, `uicontrol`

uisetpref

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`

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 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)
```

uitable

Purpose

Create 2-D graphic table GUI component

Syntax

```
uitable
uitable('PropertyName1', value1,'PropertyName2',value2,...)
uitable(parent,...)
handle = uitable(...)
```

Description

`uitable` creates a 1-by-1 `uitable` object in the current figure window, using default property values. If no figure exists, a new figure window opens.

`uitable('`*`PropertyName1`*`', value1,'`*`PropertyName2`*`',value2,...)` creates a `uitable` object with specified property values. Properties that you do not specify assume the default property values. See the Uitable Properties reference page for information about the available properties.

`uitable(parent,...)` creates a `uitable` object as a child of the specified parent handle `parent`. The parent can be a figure or `uipanel` handle. If you also specify a different value for the `Parent` property, the value of the `Parent` property takes precedence.

`handle = uitable(...)` creates a `uitable` object and returns its handle.

After creating a `uitable` object, you can set and query its property values using the `set` and `get` functions.

Remarks

Users can change values in a table if the `ColumnEditable` property is true for the column they attempt to edit. By default, this property is false for all columns. If the column contains pop-up choices, only the current choice is visible (and not the pop-up menu control) when its column cannot be edited.

After editing a value, the edited value is displayed and the `CellEditCallback` fires when the user does any of the following:

- Types **Enter**
- Clicks another table cell

- Clicks anywhere else within the table
- Clicks another control or area within the same figure window
- Clicks another window, then clicks again on the GUI containing the table (or uses **Alt+Tab** to do this) and performs any of the above four actions

When the `CellEditCallback` callback executes, the underlying data matrix (the table's `Data` property) contains the value that is displayed in the cell.

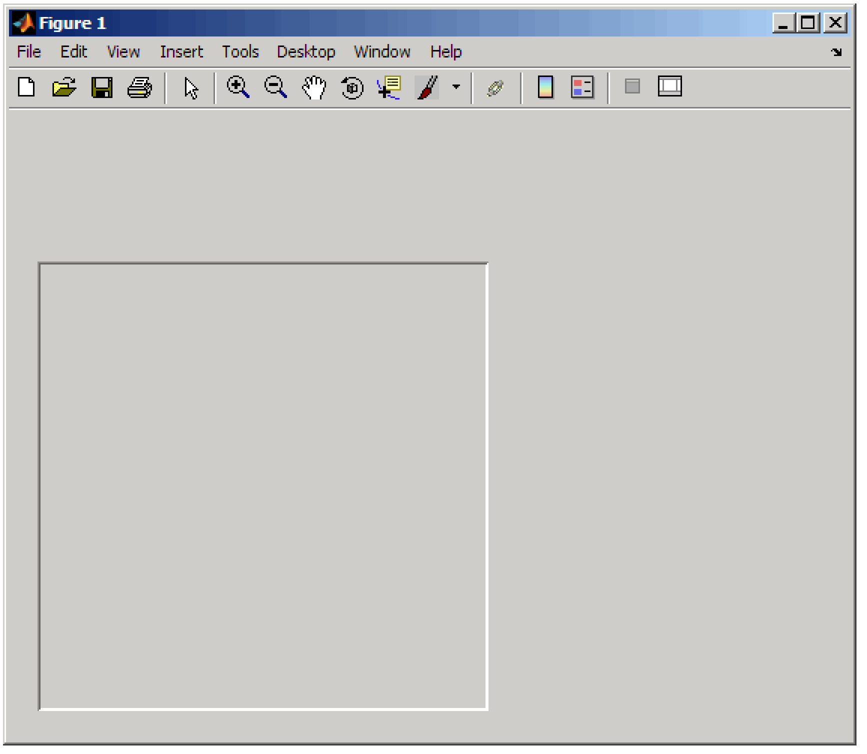
Note If you attempt to create a uitable when running MATLAB on a Linux system without a Java virtual machine (`matlab -nojvm`) or without a display (`matlab  nodisplay`), no table generates and you receive an error.

Examples

Example 1

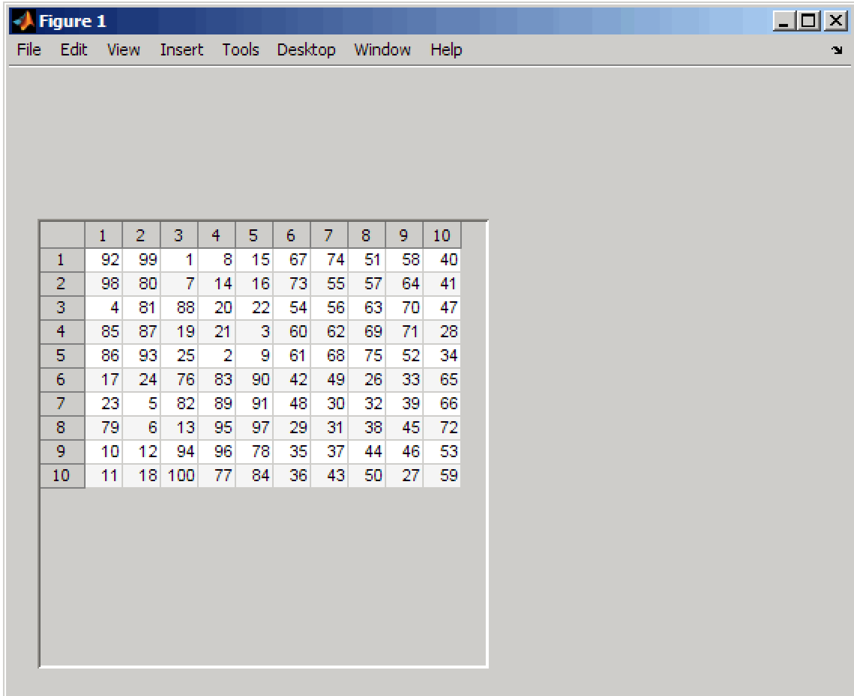
This example creates a table in the current figure. If no figure exists, one is created.

```
t = uitable;
```



As the table has no content (its `Data` property is empty), it initially displays no rows or columns. Provide data (a magic square), and set column widths to 25 pixels uniformly to make the entire table visible.

```
set(t,'Data',magic(10))
set(t,'ColumnWidth',{25})
```



The uitable `ColumnWidth` property is specified as a cell array. It can contain:

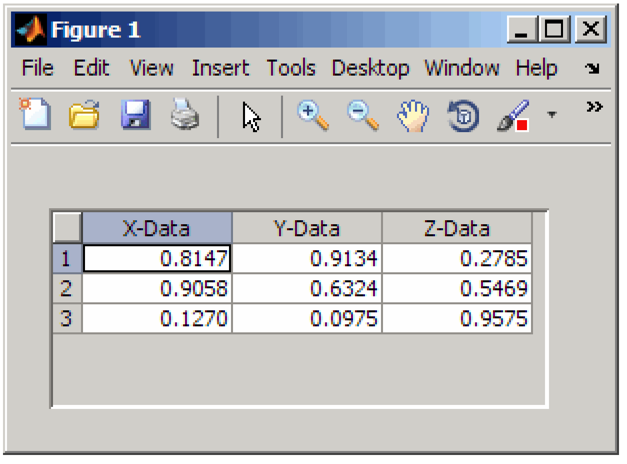
- One number (a width measured in pixels) or the string `'auto'`
- A cell array containing a list of pixel sizes having up to as many entries as the table has columns

If a list has `n` entries, where `n` is smaller than the number of columns, it sets the first `n` column widths only. You can substitute `'auto'` for any value in the cell array to have the width of that column calculated automatically.

Example 2

This example creates a table with a 3-by-3 data matrix. This example specifies the column names, parent, and position of the table:

```
f = figure('Position',[100 100 300 150]);
dat = rand(3);
cnames = {'X-Data','Y-Data','Z-Data'};
t = uitable('Data',dat,'ColumnName',cnames,...
            'Parent',f,'Position',[20 20 250 100]);
```

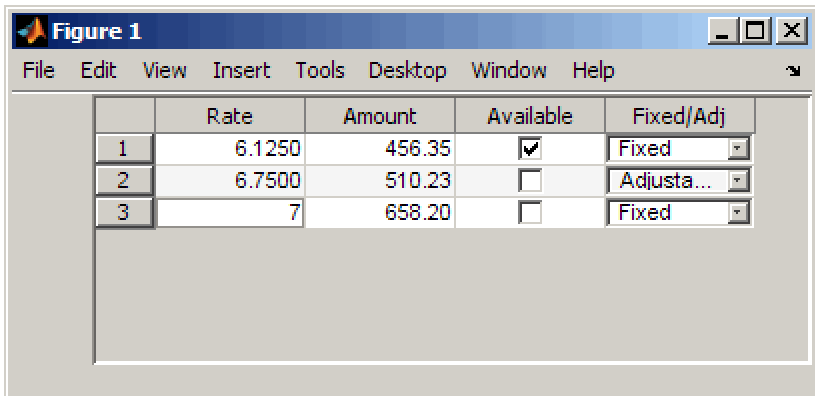


Example 3

This example creates a table with a 3-by-4 array that contains numeric, logical, and string data in the following columns:

- First column (**Rate**): Numeric, with three decimals (not editable)
- Second column (**Amount**): Currency (not editable)
- Third column (**Available**): Check box (editable)
- Fourth column (**Fixed/Adj**): Pop-up menu with two choices: `Fixed` and `Adjustable` (editable)

```
f = figure('Position',[100 100 400 150]);
dat =  {6.125, 456.3457, true,  'Fixed';...
        6.75,  510.2342, false, 'Adjustable';...
        7,     658.2,    false, 'Fixed';};
columnname =   {'Rate', 'Amount', 'Available', 'Fixed/Adj'};
columnformat = {'numeric', 'bank', [], {'Fixed' 'Adjustable'}};
columneditable =  [false false true true];
t = uitable('Units','normalized','Position',...
            [0.1 0.1 0.9 0.9], 'Data', dat,...
            'ColumnName', columnname,...
            'ColumnFormat', columnformat,...
            'ColumnEditable', columneditable);
```



For more information about working with uitables, see the following examples in the MATLAB Creating Graphical User Interfaces documentation:

- (GUIDE example)
- (programmatic example)

See Also

`figure`, `format`, `inspect`, `uicontrol`, `uimenu`, `uipanel`

Uitable Properties

Purpose

Describe table 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 uitable properties by typing:

```
set(h,'DefaultUitablePropertyName',PropertyValue...)
```

Where `h` can be the root handle (`0`), a figure handle, or a uitable handle. *`PropertyName`* is the name of the uitable 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 . For an example, see the `CreateFcn` property.

Uitable Properties

This section lists all properties useful to `uitable` objects along with valid values and descriptions of their use. In the property descriptions, curly braces { } enclose default values.

Property Name	Description
`BackgroundColor`	Background color of cells.
`BeingDeleted`	This object is being deleted.
`BusyAction`	Callback routine interruption
`ButtonDownFcn`	Button-press callback routine
`CellEditCallback`	Callback when data in a cell is changed.

Property Name	Description
CellSelectionCallback	Callback when cell is selected
Children	uitable objects have no children
Clipping	Does not apply to uitable objects
ColumnEditable	Determines data in a column as editable
ColumnFormat	Determines display and editablility of columns
ColumnName	Column header label
ColumnWidth	Width of each column in pixels
CreateFcn	Callback routine during object creation
Data	Table data
DeleteFcn	Callback routine during object deletion
Enable	Enable or disable the uitable
Extent	Size of uitable rectangle
FontAngle	Character slant of cell content
FontName	Font family for cell content
FontSize	Font size of cell content
FontUnits	Font size units for cell content
FontWeight	Weight of cell text characters
ForegroundColor	Color of text in cells
HandleVisibility	Control access to object's handle
HitTest	Selectable by mouse click
Interruptible	Callback routine interruption mode
KeyPressFcn	Key press callback function
Parent	uitable parent

Property Name	Description
`Position`	Size and location of `uitable`
`RearrangeableColumn`	Location of the column
`RowName`	Row header label names
`RowStriping`	Color striping of label rows
`Selected`	Is object selected?
`SelectionHighlight`	Object highlight when selected
`Tag`	Use-specified object label
`TooltipString`	Content of tooltip for object
`Type`	Class of graphics object
`UIContextMenu`	Associate context menu with `uitable`
`Units`	Units of measurement
`UserData`	User-specified data
`Visible`	`uitable` visibility

`BackgroundColor`

`1-by-3 or 2-by-3 matrix of RGB triples`

Cell background color. Color used to fill the `uitable` cells. Specify as an 1-by-3 or 2-by-3 matrix of RGB triples, such as `[.8 .9. .8]` or `[1 1 .9; .9 1 1]`. Each row is an RGB triplet of real numbers between 0.0 and 1.0 that defines one color. (Color names are not allowed.) The default is a 1-by-3 matrix of platform-dependent colors. See `ColorSpec` for information about RGB colors.

Row 2 of the matrix is used only if the `RowStriping` property is `on`. The table background is not striped unless both `RowStriping` is on and the `BackgroundColor` color matrix has two rows.

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

For example, 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 new 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 `DeleteFcn` or `CreateFcn` 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 `uitable`. Specifically:

- If the `uitable` `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 `uitable` or when you click the right mouse button on the control itself.
- If the `uitable` `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 the `selectmoveresize` function, 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.

`CellEditCallback`

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

Callback to edit user-entered data

Callback function executed when the user modifies a table cell. It can perform evaluations, validations, or other customizations. If this function is called as a function handle, `uitable` passes it two arguments. The first argument, `source`, is the handle of the `uitable`. The second argument, `eventdata`, is an event data structure that contains the fields shown in the following table. All fields in the event data structure are read only.

Event Data Structure Field	Type	Description
`Indices`	1-by-2 matrix	Row index and column index of the cell the user edited.
`PreviousData`	1-by-1 matrix or cell array	Previous data for the changed cell. The default is an empty matrix, `[]`.
`EditData`	String	User-entered string.

Event Data Structure Field	Type	Description
`NewData`	1-by-1 matrix or cell array	Value that `uitable` wrote to `Data`. It is either the same as `EditData` or a converted value, for example, 2 where `EditData` is `'2'` and the cell is numeric. Empty if `uitable` detected an error in the user-entered data and did not write it to `Data`.
`Error`	String	Error that occurred when `uitable` tried to convert the `EditData` string into a value appropriate for `Data`. For example, `uitable` could not convert the `EditData` string consistent with the `Column Format` property, if any, or the data type for the changed cell. Empty if `uitable` wrote the value to `Data`. If `Error` is not empty, the `CellEditCallback` can pass the error string to the user or can attempt to manipulate the data. For example, the string `'pi'` would raise an error in a numeric cell but the `CellEditCallback` could convert it to its numerical equivalent and store it in `Data` without passing the error to the user.

When a user edits a cell, `uitable` first attempts to store the user-entered value in `Data`, converting the value if necessary. It then calls the `CellEditCallback` and passes it the event data structure. If there is no `CellEditCallback` and the user-entered data results it an error, the contents of the cell reverts to its previous value and no error is displayed.

Note In order for the `CellEditCallback` to be issued, after modifying a table cell the user must hit **Enter** or click somewhere else within the figure containing the table. Editing a cell's value and then clicking another figure or other window does not save the new value to the data table, and does not fire the `CellEditCallback`.

`CellSelectionCallback`

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

Callback that executes when cell is selected. Callback function that executes when the user highlights a cell by navigating to it or clicking it. For multiple selection, this callback executes when new cells are added to the selection. The callback includes event data, a structure with one member

Event Data Structure Field	Type	Description
`Indices`	n-by-2 matrix	Row index and column index of the cells the user currently has selected

Once a cell selection has been made, cells within it can be removed one at a time by **Ctrl**-clicking them.

`Children`

matrix

The empty matrix; `uitable` objects have no children.

`Clipping`

`{on} | off`

This property has no effect on `uitable` objects.

`ColumnEditable`

logical 1–by-n matrix | scalar logical value |{ empty matrix ([])}

Determines if column is user-editable.

Determines if the data can be edited by the end user. Each value in the cell array corresponds to a column. False is default because the developer needs to have control over changes users potentially might make to data.

Specify elements of a logical matrix as `true` if the data in a column is editable by the user or `false` if it is not. An empty matrix indicates that no columns are editable.

Columns that contain check boxes or pop-up menus must be editable for the user to manipulate these controls. If a column that contains pop-up menus is not editable, the currently selected choice appears without displaying the pop-up control. The Elements of the `ColumnEditable` matrix must be in the same order as columns in the `Data` property. If you do not specify `ColumnEditable`, the default is an empty matrix (`[]`).

`ColumnFormat`

cell array of strings

Cell display formatting. Determines how the data in each column displays and is edited. Elements of the cell array must be in the same order as table columns in the `Data` property. If you do not want to specify a display format for a particular column, enter `[]` as a placeholder. If no format is specified for a column, the default display is determined by the data type of the data in the cell. Default `ColumnFormat` is an empty cell array (`{}`). In most cases, the default is similar to the command window.

Elements of the cell array must be one of the strings described in the following table.

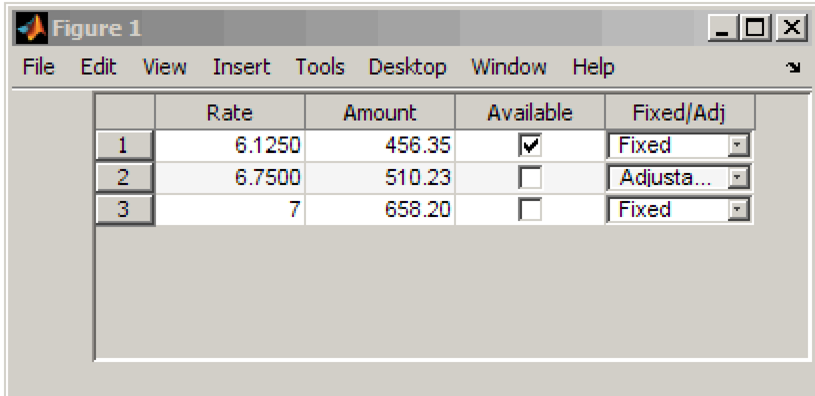
Cell Format	Description
'char'	Displays a left-aligned string. To edit, the user types a string that replaces the existing string.
'logical'	Displays a check box. To edit, the user checks or unchecks the check box. uitable sets the corresponding Data value to true or false accordingly. Initially, the check box is checked if the corresponding Data value would produce
'numeric'	Displays a right-aligned string equivalent to the command window, for numeric data. If the cell Data value is boolean, then 1 or 0 is displayed. If the cell Data value is not numeric and not boolean, then NaN is displayed. To edit, the user can enter any string. This enables a user to enter a value such as 'pi' that can be converted to its numeric equivalent by a CellEditCallback. The uitable function first attempts to convert the user-entered string to a numeric value and store it in Data. It then calls the CellEditCallback. See CellEditCallback for more information.

Cell Format	Description
1–by-n cell array of strings that define a pop-up menu, e.g., {'one' 'two' 'three'}	Displays a pop-up menu. To edit, the user makes a selection from the pop-up menu. uitable sets the corresponding Data value to the selected menu item. The initial values for the pop-up menus in the column are the corresponding strings in Data. These initial values do not have to be items in the pop-up menu. See Example 3 on the uitable reference page.
Valid string accepted by the format function, e.g.,'short' or 'bank'	Displays the Data value using the specified format. For example, for a two-column table, set(htable,'ColumnFormat',{'short','bank'}).

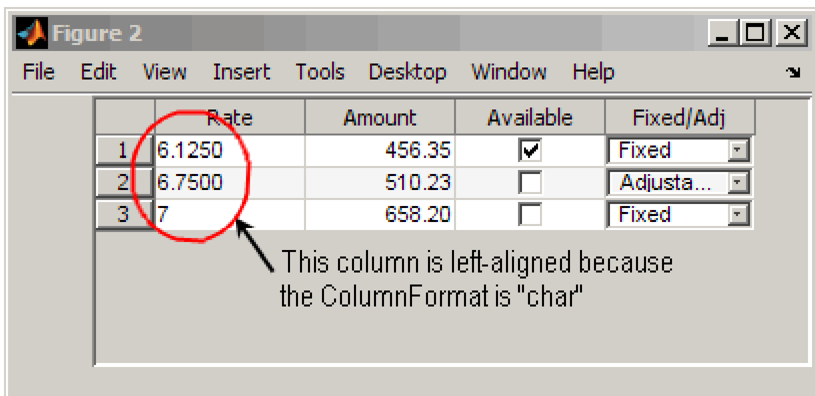
In some cases, you may need to insert an appropriate column in Data. If Data is a numerical or logical matrix, you must first convert it to a cell array using the mat2cell function.

Data and ColumnFormat

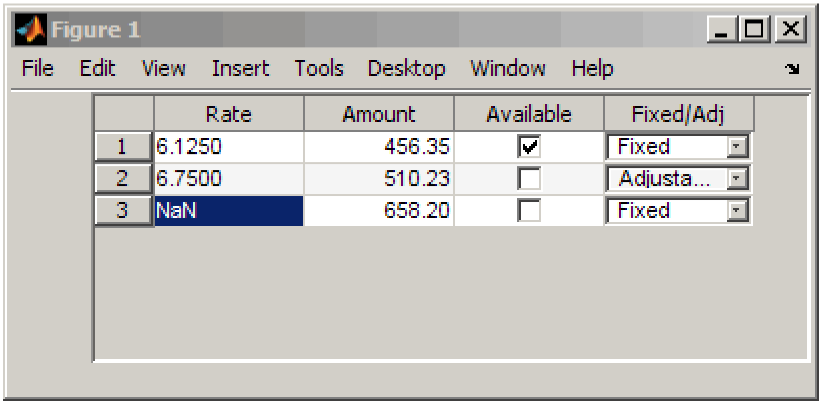
When you create a table, you must specify value of Data. The Data property dictates what type of data can exist in any given cell. By default, the value of the Data also dictates the display of the cell to the end user, unless you specify a different format using the ColumnFormat property.



ColumnFormat controls the presentation of the Data to the end user. Therefore, if you specify a ColumnFormat of char (or pick **Text** from the Table Property Editor), you are asking the table to display the Data associated with that column as a string. For example, if the Data for a particular column is numeric, and you specify the ColumnFormat as char, then the display of the numeric data will be left-aligned



If your column is editable and the user enters a number, the number will be left-aligned. However, if the user enters a text string, the table displays a **NaN**.



Another possible scenario is that the value `Data` is `char` and you set the `ColumnFormat` to be a pop-up menu. Here, if the value of the `Data` in the cell matches one of the pop-up menu choices you define in `ColumnFormat`, then the `Data` is shown in the cell. If it does not match, then the cell defaults to display the first option from the choices you specify in `ColumnFormat`. Similarly, if `Data` is numeric or logical with the `ColumnFormat` as pop-up menu, if the `Data` value in the cell does not match any of the choices you specify in `ColumnFormat`, the cell defaults to display the first option in the pop-menu choice.

This table describes how `Data` values correspond with your `ColumnFormat` when the columns are editable.

	`ColumnFormat` Selections		
	`numeric`	`char`	`logical`

Data Type	numeric	Values match. MATLAB displays numbers as is.	MATLAB converts the text string entered to a double. See `str2double` for more information. If string cannot be converted, **NaN** is displayed.	Does not work: warning is thrown. **Note** If you have defined `CellEditCallback`, this warning will not be thrown
	char	MATLAB converts the entered number to a text string.	Values match. MATLAB displays the string as is.	Does not work: warning is thrown. **Note** If you have defined `CellEditCallback`, this warning will not be thrown
	logical	Does not work: warning is thrown. **Note** If you have defined `CellEditCallback`, this warning will not be thrown	If text string entered is `true` or `false`, MATLAB converts string to the corresponding logical value and displays it. For all others, it Does not work: warning is thrown. **Note** If you have defined `CellEditCallback`, this warning will not be thrown	Values match. MATLAB displays logical value as a check box as is.

If you get a mismatch error, you have the following options:

- Change the ColumnFormat or value of Data to match.
- Implement the CellEditCallback to handle custom data conversion.

ColumnName

1–by-*n* cell array of strings | {'numbered'} | empty matrix ([])

Column heading names. Each element of the cell array is the name of a column. Multiline column names can be expressed as a string vector separated by vertical slash (|) characters, e.g.,'Standard|Deviation'

For sequentially numbered column headings starting with 1, specify ColumnName as 'numbered'. This is the default.

To remove the column headings, specify ColumnName as the empty matrix ([]).

The number of columns in the table is the larger of ColumnName and the number of columns in the Data property matrix or cell array.

ColumnWidth

1–by-*n* cell array or 'auto'

Column widths. The width of each column in units of pixels. Column widths are always specified in pixels; they do not obey the Units property. Each column in the cell array corresponds to a column in the uitable. By default, the width of the column name, as specified in ColumnName, along with some other factors, is used to determine the width of a column. If ColumnWidth is a cell array and the width of a column is set to 'auto'or if **auto** is selected for that column in the Property Inspector GUI for columns, the column width defaults to a size determined by the table. The table decides the default size using a number of factors, including the ColumnName and the minimum column size.

To default all column widths in an existing table, use

```
set(uitable_handle,'ColumnWidth','auto')
```

To default some column widths but not others, use a cell array containing a mixture of pixel values and 'auto'. For example,

```
set(uitable_handle,'ColumnWidth',{64 'auto' 40 40 'auto' 72})
```

CreateFcn
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uitable object. MATLAB sets all property values for the uitable before executing the CreateFcn callback so these values are available to the callback. Within the function, use gcbo to get the handle of the uitable being created.

Setting this property on an existing uitable object has no effect.

You can define a default CreateFcn callback for all new uitables. This default applies unless you override it by specifying a different CreateFcn callback when you call uitable. For example, the code

```
set(0,'DefaultUitableCreateFcn','set(gcbo,...
    ''BackGroundColor'',''blue'')')
```

creates a default CreateFcn callback that runs whenever you create a new uitable. It sets the default background color of all new uitables.

To override this default and create a uitable whose BackgroundColor is set to a different value, call uitable with code similar to

```
hpt = uitable(...,'CreateFcn','set(gcbo,...
''BackgroundColor'',''white'')')
```

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 `uitable` call. In the example above, if instead of redefining the `CreateFcn` property for this uitable, you had explicitly set `BackgroundColor` to `white`, the default `CreateFcn` callback would have set `BackgroundColor` back to the default, i.e., `blue`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See for information on how to use function handles to define a callback function.

`Data`

matrix or cell array of numeric, logical, or character data

Data content of uitable. The matrix or cell array must be 2–dimensional. A cell array can mix data types.

Use `get` and `set` to modify `Data`. For example,

```
data = get(tablehandle,'Data')
data(event.indices(1),event.indices(2)) = pi();
set(tablehandle,'Data',data);
```

See `CellEditCallback` for information about the event data structure. See `ColumnFormat` for information about specifying the data display format.

The number of rows in the table is the larger of `RowName` and the number of rows in `Data`. The number of columns in the table is the larger of `ColumnName` and the number of columns in `Data`.

`DeleteFcn`

string or function handle

Delete uitable callback routine. A callback routine that executes when you delete the `uitable` object (e.g., when you issue a `delete` command or clear the figure containing the `uitable`). 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 for information on how to use function handles to define a callback function.

`Enable`

`{on} | inactive | off`

Enable or disable the uitable. This property determines how uitables respond to mouse button clicks, including which callback routines execute.

- `on` – The uitable is operational (the default).
- `inactive` – The uitable is not operational, but looks the same as when `Enable` is `on`.
- `off` – The uitable is not operational and its image is grayed out.

When you left-click on a uitable whose `Enable` property is `on`, MATLAB performs these actions in this order:

1. Sets the figure's `SelectionType` property.

2 Executes the uitable's `CellSelectionCallback` routine (but only for table cells, not header cells). Row and column indices of the cells the user selects continuously update the `Indices` field in the `eventdata` passed to the callback.

3 Does not set the figure's `CurrentPoint` property and does not execute either the table's `ButtonDownFcn` or the figure's `WindowButtonDownFcn` callback.

When you left-click on a uitable whose `Enable` property is `off`, or when you right-click a uitable whose `Enable` property has any value, MATLAB performs these actions in this order:

1 Sets the figure's `SelectionType` property.

2 Sets the figure's `CurrentPoint` property.

3 Executes the figure's `WindowButtonDownFcn` callback.

`Extent`

position rectangle (read only)

Size of uitable rectangle. A four-element vector of the form `[0,0,width,height]` that contains the calculated values of the largest extent of the table based on the current `Data`, `RowNames` and `ColumnNames` property values. Calculation depends on column and row widths, when they are available. The calculated extent can be larger than the figure.

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.

When the uitable's `Units` property is set to `'Normalized'`, its `Extent` is measured relative to the figure, regardless of whether the table is contained in (parented to) a uipanel or not.

You can use this property to determine proper sizing for the uitable with respect to its content. Do this by setting the `width` and `height` of the uitable `Position` property to the width and height of the `Extent` property. However, doing this can cause the table to extend beyond the right or top edge of the figure and/or its uipanel parent, if any, for tables with large extents.

`FontAngle`
`{normal} | italic | oblique`

Character slant of cell content. 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 for cell content. The name of the font in which to display cell content. To display and print properly, this must be a font that your system supports. The default font is system dependent.

To use a fixed-width font that looks good in any locale (and displays properly in Japan, where multibyte character sets are used), set `FontName` to the string `FixedWidth` (this string value is case sensitive):

```
set(uitable_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 for cell contents. A number specifying the size of the font in which to display cell contents, in units determined by the `FontUnits` property. The default point size is system dependent. If `FontUnits` is set to `normalized`, `FontSize` is a number between 0 and 1.

`FontUnits`

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

Font size units for cell contents. This property determines the units used by the `FontSize` property. `Normalized` units interpret `FontSize` as a fraction of the height of the uitable. When you resize the uitable, 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 cell 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`

`1-by-3 matrix of RGB triples or a color name`

Color of text in cells. Determines the color of the text defined for cell contents. Text in all cells share the current color. Specify as a 1-by-3 matrix of RGB triples, such as `[0 0 .8]` or as a color name.

The default is a 1-by-3 matrix of platform-dependent colors. See `ColorSpec` for information about specifying RGB colors.

`HandleVisibility`
: `{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

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

`HitTest`
: `{on} | off`

Selectable by mouse click. When `HitTest` is `off`, the `ButtonDownFcn` callback does not execute.

`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. The 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.

KeyPressFcn
string or function handle

Key press callback function. A callback routine invoked by a key press when the callback's uitable object has focus. Focus is denoted by a border or a dotted border, respectively, in UNIX and Microsoft Windows. If no uitable 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'

Event Data Structure Field	Description	Examples:			
		a	=	Shift	Shift/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'

The uitable KeyPressFcn callback executes for all keystrokes, including arrow keys or when a user edits cell content.

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

Parent
handle

Uitable parent. The handle of the uitable's parent object. You can move a uitable 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 uitable. The rectangle defined by this property specifies the size and location of the table within the parent figure window, ui, or uibuttongroup. Specify Position as a 4–element vector:

```
[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 uitable object. width and height are the dimensions of the uitable rectangle. All measurements are in units specified by the Units property.

Note If you are specifying both `Units` and `Position` in the same call to `uitable`, specify `Units` first if you want `Position` to be interpreted using those units.

`RearrangeableColumn`

`on | {off}`

This object can be rearranged. The `RearrangeableColumn` property provides a mechanism that you can use to reorder the columns in the table. All columns are rearrangable when this property is turned on. MATLAB software sets the `RearrangeableColumn` property to `off` by default.

When this property is on, the user of a table can move any column of data (but not the row labels) at a time left or right to reorder it by clicking and dragging its header. Rearranging columns does not affect the ordering of columns in the table's `Data`, only the user's view of it.

`RowName`

1–by-n cell array of strings`|` {`'numbered'`} `|` empty matrix (`[]`)

Row heading names. Each element of the cell array is the name of a row. Row names are restricted to one line of text.

For sequentially numbered row headings starting with `1`, specify `RowName` as `'numbered'`. This is the default.

To remove the row headings, specify `RowName` as the empty matrix (`[]`).

The number of rows in the table is the larger of `RowName` and the number of rows in the `Data` property matrix or cell array.

`RowStriping`

`{on} | off`

Color striping of table rows. When RowStriping is on, the background of consecutive rows of the table display in the pair of colors that the BackgroundColor color matrix specifies. The first color matrix row applies to odd-numbered rows, and the second to even-numbered rows. If the BackgroundColor matrix has only one row, it is applied to all rows (that is, no striping occurs).

When RowStriping is off, the first color specified for BackgroundColor is applied to all rows.

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

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.

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 uitable. When the user moves the mouse pointer over the table and leaves it there, the tooltip is displayed.

`Type`

string (read only)

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

`UIContextMenu`

handle

Associate a context menu with uitable. Assign this property the handle of a `uicontextmenu` object. MATLAB displays the context menu whenever you right-click over the uitable. 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.

- `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 uitable object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Visible`

`{on} | off`

Uitable visibility. By default, all uitables are visible. When set to `off`, the uitable is not visible, but still exists and you can query and set its properties.

Note Setting `Visible` to `off` for uitables that are not displayed initially in the GUI, can result in faster startup time for the GUI.

Purpose

Create toggle button on toolbar

Syntax

```
htt = uitoggletool
htt = uitoggletool('PropertyName1',value1,'PropertyName2',
   value2,...)
htt = uitoggletool(ht,...)
```

Description

`htt = uitoggletool` creates a toggle button on the uitoolbar at the top of the current figure window, sets all its properties to default values, and returns a handle to the tool. If no uitoolbar exists, one is created. The uitoolbar is the parent of the uitoggletool. Use the returned handle `htt` to set properties of the uitoggletool. The `OnCallback`, `OffCallback` and `ClickedCallback` use the handle as their first argument. The button has no icon, but its border highlights when you hover over it with the mouse cursor. Add an icon by setting `CData` for the tool. 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.

`htt = uitoggletool('`*`PropertyName1`*`',value1,'`*`PropertyName2`*`',value2,...)` 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. You can specify properties as parameter name/value pairs, cell arrays containing parameter names and values, or structures with fields containing parameter names and values as input arguments. For a complete list, see Uitoggletool Properties. Type `get(htt)` to see a list of uipushtool object properties and their current values. Type `set(htt)` to see a list of uipushtool object properties that you can set and their legal property values.

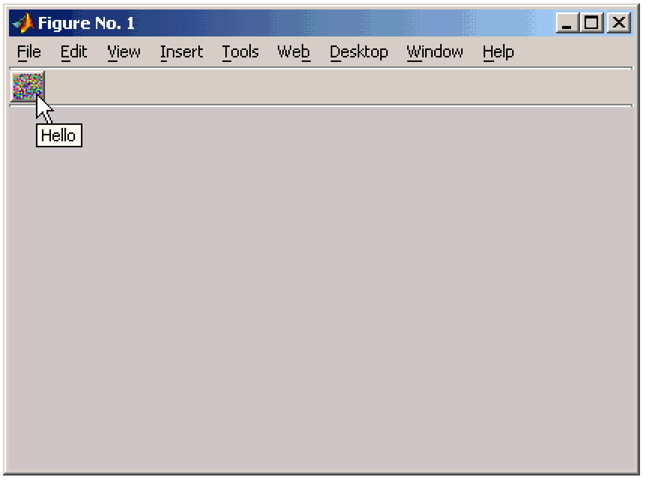
`htt = uitoggletool(ht,...)` creates a button with `ht` as a parent. `ht` must be a uitoolbar handle.

Toggle tools appear in figures whose `Window Style` is `normal` or `docked`. They do not appear in figures with a `'modal' WindowStyle`. If the `WindowStyle` property of a figure containing a tool bar and its toggle tool children changes to `modal`, the toggle tools continue to exist as `Children` of the tool bar. The toggle tools do not display until you change the `WindowStyle` to `normal` or `docked`.

Examples

Create a `uitoolbar` object and places a `uitoggletool` object on it by specifying the toolbar handle as the toggle tool parent. Generate a random set of colors for the tool icon and specify a tool tip.

```
h = figure('ToolBar','none');
ht = uitoolbar(h);
a = rand(16,16,3);
htt = uitoggletool(ht,'CData',a,'TooltipString','Hello');
```



Alternatives

You can create toolbars with toggle tools using GUIDE.

See Also

`get` | `set` | `uicontrol` | `uipushtool` | `uitoolbar`

Tutorials

-
-

How To

-
-

Uitoggletool Properties

Purpose

Describe toggle tool 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 .

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.

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.
HitTest	Whether selectable by mouse click
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.
UIContextMenu	Uicontextmenu object associated with the uitoggletool
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 software sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the`Interruptible` property for information about controlling a callback's interruptibility.

`CData`

3-dimensional array

Truecolor image displayed on control as its icon. An *n*-by-*m*-by-3 array of RGB values that defines a truecolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0. If your `CData` array is larger than 16 in the first or second dimension, it can be clipped or result in 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 independent of the toggle tool position. A routine that executes after either the `OnCallback` routine or `OffCallback` routine runs to completion. The uitoggletool `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`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

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

`DeleteFcn`
string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the uitoggletool object (e.g., when you call the `delete` function or cause the figure containing the uitoggletool to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

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

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

`Enable`

{on} | off

Enable or disable the uitoggletool. This property controls how uitoggletools respond to mouse button clicks, including which callback routines execute.

- `on` – The uitoggletool is operational (the default).
- `off` – The uitoggletool is not operational and its icon (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. Executes the toggle tool `OnCallback` or `OffCallback` routine, depending on its current state, and its `ClickedCallback` routine.
2. Does *not* set the figure `CurrentPoint` property and does *not* execute the figure's `WindowButtonDownFcn` callback.
3. Does *not* set the figure `SelectionType` property.

When you left-click a uitoggletool whose `Enable` property is `off`, or when you right-click a uitoggletool whose `Enable` property has any value, no action is reported, no callback executes, and neither the `SelectionType` nor `CurrentPoint` figure properties are modified.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

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

`HitTest`
`{on} | off`

Selectable by mouse click. This property has no effect on uitoggletool objects.

`Interruptible`
`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below).

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement.

`OffCallback`
string or function handle

Control action. A routine that executes if the uitoggletool's `Enable` property is set to `on`, and either

- The toggle tool `State` is set to `off`.
- The toggle tool is set to the `off` position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The `ClickedCallback` routine, if there is one, runs after the `OffCallback` routine runs to completion.

`OnCallback`
string or function handle

Control action. A routine that executes if the uitoggletool's `Enable` property is set to `on`, and either

- The toggle tool `State` is set to `on`.
- The toggle tool is set to the `on` position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The `ClickedCallback` routine, if there is one, runs after the `OffCallback` routine runs to completion.

`Parent`
handle

Uitoggletool parent. The handle of the uitoggletool's parent toolbar. You can move a uitoggletool object to another toolbar by setting this property to the handle of the new parent.

`Separator`
`on | {off}`

Separator line mode. Setting this property to `on` draws a dividing line to left of the uitoggletool.

`State`
: `on | {off}`

Uitoggletool state. When the state is `on`, the toggle tool appears in the down, or pressed, position. When the state is `off`, it appears in the up position. Changing the state causes the appropriate `OnCallback` or `OffCallback` routine to run.

`Tag`
: string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified toolbars) that have the `Tag` value `'Bold'`.

```
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'`.

`UIContextMenu`
handle

Associate a context menu with uicontrol. This property has no effect on uitoggletool objects.

`UserData`
array

User specified data. You can specify `UserData` as any array you want to associate with the uitoggletool object. The object does not use this data, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Uitoggletool visibility. By default, all uitoggletools are visible. When set to `off`, the uitoggletool is not visible, but still exists and you can query and set its properties.

Purpose

Create 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

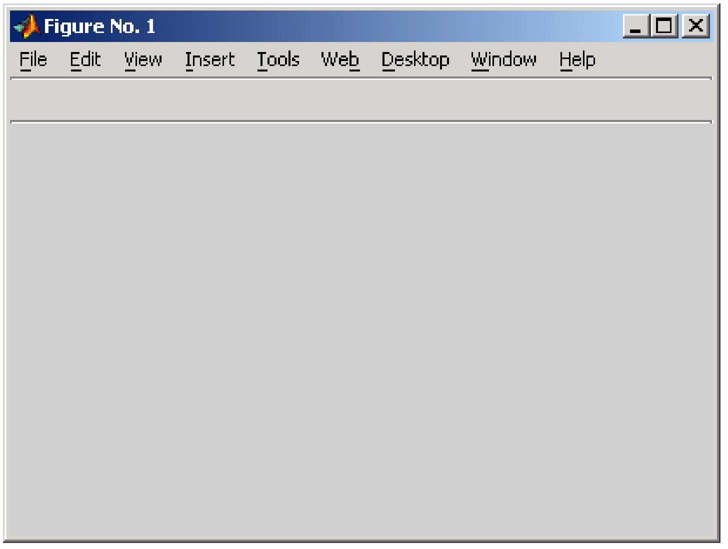
`uitoolbar` accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.

Uitoolbars appear in figures whose `Window Style` is `normal` or `docked`. They do not appear in figures whose `WindowStyle` is `modal`. If the `WindowStyle` property of a figure containing a uitoolbar is changed to `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`

Purpose

Describe toolbar properties

Modifying Properties

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

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uitoolbar properties by typing:

```
set(h,'DefaultUitoolbarPropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, or a uitoolbar handle. *`PropertyName`* is the name of the Uitoolbar property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see Setting Default Property Values.

Uitoolbar Properties

This section lists all properties useful to `uitoolbar` objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property	Purpose
`BeingDeleted`	This object is being deleted.
`BusyAction`	Callback routine interruption.
`Children`	Handles of uitoolbar's children.
`CreateFcn`	Callback routine executed during object creation.
`DeleteFcn`	Callback routine executed during object deletion.

Property	Purpose
HandleVisibility	Control access to object's handle.
HitTest	Whether selectable by mouse click
Interruptible	Callback routine interruption mode.
Parent	Handle of uitoolbar's parent.
Tag	User-specified object identifier.
Type	Object class.
UIContextMenu	Uicontextmenu object associated with the uitoolbar
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 software sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, 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.

`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`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

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

`DeleteFcn`

string or function handle

Callback routine executed during object deletion. A callback function that executes when the uitoolbar object is deleted (e.g., when you call the `delete` function or cause the figure containing the uitoolbar to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

Within the function, use `gcbo` to get the handle of the toolbar being deleted.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

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

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

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

`HitTest`
`{on} | off`

Selectable by mouse click. This property has no effect on uitoolbar objects.

`Interruptible`
`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

`Parent`

handle

Uitoolbar parent. The handle of the 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'`.

`UIContextMenu`
handle

Associate a context menu with uicontrol. This property has no effect on uitoolbar objects.

`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.

Purpose

Block execution and wait for resume

Syntax

```
uiwait
uiwait(h)
uiwait(h,timeout)
```

Description

`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. The minimum value of `timeout` is 1. If `uiwait` receives a smaller value, it issues a warning and uses a 1 second `timeout`.

Remarks

The `uiwait` and `uiresume` functions block and resume MATLAB and Simulink program execution. `uiwait` also blocks the execution of Simulink models. The functions `pause` (with no argument) and `waitfor` also block execution in this manner.`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` can block the execution of the M-file *and* restrict user interaction to the dialog only.

Example

This example creates a GUI with a **Continue** push button. The example calls `uiwait` to block MATLAB execution until `uiresume` is called. This happens when the user clicks the **Continue** push button because the push button's `Callback` callback, which responds to the click, calls `uiresume`.

```
f = figure;
h = uicontrol('Position',[20 20 200 40],'String','Continue',...
              'Callback','uiresume(gcbf)');
disp('This will print immediately');
```

```
uiwait(gcf);
disp('This will print after you click Continue');
close(f);
```

gcbf is the handle of the figure that contains the object whose callback is executing.

is a more complex example for a GUIDE GUI. See for an example for a programmatically created GUI.

See Also

dialog, figure, uicontrol, uimenu, uiresume, waitfor

Purpose

Undo previous checkout from source control system (UNIX platforms)

GUI Alternatives

As an alternative to the `undocheckout` function, select **Source Control > Undo Checkout** in the **File** menu of the Editor, Simulink software, or Stateflow software, or in the context menu of the Current Folder browser.

Syntax

```
undocheckout('filename')
undocheckout({'filename1','filename2', ...,'filenamen'})
```

Description

`undocheckout('filename')` makes the file `filename` available for checkout, where `filename` does not reflect any of the changes you made after you last checked it out. Use the full path for `filename` and include the file extension.

`undocheckout({'filename1','filename2', ...,'filenamen'})` makes `filename1` through `filenamen` available for checkout, where the files do not reflect any of the changes you made after you last checked them out. Use the full paths for the file names and include the file extensions.

Examples

Undo the checkouts of `/myserver/mymfiles/clock.m` and `/myserver/mymfiles/calendar.m` from the source control system:

```
undocheckout({'/myserver/mymfiles/clock.m', ...
'/myserver/mymfiles/calendar.m'})
```

See Also

`checkin`, `checkout`

- For Microsoft Windows platforms, use `verctrl`.
- For more information, see .

unicode2native

Purpose

Convert Unicode characters to numeric bytes

Syntax

```
bytes = unicode2native(unicodestr)
bytes = unicode2native(unicodestr, encoding)
```

Description

`bytes = unicode2native(unicodestr)` takes a `char` vector of Unicode characters, `unicodestr`, converts it to the MATLAB default character encoding scheme, and returns the bytes as a `uint8` vector, `bytes`. Output vector `bytes` has the same general array shape as the `unicodestr` input. You can save the output of `unicode2native` to a file using the `fwrite` function.

`bytes = unicode2native(unicodestr, encoding)` converts the Unicode characters to the character encoding scheme specified by the string `encoding`. `encoding` must be the empty string (`''`) or a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. For common names and aliases, see the Web site `http://www.iana.org/assignments/character-sets`. If `encoding` is unspecified or is the empty string (`''`), the MATLAB default encoding scheme is used.

Examples

This example begins with two strings containing Unicode characters. It assumes that string `str1` contains text in a Western European language and string `str2` 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 `unicode2native` to convert the two strings to the appropriate encoding schemes.

```
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);
```

See Also

`native2unicode`

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` ∪ `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. MATLAB ignores the rows flag for all cell arrays.

`[c, ia, ib] = union(...)` also returns index vectors `ia` and `ib` such that `c = a(ia)` ∪ `b(ib)`, or for row combinations, `c = a(ia,:)` ∪ `b(ib,:)`. If a value appears in both `a` and `b`, `union` indexes its occurrence in `b`. If a value appears more than once in `b` or in `a` (but not in `b`), `union` indexes the last occurrence of the value.

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

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` can be

- `'first'`, which returns the vector `m` to index the first occurrence of each unique value in `A`, or
- `'last'`, which returns the vector `m` to index the last occurrence.

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
```

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`, `sort`, `issorted`, `setdiff`, `setxor`, `union`

unix

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[19] 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 `result`.

Definitions

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`.

Try to execute a string that isn't a UNIX command.

```
[s,w] = unix('why')
s =
     1
w =
why: Command not found.
```

19. UNIX is a registered trademark of The Open Group in the United States and other countries.

MATLAB returns a nonzero value in s to indicate failure, and returns an error message in w because why is not a UNIX command.

Algorithm

The MATLAB software 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.

See Also

dos | ! (exclamation point) | perl | system

Tutorials

-

unloadlibrary

Purpose

Unload shared library from memory

Syntax

```
unloadlibrary('libname')
unloadlibrary libname
```

Description

`unloadlibrary('libname')` unloads the shared library `libname` from memory. If you need to use functions in this library, you must reload the library using the `loadlibrary` function.

`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`

Purpose

Convert edge matrix to coordinate and Laplacian matrices

Syntax

`[L,XY] = UNMESH(E)`

Description

`[L,XY] = UNMESH(E)` returns the Laplacian matrix `L` and mesh vertex coordinate matrix `XY` for the `M`-by-4 edge matrix `E`. Each row of the edge matrix must contain the coordinates `[x1 y1 x2 y2]` of the edge endpoints.

Inputs

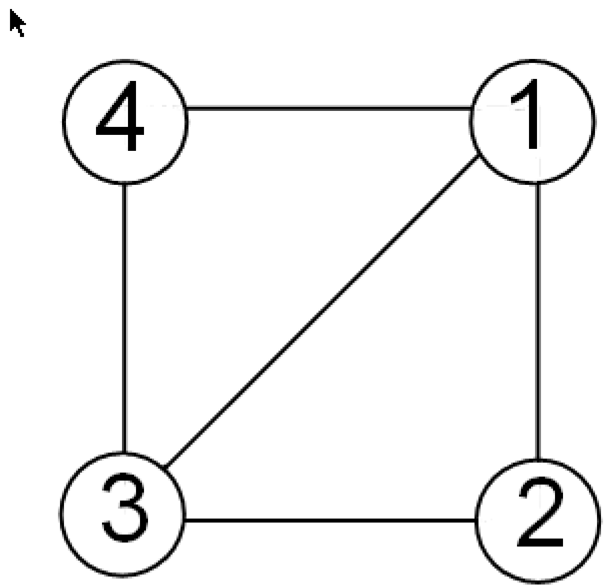
`E`	M-by-4 edge matrix `E`.

Outputs

`L`	Laplacian matrix representation of the graph.
`XY`	Mesh vertex coordinate matrix.

Examples

Take a simple example of a square with vertices at (1,1), (1,–1),(–1,–1), and (–1,1), where the connections between vertices are the four perpendicular edges of the square plus one diagonal connection between (–1, –1) and (1,1).



The edge matrix E for this graph is:

```
E=[1 1 1 -1;  % edge from 1 to 2
1 -1 -1 -1;   % edge from 2 to 3
-1 -1 -1 1;   % edge from 3 to 4
-1 -1 1 1;    % edge from 4 to 1
-1 1 1 1]     % edge from 3 to 1
```

Use unmesh to create the output matrices,

```
[A,XY]=unmesh(E);
4 vertices:
4/4
```

The Laplacian matrix is defined as

$$L_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

`unmesh` returns the Laplacian matrix `L` in sparse notation.

```
L

L =

   (1,1)        3
   (2,1)       -1
   (3,1)       -1
   (4,1)       -1
   (1,2)       -1
   (2,2)        2
   (4,2)       -1
   (1,3)       -1
   (3,3)        2
   (4,3)       -1
   (1,4)       -1
   (2,4)       -1
   (3,4)       -1
```

To see `L` in regular matrix notation, use the `full` command.

```
full(L)

ans =

     3    -1    -1    -1
    -1     2     0    -1
    -1     0     2    -1
    -1    -1    -1     3
```

The mesh coordinate matrix `XY` returns the coordinates of the corners of the square.

```
XY

XY =

    -1    -1
    -1     1
     1    -1
     1     1
```

See Also

`gplot`
`treeplot`

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`

unregisterallevents

Purpose

Unregister all event handlers associated with COM object events at run time

Syntax

```
h.unregisterallevents
unregisterallevents(h)
```

Description

`h.unregisterallevents` unregisters all events previously registered with COM object `h`. After calling `unregisterallevents`, the object no longer responds to any events until you register them again using the `registerevent` function.

`unregisterallevents(h)` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Register and unregister events for an instance of the `mwsamp` control, using the `eventlisteners` function to see the event handler associated with each event:

1. Register three events and their respective handler routines.

   ```
   f = figure ('position', [100 200 200 200]);
   h = actxcontrol('mwsamp.mwsampctrl.2', ...
       [0 0 200 200], f, ...
       {'Click' 'myclick'; 'DblClick' 'my2click'; ...
       'MouseDown' 'mymoused'});
   h.eventlisteners
   ```

   MATLAB displays:

   ```
   ans =
       'click'         'myclick'
       'dblclick'      'my2click'
       'mousedown'     'mymoused'
   ```

2. Unregister all events simultaneously with `unregisterallevents`. `eventlisteners` returns an empty cell array, indicating that there are no longer any events registered with the control:

```
h.unregisterallevents;
h.eventlisteners
```

MATLAB displays:

```
ans =
     {}
```

See Also

events (COM) | eventlisteners | registerevent | unregisterevent | isevent

unregisterevent

Purpose

Unregister event handler associated with COM object event at run time

Syntax

```
h.unregisterevent(eventhandler)
unregisterevent(h, eventhandler)
```

Description

`h.unregisterevent(eventhandler)` unregisters specific event handler routines from their corresponding events. Once you unregister an event, the object no longer responds to the event.

`unregisterevent(h, eventhandler)` is an alternate syntax.

You can unregister events at any time after creating a control. The `eventhandler` argument, which is a cell array, specifies both events and event handlers.

```
h.unregisterevent({'event_name',@event_handler});
```

Specify events in the `eventhandler` argument using the names of the events. Strings used in the `eventhandler` argument are not case sensitive. `unregisterevent` does not accept numeric event identifiers.

COM functions are available on Microsoft Windows systems only.

Examples

Unregister events for a control:

1. Create an `mwsamp` control and register all events with the same handler routine, `sampev`. Use `eventlisteners` to see the event handler used by each event. In this case, each event, when fired, calls `sampev.m`:

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

   MATLAB displays:

```
ans =
    'Click'          'sampev'
    'DblClick'       'sampev'
    'MouseDown'      'sampev'
    'Event_Args'     'sampev'
```

2 Unregister just the `dblclick` event. Now, when you list the registered events using `eventlisteners`, `dblclick` is no longer registered and the control does not respond when you double-click the mouse over it:

```
h.unregisterevent({'dblclick' 'sampev'});
h.eventlisteners
```

MATLAB displays:

```
ans =
    'Click'          'sampev'
    'MouseDown'      'sampev'
    'Event_Args'     'sampev'
```

3 Now, 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
```

MATLAB displays:

```
ans =
    'click'        'myclick'
    'dblclick'     'my2click'
```

4 Unregister these same events by specifying event names and their handler routines in a cell array. `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
```

MATLAB displays:

```
ans =
     {}
```

Unregister Microsoft Excel workbook events:

1. Create a `Workbook` object and register two events with the event handler routines, `EvtActivateHndlr` and `EvtDeactivateHndlr`:

```
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
wb = wbs.Add;wb.registerevent({'Activate' 'EvtActivateHndlr'; ...
    'Deactivate' 'EvtDeactivateHndlr'})
wb.eventlisteners
```

   MATLAB shows the events with the corresponding event handlers.

```
ans =
    'Activate'      'EvtActivateHndlr'
    'Deactivate'    'EvtDeactivateHndlr'
```

2. Next, unregister the `Deactivate` event handler:

```
wb.unregisterevent({'Deactivate' 'EvtDeactivateHndlr'})
wb.eventlisteners
```

   MATLAB shows the remaining registered event (`Activate`) with its corresponding event handler.

```
ans =
    'Activate'    'EvtActivateHndlr'
```

See Also

events (COM) | eventlisteners | registerevent | unregisterallevents | isevent

How To

•

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, if you rerun `untar` on the same `tarfilename`, MATLAB software does not overwrite files with a read-only attribute; instead, `untar` displays a warning for such files. On Microsoft 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`. `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`. If `outputdir` does not exist, MATLAB creates it.

`untar(url, ...)` extracts the tar archive from an Internet URL. The URL must include the protocol type (for example, `'http://'` or `'ftp://'`). MATLAB downloads the URL is to a temporary directory, and then deletes it.

`filenames = untar(...)` extracts the tar archive and returns the names of the extracted files in the string cell array `filenames`. If `outputdir` specifies a relative path, `filenames` contains the relative path. If `outputdir` specifies an absolute path, `filenames` contains the absolute path.

Examples

Using tar and untar to Copy Files

Copy all .m files in the current directory to the directory backup.

```
tar('mymfiles.tar.gz','*.m');
untar('mymfiles','backup');
```

Using untar with URL

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.

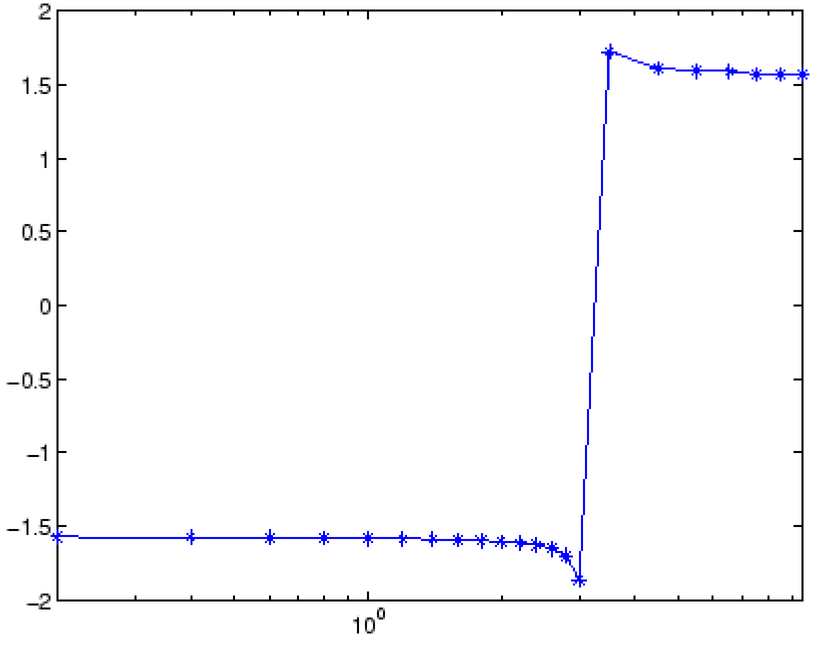
Examples

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`.

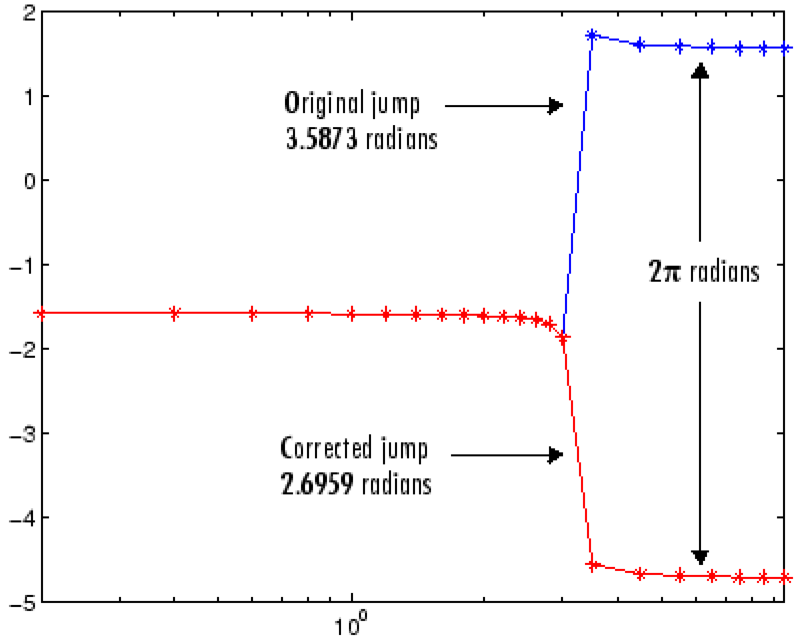
```
w = [0:.2:3,3.5:1:10];
p = [     0
      -1.5728
      -1.5747
      -1.5772
```

```
         -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
```



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 , 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
      0.5890     1.3744     2.1598     2.9452 ]
```

The function `Q = 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 folder and sets the files' attributes, preserving the timestamps. 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 folder name; otherwise, the file must be in the current folder or in a folder on the MATLAB path.

`unzip(zipfilename, outputdir)` extracts the contents of *`zipfilename`* into the folder *`outputdir`*.

`unzip(url, ...)` extracts the zipped contents from an Internet URL. The URL must include the protocol type (for example, `http://`). The URL is downloaded to the temp folder and deleted.

`filenames = unzip(...)` extracts the zip archive and returns the names of the extracted files in the string cell array *`filenames`*. If *`outputdir`* specifies a relative path, *`filenames`* contains the relative path. If *`outputdir`* specifies an absolute path, *`filenames`* contains the absolute path.

`unzip` does not support password-protected or encrypted zip archives.

Examples

Using zip and unzip to Copy Files

Copy the demos HTML files to the folder `archive`:

```
% Zip the demo MAT-files to demos.zip
```

```
zip('demos.zip','*.mat',...
    fullfile(matlabroot,'toolbox','matlab','demos'))

% Unzip demos.zip to the folder 'archive'
unzip('demos','archive')
```

Using unzip with URL

Run `unzip` to list Cleve Moler's "Numerical Computing with MATLAB" examples to the output folder `ncm`.

```
url ='http://www.mathworks.com/moler/ncm.zip';
ncmFiles = unzip(url,'ncm')
```

See Also

`fileattrib`, `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`

Purpose Download content at URL into MATLAB string

Syntax

```
str = urlread(URL)
str = urlread(URL, method, params)
[str, status] = urlread(...)
```

Description

`str = urlread(URL)` reads Web content at the specified *`URL`* into the string *`str`*. If the server returns binary data, *`str`* is unreadable.

`str = urlread(URL, method, params)` uses a *`method`* of `'get'` or `'post'`, and passes information in *`params`* to the server. *`params`* is a cell array of parameter name/value pairs.

`[str, status] = urlread(...)` returns a *`status`* of `1` when the operation is successful. Otherwise, *`status`* is `0`.

To save Web content to a file instead of a string, use `urlwrite`.

Examples

Download the page on the MATLAB Central File Exchange that lists submissions related to `urlread`, found at `http://www.mathworks.com/matlabcentral/fileexchange/?term=urlread`.

```
samples = urlread(...
     'http://www.mathworks.com/matlabcentral/fileexchange',...
     'get', ...
     {'term','urlread'});
```

Alternatives

`urlread` and `urlwrite` can download content from FTP sites. Alternatively, use the `ftp` function to connect to an FTP server and the `mget` function to download a file.

See Also

`urlwrite` | `ftp` | `web`

How To

-

Purpose

Download content at URL and save to file

Syntax

```
urlwrite(URL, filename)
urlwrite(URL, filename, method, params)
f = urlwrite(...)
[f, status] = urlwrite(...)
```

Description

`urlwrite(URL, filename)` reads Web content at the specified *URL* and saves it to *filename*. If you do not specify the path for *filename*, `urlwrite` saves the file in the MATLAB current folder.

`urlwrite(URL, filename, method, params)` uses a *method* of `'get'` or `'post'`, and passes information in *params* to the server. *params* is a cell array of parameter name/value pairs.

`f = urlwrite(...)` assigns *filename* to *f*.

`[f, status] = urlwrite(...)` returns a *status* of `1` when the operation is successful. Otherwise, *status* is `0`.

Examples

Download the page on the MATLAB Central File Exchange that lists submissions related to `urlwrite`, found at `http://www.mathworks.com/matlabcentral/fileexchange/?term=urlwrite`. Save the results to `samples.html` in the current directory.

```
urlwrite(...
    'http://www.mathworks.com/matlabcentral/fileexchange',...
    'samples.html', ...
    'get', ...
    {'term','urlwrite'});
```

View the file in the Help browser:

```
open('samples.html')
```

Alternatives

`urlread` and `urlwrite` can download content from FTP sites. Alternatively, use the `ftp` function to connect to an FTP server and the `mget` function to download a file.

See Also

urlread | ftp | web

How To

-

Purpose

Determine whether Sun Java feature is supported in MATLAB software

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 software (JVM) is running
`'swing'`	Swing components[2] are available

1. Java GUI components in the Abstract Window Toolkit

2. Java 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 JVM software.

```
if ~usejava('jvm')
   error([mfilename ' requires Java to run.']);
end
```

See Also `javachk`

Purpose

View or change user portion of search path

Syntax

```
userpath
userpath('newpath')
userpath('reset')
userpath('clear')
```

Description

`userpath` returns a string specifying the user portion of the search path. The user portion of the search path is the first folder on the search path, above the folders supplied by The MathWorks. The default folder is `My Documents/MATLAB` on Microsoft Windows platforms, and `Documents/MATLAB` on Microsoft Windows Vista™ platforms.

On Apple Macintosh and UNIX[20] platforms, the default value is *userhome*/Documents/MATLAB. You can define the *userpath* folder to also be the MATLAB startup folder. On Windows platforms, *userpath* is the startup folder, unless the startup folder is otherwise specified, such as by the MATLAB shortcut properties **Start in** field. On UNIX and Macintosh platforms, the startup folder is *userpath* if the value of the environment variable MATLAB_USE_USERPATH is set to 1 prior to startup and if the startup folder is not otherwise specified, such as via a startup.m file. On Macintosh and UNIX platforms, you can automatically add additional subfolders to the top of the search path upon startup by specifying the path for the subfolders via the MATLABPATH environment variable.

userpath('newpath') sets the *userpath* value to newpath. The newpath folder appears at the top of the search path immediately and at startup in future sessions. MATLAB removes the folder previously specified by *userpath* from the search path. newpath must be an absolute path. userpath('newpath') does not work when the -nojvm startup option is used. Upon the next startup, newpath, can become the current folder, as described in the syntax for userpath with no arguments.

userpath('**reset**') sets the *userpath* value to the default for that platform, creating the Documents/MATLAB (or My Documents/MATLAB) folder, if it does not exist. MATLAB immediately adds the default folder to the top of the search path, and also adds it to the search path at startup in future sessions. It can become the startup folder as described for the *userpath* syntax with no arguments. MATLAB removes the folder previously specified by *userpath* from the search path. userpath('reset') does not work when the -nojvm startup option is used.

userpath('**clear**') clears the value for *userpath*. MATLAB removes the folder previously specified by *userpath* from the search path. This does not work when the -nojvm startup option is used. You can otherwise specify the startup folder—see .

20. UNIX is a registered trademark of The Open Group in the United States and other countries.

Examples

Viewing *userpath*

This example assumes `userpath` is set to the default value on the Windows XP platform, `My Documents\MATLAB`. Start MATLAB and display the current folder:

```
cd
```

MATLAB returns:

```
H:\My Documents\MATLAB
```

where `H` is the drive at which `My Documents` is located for this example. Confirm the current folder is the `userpath`:

```
userpath
```

MATLAB returns:

```
H:\My Documents\MATLAB;
```

Display the search path:

```
path
```

MATLAB returns the search path. The `userpath` portion is at the top:

```
MATLABPATH

 H:\My Documents\MATLAB
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
```

Setting a New Value for *userpath*

This example assumes `userpath` is set to the default value on the Windows XP platform, `My Documents\MATLAB`. Change the value from the default for `userpath` to `C:\Research_Project`:

```
userpath('C:\Research_Project')
```

View the effect of the change on the search path:

```
path
```

MATLAB displays the search path, with the new value for `userpath` portion at the top:

```
MATLABPATH

 C:\Research_Project
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Note that MATLAB automatically removed the previous value of `userpath`, `H:\My Documents\MATLAB`, from the search path when you assigned a new value to `userpath`. The next time you start MATLAB, the current folder will be `C:\Research_Project` on Windows platforms.

Clearing the Value for *userpath*, and Specifying a New Startup Folder on Windows Platforms

`userpath` is set to the default value and you do not want any folders to be added to the search path upon startup. Confirm the default is currently set:

```
userpath
```

MATLAB returns:

```
H:\My Documents\MATLAB
```

Verify that the *userpath* folder is at the top of the search path:

```
path
```

MATLAB returns:

```
MATLABPATH

H:\My Documents\MATLAB
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Clear the value:

```
userpath('clear')
```

Verify the result:

```
userpath
```

MATLAB returns:

```
ans =
     ''
```

Confirm the *userpath* folder was removed from the search path:

```
path
```

MATLAB returns

```
MATLABPATH
```

```
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

After clearing the `userpath` value, unless you otherwise specify the startup folder, the startup folder will be the desktop on Windows platforms. There are a number of ways to specify the startup folder. For example, right-click the Windows shortcut icon for MATLAB and select **Properties** from the context menu. In the Properties dialog box **Shortcut** tab, enter the full path to the new startup folder in the **Start in** field, for example, `I\:my_matlab_files\my_mfiles`. The next time you start MATLAB, the current folder will be `I\:my_matlab_files\my_mfiles`, but that folder will *not* be on the search path. Note that you do not have to clear `userpath` to specify a different startup folder; when you otherwise specify a startup folder, the `userpath` folder is added to the search path upon startup, but is not the startup folder.

Removing *userpath* from the Search Path; Resets the Startup Folder

In this example, `userpath` is set to the default value and you remove the `userpath` folder from the search path, then save the changes. This has the same effect as clearing the value for `userpath`. Confirm the default is currently set:

```
userpath
```

MATLAB returns:

```
H:\My Documents\MATLAB
```

See the `userpath` folder at the top of the search path:

```
path
```

MATLAB returns:

```
MATLABPATH
```

```
H:\My Documents\MATLAB
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Remove `H:\My Documents\MATLAB` from the search path and confirm the result:

```
rmpath('H:\My Documents\MATLAB')
path
```

MATLAB returns:

```
MATLABPATH

 C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Verify the value:

```
userpath
```

MATLAB returns:

```
H:\My Documents\MATLAB
```

Save changes to the search path:

```
savepath
```

View the value:

```
userpath
```

MATLAB returns:

```
ans =
     ''
```

The value is now cleared. Removing the folder from the search path *and* saving the changes to the path has the same effect as clearing the value for *userpath*. At the next startup, the startup folder will *not* be `H:\My Documents\MATLAB`, and `H:\My Documents\MATLAB` will *not* be on the search path.

Assigning *userpath* as the Startup Folder on a UNIX or Macintosh Platform

userpath is set to the default value on a Macintosh platform and you start MATLAB using a bash X11 shell, where `smith` is the home directory. Set the `MATLAB_USE_USERPATH` environment variable so that *userpath* will be used as the startup folder:

```
export MATLAB_USE_USERPATH=1
```

From that shell, start MATLAB. After MATLAB starts, verify the current folder in MATLAB:

```
pwd
```

MATLAB returns:

```
/Users/smith/Documents/MATLAB
```

That is the value defined for *userpath*, which you can confirm:

```
userpath
```

MATLAB returns:

```
/Users/smith/Documents/MATLAB
```

The *userpath* is at the top of the search path, which you can confirm:

```
path
```

MATLAB returns:

```
/Users/smith/Documents/MATLAB
/Users/smith/Applications/MATLAB/R2009a/toolbox/matlab/general
```

```
/Users/smith/Applications/MATLAB/R2009a/toolbox/matlab/ops

...
```

Adding Folders to the Search Path Upon Startup on a UNIX or Macintosh Platform

userpath is set to the default value on a UNIX platform with a csh shell, where j is the user's home directory.

To add additional folders to the search path upon startup, for example, /home/j/Documents/MATLAB/mine and /home/j/Documents/MATLAB/mine/research, run the following in an X11 terminal:

```
setenv MATLABPATH '/home/j/Documents/MATLAB/mine':'/home/j/Documents/MATLAB/mine/research'
```

Separate multiple folders using a : (colon).

MATLAB displays

```
MATLABPATH

home/j/Documents/MATLAB
home/j/Documents/MATLAB/mine
home/j/Documents/MATLAB/mine/research
home/j/Applications/MATLAB/R2009a/toolbox/matlab/general
home/j/Applications/MATLAB/R2009a/toolbox/matlab/ops
...
```

See Also

addpath, path, pathtool, rmpath, savepath, startup,

Topics in the User Guide:

-
-

Purpose

Check validity of array

Syntax

```
validateattributes(A, classes, attributes)
validateattributes(A, classes, attributes, position)
validateattributes(A, classes, attributes, funcname)
validateattributes(A, classes, attributes,
funcname, varname)
validateattributes(A, classes, attributes,
funcname, varname,
   position)
```

Description

`validateattributes(A, classes, attributes)` validates that array `A` belongs to at least one of the classes specified by the *`classes`* input and has all of the attributes specified by the *`attributes`* input. If the validation succeeds, the command completes without displaying any output and without throwing an error. If the validation does not succeed, MATLAB issues a formatted error message.

The *`classes`* input is a cell array containing one or more strings from the Class Values on page 2-4115 table shown below.

The *`attributes`* input is a cell array containing one or more strings from the Attribute Values on page 2-4116 table shown below. Size validation requires two inputs: the `'size'` keyword and the length of each dimension (e.g., `{'size', [4,3,7]}`). Value range validation requires two inputs for each aspect of the range being validated (e.g., `{'>', 10, '<=', 65}`).

`validateattributes(A, classes, attributes, position)` validates array *`A`* and, if the validation fails, displays an error message that includes the position of the failing variable in the function argument list. The *`position`* input must be a positive integer.

`validateattributes(A, classes, attributes, funcname)` validates array *`A`* and, if the validation fails, displays an error message that includes the name of the function performing the validation (*`funcname`*). The *`funcname`* input must be a string.

`validateattributes(A, classes, attributes, funcname, varname)` validates array *A* and, if the validation fails, displays an error message that includes the name of the function performing the validation (*funcname*), and the name of the variable being validated (*varname*). The *funcname* and `varname` inputs must be strings enclosed in single quotation marks.

`validateattributes(A, classes, attributes, funcname, varname, position)` validates array *A* and, if the validation fails, displays an error message that includes the name of the function performing the validation (*funcname*), the name of the variable being validated (`varname`), and the position of this variable in the function argument list (*position*). The *funcname* and *varname* inputs must be strings enclosed in single quotation marks. The *position* input must be a positive integer.

Class Values

`classes` Argument	Contents of Array `A`
`'numeric'`	Any numeric value
`'single'`	Single-precision number
`'double'`	Double-precision number
`'int8'`	Signed 8-bit integer
`'int16'`	Signed 16-bit integer
`'int32'`	Signed 32-bit integer
`'int64'`	Signed 64-bit integer
`'uint8'`	Unsigned 8-bit integer
`'uint16'`	Unsigned 16-bit integer
`'uint32'`	Unsigned 32-bit integer
`'uint64'`	Unsigned 64-bit integer
`'logical'`	Logical `true` or `false`

Class Values (Continued)

`classes` Argument	Contents of Array `A`
`’'char'`	Character or string
`'struct'`	MATLAB structure
`'cell'`	Cell array
`'function_handle'`	Scalar function handle
class name	Object of any MATLAB class

Attribute Values

`attributes` Argument	Description of array `A`
`'>', N`	Array in which all values are greater than `N`.
`'>=', N`	Array in which all values are greater than or equal to `N`.
`'<', N`	Array in which all values are less than `N`.
`'<=', N`	Array in which all values are lass than or equal to `N`.
`'2d'`	Array having dimensions `M`-by-`N` (includes scalars, vectors, 2-D matrices, and empty arrays)
`'binary'`	Array of ones and zeros
`'column'`	Array having dimensions `N`-by-1
`'even'`	Numeric or logical array in which all elements are even (includes zero)
`'finite'`	Numeric array in which all elements are finite
`'integer'`	Numeric array in which all elements are integer-valued
`'nonempty'`	Array having no dimension equal to zero

Attribute Values (Continued)

attributes Argument	Description of array A
'nonnan'	Numeric array in which there are no elements equal to NaN (Not a Number)
'nonnegative'	Numeric array in which all elements are zero or greater than zero
'nonsparse'	Array that is not sparse
'nonzero'	Numeric or logical array in which all elements are less than or greater than zero
'odd'	Numeric or logical array in which all elements are odd integers
'positive'	Numeric or logical array in which all elements are greater than zero
'real'	Numeric array in which all elements are real
'row'	Array having dimensions 1-by-N
'scalar'	Array having dimensions 1-by-1
'size', [M,N,...]	Array having dimensions M-by-N-by- ....
'vector'	Array having dimensions N-by-1 or 1-by-N (includes scalar arrays)

Numeric properties, such as `positive` and `nonnan`, do not apply to strings. If you attempt to validate numeric properties on a string, `validateattributes` generates an error.

Examples

Example 1

In this example, the `empl_profile1` function compares the values passed in each argument to the specified classes and attributes and throws an error if they are not correct:

```
function empl_profile1(empl_id, empl_info, healthplan, ...
   vacation)
validateattributes(empl_id, {'numeric'}, ...
   {'integer', 'nonempty'});
validateattributes(empl_info, {'struct'}, {'vector'});
validateattributes(healthplan, {'cell', 'char'}, ...
   {'vector'});
validateattributes(vacation, {'numeric'}, ...
   {'nonnegative', 'scalar'});
```

Call the `empl_profile1` function, passing the expected argument types, and the example completes without error:

```
empl_id = 51723;
empl_info.name = 'John Miller';
empl_info.address = '128 Forsythe St.';
empl_info.town = 'Duluth';  empl_info.state='MN';

empl_profile1(empl_id, empl_info, 'HCP Medical Plus', 14.3)
```

If you accidentally pass the argument values out of their correct sequence, MATLAB throws an error in response to the first argument that is not a match:

```
empl_profile1(empl_id, empl_info, 14.3, 'HCP Medical Plus')

??? Error using ==> empl_profile1 at 6
Expected input to be one of these types:

  cell, char

Instead its type was double.
```

Example 2

Write a new function `empl_profile2` that displays the function name, variable name, and position of the argument:

```
function empl_profile2(empl_id, empl_info, healthplan, ...
   vacation)

validateattributes(empl_id, ...
   {'numeric'}, {'integer', 'nonempty'}, ...
   mfilename, 'Employee Identification', 1);

validateattributes(empl_info, ...
   {'struct'}, {'vector'}, ...
   mfilename, 'Employee Info', 2);

validateattributes(healthplan, ...
   {'cell', 'char'}, {'vector'}, ...
   mfilename, 'Health Plan', 3);

validateattributes(vacation, ...
   {'numeric'}, {'nonnegative', 'scalar'}, ...
   mfilename, 'Vacation Accrued', 4);
```

Call `empl_profile2` with the argument values out of sequence. MATLAB throws an error that includes the name of the function validating the attributes, the name of the variable that was in error, and its position in the input argument list:

```
??? Error using ==> empl_profile2
Expected input number 3, Health Plan, to be one of
these types:

  cell, char

Instead its type was double.

Error in ==> empl_profile2 at 12
validateattributes(healthplan, ...
```

Example 3

Write a new function empl_profile3 that checks the input parameters with inputParser. Use validateattributes as the validating function for the inputParser methods:

```
function empl_profile3(empl_id, varargin)
p = inputParser;

% Validate the input arguments.
addRequired(p, 'empl_id', ...
   @(x)validateattributes(x, {'numeric'}, {'integer'}));
addOptional(p, 'empl_info', '', ...
   @(x)validateattributes(x, {'struct'}, {'nonempty'}));
addParamValue(p, 'health', 'HCP Medical Plus', ...
   @(x)validateattributes(x, {'cell', 'char'}, ...
   {'vector'}));
addParamValue(p, 'vacation', [], ...
   @(x)validateattributes(x, {'numeric'}, ...
   {'nonnegative', 'scalar'}));
parse(p, empl_id, varargin{:});
p.Results
```

Call empl_profile3 using appropriate input arguments:

```
empl_info.name = 'John Miller';
empl_info.address = '128 Forsythe St.';
empl_info.town = 'Duluth';  empl_info.state='MN';

empl_profile3(51723, empl_info, 'vacation', 14.3)

ans =
     empl_id: 51723
   empl_info: [1x1 struct]
      health: 'HCP Medical Plus'
    vacation: 14.3000
```

Call `empl_profile3` using a character string where a structure is expected:

```
empl_profile3(51723, empl_info.name,  'vacation', 14.3)

??? Error using ==> empl_profile3 at 12
Argument 'empl_info' failed validation with error:
Expected input to be one of these types:

  struct

Instead its type was char.
```

Example 4

Create a 4-by-2-by-6 array and then validate its size:

```
x = rand(4,2,6);

validateattributes(x, {'numeric'}, {'size', [4,2,6]});
```

Create an array of integers between 50 and 200 and then validate that these values are within the intended range:

```
y = uint8(50:10:200);

validateattributes(y, {'uint8'}, {'>=', 50, '<=', 200})
```

This next statement fails for `y(end)`:

```
validateattributes(y, {'uint8'}, {'>=', 50, '<', 200})
??? Expected input to be an array with all of the values < 200.
```

Example 5

Generate a new array `z` and validate that it is a 4-by-2-by-6 nonsparse array of class `double`, with all elements being between 0.005 and 50, inclusive:

```
z = rand(4,2,6) * 50;
```

```
validateattributes(z, {'numeric', 'double'}, ...
   {'<', 50, 'size', [4 2 6], 'nonsparse', '>=', .005});
```

There are several things to note in the above statement:

- All class arguments are enclosed in just one set of curly braces `{}`. All attribute arguments the same way.
- The placement of the `<`, `<=`, `>`, and `>=` arguments in the argument list is unimportant. However, you must immediately follow any of these arguments with the numeric argument it relates to.
- The placement of the `'size'` argument in the argument list is unimportant. However, you must immediately follow this argument with the numeric vector argument it relates to.

If you add to this a requirement that `z` be two-dimensional, `validateattributes` throws an error because `z` has three dimensions:

```
validateattributes(z, {'double'}, ...
   {z, '<', 50, 'size', [4 2 6], '2d', 'positive', '>', 0});
Warning: Failed to find attribute in list.
??? Expected input to be two-dimensional.
```

See Also

`validatestring`, `is*`, `isa`, `inputParser`

Purpose

Check validity of text string

Syntax

```
validstr = validatestring(str, strarray)
validstr = validatestring(str, strarray, position)
validstr = validatestring(str, strarray, funname)
validstr = validatestring(str, strarray, funname, varname)
validstr = validatestring(str, strarray, funname, varname,
   position)
```

Description

`validstr = validatestring(str, strarray)` checks the validity of text string `str`. If `str` matches one or more of the text strings in the cell array `strarray`, MATLAB returns the matching string in `validstr`. If `str` does not match any of the strings in `strarray`, MATLAB issues a formatted error message. MATLAB compares the strings without respect to letter case.

This table shows how `validatestring` determines what value to return. If multiple matches are found, `validatestring` returns the shortest matching string.

Type of Match	Example — Match 'ball' with . . .	Return Value
Exact match	`ball`, `barn`, `bell`	`ball`
Partial match (leading characters)	`balloon`, `barn`	`balloon`
Multiple partial matches where each string is a subset of another	`ball`, `ballo`, `balloo`, `balloon`	`ball`
Multiple partial matches where strings are unique	`balloon`, `ballet`	Error
No match	`barn`, `bell`	Error

`validstr = validatestring(str, strarray, position)` checks the validity of text string `str` and, if the validation fails, displays an error message that includes the position of the failing variable in the function argument list. The `position` input must be a positive integer.

validstr = validatestring(str, strarray, funname) checks the validity of text string str and, if the validation fails, displays an error message that includes the name of the function performing the validation (funname). The funname input must be a string enclosed in single quotation marks.

validstr = validatestring(str, strarray, funname, varname) checks the validity of text string str and, if the validation fails, displays an error message that includes the name of the function performing the validation (funname) and the name of the variable being validated (varname). The funname and varname inputs must be strings enclosed in single quotation marks.

validstr = validatestring(str, strarray, funname, varname, position) checks the validity of text string str and, if the validation fails, displays an error message that includes the name of the function performing the validation (funname), the name of the variable being validated (varname), and the position of this variable in the function argument list (position). The funname and varname inputs must be strings enclosed in single quotation marks. The position input must be a positive integer.

Examples

Example 1

Use validatestring to find the word won in the cell array of strings:

```
validatestring('won', {'wind', 'won', 'when'})
ans =
   won
```

Replace the word won with wonder in the string array. Because the leading characters of the input string and wonder are the same, validatestring finds a partial match between the two words and returns the full word wonder:

```
validatestring('won', {'wind', 'wonder', 'when'})
ans =
   wonder
```

If there is more than one partial match, and each string in the array is a subset or superset of the others, `validatestring` returns the shortest matching string:

```
validatestring('wond', {'won', 'wonder', 'wonderful'})
ans =
   wonder
```

However, if each string in the array is not subset or superset of each other, MATLAB throws an error because there is no exact match and it is not clear which of the two partial matches should be returned:

```
validatestring('wond', {'won', 'wonder', 'wondrous'})
??? Error using ==> validatestring at 89
Function VALIDATESTRING expected its input argument to
   match one of these strings:

  won, wonder, wondrous

The input, 'wond', matched more than one valid string.
```

Example 2

In this example, the `get_flight_numbers` function returns the flight numbers for routes between two cities: a point of origin and point of destination. The function uses `validatestring` to see if the origin and destination are among those covered by the airline. If not, an error message is displayed:

```
function get_flight_numbers(origin, destination)
% Only part of the airline's flight data is shown here.
   flights.chi2rio = [503, 196, 331, 373, 1475];
   flights.chi2par = [718, 9276, 172, 903, 7724 992, 1158];
   flights.chi2hon = [9193, 880, 471, 391];

   routes = {'Athens', 'Paris', 'Chicago', 'Sydney', ...
      'Cancun', 'London', 'Rio de Janeiro', 'Honolulu', ...
      'Rome', 'New York City'};
   orig = '';  dest = '';
```

```
         % Does the airline cover these cities?
         try
            orig = validatestring(origin, routes);
            dest = validatestring(destination, routes);
         catch
            % If not covered, then display error message.
            if isempty(orig)
               fprintf(...
                  'We have no flights with origin: %s.\n', ...
                  origin)
            elseif isempty(dest)
               fprintf('%s%s%s.\n', 'We have no flights ', ...
                       'with destination: ', destination)
            end
         return
         end

      % If covered, display the flights from 'orig' to 'dest'.
      fprintf(...
         'Flights available from %s to %s are:\n', orig, dest)
      reply = eval(...
        ['flights.' lower(orig(1:3)) '2' lower(dest(1:3))])';
      fprintf('   Flight %d\n', reply)
```

Enter a point of origin that is not covered by this airline:

```
get_flight_numbers('San Diego', 'Rio de Janeiro')
ans =
We have no flights with origin: San Diego.
```

Enter a destination that is misspelled:

```
get_flight_numbers('Chicago', 'Reo de Janeiro')
ans =
We have no flights with destination: Reo de Janeiro.
```

Enter a route that is covered:

```
get_flight_numbers('Chicago', 'Rio de Janeiro')
ans =
Flights available from Chicago to Rio de Janeiro are:
   Flight 503
   Flight 196
   Flight 331
   Flight 373
   Flight 1475
```

Example 3

Rewrite the try-catch block of Example 2 by adding funname, varname, and position arguments to the call to validatestring and replacing the return statement with rethrow:

```
% See if the cities entered are covered by this airline.
try
   orig = validatestring(...
      origin, routes, mfilename, 'Flight Origin', 1);
   dest = validatestring(...
      destination, routes, mfilename, ...
        'Flight Destination', 2);
catch e
   % If not covered, then display error message.
   if isempty(orig)
       fprintf(...
          'We have no flights with origin: %s.\n', ...
          origin)
   elseif isempty(dest)
       fprintf('%s%s%s.\n', 'We have no flights ', ...
               'with destination: ', destination)
   end
   rethrow(e);
end
```

In response to the rethrow command, MATLAB displays an error message that includes the function name get_flight_numbers, the failing variable name `Flight Destination`', and its position in the argument list, 2:

```
get_flight_numbers('Chicago', 'Reo de Janeiro')
We have no flights with destination: Reo de Janeiro.

??? Error using ==> validatestring at 89
Function GET_FLIGHT_NUMBERS expected its input argument
    number 2, Flight Destination, to match one of these
    strings:

  Athens, Paris, Chicago, Sydney, Cancun, London, Rio de
Janeiro, Honolulu, Rome

The input, 'Reo de Janeiro', did not match any of the valid
   strings.

Error in ==> get_flight_numbers at 17
   dest = validatestring(destination, routes, mfilename,
   'destination', 2);
```

See Also

`validateattributes`, `is*`, `isa`, `inputParser`

Purpose

Return values of `containers.Map` object

Syntax

```
v = values(M)
v = values(M, keys)
```

Description

`v = values(M)` returns in cell array `v` the values that correspond to all keys in `Map` object `M`.

`v = values(M, keys)` returns in cell array `v`, those values in `Map` object `M` that correspond to the keys specified by the `keys` argument.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

Examples

Create a `Map` object of four US states and their capital cities:

```
US_Capitals = containers.Map( ...
   {'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...
   {'Atlanta', 'Juneau', 'Montpelier', 'Salem'})
```

Find the capital cities of all states contained in the map:

```
v = values(US_Capitals)
v =
    'Juneau'    'Atlanta'    'Salem'    'Montpelier'
```

Find the capital cities of selected states:

```
 = values(US_Capitals, {'Oregon', 'Alaska'})
v =
    'Salem'    'Juneau'
```

See Also

`containers.Map`, `keys(Map)`, `size(Map)`, `length(Map)`, `isKey(Map)`, `remove(Map)`, `handle`

Purpose Vandermonde matrix

Syntax

```
A = vander(v)
```

Description `A = vander(v)` returns the Vandermonde matrix whose columns are powers of the vector `v`, that is, `A(i,j) = v(i)^(n-j)`, where `n = length(v)`.

Examples

```
vander(1:.5:3)

ans =

    1.0000    1.0000    1.0000    1.0000    1.0000
    5.0625    3.3750    2.2500    1.5000    1.0000
   16.0000    8.0000    4.0000    2.0000    1.0000
   39.0625   15.6250    6.2500    2.5000    1.0000
   81.0000   27.0000    9.0000    3.0000    1.0000
```

See Also `gallery`

Purpose

Variance

Syntax

```
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

Example 1

Write an M-file function that displays the expected and optional arguments you pass to it

```
function vartest(argA, argB, varargin)

optargin = size(varargin,2);
stdargin = nargin - optargin;

fprintf('Number of inputs = %d\n', nargin)

fprintf('  Inputs from individual arguments(%d):\n', ...
        stdargin)
if stdargin >= 1
    fprintf('     %d\n', argA)
end
if stdargin == 2
    fprintf('     %d\n', argB)
end

fprintf('  Inputs packaged in varargin(%d):\n', optargin)
 for k= 1 : size(varargin,2)
     fprintf('     %d\n', varargin{k})
 end
```

Call this function and observe that the MATLAB software extracts those arguments that are not individually-specified from the `varargin` cell array:

```
vartest(10,20,30,40,50,60,70)
Number of inputs = 7
  Inputs from individual arguments(2):
     10
     20
  Inputs packaged in varargin(5):
     30
     40
     50
     60
     70
```

Example 2

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`

varargout

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`

Purpose

Vectorize expression

Syntax

```
vectorize(s)
vectorize(fun)
```

Description

`vectorize(s)` where `s` is a string expression, inserts a `.` before any `^`, `*` or `/` in `s`. The result is a character string.

`vectorize(fun)` when `fun` is an inline function object, vectorizes the formula for `fun`. The result is the vectorized version of the inline function.

See Also

`inline`, `cd`, `dbtype`, `delete`, `dir`, `path`, `what`, `who`

Purpose

Version information for MathWorks products

GUI Alternatives

As an alternative to the ver function, select **Help > About** in any tool that has a **Help** menu.

Syntax

```
ver
ver product
v = ver('product')
```

Description

ver displays a header containing the current MathWorks product family version number, license number, operating system, and version of Sun Microsystems JVM software for the MATLAB product. This is followed by the version numbers for MATLAB, Simulink, if installed, and all other installed MathWorks products.

ver product displays the MathWorks product family header information followed by the current version number for product. The name product corresponds to the folder name that holds the Contents.m file for that product. For example, Contents.m for the Control System Toolbox product resides in the control folder. You therefore use ver control to obtain the version of this toolbox.

v = ver('product') returns the version information to structure array, v, having fields Name, Version, Release, and Date.

Remarks

To use ver with your own collection of files, see .

Examples

Using R2009b, return version information for MathWorks products, and specifically the Control System Toolbox product:

```
ver control
```

MATLAB returns:

```
-------------------------------------------------------------------------------------
MATLAB Version 7.9.0.3512 (R2009b)
MATLAB License Number: [not shown]
Operating System: Microsoft Windows XP Version 5.1 (Build 2600: Service Pack 3)
```

```
Java VM Version: Java 1.6.0_12-b04 with Sun Microsystems Inc. Java HotSpot(TM) Client VM mixed mo
-------------------------------------------------------------------------------------
Control System Toolbox                                 Version 8.3       (R2009b)
```

Return version information for the Control System Toolbox product in a structure array, `v`.

```
v = ver('control')
v =

       Name: 'Control System Toolbox'
    Version: '8.4'
    Release: '(R2009b)'
       Date: '24-Sep-2009'
```

Display version information for 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 3.4
Real-Time Workshop, Version 7.4
Real-Time Workshop Embedded Coder, Version 5.4
```

See Also

`computer`, `help`, `hostid`, `license`, `verlessthan`, `version`, `whatsnew`

Purpose

Source control actions (Windows platforms)

GUI Alternatives

As an alternative to the verctrl function, use **Source Control** in the **File** menu of the Editor, the Simulink product, or the Stateflow product, or in the context menu of the Current Folder 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 file name and include the extensions. Specify `0` as the last argument. Complete the resulting dialog box to execute the operation. 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 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.
`'runscc'`	Starts the source control system. The file name 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 absolute path 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`

in MATLAB Desktop Tools and Development Environment documentation

verLessThan

Purpose

Compare toolbox version to specified version string

Syntax

```
verLessThan(toolbox, version)
```

Description

`verLessThan(toolbox, version)` returns logical 1 (`true`) if the version of the toolbox specified by the string `toolbox` is older than the version specified by the string `version`, and logical 0 (`false`) otherwise. Use this function when you want to write code that can run across multiple versions of the MATLAB software, when there are differences in the behavior of the code in the different versions.

The `toolbox` argument is a string enclosed within single quotation marks that contains the name of a MATLAB toolbox folder. The `version` argument is a string enclosed within single quotation marks that contains the version to compare against. This argument must be in the form `major[.minor[.revision]]`, such as `7`, `7.1`, or `7.0.1`. If `toolbox` does not exist, MATLAB generates an error.

To specify `toolbox`, find the folder that holds the `Contents.m` file for the toolbox and use that folder name. To see a list of all toolbox folder names, enter the following statement in the MATLAB Command Window:

```
dir([matlabroot '/toolbox'])
```

Remarks

The `verLessThan` function is available with MATLAB Version 7.4 and subsequent versions. If you are running a version of MATLAB prior to 7.4, you can download the `verLessThan` M-file from the following MathWorks Technical Support solution. You must be running MATLAB Version 6.0 or higher to use this M-file:

`http://www.mathworks.com/support/solutions/data/1-38LI61.html?solution=1-`

Examples

These examples illustrate usage of the `verLessThan` function.

Example 1 – Checking For the Minimum Required Version

```
if verLessThan('simulink', '4.0')
    error('Simulink 4.0 or higher is required.');
```

```
end
```

Example 2 – Choosing Which Code to Run

```
if verLessThan('matlab', '7.0.1')
% -- Put code to run under MATLAB 7.0.0 and earlier here --
else
% -- Put code to run under MATLAB 7.0.1 and later here --
end
```

Example 3 – Looking Up the Folder Name

Find the name of the Data Acquisition Toolbox folder:

```
dir([matlabroot '/toolbox/d*'])

   daq        database   des      distcomp   dotnetbuilder
   dastudio   datafeed   dials    dml        dspblks
```

Use the toolbox folder name, `daq`, to compare the Data Acquisition Toolbox software version that MATLAB is currently running against version number `3`:

```
verLessThan('daq', '3')
ans =
     1
```

See Also

`ver`, `version`, `license`, `ispc`, `isunix`, `ismac`, `dir`

version

Purpose

Version number for MATLAB and libraries

Syntax

```
version
version('-date')
version('-description')
version('-release')
version('-java')
version -versionOption
v = version('-versionOption')
```

Description

`version` displays the version and release number for the MATLAB software currently running.

`version('-date')` displays the release date for the MATLAB software.

`version('-description')` displays a description of the version. Usually, the description is for special versions, such as beta versions.

`version('-release')` displays the release number for the MATLAB software currently running.

`version('-java')` displays the version of the Sun Microsystems JVM software that MATLAB is using.

`version -`*`versionOption`* is an alternate form of the syntax.

`v = version('-`*`versionOption`*`')` returns a string containing the result of `version`.

Examples

Display the version:

```
version
```

MATLAB returns:

```
7.9.0.2601 (R2009b)
```

Display the release, prefaced by a descriptor:

```
['Release R' version('-release')]
```

MATLAB returns:

```
Release R2009b
```

View the Java version:

```
version -java
```

MATLAB returns:

```
Java 1.6.0_12-b04 with Sun Microsystems Inc. Java HotSpot(TM) Clien
```

Alternatives

To view version information, select **Help > About MATLAB** in the MATLAB desktop.

See Also

computer | ver | verlessthan | whatsnew

How To

-
-

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`

vertcat (tscollection)

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`

TriRep.vertexAttachments

Purpose Return simplices attached to specified vertices

Syntax `SI = vertexAttachments(TR, VI)`

Description `SI = vertexAttachments(TR, VI)` returns the vertex-to-simplex information for the specified vertices `VI`. In relation to 2-D triangulations, if the triangulation has a consistent orientation the triangles in each cell will be ordered consistently around each vertex.

Inputs

`TR`	Triangulation representation
`VI`	`VI` is a column vector of indices into the array of points representing the vertex coordinates, `TR.X`. The simplices associated with vertex `i` are the `i`'th entry in the cell array. If `VI` is not specified the vertex-simplex information for the entire triangulation is returned.

Outputs

`SI`	Cell array of indices of the simplices attached to a vertex. A cell array is used to store the information because the number of simplices associated with each vertex can vary. The simplices associated with vertex `i` are in the `i`'th entry in the cell array `SI`.

Definitions A simplex is a triangle/tetrahedron or higher dimensional equivalent.

Examples

Example 1

Load a 2-D triangulation and use `TriRep` to compute the vertex-to-triangle relations.

```
load trimesh2d
```

Find the indices of the tetrahedra attached to the first vertex:

```
Tv = vertexAttachments(trep, 1)
Tv{:}
```

Example 2

Perform a direct query of a 2-D triangulation created using `DelaunayTri`.

```
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y);
```

Find the triangles attached to vertex 5:

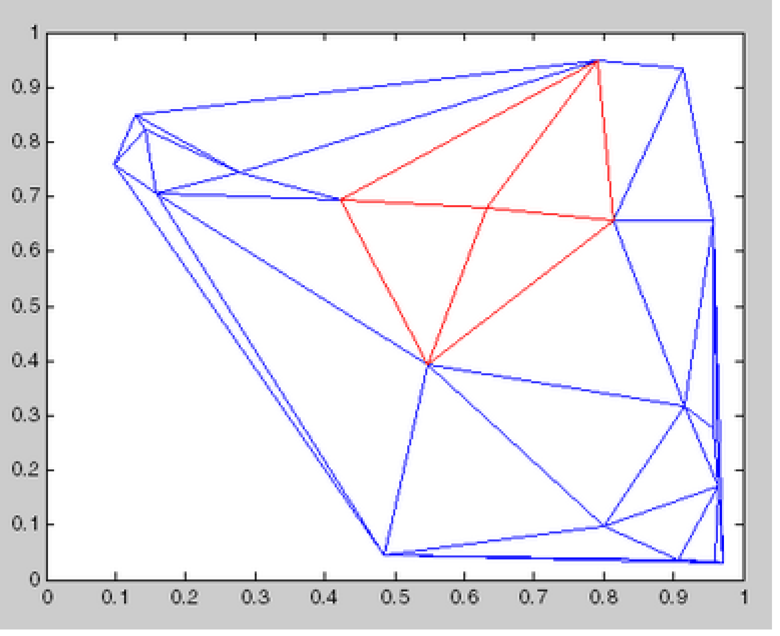
```
t = vertexAttachments(dt,5);
```

Plot the triangulation:

```
triplot(dt);
hold on;
```

Plot the triangles attached to vertex 5 (in red):

```
triplot(dt(t{:},:),x,y,'Color','r');
hold off;
```



See Also `DelaunayTri`

Purpose

Viewpoint specification

Syntax

```
view(az,el)
view([az,el])
view([x,y,z])
view(2)
view(3)
view(ax,...)
[az,el] = view
T = view
```

Description

The position of the viewer (the viewpoint) determines the orientation of the axes. You specify the viewpoint in terms of azimuth and elevation, or by a point in three-dimensional space.

`view(az,el)` and `view([az,el])` set the viewing angle for a three-dimensional plot. The azimuth, `az`, is the horizontal rotation about the *z*-axis as measured in degrees from the negative *y*-axis. Positive values indicate counterclockwise rotation of the viewpoint. `el` is the vertical elevation of the viewpoint in degrees. Positive values of elevation correspond to moving above the object; negative values correspond to moving below the object.

`view([x,y,z])` sets the viewpoint to the Cartesian coordinates `x`, `y`, and `z`. The magnitude of `(x,y,z)` is ignored.

`view(2)` sets the default two-dimensional view, `az = 0, el = 90`.

`view(3)` sets the default three-dimensional view, `az =  37.5, el = 30`.

`view(ax,...)` uses axes `ax` instead of the current axes.

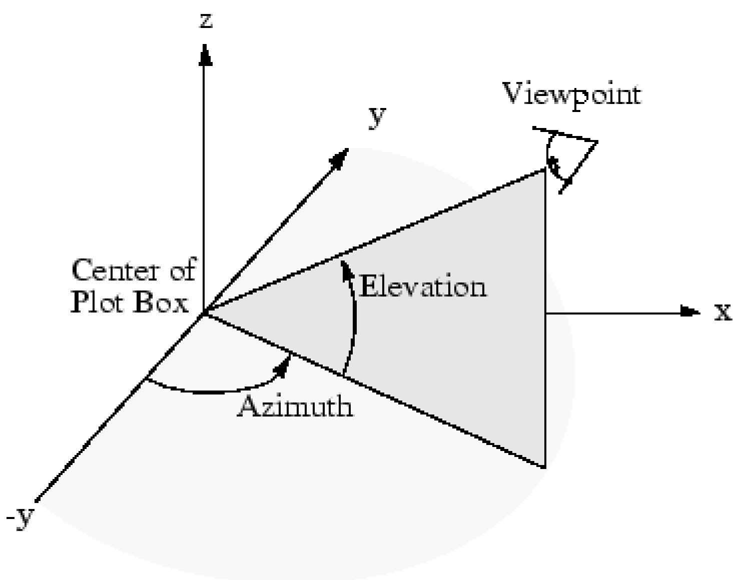
`[az,el] = view` returns the current azimuth and elevation.

`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 180º.

```
az = 180;
el = 90;
```

```
view(az, el);
```

See Also

viewmtx, hgtransform, rotate3d

"Camera Viewpoint" on page 1-104 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

viewmtx

Purpose

View transformation matrices

Syntax

```
viewmtx
T = viewmtx(az,el)
T = viewmtx(az,el,phi)
T = viewmtx(az,el,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,el)` returns an *orthographic* transformation matrix corresponding to azimuth `az` and elevation `el`. `az` is the azimuth (i.e., horizontal rotation) of the viewpoint in degrees. `el` is the elevation of the viewpoint in degrees. This returns the same matrix as the commands

```
view(az,el)
T = view
```

but does not change the current view.

`T = viewmtx(az,el,phi)` returns a *perspective* transformation matrix. `phi` is the perspective viewing angle in degrees. `phi` is the subtended view angle of the normalized plot cube (in degrees) and controls the amount of perspective distortion.

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

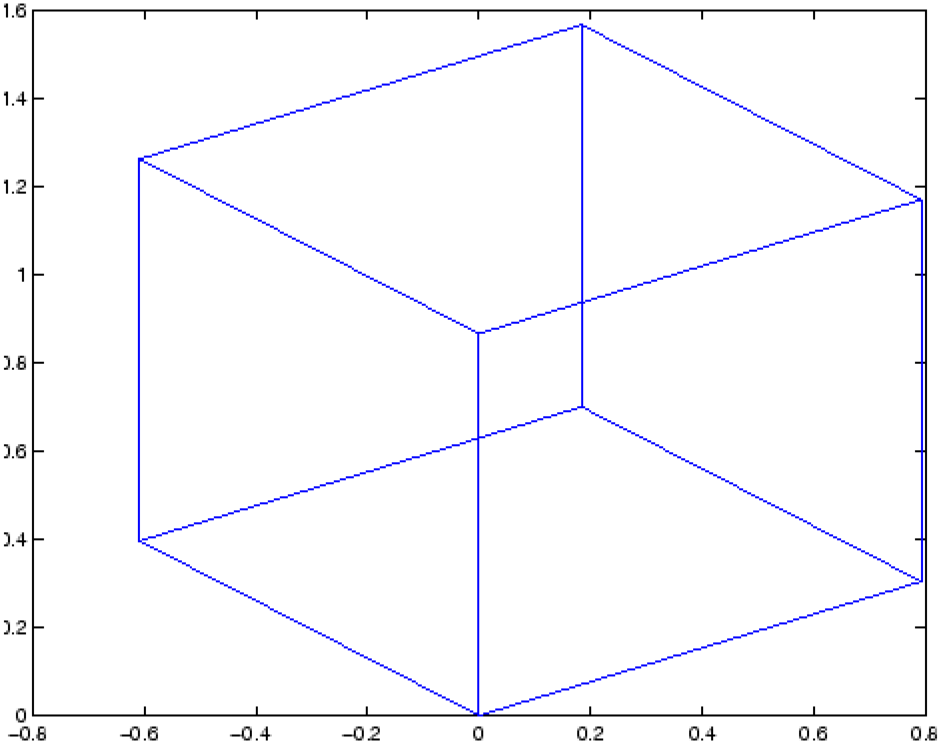
`T = viewmtx(az,el,phi,xc)` returns the perspective transformation matrix using `xc` as the target point within the normalized plot cube (i.e., the camera is looking at the point `xc`). `xc` is the target point that is the center of the view. You specify the point as a three-element vector, `xc = [xc,yc,zc]`, in the interval [0,1]. The default value is `xc = [0,0,0]`.

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)

% 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  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)
```

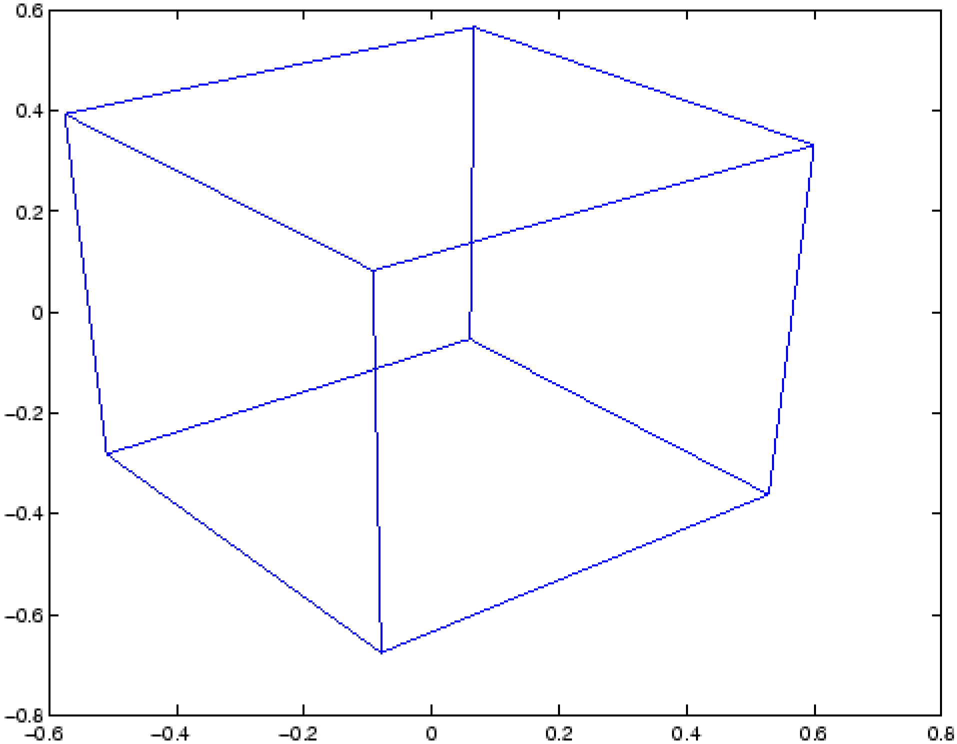


Use a perspective transformation with a 25 degree viewing angle:

```
A = viewmtx(-37.5,30,25);
x4d = [.5  0  -3  1]';
x2d = A*x4d;
x2d = x2d(1:2)/x2d(4)   % Normalize
x2d =
     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`

Tutorials

- Defining the View

Purpose

Compare two text files, MAT-Files, or binary files

GUI Alternative

As an alternative to the `visdiff` function:

1. Select **Desktop > File and Folder Comparisons**.
2. In the File and Folder Comparisons tool select one of the following:
   - **File > New File Comparison**
   - **File > New Folder Comparison**

Syntax

```
visdiff('fname1', 'fname2')
visdiff('fname1', 'fname2', showchars)
S = visdiff('fname1', 'fname2')
```

Description

`visdiff('fname1', 'fname2')` opens the File and Folder Comparisons tool and presents the differences between the two files. The two files must be on the MATLAB path, or you must provide the full path for each file.

`visdiff('fname1', 'fname2', showchars)` opens the File and Folder Comparisons tool with the width of each column in the display set to `showchars` characters wide. MATLAB ignores the `showchars` option when you compare two binary files or two MAT-Files.

`S = visdiff('fname1', 'fname2')` creates an HTML report describing the differences between the two files and returns it in the string `S`.

Remarks

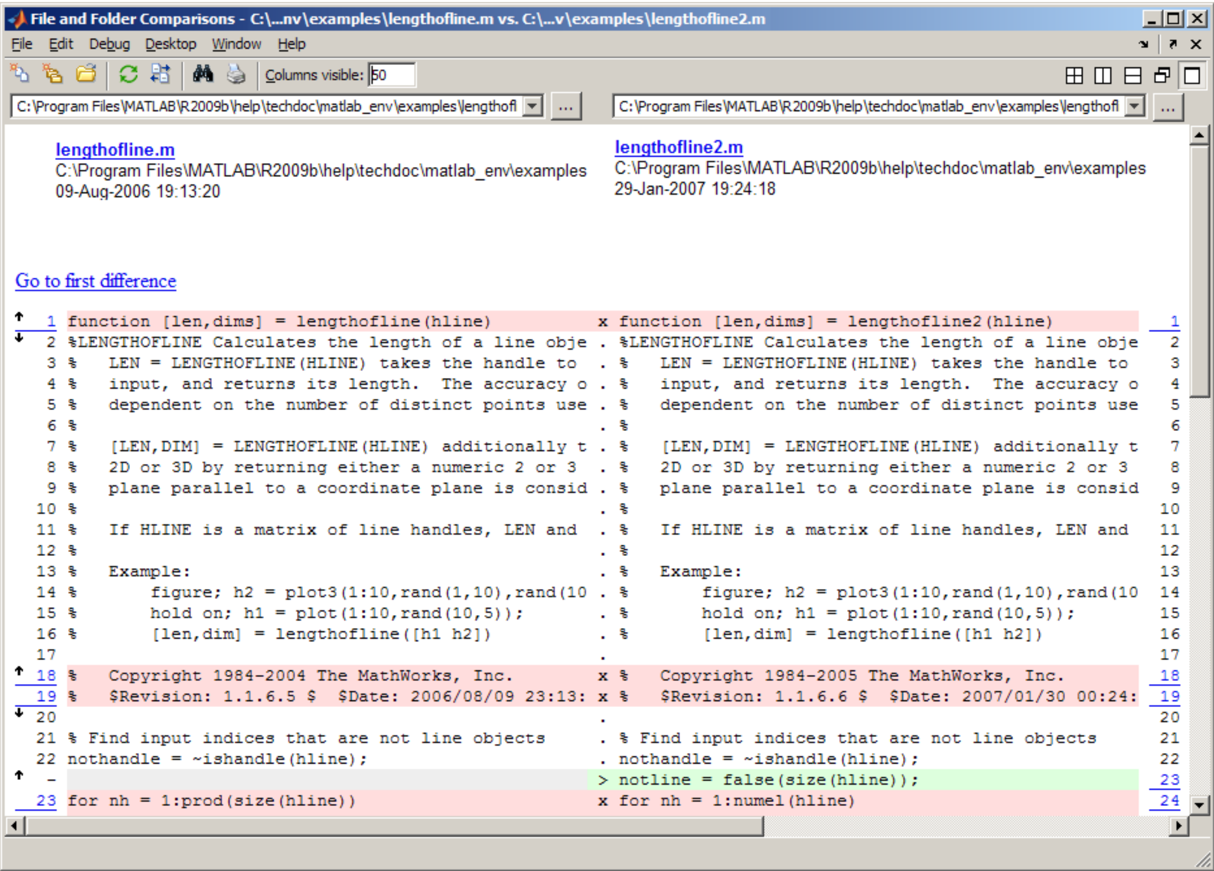
MATLAB supports displaying the differences in the File and Folder Comparisons tool only if Java software is installed. However, MATLAB supports assigning the HTML report to a string, even if the Java software is not installed.

Examples

Compare Two Text Files in the Current Folder

Compare the example files, `lengthofline.m` and `lengthofline2.m` and set the width of each column to 50 characters wide. :

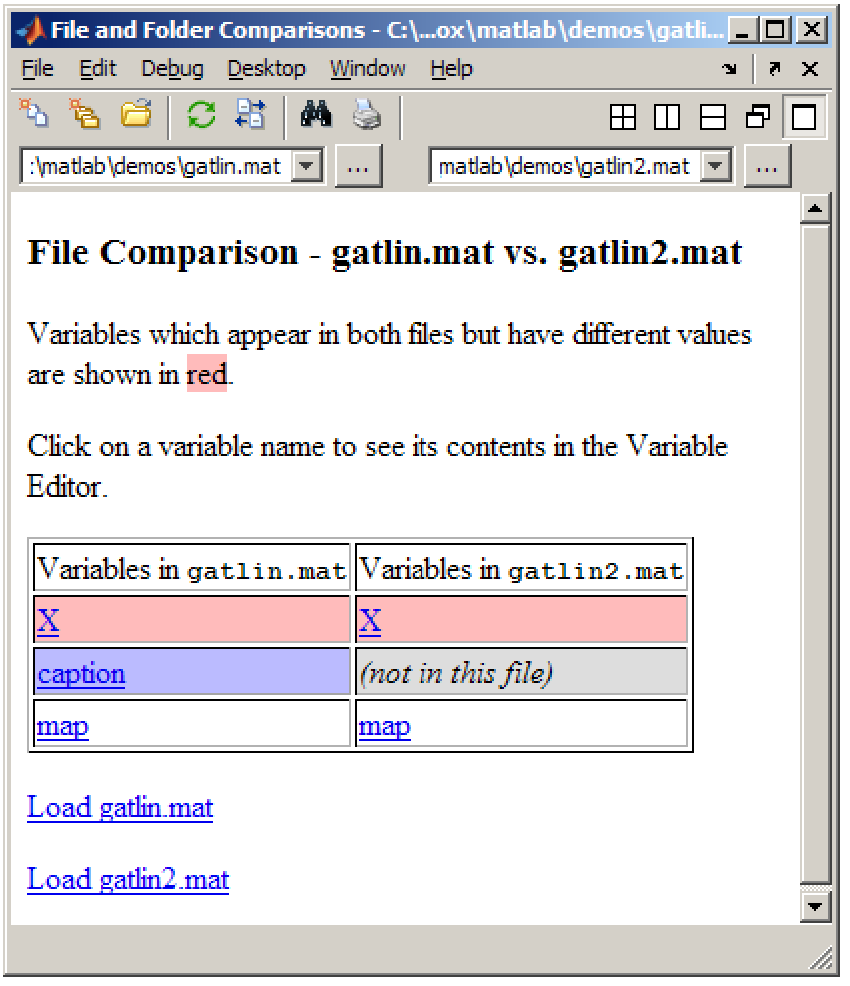
```
visdiff(fullfile(matlabroot,'help','techdoc','matlab_env',...
'examples','lengthofline.m'), fullfile(matlabroot,'help',...
'techdoc','matlab_env','examples','lengthofline2.m'), 50)
```



Compare Two MAT-Files Off the MATLAB Path

Compare two binary files that are on the MATLAB path.

```
visdiff(fullfile(matlabroot,'toolbox','matlab','demos','gatlin.mat'), ...
fullfile(matlabroot,'toolbox','matlab','demos','gatlin2.mat'))
```



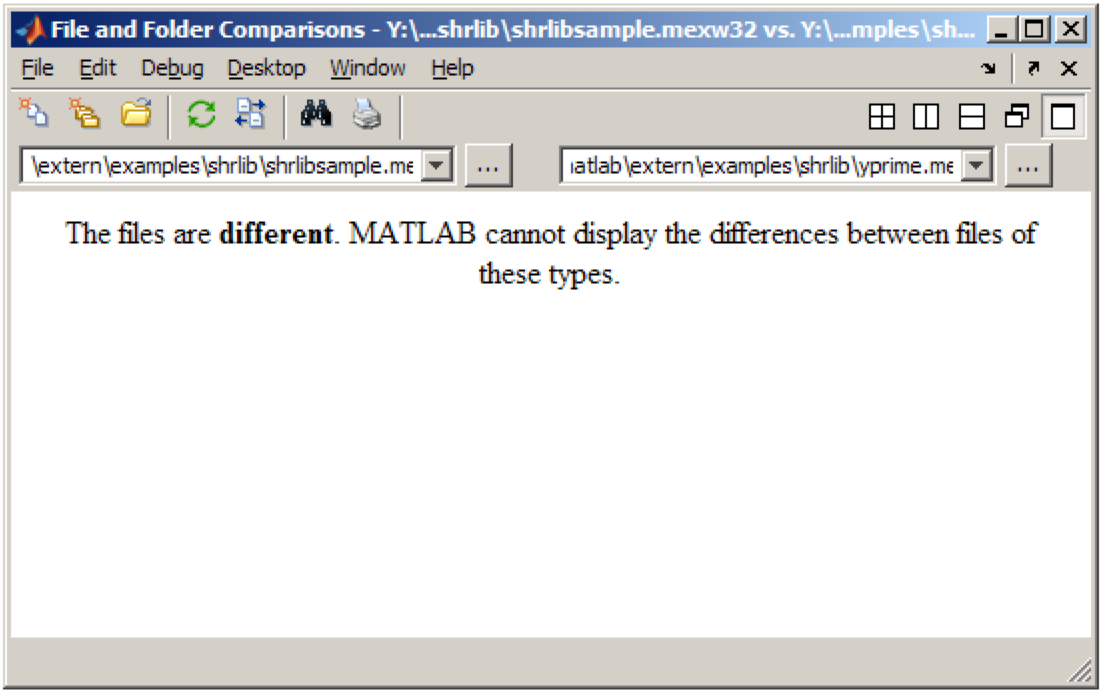
Compare Two Binary Files on the MATLAB Path

Add the folder containing two MEX-files to the MATLAB path, and then compare the files, by issuing these commands:

```
addpath([matlabroot '\extern\examples\shrlib'])
```

```
visdiff('shrlibsample.mexw32', 'yprime.mexw32')
```

The File and Folder Comparisons tool opens and indicates that the files are different, but does not provide details about the differences.



See Also

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

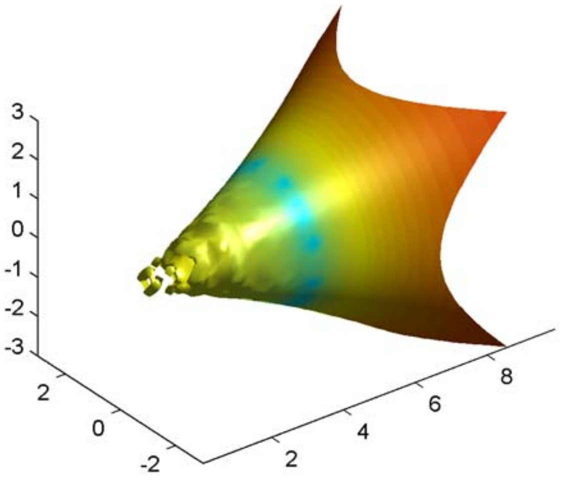
```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

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 1-106 for related functions

voronoi

Purpose

Voronoi diagram

Syntax

```
voronoi(x,y)
voronoi(x,y,TRI)
voronoi(dt)
voronoi(AX,...)
voronoi(...,'LineSpec')
h = voronoi(...)
[vx,vy] = voronoi(...)
```

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(dt)` uses the Delaunay triangulation `dt` instead of computing it.

`voronoi(AX,...)` plots into `AX` instead of `gca`.

`voronoi(...,'LineSpec')` plots the diagram with color and line style specified.

`h = voronoi(...)` returns, in `h`, handles to the line objects created.

`[vx,vy] = voronoi(...)` returns the finite vertices of the Voronoi edges in `vx` and `vy` so that `plot(vx,vy,'-',x,y,'.')` creates the Voronoi diagram. 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(:)])
```

`voronoi(X,Y,options)` specifies a cell array of strings that were previously used by Qhull. Qhull-specific `options` are no longer required and are currently ignored. Support for these options will be removed in a future release.

`convhull` uses CGAL, see http://www.cgal.org.

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*.

Visualization

Use one of these methods to plot a Voronoi diagram:

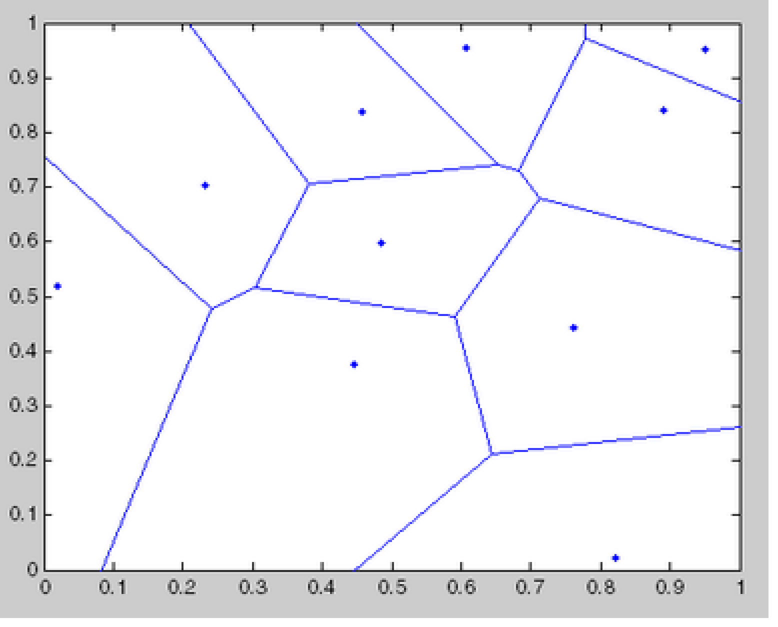
- If you provide no output argument, `voronoi` plots the diagram. See Example 1.
- To gain more control over color, line style, and other figure properties, use the syntax `[vx,vy] = voronoi(...)`. This syntax returns the vertices of the finite Voronoi edges, which you can then plot with the `plot` function. 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.

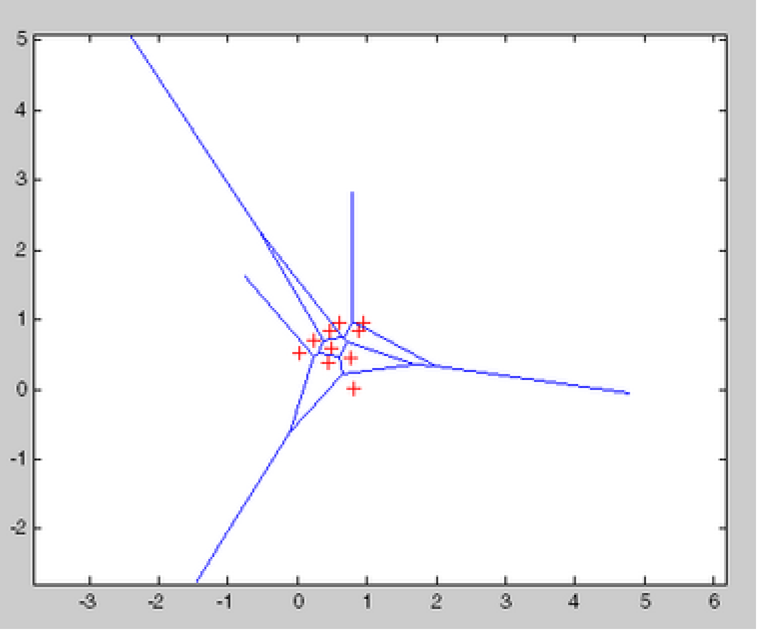
```
x = gallery('uniformdata',[1 10],0);
y = gallery('uniformdata',[1 10],1);
voronoi(x,y)
```



Example 2

This code uses the vertices of the finite Voronoi edges to plot the Voronoi diagram for the same 10 points.

```
x = gallery('uniformdata',[1 10],0);
y = gallery('uniformdata',[1 10],1);
[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.

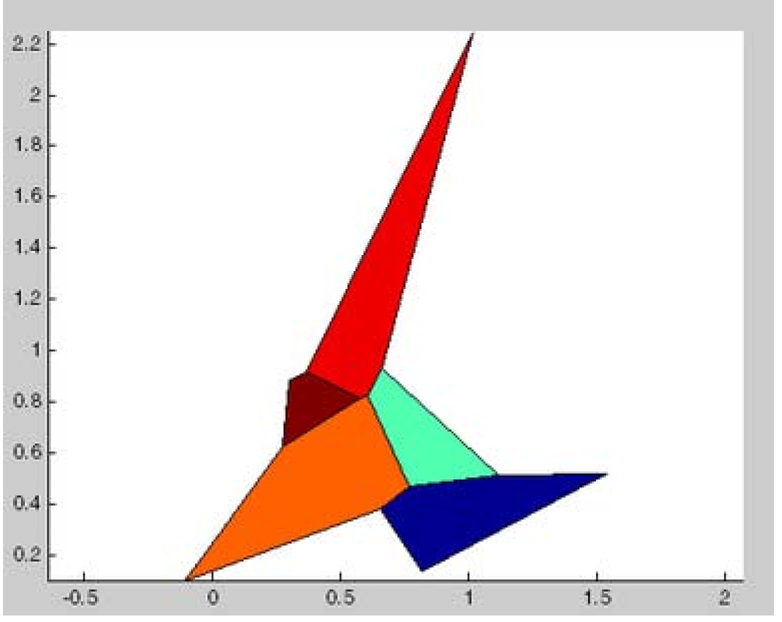
```
xlim([min(x) max(x)])
ylim([min(y) max(y)])
```

Example 3

This code uses `voronoin` and `patch` to fill the bounded cells of the same Voronoi diagram with color.

```
x = gallery('uniformdata',[10 2],5);
[v,c]=voronoin(x);
for i = 1:length(c)
if all(c{i}~=1)   % If at least one of the indices is 1,
                  % then it is an open region and we can't
                  % patch that.
patch(v(c{i},1),v(c{i},2),i); % use color i.
end
```

```
end
```



See Also

`DelaunayTri`, `convhull`, `delaunay`, `LineSpec`, `plot`, `voronoin`

Purpose

Voronoi diagram

Syntax

`[V, R] = voronoiDiagram(DT)`

Description

`[V, R] = voronoiDiagram(DT)` returns the vertices `V` and regions `R` of the Voronoi diagram of the points `DT.X`. The region `R{i}` is a cell array of indices into `V` that represents the Voronoi vertices bounding the region. The Voronoi region associated with the `i`'th point, `DT.X(i)` is `R{i}`. For 2-D, vertices in `R{i}` are listed in adjacent order, i.e. connecting them will generate a closed polygon (Voronoi diagram). For 3-D the vertices in `R{i}` are listed in ascending order.

The Voronoi regions associated with points that lie on the convex hull of `DT.X` are unbounded. Bounding edges of these regions radiate to infinity. The vertex at infinity is represented by the first vertex in `V`.

Inputs

`DT`	Delaunay triangulation.

Outputs

`V`	`numv`-by-`ndim` matrix representing the coordinates of the Voronoi vertices, where `numv` is the number of vertices and `ndim` is the dimension of the space where the points reside.
`R`	Vector cell array of `length(DR.X)`, representing the Voronoi cell associated with each point.

Definitions

The *Voronoi diagram* of a discrete set of points `X` decomposes the space around each point `X(i)` into a region of influence `R{i}`. Locations within the region are closer to point `i` than any other point. The region of influence is called the Voronoi region. The collection of all the Voronoi regions is the Voronoi diagram.

The *convex hull* of a set of points X is the smallest convex polygon (or polyhedron in higher dimensions) containing all of the points of X.

Examples

Compute the Voronoi Diagram of a set of points:

```
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 ]
dt = DelaunayTri(X)
[V,R] = voronoiDiagram(dt)
```

See Also

voronoi
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.

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
    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

`DelaunayTri`, `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.

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 or update a wait bar dialog box

Syntax

```
h = waitbar(x,'message')
waitbar(x,'message','CreateCancelBtn','button_callback')
waitbar(x,'message',property_name,property_value,...)
waitbar(x)
waitbar(x,h)
waitbar(x,h,'updated message')
```

Description

A wait bar is a figure that displays what percentage of a calculation is complete as the calculation proceeds by progressively filling a bar with red from left to right.

`h = waitbar(x,'message')` displays a wait bar of fractional length `x`. The wait bar figure displays until the code that controls it closes it or the use clicks its Close Window button. Its (figure) handle is returned in `h`. The argument `x` must be between `0` and `1`.

Note Wait bars are not modal figures (their `WindowStyle` is `'normal'`). They often appear to be modal because the computational loops within which they are called prevent interaction with the Command Window until they terminate. For more information, see `WindowStyle` in the MATLAB Figure Properties documentation.

`waitbar(x,'message','CreateCancelBtn','button_callback')` specifying `CreateCancelBtn` adds a **Cancel** button to the figure that executes the MATLAB commands specified in `button_callback` when the user clicks the **Cancel** button or the **Close Figure** button. `waitbar` sets both the **Cancel** button callback and the figure `CloseRequestFcn` to the string specified in `button_callback`.

`waitbar(x,'message',property_name,property_value,...)` optional arguments `property_name` and `property_value` enable you to set figure properties for the `waitbar`.

`waitbar(x)` subsequent calls to `waitbar(x)` extend the length of the bar to the new position `x`. Successive values of `x` normally increase. If they decrease, the wait bar runs in reverse.

`waitbar(x,h)` extends the length of the bar in the wait bar `h` to the new position `x`.

`waitbar(x,h,'updated message')` updates the message text in the `waitbar` figure, in addition to setting the fractional length to `x`.

Examples

Example 1 — Basic Wait Bar

Typically, you call `waitbar` repeatedly inside a `for` loop that performs a lengthy computation. For example:

```
h = waitbar(0,'Please wait...');
steps = 1000;
for step = 1:steps
    % computations take place here
    waitbar(step / steps)
end
close(h)
```

Example 2 — Wait Bar with Dynamic Text and Cancel Button

Adding a **Cancel** button allows user to abort the computation. Clicking it sets a logical flag in the figure's application data (appdata). The function tests for that value within the main loop and exits the loop as soon as the flag has been set. The example iteratively approximates the value of π. At each step, the current value is encoded as a string and displayed in the wait bar's message field. When the function finishes, it destroys the wait bar and returns the current estimate of π and the number of steps it ran.

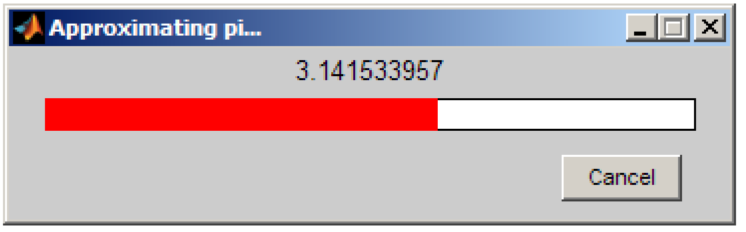
Copy the following function to an M-file and save it as `approxpi.m`. Execute it as follows, allowing it to run for 10,000 iterations.

```
[estimated_pi steps] = approxpi(10000)
```

You can click **Cancel** or close the window to abort the computation and return the current estimate of π.

```
function [valueofpi step] = approxpi(steps)
% Converge on pi in steps iterations, displaying waitbar.
% User can click Cancel or close button to exit the loop.
% Ten thousand steps yields error of about 0.001 percent.

h = waitbar(0,'1','Name','Approximating pi...',...
            'CreateCancelBtn',...
            'setappdata(gcbf,''canceling'',1)');
setappdata(h,'canceling',0)
% Approximate as pi^2/8 = 1 + 1/9 + 1/25 + 1/49 + ...
pisqover8 = 1;
denom = 3;
valueofpi = sqrt(8 * pisqover8);
for step = 1:steps
    % Check for Cancel button press
    if getappdata(h,'canceling')
        break
    end
    % Report current estimate in the waitbar's message field
    waitbar(step/steps,h,sprintf('%12.9f',valueofpi))
    % Update the estimate
    pisqover8 = pisqover8 + 1 / (denom * denom);
    denom = denom + 2;
    valueofpi = sqrt(8 * pisqover8);
end
delete(h)       % DELETE the waitbar; don't try to CLOSE it.
```

The function sets the figure `Name` property to describe what is being computed. In the `for` loop, calling `waitbar` sets the fractional progress indicator and displays intermediate results. the code `waitbar(i/steps,h,sprintf('%12.9f',valueofpi))` sets the wait bar's `message` variable to a string representation of the current estimate of *pi*. Naturally, the extra computation involved makes iterations last longer than they need to, but such feedback can be helpful to users.



Note You should call `delete` to remove a wait bar when you give it a `CloseRequestFcn`, as in the preceding code; calling `close` does not close it, and makes its Cancel and Close Window buttons unresponsive. This happens because the figure's `CloseRequestFcn` recursively calls itself. In such a situation you must forcibly remove the wait bar, for example like this:

```
set(0,'ShowHiddenHandles','on')
delete(get(0,'Children'))
```

However, as issuing these commands will delete all open figures—not just the wait bar—it is best never to use `close` in a `CloseRequestFcn` to close a window.

See Also

for related functions

`close`, `delete`, `dialog`, `msgbox`, `getappdata`, `setappdata`

Purpose

Wait for condition before resuming execution

Syntax

```
waitfor(h)
waitfor(h,'PropertyName')
waitfor(h,'PropertyName',PropertyValue)
```

Description

`waitfor(h)` blocks the caller execution stream until the graphics object identified by handle `h` is deleted or you type **Ctrl+C** in the Command Window. `h` must be scalar. When either of those events occur, `waitfor` stops blocking execution and returns. If `h` does not exist, `waitfor` returns immediately without processing any events.

`waitfor(h,'PropertyName')`, in addition to the conditions in the previous syntax, stops blocking and returns when the value of `'PropertyName'` (any property of the graphics object `h`) changes. If `'PropertyName'` is not a valid property for the object, `waitfor` returns immediately without processing any events.

`waitfor(h,'PropertyName',PropertyValue)` stops blocking and returns when the value of `'PropertyName'` for the graphics object `h` changes to `PropertyValue`. If you previously set `'PropertyName'` to `PropertyValue`, `waitfor` returns immediately without processing any events.

Definitions

`waitfor` blocks the caller execution stream so that command-line expressions and statements in the blocked file do not execute until a specified condition occurs. While `waitfor` blocks an execution stream, other execution streams generated by callbacks that respond to various events (for example, pressing a mouse button) can run, unaffected by `waitfor`. It also blocks Simulink models from executing. However, callbacks do execute during the blocking of the execution stream. `waitfor` can block nested execution streams. For example, a callback invoked during a `waitfor` statement can invoke `waitfor`.

Examples

Create a plot and pause execution of the rest of the statements until you delete the figure window:

```
h = figure;
plot(rand(10,1));
disp('Waiting for you to delete the figure...')
drawnow     % Necessary to plot and put message on the screen
waitfor(h)
% The next line only executes when the figure is deleted
disp('Thank you.')
```

Display the current date and time only while a button is depressed

```
figure('Position',[560 526 420 315]);
hb = uicontrol('Style','togglebutton','Value',0,...
               'Units','normalized',...
               'Position',[.4 .6 .2 .05],...
               'String','Start/Stop');
ht = uicontrol('Style','text','Units','normalized',...
               'Position',[.275 .5 .425 .04],...
               'FontSize',10,...
               'String',datestr(now));
% Iterate 100,000 times then quit
% Typing Ctrl+C in Command Window will also stop the count
count = 0;
while count < 100000        % Exit condition
    waitfor(hb,'Value',1)   %Until togglebutton is down
    % Text only updates while Start/Stop button is down
    set(ht,'String',datestr(now))  % Update date and time
    drawnow                 % Update text field
    count = count+1;
end
```

If you close the figure while the code is executing, an error occurs because the code attempts to access handles of objects that no longer exist. You can handle the error by enclosing code in the loop in a try/catch block, as follows:

```
...
while count < 100000        % Exit condition
     try  % An error occurs if you delete the figure here
        waitfor(hb,'Value',1)   %Until togglebutton is down
        % Text only updates while Start/Stop button is down
        set(ht,'String',datestr(now))  % Update date and time
        drawnow                 % Update text field
    catch ME  % Catch the error and exit gracefully
     % You can place more code to respond to the error here
        return
    end
end
```

The ME variable is a MATLAB Exception object that you can use to determine the type of error that occurred. For more information, see .

See Also

`drawnow` | `keyboard` | `pause` | `uiresume` | `uiwait` | `waitforbuttonpress`

How To

-
- Developing User Interfaces

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.

You can interrupt `waitforbuttonpress` by typing **Ctrl**+C, but an error results unless the function is called from within a `try-catch` block. You also receive an error from `waitforbuttonpress` if you close the figure by clicking the **X** close box unless you call `waitforbuttonpress` within a `try-catch` block.

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`

for related functions

Purpose

Open warning dialog box

Syntax

```
h = warndlg
h = warndlg(warningstring)
h = warndlg(warningstring,dlgname)
h = warndlg(warningstring,dlgname,createmode)
```

Description

`h = warndlg` displays a dialog box named `Warning Dialog` containing the string `This is the default warning string.` The `warndlg` function returns the handle of the dialog box in `h`. The warning dialog box disappears after the user clicks **OK**.

`h = warndlg(warningstring)` displays a dialog box with the title `Warning Dialog` containing the string specified by `warningstring`. The `warningstring` argument can be any valid string format – cell arrays are preferred.

To use multiple lines in your warning, define `warningstring` using either of the following:

- `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'})
  ```

`h = warndlg(warningstring,dlgname)` displays a dialog box with title `dlgname`.

`h = warndlg(warningstring,dlgname,`*`createmode`*`)` specifies whether the warning dialog box is modal or nonmodal. Optionally, it can also specify an interpreter for `warningstring` and `dlgname`. The *`createmode`* argument can be a string or a structure.

If *`createmode`* is a string, it must be one of the values shown in the following table.

createmode Value	Description
`modal`	Replaces the warning dialog box having the specified `Title`, that was last created or clicked on, with a modal warning dialog box as specified. All other warning dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.
`non-modal` (default)	Creates a new nonmodal warning dialog box with the specified parameters. Existing warning dialog boxes with the same title are not deleted.
`replace`	Replaces the warning dialog box having the specified `Title`, that was last created or clicked on, with a nonmodal warning dialog box as specified. All other warning dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function.

If you open a dialog with `errordlg`, `msgbox`, or `warndlg` using `'CreateMode','modal'` and a non-modal dialog created with any of these functions is already present and *has the same name as the modal dialog*, the non-modal dialog closes when the modal one opens.

For more information about modal dialog boxes, see `WindowStyle` in the Figure Properties.

If `CreateMode` is a structure, it can have fields `WindowStyle` and `Interpreter`. `WindowStyle` must be one of the options shown in the table above. `Interpreter` is one of the strings `'tex'` or `'none'`. The default value for `Interpreter` is `'none'`.

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`

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 descriptive text `message` and sets the warning state that `lastwarn` returns. If `message` is an empty string (`''`), `warning` resets the warning state but does not display any text.

`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 and in the MATLAB Programming Fundamentals documentation for more information on the `message_id` argument and how to use it.

`warning('message_id', 'message', a1, a2, ..., an)` includes formatting conversion characters in `message`, and the character translations in arguments `a1, a2, ..., an`.

`s = warning(state, 'message_id')` is a warning control statement that enables you to indicate how you want MATLAB to act on certain warnings. The `state` argument can be `'on'`, `'off'`, or `'query'`.

The message_id argument can be a message identifier string, 'all', or 'last'. See in the MATLAB Programming Fundamentals 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 in the MATLAB Programming Fundamentals 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 in the MATLAB Programming Fundamentals 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');
```

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

waterfall

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:size(Z,1)` and `y = 1:size(Z,1)`. `Z` determines the color, so color is proportional to surface height.

`waterfall(X,Y,Z)` creates a waterfall plot using the values specified in `X`, `Y`, and `Z`. `Z` also determines the color, so color is proportional to the surface height. If `X` and `Y` are vectors, `X` corresponds to the columns of `Z`, and `Y` corresponds to the rows, where `length(x) = n`, `length(y) = m`, and `[m,n] = size(Z)`. `X` and `Y` are vectors or matrices that define the *x*- and *y*-coordinates of the plot. `Z` is a matrix that defines the *z*-coordinates of the plot (i.e., height above a plane). If `C` is omitted, color is proportional to `Z`.

`waterfall(...,C)` uses scaled color values to obtain colors from the current colormap. Color scaling is determined by the range of `C`, which must be the same size as `Z`. MATLAB performs a linear transformation on `C` to obtain colors from the current colormap.

`waterfall(axes_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.

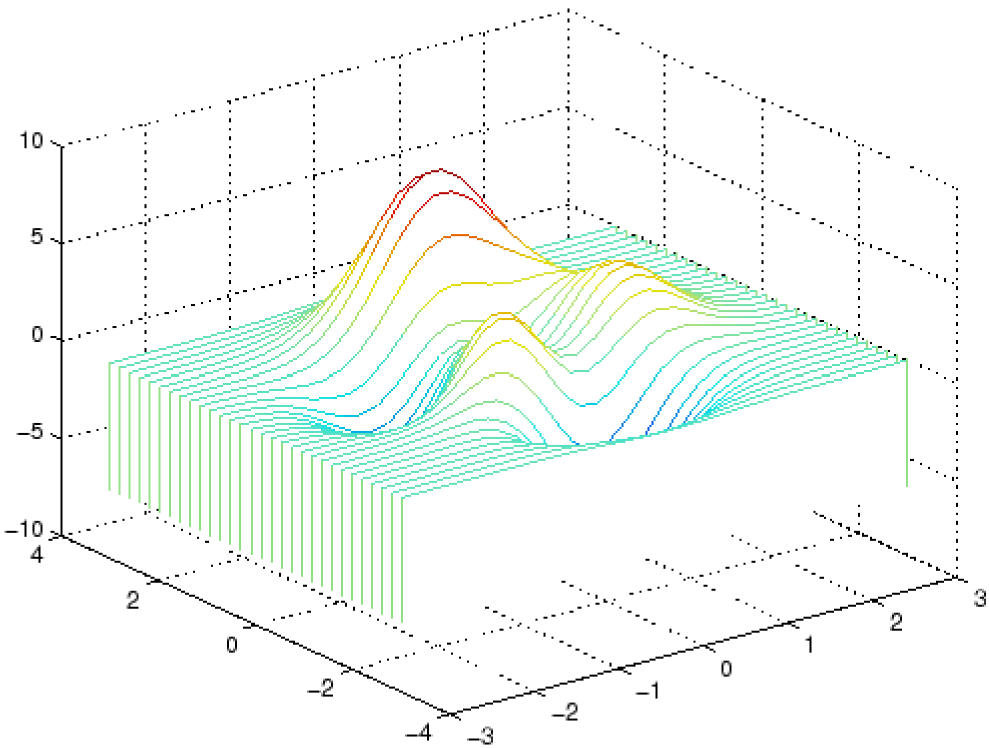
Remarks

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'. 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 WAVE (`.wav`) sound file

Syntax

```
[m d] = wavfinfo(filename)
```

Description `[m d] = wavfinfo(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 `wavplay`, `wavread`, `wavrecord`, `wavwrite`

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

`wavfinfo`, `wavread`, `wavrecord`, `wavwrite`

wavread

Purpose

Read WAVE (.wav) sound file

Graphical Interface

As an alternative to `wavread`, use the Import Wizard. To activate the Import Wizard, select **File > Import Data**.

Syntax

```
y = wavread(filename)
[y, Fs] = wavread(filename)
[y, Fs, nbits] = wavread(filename)
[y, Fs, nbits, opts] = wavread(filename)
[...] = wavread(filename, N)
[...] = wavread(filename, [N1 N2])
[...] = wavread(..., fmt)
siz = wavread(filename,'size')
```

Description

`y = wavread(filename)` loads a WAVE file specified by the string *filename*, returning the sampled data in *y*. If *filename* does not include an extension, `wavread` appends `.wav`.

`[y, Fs] = wavread(filename)` returns the sample rate (*Fs*) in Hertz used to encode the data in the file.

`[y, Fs, nbits] = wavread(filename)` returns the number of bits per sample (*nbits*).

`[y, Fs, nbits, opts] = wavread(filename)` returns a structure *opts* of additional information contained in the WAV file. The content of this structure differs from file to file. Typical structure fields include `opts.fmt` (audio format information) and `opts.info` (text that describes the title, author, etc.).

`[...] = 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.

`[...] = wavread(..., fmt)` specifies the data format of *y* used to represent samples read from the file. *fmt* can be either of the following values, or a partial match (case-insensitive):

'double'	Double-precision normalized samples (default).
'native'	Samples in the native data type found in the file.

siz = wavread(*filename*,'size') returns the size of the audio data contained in *filename* instead of the actual audio data, returning the vector *siz* = [*samples channels*].

Output Scaling

The range of values in *y* depends on the data format *fmt* specified. Some examples of output scaling based on typical bit-widths found in a WAV file are given below for both 'double' and 'native' formats.

Native Formats

Number of Bits	MATLAB Data Type	Data Range
8	uint8 (unsigned integer)	0 <= *y* <= 255
16	int16 (signed integer)	-32768 <= *y* <= +32767
24	int32 (signed integer)	-2^23 <= *y* <= 2^23-1
32	single (floating point)	-1.0 <= *y* < +1.0

Double Formats

Number of Bits	MATLAB Data Type	Data Range
N<32	double	-1.0 <= *y* < +1.0
N=32	double	-1.0 <= *y* <= +1.0 Note: Values in *y* might exceed -1.0 or +1.0 for the case of N=32 bit data samples stored in the WAV file.

wavread supports multi-channel data, with up to 32 bits per sample.

wavread supports Pulse-code Modulation (PCM) data format only.

Examples

Create a WAV file from the demo file handel.mat, and read portions of the file back into MATLAB.

```
% Create WAV file in current folder.
load handel.mat

hfile = 'handel.wav';
wavwrite(y, Fs, hfile)
clear y Fs

% Read the data back into MATLAB, and listen to audio.
[y, Fs, nbits, readinfo] = wavread(hfile);
sound(y, Fs);

% Pause before next read and playback operation.
duration = numel(y) / Fs;
pause(duration + 2)

% Read and play only the first 2 seconds.
nsamples = 2 * Fs;
[y2, Fs] = wavread(hfile, nsamples);
sound(y2, Fs);
pause(4)

% Read and play the middle third of the file.
sizeinfo = wavread(hfile, 'size');

tot_samples = sizeinfo(1);
startpos = tot_samples / 3;
endpos = 2 * startpos;

[y3, Fs] = wavread(hfile, [startpos endpos]);
sound(y3, Fs);
```

See Also

`audioplayer`, `audiorecorder`, `mmfileinfo`, `sound`, `wavwrite`

wavrecord

Purpose

Record sound using PC-based audio input device

Syntax

```
y = wavrecord(n,Fs)
y = wavrecord(...,ch)
y = wavrecord(...,'dtype')
```

Description

`y = wavrecord(n,Fs)` records `n` samples of an audio signal, sampled at a rate of `Fs` Hz (samples per second). The default value for `Fs` is 11025 Hz.

`y = wavrecord(...,ch)` uses `ch` number of input channels from the audio device. `ch` can be either 1 or 2, for mono or stereo, respectively. The default value for `ch` is 1.

`y = wavrecord(...,'dtype')` uses the data type specified by the string `'dtype'` to record the sound. The following table lists the string values for `'dtype'` along with the corresponding bits per sample and acceptable data range for `y`.

dtype	Bits/sample	y Data Range
'double'	16	–1.0 <= y < +1.0
'single'	16	–1.0 <= y < +1.0
'int16'	16	–32768 <= y <= +32767
'uint8'	8	0 <= y <= 255

Remarks

Standard sampling rates for PC-based audio hardware are 8000, 11025, 22050, 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.

The `wavrecord` function is for use only with 32-bit Microsoft Windows operating systems. To record audio data from audio input devices on other platforms, use `audiorecorder`.

Examples

Record 5 seconds of 16-bit audio sampled at 11025 Hz. Play back the recorded sound using `wavplay`. Speak into your audio device (or produce your audio signal) while the `wavrecord` command runs.

```
Fs = 11025;
y = wavrecord(5*Fs,Fs,'int16');
wavplay(y,Fs);
```

See Also

`audiorecorder`, `wavfinfo`, `wavplay`, `wavread`, `wavwrite`

wavwrite

Purpose

Write WAVE (`.wav`) sound file

Syntax

```
wavwrite(y,filename)
wavwrite(y,Fs,filename)
wavwrite(y,Fs,N,filename)
```

Description

`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.

`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.

`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.

Input Data Ranges

The range of values in `y` depends on the number of bits specified by `N` and the data type of `y`. The following tables list the valid input ranges based on the value of `N` and the data type of `y`.

If `y` contains integer data:

N Bits	y Data Type	y Data Range	Output Format
8	`uint8`	0 <= *y* <= 255	`uint8`
16	`int16`	–32768 <= *y* <= +32767	`int16`
24	`int32`	–2^23 <= *y* <= 2^23 – 1	`int32`

If y contains floating-point data:

N Bits	y Data Type	y Data Range	Output Format
8	single or double	–1.0 <= *y* < +1.0	uint8
16	single or double	–1.0 <= *y* < +1.0	int16
24	single or double	–1.0 <= *y* < +1.0	int32
32	single or double	–1.0 <= *y* <= +1.0	single

For floating point data where *N* < 32, amplitude values are clipped to the range –1.0 <= *y* < +1.0.

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, wavfinfo, wavplay, wavread, wavrecord

web

Purpose

Open Web site or file in Web 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.

`web url` displays the page specified by `url` in the MATLAB Web browser. If any MATLAB Web browsers are already open, it displays the page in the browser that was used last. Files up to 1.5 MB in size display in the MATLAB Web browser, while larger files instead display in the system Web browser. The `web` function accepts a valid URL such as a web site address, a full path to a file, or a relative path to a file (using `url` within the current folder if it exists there). If `url` is located in the folder returned when you run `docroot` (an unsupported utility function), the page displays in the MATLAB Help browser instead of the MATLAB Web browser.

`web url -new` displays the page specified by `url` in a new MATLAB Web browser.

`web url -notoolbar` displays the page specified by `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 was used last, regardless of its toolbar status.

`web url -noaddressbox` displays the page specified by `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 was used last, regardless of its address field status.

`web url` **`-helpbrowser`** displays the page specified by `url` in the MATLAB Help browser.

`web url -`**`browser`** displays `url` in a system Web browser window. `url` can be in any form that the browser supports. On Microsoft Windows and Apple Macintosh platforms, the system Web browser is determined by the operating system. On UNIX[21] platforms, the default system Web browser for MATLAB is Mozilla Firefox®. To specify a different browser, use MATLAB Web preferences.

`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 `stat`	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 Sun Microsystems Java class, `h1`, for the last active browser. You can use `close(h1)` to clear the displayed page from the browser. The browser, `h1`, could have been opened when the `web` function was previously executed, or when a tool ran the `web` function. For example, clicking a link to an external site from the Help browser runs `web` to open the Web site in a system browser, and `h1` would be the handle for that browser instance.

`[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`.

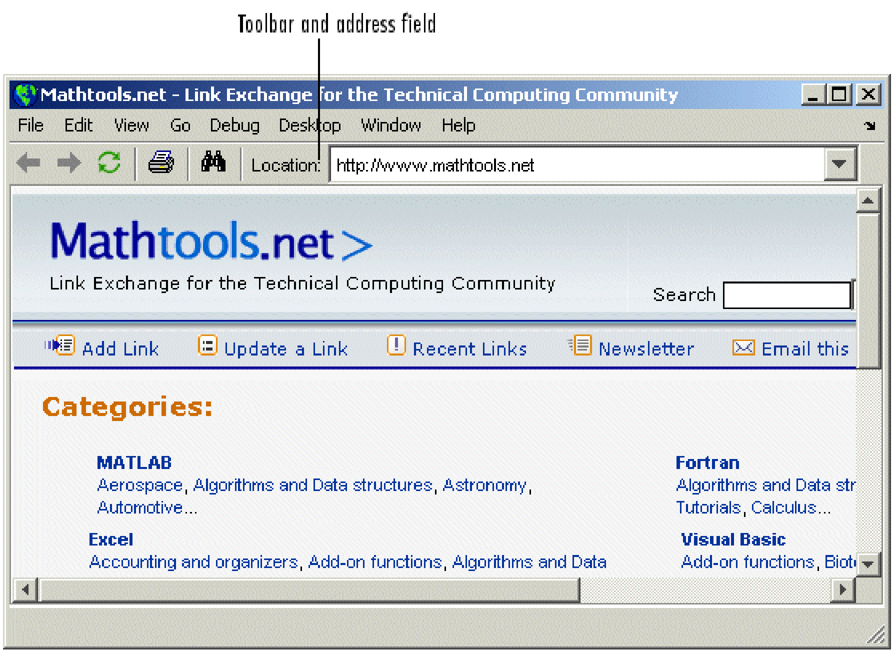
Examples

Display the Mathtools Web site:

```
web http://www.mathtools.net
```

MATLAB displays:

21. UNIX is a registered trademark of The Open Group in the United States and other countries.



`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 mydir/myfile.html` opens `myfile.html` in the MATLAB Web browser, where `mydir` is in the current folder.

`web(['file:///' which('foo.html')])` opens `foo.html` if the file is in a folder on the search path or in the current folder for MATLAB.

`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 the system browser's default e-mail application to send a message to `email_address`.

`web http://www.mathtools.net -browser` opens the system Web browser at `mathtools.net`.

`[stat,h1]=web('http://www.mathworks.com');` opens `mathworks.com` in a MATLAB Web browser. Then, `close(h1)` clears the displayed URL, `mathworks.com`, from the browser window.

See Also

`doc`, `helpbrowser`, `matlabcolon`, `urlread`, `urlwrite`

Related topics in the User Guide:

- in the MATLAB Desktop Tools and Development Environment documentation
-
-

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`

Purpose

List MATLAB files in folder

Graphical Interface

As an alternative to the `what` function, use the Current Folder browser.

Syntax

```
what
what folderName
what className
what packageName
s = what('folderName')
```

Description

`what` lists the path for the current folder, and lists all files and folders relevant to MATLAB found in the current folder. Files listed are M, MAT, MEX, MDL, and P-files. Folders listed are all class and package folders.

`what` *`folderName`* lists path, file, and folder information for *`folderName`*. Use an absolute or partial path for *`folderName`*.

`what` *`className`* lists path, file, and folder information for method folder `@`*`className`*. For example, `what cfit` lists the MATLAB files and folders in `toolbox/curvefit/curvefit/@cfit`.

`what` *`packageName`* lists path, file, and folder information for package folder `+`*`packageName`*. For example, `what commsrc` lists the MATLAB files and folders in `toolbox/comm/comm/+commsrc`.

`s = what('`*`folderName`*`')` returns the results in a structure array with the fields shown in the following table.

Field	Description
`path`	Path to folder
`m`	Cell array of M-file names
`mat`	Cell array of MAT-file names
`mex`	Cell array of MEX-file names

Field	Description
mdl	Cell array of MDL-file names
p	Cell array of P-file names
classes	Cell array of class folders
packages	Cell array of package folders

Examples

List Files and Folders Relevant to MATLAB

List the MATLAB files and folders in `toolbox/matlab/audiovideo`:

```
what audiovideo

M-files in directory matlabroot\toolbox\matlab\audiovideo

Contents            avifinfo            sound
audiodevinfo        aviinfo             soundsc
audioplayerreg      aviread             wavfinfo
audiorecorderreg    lin2mu              wavplay
audiouniquename     mmcompinfo          wavread
aufinfo             mmfileinfo          wavrecord
auread              movie2avi           wavwrite
auwrite             mu2lin
avgate              prefspanel

MAT-files in directory matlabroot\toolbox\matlab\audiovideo

chirp         handel        splat
gong          laughter      train

MEX-files in directory matlabroot\toolbox\matlab\audiovideo

winaudioplayer
winaudiorecorder

Classes in directory matlabroot\toolbox\matlab\audiovideo
```

```
audioplayer    avifile
audiorecorder  mmreader
```

Return Names to a Structure

Obtain a structure array containing the file and folder names in `toolbox/matlab/general` that are relevant to MATLAB:

```
s = what('general')
s =
       path: 'matlabroot:\toolbox\matlab\general'
          m: {89x1 cell}
        mat: {0x1 cell}
        mex: {2x1 cell}
        mdl: {0x1 cell}
          p: {'callgraphviz.p'}
    classes: {'char'}
   packages: {0x1 cell}
```

List M-Files in a Package

Find the supporting M-files for one of the packages in the Communications Toolbox product:

```
p1 = what('comm');
p1.packages
ans =
    'commdevice'
    'crc'
    'commsrc'

p2 = what('commsrc');
p2.m
ans =
    'abstractJitter.m'
    'abstractPulse.m'
```

```
'combinedjitter.m'
'diracjitter.m'
'periodicjitter.m'
'randomjitter.m'
```

See Also

`dir`, `exist`, `lookfor`, `ls`, `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`

Purpose

Locate functions and files

Graphical Interface

As an alternative to the `which` function, you can use the to find files. You can find functions using the Function Browser in the Command Window or Editor.

Syntax

```
which fun
which classname/fun
which private/fun
which classname/private/fun
which fun1 in fun2
which fun(a,b,c,...)
which file.ext
which fun -all
s = which('fun',...)
```

Description

`which fun` displays the full pathname for the argument `fun`. If `fun` is a

- MATLAB function or Simulink model in an M, P, or MDL file on the MATLAB path, then `which` displays the full pathname for the corresponding file
- Workspace variable, then `which` displays a message identifying `fun` as a variable
- Method in a loaded Java class, then `which` displays the package, class, and method name for that method

If `fun` is an overloaded function or method, then `which fun` returns only the pathname of the first function or method found.

`which classname/fun` displays the full pathname for the M-file defining the `fun` method in MATLAB class, `classname`. For example, `which serial/fopen` displays the path for `fopen.m` in the MATLAB class directory, `@serial`.

`which private/fun` limits the search to private functions. For example, `which private/orthog` displays the path for `orthog.m` in the `/private` subdirectory of `toolbox/matlab/elmat`.

`which classname/private/fun` limits the search to private methods defined by the MATLAB class, `classname`. For example, `which dfilt/private/todtf` displays the path for `todtf.m` in the `private` directory of the `dfilt` class.

`which fun1 in fun2` displays the pathname to function `fun1` in the context of the M-file `fun2`. You can use this form to determine whether a subfunction 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. To display the path for a file that has no file extension, type "`which file.`" (the period following the filename is required). Use `exist` to check for the existence of files anywhere else.

`which fun -all` d isplays the paths to all items on the MATLAB path with the name `fun`. You may use the **`-all`** qualifier with any of the above formats of the `which` function.

`s = which('fun',...)` returns the results of `which` in the string `s`. For workspace variables, `s` is the string '`variable`'. You may specify an output variable in any of the above formats of the `which` function.

If **`-all`** is used with this form, the output `s` is always a cell array of strings, even if only one string is returned.

Examples

The statement below indicates that `pinv` is in the `matfun` directory of MATLAB.

```
which pinv
matlabroot\toolbox\matlab\matfun\pinv.m
```

To find the `fopen` function used on MATLAB `serial` class objects

```
which serial/fopen
matlabroot\toolbox\matlab\iofun\@serial\fopen.m  % serial method
```

To find the `setMonth` method used on objects of the Java `Date` class, the class must first be loaded into MATLAB. The class is loaded when you create an instance of the class:

```
myDate = java.util.Date;
which setMonth
```

MATLAB displays:

```
setMonth is a Java method  % java.util.Date 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

```
while expression, statements, end
```

Description `while expression, statements, end` repeatedly executes one or more MATLAB *`statements`* in a loop, continuing until *`expression`* no longer holds true or until MATLAB encounters a `break`, or `return` instruction. thus forcing an immediately exit of the loop. If MATLAB encounters a `continue` statement in the loop code, it immediately exits the current pass at the location of the `continue` statement, skipping any remaining code in that pass, and begins another pass at the start of the loop *`statements`*.

`expression` is a MATLAB expression that evaluates to a result of logical 1 (`true`) or logical 0 (`false`). *`expression`* can be scalar or an array. It must contain all real elements, and the statement `all(A(:))` must be equal to logical 1 for the expression to be true.

`expression` usually consists of variables or smaller expressions joined by relational operators (e.g., `count < limit`) or logical functions (e.g., `isreal(A)`). Simple expressions can be combined by logical operators (`&&`, `||`, `~`) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to rules.

```
(count < limit) && ((height - offset) >= 0)
```

`statements` is one or more MATLAB statements to be executed only while the *`expression`* is `true` or nonzero.

The scope of a `while` statement is always terminated with a matching `end`.

See in the MATLAB Programming Fundamentals documentation for more information on controlling the flow of your program code.

Remarks

Nonscalar Expressions

If the evaluated `expression` yields a nonscalar value, then every element of this value must be `true` or nonzero for the entire expression to be considered `true`. For example, the statement `while (A < B)` is `true` only if each element of matrix `A` is less than its corresponding element in matrix `B`. See “Example 2 – Nonscalar Expression” on page 2-4227, 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 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.

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

```
whitebg
whitebg(fig)
whitebg(ColorSpec)
whitebg(fig, ColorSpec)
whitebg(fig, ColorSpec)
whitebg(fig)
```

Description

`whitebg` complements the colors in the current figure.

`whitebg(fig)` complements colors in all figures specified in the vector `fig`.

`whitebg(ColorSpec)` and `whitebg(fig, ColorSpec)` change the color of the axes, which are children of the figure, to the color specified by `ColorSpec`. Without a figure specification, `whitebg` or `whitebg(ColorSpec)` affects the current figure and the root's default properties so subsequent plots and new figures use the new colors.

`whitebg(fig, ColorSpec)` sets the default axes background color of the figures in the vector `fig` to the color specified by `ColorSpec`. Other axes properties and the figure background color can change as well so that graphs maintain adequate contrast. `ColorSpec` can be a 1-by-3 RGB color or a color string such as `'white'` or `'w'`.

`whitebg(fig)` 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 `whitebg` gets its name. Include the root window handle `(0)` in `fig` to affect the default properties for new windows or for `clf reset`.

Remarks

`whitebg` works best in cases where all the axes in the figure have the same background color.

`whitebg` changes the colors of the figure's children, with the exception of shaded surfaces. This ensures that all objects are visible against the new background color. `whitebg` sets the default properties on the root such that all subsequent figures use the new background color.

Examples

Set the background color to blue-gray.

```
whitebg([0 .5 .6])
```

Set the background color to blue.

```
whitebg('blue')
```

See Also

`ColorSpec`, `colordef`

The figure graphics object property `InvertHardCopy`

"Color Operations" on page 1-103 for related functions

Purpose

List variables in workspace

Graphical Interface

As an alternative to `whos`, use the Workspace browser. Or use the 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 applies 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, the MATLAB software 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 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.	whos('global')
'-file', filename	List variables in the specified MAT-file. Use the full path for filename.	whos('-file', 'mydata')
'-regexp', exprlist	List variables that match any of the regular expressions in exprlist.	whos('-regexp', '[AB].', '\w\d')

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: • `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

  javaChrArray        3x1               java.lang.String[][][]
  javaDblArray        4x1               java.lang.Double[][]
  javaIntArray       14x1               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
```

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`

in the Desktop Tools and Development Environment documentation

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 associated Microsoft Windows application. The application is associated with the extension in `fileName` in the Windows operating system. `filename` is a string enclosed in single quotes. `winopen` uses a Windows shell command, and performs the same action as double-clicking the file in the Windows Explorer program. Use an absolute or relative path for `fileName`.

Examples

Open the file `thesis.doc`, located in the current folder, in the Microsoft Word program:

```
winopen('thesis.doc')
```

Open `myresults.html` in the system Web browser:

```
winopen('D:/myfiles/myresults.html')
```

On Microsoft Windows platforms, open the current folder in the Windows Explorer tool:

```
winopen(cd)
```

See Also

`dos`, `open`, `web`

winqueryreg

Purpose Item from 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` of Microsoft Windows operating system registry to 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 the MATLAB software 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 Microsoft 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
SwapMouseButtons = 0
```

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 Lotus 1-2-3 WK1 spreadsheet file into matrix

Syntax

```
M = wk1read(filename)
M = wk1read(filename,r,c)
M = wk1read(filename,r,c,range)
```

Description

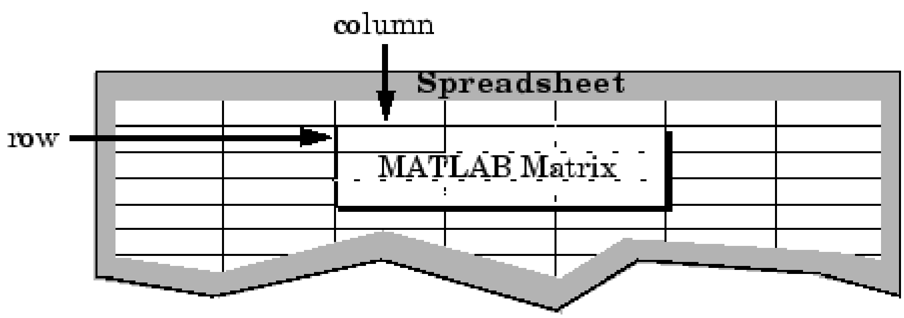
`M = wk1read(filename)` reads a Lotus1-2-3 WK1 spreadsheet file into the matrix `M`. The `filename` input is a string enclosed in single quotes.

`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 =
```

```
     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

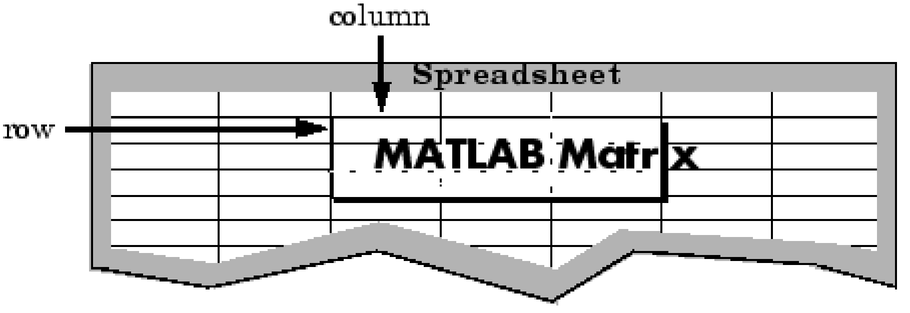
Syntax

```
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 =
```

```
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.

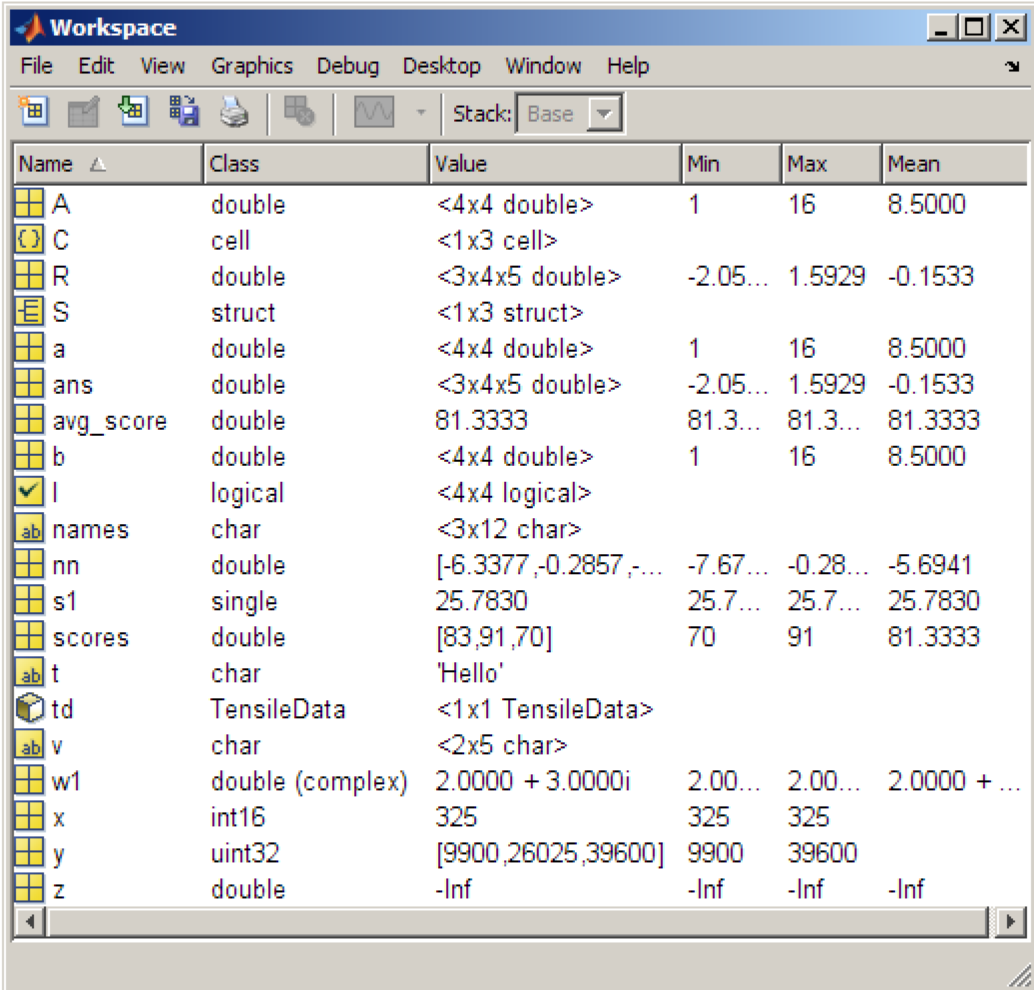
Syntax

```
workspace
```

Description

`workspace` displays the Workspace browser, a graphical user interface that allows you to view and manage the contents of the workspace in MATLAB. 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, which you can choose to show or hide.



You can edit a 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 classes, double-click the variable in the Workspace browser. The variable displays in the Variable Editor, where you can view the full contents and make changes.

See Also

`openvar`, `who`

Tiff.write

Purpose Write entire image

Syntax

```
tiffobj.write(imageData)
tiffobj.write(Y,Cb,Cr)
```

Description

`tiffobj.write(imageData)` writes imageData to TIFF file associated with the Tiff object, `tiffobj`. The `write` method breaks the data into strips or tiles, depending on the value of the `RowsPerStrip` tag, or the `TileLength` and `TileWidth` tags.

`tiffobj.write(Y,Cb,Cr)` writes the YCbCr component data to the TIFF file.

See Also `Tiff.writeDirectory`

Tutorials

-

Purpose Create new IFD and make it current IFD

Syntax `tiffobj.writeDirectory()`

Description `tiffobj.writeDirectory()` create a new image file directory (IFD) and makes it the current IFD. `Tiff` object methods operate on the current IFD. If you are creating a TIFF file that only contains one image, you do not need to use this method. With single-image TIFF files, just close the `Tiff` object to write data to the file.

Examples Open a TIFF file for modification and create a new IFD in the file. `writeDirectory` makes the newly created IFD the current IFD. Replace the name `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r+');
dnum = t.currentDirectory();
t.writeDirectory();
dnum = t.currentDirectory();
```

References This method corresponds to the `TIFFWriteDirectory` function in the LibTIFF C API. To use this method, you must be familiar with LibTIFF version 3.7.1, as well as the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also `Tiff.write` | `Tiff.close`

Tutorials

-
-

Tiff.writeEncodedStrip

Purpose

Write data to specified strip

Syntax

```
tiffobj.writeEncodedStrip(stripNumber,imageData)
tiffobj.writeEncodedStrip(stripNumber,Y,Cb,Cr)
```

Description

`tiffobj.writeEncodedStrip(stripNumber,imageData)` writes the data in `imageData` to the strip specified by `stripNumber`. Strip identification numbers are one-based. If `imageData` has fewer bytes than fit into a strip, `writeEncodedStrip` silently pads the strip. If `imageData` has more bytes than fit into a strip, `writeEncodedStrip` issues a warning and truncates the data. To determine the size of a strip, view the value of the `RowsPerStrip` tag.

`tiffobj.writeEncodedStrip(stripNumber,Y,Cb,Cr)` writes the YCbCr component data to the specified tile. You must set the `YCbCrSubSampling` tag.

Examples

Open a `Tiff` object for modification. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path. The folder in which you run the example must be writable.

```
t = Tiff('myfile.tif', 'r+');

if ~t.isTiled()
   width = t.getTag('ImageWidth');
  height = t.getTag('RowsPerStrip');
  numSamples = t.getTag('SamplesPerPixel');
  imageData = zeros(height,width,numSamples,'uint8');
  t.writeEncodedStrip(1,imageData);
end
```

References

This method corresponds to the `TIFFWriteEncodedStrip` function in the LibTIFF C API. To use this method, you must be familiar with LibTIFF version 3.7.1, as well as the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.writeEncodedTile`

Tutorials

-
-

Tiff.writeEncodedTile

Purpose

Write data to specified tile

Syntax

```
writeEncodedTile(tileNumber,imageData)
tiffobj.writeEncodedTile(tileNumber,Y,Cb,Cr)
```

Description

`writeEncodedTile(tileNumber,imageData)` writes the data in `imageData` to the tile specified by `tileNumber`. Tile identification numbers are one-based. If `imageData` has fewer bytes than fit into a tile, `writeEncodedTile` silently pads the tile. If `imageData` has more bytes than fit into a tile, `writeEncodedTile` issues a warning and truncates the data. To determine the size of a tile, view the value of the `tileLength` and `tileWidth` tags.

`tiffobj.writeEncodedTile(tileNumber,Y,Cb,Cr)` writes the YCbCr component data to the specified tile. You must set the `YCbCrSubSampling` tags.

Examples

Open a TIFF file for modification. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r+');

if t.isTiled()
   width = t.getTag('tileWidth');
  height = t.getTag('tileLength');
  numSamples = t.getTag('SamplesPerPixel');
  imageData = zeros(height,width,numSamples,'uint8');
  t.writeEncodedTile(1,imageData);
end
```

References

This method corresponds to the `TIFFWriteEncodedTile` function in the LibTIFF C API. To use this method, you must be familiar with LibTIFF version 3.7.1, as well as the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.writeEncodedStrip`

Tutorials

-
-

xlabel, ylabel, zlabel

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(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 1-92 for related functions

for more information about labeling axes

xlim, ylim, zlim

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

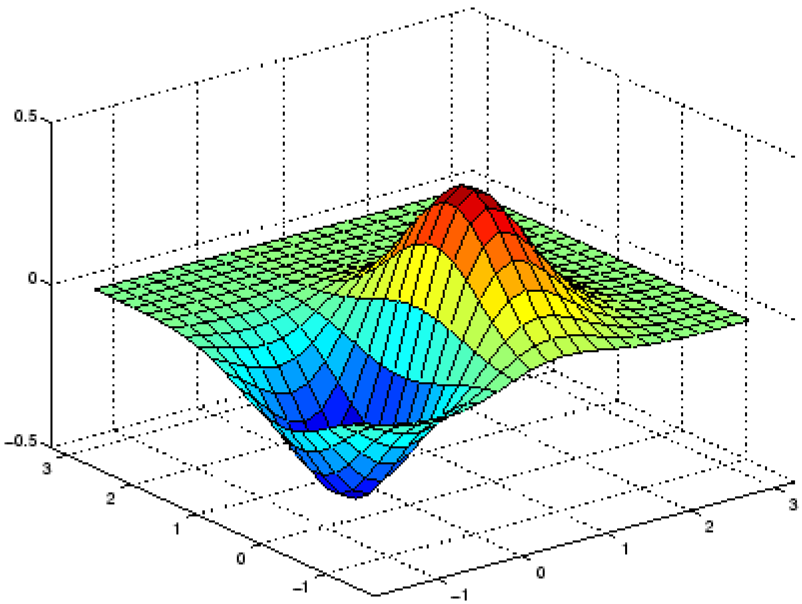
`xlim`, `ylim`, and `zlim` set or query values of the axes object `XLim`, `YLim`, `ZLim`, and `XLimMode`, `YLimMode`, `ZLimMode` properties.

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. 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

"Aspect Ratio and Axis Limits" on page 1-105 for related functions

Understanding Axes Aspect Ratio for more information on how axis limits affect the axes

Purpose

Determine whether file contains a Microsoft Excel spreadsheet

Syntax

```
typ = xlsfinfo(filename)
[typ, desc] = xlsfinfo(filename)
[typ, desc, fmt] = xlsfinfo(filename)
xlsfinfo filename
```

Description

`typ = xlsfinfo(filename)` returns the string `'Microsoft Excel Spreadsheet'` if the file specified by `filename` is an Excel 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 quotation marks.

`[typ, desc] = xlsfinfo(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] = xlsfinfo(filename)` returns in the `fmt` output a string containing the Excel-reported file format. On UNIX systems, or on Windows systems without Excel software installed, `xlsfinfo` returns `fmt` as an empty string, (`''`).

`xlsfinfo filename` is the command format for `xlsfinfo`. It returns only the first output, `typ`, assigning it to the MATLAB default variable `ans`.

Remarks

If your system has Excel for Windows installed, `xlsfinfo` uses the COM server to obtain information. This server is part of the typical installation of Excel for Windows. If the COM server is unavailable, `xlsfinfo` returns a warning indicating that it cannot start an ActiveX server. To establish connectivity with the COM server, you might need to reinstall your Excel software.

Examples

Get information about an `.xls` file:

```
[typ, desc, fmt] = xlsfinfo('myaccount.xls')
```

```
typ =
    Microsoft Excel Spreadsheet

desc =
    'Sheet1'    'Income'    'Expenses'

fmt =
    xlWorkbookNormal
```

Export the `.xls` file to comma-separated value (CSV) format. Use `xlsfinfo` to see the format of the exported file:

```
[typ, desc, fmt] = xlsfinfo('myaccount.csv');
fmt

fmt =
    xlCSV
```

Export the `.xls` file to HTML format. `xlsfinfo` returns the following format string:

```
[typ, desc, fmt] = xlsfinfo('myaccount.html');
fmt

fmt =
    xlHtml
```

Export the `.xls` file to XML format. `xlsfinfo` returns the following format string:

```
[typ, desc, fmt] = xlsfinfo('myaccount.xml');
fmt

fmt =
    xlXMLSpreadsheet
```

See Also

`xlsread`, `xlswrite`

Purpose

Read Microsoft Excel spreadsheet file

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` argument is a string enclosed in single quotation marks.

`xlsread` ignores any *outer* rows or columns of the spreadsheet that contain no numeric data. If there are single or multiple nonnumeric rows at the top or bottom, or single or multiple nonnumeric columns to the left or right, `xlsread` does not include these rows or columns in the output. For example, `xlsread` ignores one or more header lines appearing at the top of a spreadsheet.

Any *inner* rows or columns in which some or all cells contain nonnumeric data are *not* ignored. Instead, `xlsread` assigns a value of `NaN` to the nonnumeric cells.

`num = xlsread(filename, -1)` opens the file `filename` in an Excel window, enabling you to interactively select the worksheet to 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 2-4266 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`). (See "COM Server Requirements" on page 2-4266 below.)

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 reference style. For more information on this reference style, see Excel help.

Note If you specify only two inputs, `xlsread` must decide whether the second input refers to a `sheet` or a `range`. To specify a `range` (even a `range` of a single cell), include a colon character in the input string (e.g., `'D2:H4'`). If you do not include a colon character (e.g., `'sales'` or `'D2'`), `xlsread` interprets the second input as the name or index of a worksheet.

`num = xlsread(filename, sheet, range)` reads data from a specific rectangular region (`range`) of the worksheet specified by `sheet`. If you specify both `sheet` and `range`, `range` can refer to a named range that you defined in the Excel file. (For more information on named ranges, see the Excel help.) See the previous two syntax formats for further explanation of the `sheet` and `range` inputs. (Also, see "COM Server Requirements" on page 2-4266 below.)

`num = xlsread(filename, sheet, range, '`**`basic`**`')` imports data from the spreadsheet in basic import mode. `xlsread` uses this mode on systems where Excel software is not installed. Import ability is limited. `xlsread` ignores the value for `range` and, consequently, imports the whole active range of a sheet. (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 2-4266 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 the Excel application 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.

For more information, see in the MATLAB Programming Fundamentals documentation.

`[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.

`[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 2-4266 below.)

If the Excel file includes cells with undefined values (such as `'#N/A'`), `xlsread` returns these values as `'#N/A'` in the `txt` output, and as `'ActiveX VT_ERROR:'` in the `raw` output.

`[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 2-4266 below.)

`xlsread filename sheet range basic` is an example of the command format for `xlsread`, showing its usage with all input arguments specified. When using this format, you must specify `sheet` as a string, (for example, `Income` or `Sheet4`) and not a numeric index. If the `sheet` name contains space characters, then quotation marks are required around the string, (for example, `'Income 2002'`).

Remarks

COM Server Requirements

The typical installation of Excel for Windows includes the ability to start a COM server. With Excel for Windows installed, you can use `xlsread` to read any file format recognized by your version of Excel, including XLS, XLSX, XLSB, XLSM, and HTML-based formats.

If your system does not have Excel for Windows installed, or MATLAB cannot access the COM server, `xlsread` operates in `basic` mode. In this mode, `xlsread` only reads XLS files.

The following five syntax formats are supported only on computer systems able to start a COM server from a MATLAB session. They are not supported in `basic` mode.

```
num = xlsread(filename, -1)
num = xlsread(filename, 'range')
num = xlsread(filename, sheet, 'range')
num = xlsread(filename, ..., functionhandle)
[num, txt, raw, opt] = xlsread(filename, ..., functionhandle)
```

Handling Excel Date Values

MATLAB functions import all formatted dates as strings. To import a numeric date, the date field in Excel must have a numeric format.

Both Excel and MATLAB applications represent numeric dates as a number of serial days elapsed from a specific reference date. However, Excel and MATLAB use different reference dates:

Application	Reference Date
MATLAB	January 0, 0000

Application	Reference Date
Excel for Windows	January 1, 1900
Excel for the Macintosh	January 2, 1904

Therefore, you must convert any numeric date that you import before you process it in MATLAB. For more information, see in the MATLAB Data Import and Export documentation.

Consider using the `functionhandle` parameter for this conversion, discussed in the Syntax Description and in Example 5 and Example 6.

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 worksheet labeled `Temperatures` in the 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. A user-written function, `setMinMax`, that you pass as a function handle in the call to `xlsread`, performs these modifications. 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. The returned values 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 need to pass this function in the call to `xlsread`, which then executes the function on the data it has read before returning it to you.

When `xlsread` calls your function, it passes an Excel range interface 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
```

Now call `xlsread`, passing a function handle for the `setMinMax` function as the final argument, using `''` as placeholders for `sheet`, `range`, and import mode. After this call, all values are between 0 and 2000:

```
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 returns the indices of the altered elements. 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
        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`:

```
[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`, `xlsfinfo`, `importdata`, `uiimport`, `textscan`, `function_handle`

Purpose

Write Microsoft Excel spreadsheet file

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 quotation marks, and should include the file extension. The matrix `M` is an `m`-by-`n` numeric or character array. `xlswrite` writes the matrix data to the first worksheet in the file, starting at cell `A1`.

The matrix `M` can also be an `m`-by-`n` cell array if each cell includes a single element (see Example 2).

If `filename` does not exist, `xlswrite` creates a new file. The file extension you provide as part of `filename` determines the Excel format that `xlswrite` uses for the new file. An extension of `.xls` creates a worksheet compatible with Excel 97-2003 software. Use extensions `.xlsx`, `.xlsb`, or `.xlsm` to create worksheets in Excel 2007 file formats. The maximum size of the matrix `M` depends on the associated Excel version. (For more information on Excel specifications and limits, see Excel help.)

`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. The `sheet` argument cannot contain a colon.

If `sheet` does not exist, `xlswrite` adds a new sheet at the end of the worksheet collection. If `sheet` is an index larger than the number of worksheets, `xlswrite` appends empty sheets until the number of worksheets in the workbook equals `sheet`. In either case, `xlswrite` 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 the syntax `'C1:C2'`, where `C1` and `C2` are two opposing corners that define the region to write. For example, 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 the Excel A1 reference style. (For more information on this reference style, see Excel help.) `xlswrite` does not recognize named ranges.

The size defined by `range` should fit the size of `M`. If `range` is larger than the size of `M`, Excel software fills the remainder of the region with `#N/A`. If `range` is smaller than the size of `M`, `xlswrite` writes only the submatrix that fits into `range` to the file specified by `filename`.

Note If you specify only three inputs, `xlswrite` must decide whether the third input refers to a `sheet` or a `range`. To specify a `range`, include a colon character in the input string (such as `'D2:H4'`). If you do not include a colon character (such as `'sales'` or `'D2'`), `xlswrite` interprets the third input as a value for `sheet`.

`xlswrite(filename, M, sheet, range)` writes matrix `M` to a rectangular region specified by `range` in worksheet `sheet` of the file `filename`. If you specify both `sheet` and `range`, the `range` can either fit the size of `M` or contain only the first cell (such as `'A2'`). 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 completes successfully, `status` is equal to logical `1` (`true`). Otherwise, `status` is logical `0` (`false`). Unless you specify an output parameter, `xlswrite` does not display a status value 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 you must place quotation marks around the string (for example, `'Income 2002'`).

Remarks

Full functionality of `xlswrite` depends on the use of the Microsoft Excel COM server. The typical installation of Excel for Windows includes access to this server. If your system does not have Excel for Windows installed, or if the COM server is unavailable, `xlswrite`:

- Writes matrix `M` as a text file in comma-separated value (CSV) format.
- Ignores the `sheet` and `range` arguments.
- Generates an error if the input matrix `M` is a cell array.

If your system has Microsoft Office 2003 software installed, but you want to create a file in an Excel 2007 format, you must install the Office 2007 Compatibility Pack.

Both Excel and MATLAB applications represent numeric dates as a number of serial days elapsed from a specific reference date. However, Excel and MATLAB use different reference dates:

Application	Reference Date
MATLAB	January 0, 0000

Application	Reference Date
Excel for Windows	January 1, 1900
Excel for the Macintosh	January 2, 1904

For more information, see in the MATLAB Data Import and Export documentation.

Examples

Example 1 — Writing Numeric Data to the Default Worksheet

Write a 7-element vector to Microsoft Excel file `testdata.xls`. By default, `xlswrite` writes the data to cells `A1` through `G1` in the first worksheet in the file:

```
xlswrite('testdata.xls', [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. `xlswrite` writes the 4-by-2 matrix 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, `xlswrite` appends a new sheet to the workbook, calling it by the name you supplied in the `sheets` input argument, `'NewTemp'`. `xlswrite` 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 with this command:

```
warning off MATLAB:xlswrite:AddSheet
```

Now try the write command again, this time creating another new worksheet, `NewTemp2`. Although the message does not appear this time, you can still retrieve it and its identifier from the second output argument, `msg`:

```
[stat msg] = xlswrite('tempdata.xls', d, 'NewTemp2', 'E1');

msg
msg =
       message: 'Added specified worksheet.'
    identifier: 'MATLAB:xlswrite:AddSheet'
```

See Also

`xlsread`, `xlsfinfo`

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.',filename);
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;
```

```
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

```
xmlwrite(filename, DOMnode)
str = xmlwrite(DOMnode)
```

Description

`xmlwrite(filename, DOMnode)` serializes the Document Object Model node `DOMnode` to the file specified by `filename`. The `filename` input is a string enclosed in single quotes.

`str = xmlwrite(DOMnode)` serializes the Document Object Model node `DOMnode` and returns the node tree as a string, `s`.

Remarks

Find out more about the Document Object Model at the World Wide Web Consortium (W3C) Web site, `http://www.w3.org/DOM/`.

Example

```
% Create a sample XML document.
docNode = com.mathworks.xml.XMLUtils.createDocument...
    ('root_element')
docRootNode = docNode.getDocumentElement;
for i=1:20
    thisElement = docNode.createElement('child_node');
    thisElement.appendChild...
        (docNode.createTextNode(sprintf('%i',i)));
    docRootNode.appendChild(thisElement);
end
docNode.appendChild(docNode.createComment('this is a comment'));

% Save the sample XML document.
xmlFileName = [tempname,'.xml'];
xmlwrite(xmlFileName,docNode);
edit(xmlFileName);
```

See Also

`xmlread`, `xslt`

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

```
result = xslt(source, style, dest)
[result,style] = xslt(...)
xslt(...,'-web')
```

Description

`result = xslt(source, style, dest)` transforms an XML document using a stylesheet and returns the resulting document's URL. The function uses these inputs, the first of which is required:

- `source` is the filename or URL of the source XML file. `source` can also specify a DOM node.
- `style` is the filename or URL of an XSL stylesheet.
- `dest` is the filename or URL of the desired output document. If `dest` is absent or empty, the function uses a temporary filename. If `dest` is `'-tostring'`, the function returns the output document as a MATLAB string.

`[result,style] = xslt(...)` returns a processed stylesheet appropriate for passing to subsequent XSLT calls as `style`. This prevents costly repeated processing of the stylesheet.

`xslt(...,'-web')` displays the resulting document in the Help Browser.

Remarks

Find out more about XSL stylesheets and how to write them at the World Wide Web Consortium (W3C) web site, `http://www.w3.org/Style/XSL/`.

Example

This example converts the file `info.xml` using the stylesheet `info.xsl`, writing the output to the file `info.html`. It launches the resulting HTML file in the Help Browser. MATLAB has several `info.xml` files that are used by the **Start** menu.

```
xslt info.xml info.xsl info.html -web
```

See Also

`xmlread`, `xmlwrite`

Purpose

Create 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 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,rootfolder)
entrynames = zip(...)
```

Description

`zip(`*`zipfile`*`,`*`files`*`)` creates a zip file with the name *`zipfile`* from the list of files and folders specified in *`files`*. Folders recursively include all of their content. If *`files`* includes relative paths, the zip file also contains relative paths. The zip file does not include absolute paths.

`zipfile` is a string specifying the name of the zip file. If *`zipfile`* has no extension, MATLAB appends the `.zip` extension.

`files` is a string or cell array of strings containing the list of files or folders included in *`zipfile`*.

Individual files that are on the MATLAB path can be specified as partial path names. Otherwise an individual file can be specified relative to the current folder or with an absolute path.

Folders must be specified relative to the current folder or with absolute paths. On UNIX systems, folders can also start with `~/` or `~`*`username`*`/`, which expands to the current user's home folder or the specified user's home folder, respectively. The wildcard character `*` can be used when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

`zip(`*`zipfile`*`,`*`files`*`,`*`rootfolder`*`)` specifies the path for *`files`* relative to *`rootfolder`* instead of the current folder. Relative paths in the zip file reflect the relative paths in *`files`*, and do not include path information from *`rootfolder`*.

`entrynames`` = zip(...)` returns a string cell array of the names of the files contained in *`zipfile`*. If *`files`* includes relative paths, *`entrynames`* also contains relative paths.

Examples

Zip a File

Create a zip file of the file `membrane.m`, which is in the MATLAB `demos` folder. Save the zip file in `tmwlogo.zip` in the current folder.

```
file = fullfile(matlabroot,'toolbox','matlab','demos','membrane.m');
zip('tmwlogo',file);
```

Run `zip` for the files `membrane.m` and `logo.m` and save the zip file, `tmwlogo.zip`, in the specified folder. The source files are on the MATLAB search path.

```
myfile = fullfile('d:','myfiles','tmwlogo.zip');
zip(myfile,{'membrane.m','logo.m'});
```

Zip Selected Files

Run `zip` for all `.m` and `.mat` files in the current folder to the file `backup.zip`:

```
zip('backup',{'*.m','*.mat'});
```

Zip a Folder

Run `zip` for the folder `mywork`, which is a subfolder of the current folder. The zip file `myfiles.zip` recursively includes the contents of all subfolders of `mywork`, and stores the relative paths.

```
zip('myfiles.zip','mywork');
```

Zip Between Folders

Run `zip` for the files `thesis.doc` and `defense.ppt`, which are located in `d:/PhD`, to the zip file `thesis.zip` in the folder one level up from the current folder.

```
zip('../thesis.zip',{'thesis.doc','defense.ppt'},'d:/PhD');
```

See Also

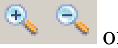
`gzip`, `gunzip`, `tar`, `untar`, `unzip`

zoom

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 in the MATLAB Graphics documentation.

Syntax

```
zoom on
zoom off
zoom out
zoom reset
zoom
zoom xon
zoom yon
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/factor`.

`zoom(fig, option)` Any of the preceding 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 the figure `figure_handle` for you to customize the mode's behavior.

Using Zoom Mode Objects

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, a read-only property that 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'|'PostContextMenu'` — The behavior of a right-click action

  A value of `'InverseZoom'` causes a right-click to zoom out. A value of `'PostContextMenu'` displays a context menu. This setting persists between MATLAB sessions.
- `UIContextMenu <handle>` — Specifies a custom context menu to be displayed during a right-click action

  This property is ignored if the `RightClickAction` property has been set to `'on'`.

Zoom Mode Callbacks

You can program the following callbacks for zoom mode operations.

- `ButtonDownFilter <function_handle>` — Function to intercept `ButtonDown` events

  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 Graphics object callbacks), as follows:

  ```
  function [res] = myfunction(obj,event_obj)
  % obj          handle to the object that has been clicked on
  % event_obj    struct for event data (empty in this release)
  % res [output] 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>` — Function to execute before zooming

Set this callback if you want to execute code when a zoom operation starts. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

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

The event data has the following field.

Axes	The handle of the axes that is being zoomed

- ActionPostCallback <function_handle> — Function to execute after zooming

  Set this callback if you want to execute code when a zoom operation finishes. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

```
function myfunction(obj,event_obj)
% obj         handle to the figure that has been clicked on
% event_obj   object containing struct of event data (same as the
%             event data of the 'ActionPreCallback' callback)
```

Zoom Mode Utility Functions

The following functions in zoom mode query and set certain of its properties.

- flags = isAllowAxesZoom(h,axes) — Function querying permission to zoom axes

  Calling the function isAllowAxesZoom on the zoom object, h, with a vector of axes handles, axes, as input returns a logical array of the same dimension as the axes handle vector, which indicates whether a zoom operation is permitted on the axes objects.

- `setAllowAxesZoom(h,axes,flag)` — Function to set permission to zoom axes

  Calling the function `setAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, either allows or disallows a zoom operation on the axes objects.

- `*info* = getAxesZoomMotion(h,axes)` — Function to get style of zoom operations

  Calling the function `getAxesZoomMotion` on the zoom object, `H`, with a vector of axes handles, `axes`, as input returns 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)` — Function to set style of zoom operations

  Calling the function `setAxesZoomMotion` on the zoom object, `h`, with a vector of axes handles, `axes`, and a character array, `style`, ses the style of zooming on each axes.

Examples

Example 1 — Entering Zoom Mode

Plot a graph and turn on Zoom mode:

```
plot(1:10);
zoom on
% zoom in on the plot
```

Example 2 — Constrained Zoom

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 — Constrained Zoom in Subplots

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 — Coding a ButtonDown Callback

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 — Coding Pre- and Post-Callback Behavior

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));
```

Example 6 — Creating a Context Menu for Zoom Mode

Coding a context menu that lets the user to switch to Pan mode by right-clicking:

```
figure;plot(magic(10))
hCMZ = uicontextmenu;
```

```
hZMenu = uimenu('Parent',hCMZ,'Label','Switch to pan','Callback','p
hZoom = zoom(gcf);
set(hZoom,'UIContextMenu',hCMZ);
zoom('on')
```

You cannot add items to the built-in zoom context menu, but you can replace it with your own.

Remarks

`zoom` changes the axes limits by a factor of 2 (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.

Note Do not change figure callbacks within an interactive mode. While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a GUI that updates a figure's callbacks, the GUI should some keep track of which interactive mode is active, if any, before attempting to do this.

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 carries 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 1-105 for related functions

Index

Symbols and Numerics

A

D

E

F

H

I

N

O

Q

T

U

X

Z