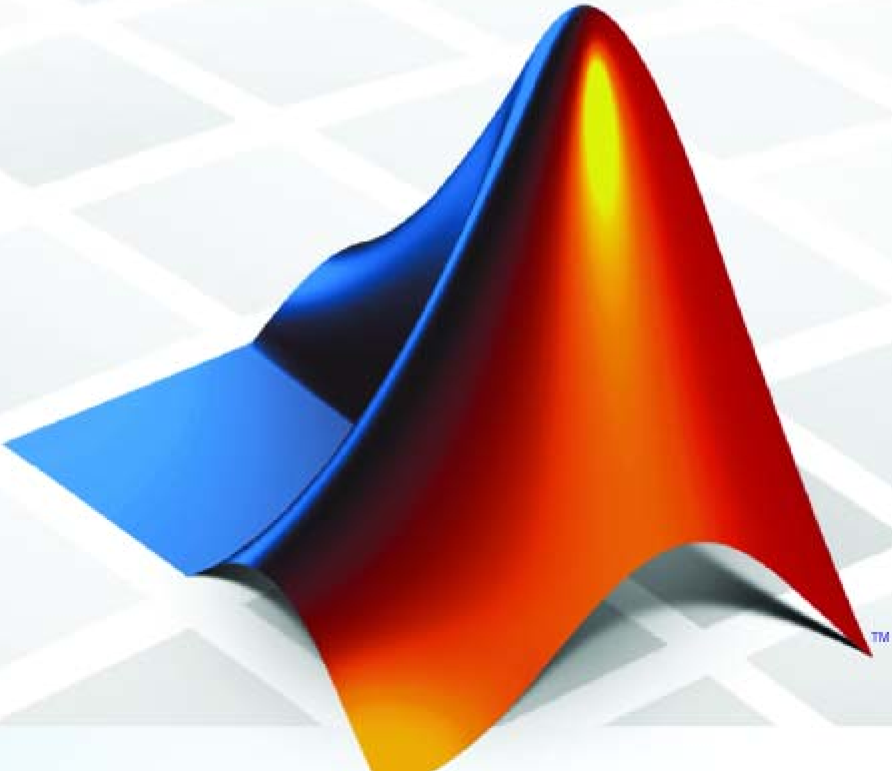
MATLAB® 7

Function Reference: Volume 2 (F-O)



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

© COPYRIGHT 1984–2009 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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)

Function Reference

1

Alphabetical List

2

Index

1

Function Reference

Desktop Tools and Development Environment (p. 1-3)	Startup, Command Window, help, editing and debugging, tuning, other general functions
Data Import and Export (p. 1-13)	General and low-level file I/O, plus specific file formats, like audio, spreadsheet, HDF, images
Mathematics (p. 1-24)	Arrays and matrices, linear algebra, other areas of mathematics
Data Analysis (p. 1-53)	Basic data operations, descriptive statistics, covariance and correlation, filtering and convolution, numerical derivatives and integrals, Fourier transforms, time series analysis
Programming and Data Types (p. 1-61)	Function/expression evaluation, program control, function handles, object oriented programming, error handling, operators, data types, dates and times, timers
Object-Oriented Programming (p. 1-87)	Functions for working with classes and objects
Graphics (p. 1-91)	Line plots, annotating graphs, specialized plots, images, printing, Handle Graphics
3-D Visualization (p. 1-102)	Surface and mesh plots, view control, lighting and transparency, volume visualization

GUI Development (p. 1-108)	GUIDE, programming graphical user interfaces
External Interfaces (p. 1-113)	Interfaces to shared librariess, Java, .NET, COM and ActiveX, Web services, and serial port devices, and C and Fortran routines

Desktop Tools and Development Environment

Startup and Shutdown (p. 1-3)	Startup and shutdown options, preferences
Command Window and History (p. 1-4)	Control Command Window and History, enter statements and run functions
Help for Using MATLAB (p. 1-5)	Command line help, online documentation in the Help browser, demos
Workspace, Search Path, and File Operations (p. 1-6)	Work with files, MATLAB search path, manage variables
Programming Tools (p. 1-8)	Edit and debug M-files, improve performance, source control, publish results
System (p. 1-10)	Identify current computer, license, product version, and more

Startup and Shutdown

exit	Terminate MATLAB® program (same as `quit`)
finish	Termination M-file for MATLAB program
matlab (UNIX)	Start MATLAB program (UNIX® platforms)
matlab (Windows)	Start MATLAB program (Windows® platforms)
matlabrc	Startup M-file for MATLAB program
prefdir	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 (p. 1-6)	Manage variables
Search Path (p. 1-6)	View and change MATLAB search path
File Operations (p. 1-7)	View and change files and directories

Workspace

assignin	Assign value to variable in specified workspace
clear	Remove items from workspace, freeing up system memory
evalin	Execute MATLAB expression in specified workspace
exist	Check existence of variable, function, directory, or 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 (p. 1-8)	Edit and debug M-files
M-File Performance (p. 1-9)	Improve performance and find potential problems in M-files
Source Control (p. 1-10)	Interface MATLAB with source control system
Publishing (p. 1-10)	Publish M-file code and results

M-File Editing and Debugging

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

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

File Name Construction (p. 1-13)	Get path, directory, filename information; construct filenames
File Opening, Loading, and Saving (p. 1-14)	Open files; transfer data between files and MATLAB workspace
Memory Mapping (p. 1-14)	Access file data via memory map using MATLAB array indexing
Low-Level File I/O (p. 1-14)	Low-level operations that use a file identifier
Text Files (p. 1-15)	Delimited or formatted I/O to text files
XML Documents (p. 1-16)	Documents written in Extensible Markup Language
Spreadsheets (p. 1-16)	Excel and Lotus 1-2-3 files
Scientific Data (p. 1-17)	CDF, FITS, HDF formats
Audio and Video (p. 1-20)	Read and write audio and video, record and play audio
Images (p. 1-22)	Graphics files
Internet Exchange (p. 1-22)	URL, FTP, zip, tar, and e-mail

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

File Name Construction

filemarker	Character to separate file name and internal function name
fileparts	Parts of file name and path
filesep	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 (p. 1-17)	Work with CDF files
Network Common Data Form (p. 1-17)	Work with netCDF files
Flexible Image Transport System (p. 1-19)	Work with FITS files
Hierarchical Data Format (p. 1-19)	Work with HDF files
Band-Interleaved Data (p. 1-20)	Work with band-interleaved files

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 (p. 1-25)	Basic array operators and operations, creation of elementary and specialized arrays and matrices
Linear Algebra (p. 1-30)	Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization
Elementary Math (p. 1-34)	Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math
Polynomials (p. 1-39)	Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion
Interpolation and Computational Geometry (p. 1-39)	Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation
Cartesian Coordinate System Conversion (p. 1-43)	Conversions between Cartesian and polar or spherical coordinates
Nonlinear Numerical Methods (p. 1-43)	Differential equations, optimization, integration
Specialized Math (p. 1-47)	Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions
Sparse Matrices (p. 1-48)	Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations
Math Constants (p. 1-51)	Pi, imaginary unit, infinity, Not-a-Number, largest and smallest positive floating point numbers, floating point relative accuracy

Arrays and Matrices

Basic Information (p. 1-25)	Display array contents, get array information, determine array type
Operators (p. 1-26)	Arithmetic operators
Elementary Matrices and Arrays (p. 1-27)	Create elementary arrays of different types, generate arrays for plotting, array indexing, etc.
Array Operations (p. 1-28)	Operate on array content, apply function to each array element, find cumulative product or sum, etc.
Array Manipulation (p. 1-29)	Create, sort, rotate, permute, reshape, and shift array contents
Specialized Matrices (p. 1-30)	Create Hadamard, Companion, Hankel, Vandermonde, Pascal matrices, etc.

Basic Information

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

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

Trigonometric (p. 1-35)	Trigonometric functions with results in radians or degrees
Exponential (p. 1-36)	Exponential, logarithm, power, and root functions
Complex (p. 1-37)	Numbers with real and imaginary components, phase angles

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 (p. 1-40)	Data interpolation, data gridding, polynomial evaluation, nearest point search
Delaunay Triangulation and Tessellation (p. 1-41)	Delaunay triangulation and tessellation, triangular surface and mesh plots
Convex Hull (p. 1-42)	Plot convex hull, plotting functions
Voronoi Diagrams (p. 1-42)	Plot Voronoi diagram, patch graphics object, plotting functions
Domain Generation (p. 1-43)	Generate arrays for 3-D plots, or for N-D functions and interpolation

Interpolation

dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
griddata	Data gridding
griddata3	Data gridding and hypersurface fitting for 3-D data
griddatan	Data gridding and hypersurface fitting (dimension >= 2)
interp1	1-D data interpolation (table lookup)
interp1q	Quick 1-D linear interpolation
interp2	2-D data interpolation (table lookup)
interp3	3-D data interpolation (table lookup)
interpft	1-D interpolation using FFT method
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 (p. 1-53)	Sums, products, sorting
Descriptive Statistics (p. 1-53)	Statistical summaries of data
Filtering and Convolution (p. 1-54)	Data preprocessing
Interpolation and Regression (p. 1-54)	Data fitting
Fourier Transforms (p. 1-55)	Frequency content of data
Derivatives and Integrals (p. 1-55)	Data rates and accumulations
Time Series Objects (p. 1-56)	Methods for `timeseries` objects
Time Series Collections (p. 1-59)	Methods for `tscollection` objects

Basic Operations

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

Descriptive Statistics

corrcoef	Correlation coefficients
cov	Covariance matrix

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

Filtering and Convolution

conv	Convolution and polynomial multiplication
conv2	2-D convolution
convn	N-D convolution
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 (p. 1-56)	Combine `timeseries` objects, query and set `timeseries` object properties, plot `timeseries` objects
Data Manipulation (p. 1-57)	Add or delete data, manipulate `timeseries` objects
Event Data (p. 1-58)	Add or delete events, create new `timeseries` objects based on event data
Descriptive Statistics (p. 1-58)	Descriptive statistics for `timeseries` objects

Utilities

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

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

Data Manipulation

addsample	Add data sample to `timeseries` object
ctranspose (timeseries)	Transpose `timeseries` object
delsample	Remove sample from `timeseries` object
detrend (timeseries)	Subtract mean or best-fit line and all `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 (p. 1-59)	Query and set `tscollection` object properties, plot `tscollection` objects
Data Manipulation (p. 1-60)	Add or delete data, manipulate `tscollection` objects

Utilities

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

Data Manipulation

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 (p. 1-61)	Numeric, character, structures, cell arrays, and data type conversion
Data Type Conversion (p. 1-69)	Convert one numeric type to another, numeric to string, string to numeric, structure to cell array, etc.
Operators and Special Characters (p. 1-71)	Arithmetic, relational, and logical operators, and special characters
Strings (p. 1-74)	Create, identify, manipulate, parse, evaluate, and compare strings
Bit-Wise Operations (p. 1-77)	Perform set, shift, and, or, compare, etc. on specific bit fields
Logical Operations (p. 1-77)	Evaluate conditions, testing for true or false
Relational Operations (p. 1-78)	Compare values for equality, greater than, less than, etc.
Set Operations (p. 1-78)	Find set members, unions, intersections, etc.
Date and Time Operations (p. 1-79)	Obtain information about dates and times
Programming in MATLAB (p. 1-79)	M-files, function/expression evaluation, program control, function handles, object oriented programming, error handling

Data Types

Numeric Types (p. 1-62)	Integer and floating-point data
Characters and Strings (p. 1-63)	Characters and arrays of characters
Structures (p. 1-64)	Data of varying types and sizes stored in fields of a structure

Cell Arrays (p. 1-65)	Data of varying types and sizes stored in cells of array
Function Handles (p. 1-66)	Invoke a function indirectly via handle
Java Classes and Objects (p. 1-66)	Access Java classes through MATLAB interface
Data Type Identification (p. 1-68)	Determine data type of a variable

Numeric Types

arrayfun	Apply function to each element of array
cast	Cast variable to different data type
cat	Concatenate arrays along specified dimension
class	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 (p. 1-69)	Convert data of one numeric type to another numeric type
String to Numeric (p. 1-70)	Convert characters to numeric equivalent
Numeric to String (p. 1-70)	Convert numeric to character equivalent
Other Conversions (p. 1-71)	Convert to structure, cell array, function handle, etc.

Numeric

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

String to Numeric

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

Numeric to String

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

Other Conversions

cell2mat	Convert cell array of matrices to single matrix
cell2struct	Convert cell array to structure array
datestr	Convert date and time to string format
func2str	Construct function name string from function handle
logical	Convert numeric values to logical
mat2cell	Divide matrix into cell array of matrices
num2cell	Convert numeric array to cell array
num2hex	Convert singles and doubles to IEEE® hexadecimal strings
str2func	Construct function handle from function name string
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 (p. 1-74)	Basics of string handling in MATLAB
String Creation (p. 1-74)	Create strings, cell arrays of strings, concatenate strings together
String Identification (p. 1-75)	Identify characteristics of strings
String Manipulation (p. 1-75)	Convert case, strip blanks, replace characters
String Parsing (p. 1-76)	Formatted read, regular expressions, locate substrings
String Evaluation (p. 1-76)	Evaluate stated expression in string
String Comparison (p. 1-76)	Compare contents of strings

Description of Strings in MATLAB

strings	String handling

String Creation

blanks	Create string of blank characters
cellstr	Create cell array of strings from character array
char	Convert to character array (string)
sprintf	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 (p. 1-80)	Declare functions, handle arguments, identify dependencies, etc.
Evaluation (p. 1-81)	Evaluate expression in string, apply function to array, run script file, etc.
Timer (p. 1-82)	Schedule execution of MATLAB commands

Variables and Functions in Memory (p. 1-83)	List files in memory, clear M-files in memory, assign to variable in nondefault workspace, refresh caches
Control Flow (p. 1-84)	if-then-else, for loops, switch-case, try-catch
Error Handling (p. 1-85)	Generate warnings and errors, test for and catch errors, retrieve most recent error message
MEX Programming (p. 1-86)	Compile MEX function from C or Fortran code, list MEX-files in memory, debug MEX-files

M-Files and Scripts

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

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 (p. 1-87)	Get information about classes and objects
Handle Classes (p. 1-88)	Define and use handle classes
Events and Listeners (p. 1-89)	Define and use events and listeners
Meta-Classes (p. 1-89)	Access information about classes without requiring instances

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

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

Events and Listeners

addlistener (handle)	Create event listener
event.EventData	Base class for all data objects passed to event listeners
event.listener	Class defining listener objects
event.PropertyEvent	Listener for property events
event.proplistener	Define listener object for property events
events	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 (p. 1-91)	Linear line plots, log and semilog plots
Plotting Tools (p. 1-92)	GUIs for interacting with plots
Annotating Plots (p. 1-92)	Functions for and properties of titles, axes labels, legends, mathematical symbols
Specialized Plotting (p. 1-93)	Bar graphs, histograms, pie charts, contour plots, function plotters
Bit-Mapped Images (p. 1-96)	Display image object, read and write graphics file, convert to movie frames
Printing (p. 1-97)	Printing and exporting figures to standard formats
Handle Graphics (p. 1-97)	Creating graphics objects, setting properties, finding handles

Basic Plots and Graphs

box	Axes border
errorbar	Plot error bars along curve
hold	Retain current graph in figure
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 (p. 1-93)	1-D, 2-D, and 3-D graphs and charts
Contour Plots (p. 1-94)	Unfilled and filled contours in 2-D and 3-D
Direction and Velocity Plots (p. 1-94)	Comet, compass, feather and quiver plots
Discrete Data Plots (p. 1-94)	Stair, step, and stem plots
Function Plots (p. 1-94)	Easy-to-use plotting utilities for graphing functions
Histograms (p. 1-95)	Plots for showing distributions of data
Polygons and Surfaces (p. 1-95)	Functions to generate and plot surface patches in two or more dimensions
Scatter/Bubble Plots (p. 1-96)	Plots of point distributions
Animation (p. 1-96)	Functions to create and play movies of plots

Area, Bar, and Pie Plots

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

Contour Plots

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

Direction and Velocity Plots

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

Discrete Data Plots

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

Function Plots

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

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

Graphics Object Identification (p. 1-98)	Find and manipulate graphics objects via their handles
Object Creation (p. 1-99)	Constructors for core graphics objects
Plot Objects (p. 1-99)	Property descriptions for plot objects
Figure Windows (p. 1-100)	Control and save figures

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 (p. 1-102)	Plot matrices, visualize functions of two variables, specify colormap
View Control (p. 1-104)	Control the camera viewpoint, zooming, rotation, aspect ratio, set axis limits
Lighting (p. 1-106)	Add and control scene lighting
Transparency (p. 1-106)	Specify and control object transparency
Volume Visualization (p. 1-106)	Visualize gridded volume data

Surface and Mesh Plots

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

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

Camera Viewpoint

camdolly	Move camera position and target
cameratoolbar	Control camera toolbar programmatically
camlookat	Position camera to view object or group of objects
camorbit	Rotate camera position around camera target
campan	Rotate camera target around camera position
campos	Set or query camera position
camproj	Set or query projection type
camroll	Rotate camera about view axis
camtarget	Set or query location of camera target

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

Predefined Dialog Boxes

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

Shared Libraries (p. 1-113)	Access functions stored in external shared library files
Java (p. 1-114)	Work with objects constructed from Java API and third-party class packages
.NET (p. 1-115)	Work with objects constructed from .NET assemblies
Component Object Model and ActiveX (p. 1-115)	Integrate COM components into your application
Web Services (p. 1-118)	Communicate between applications over a network using SOAP and WSDL
Serial Port Devices (p. 1-118)	Read and write to devices connected to your computer's serial port

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

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)

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

asin
asind
asinh
assert
assignin
atan
atan2
atand
atanh
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

betaln
bicg
bicgstab
bicgstabl
bin2dec
binary
bitand
bitcmp
bitget
bitmax
bitor
bitset
bitshift
bitxor
blanks
blkdiag
box
break
brighten
brush
bsxfun
builddocsearchdb
builtin
bvp4c
bvp5c
bvpget
bvpinit
bvpset
bvpxtend
calendar
calllib
callSoapService
camdolly
cameratoolbar
camlight
camlookat
camorbit

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

cholinc
cholupdate
circshift
circumcenters
cla
clabel
class
classdef
clc
clear
clearvars
clear (serial)
clf
clipboard
clock
close
close
close (avifile)
close (ftp)
closereq
cmopts
cmpermute
cmunique
colamd
colorbar
colordef
colormap
colormapeditor
ColorSpec (Color Specification)
colperm
comet
comet3
commandhistory
commandwindow
compan
compass
complex

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

cross
csc
cscd
csch
csvread
csvwrite
ctranspose (timeseries)
cumprod
cumsum
cumtrapz
curl
currentDirectory
customverctrl
cylinder
daqread
daspect
datacursormode
datatipinfo
date
datenum
datestr
datetick
datevec
dbclear
dbcont
dbdown
dblquad
dbmex
dbquit
dbstack
dbstatus
dbstep
dbstop
dbtype
dbup
dde23
ddeget

ddesd
ddeset
deal
deblank
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

dir
dir (ftp)
disp
disp (memmapfile)
disp (MException)
disp (serial)
disp (timer)
display
dither
divergence
dlmread
dlmwrite
dmperm
doc
docopt
docsearch
dos
dot
double
dragrect
drawnow
dsearch
dsearchn
dynamicprops
echo
echodemo
edgeAttachments
edges
edit
eig
eigs
ellipj
ellipke
ellipsoid
else
elseif
enableNETfromNetworkDrive

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

ezmeshc
ezplot
ezplot3
ezpolar
ezsurf
ezsurfc
faceNormals
factor
factorial
false
fclose
fclose (serial)
feather
featureEdges
feof
ferror
feval
Feval (COM)
fft
fft2
fftn
fftshift
fftw
fgetl
fgetl (serial)
fgets
fgets (serial)
fieldnames
figure
Figure Properties
figurepalette
fileattrib
filebrowser
File Formats
filemarker
fileparts
fileread

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

fscanf
fscanf (serial)
fseek
ftell
ftp
full
fullfile
func2str
function
function_handle (@)
functions
funm
fwrite
fwrite (serial)
fzero
gallery
gamma, gammainc, gammaln
gammaincinv
gca
gcbf
gcbo
gcd
gcf
gco
ge
genpath
genvarname
get
get (COM)
get (hgsetget)
get (memmapfile)
get (RandStream)
get (serial)
get (timer)
get (timeseries)
get (tscollection)
getabstime (timeseries)

getabstime (tscollection)
getappdata
GetCharArray
getdatasamplesize
getDefaultStream (RandStream)
getdisp (hgsetget)
getenv
getfield
getframe
GetFullMatrix
getinterpmethod
getpixelposition
getpref
getqualitydesc
getReport (MException)
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

isosurface
ispc
ispref
isprime
isprop
isreal
isscalar
issorted
isspace
issparse
isstr
isstrprop
isstruct
isstudent
isTiled
isunix
isvalid (handle)
isvalid (serial)
isvalid (timer)
isvarname
isvector
j
javaaddpath
javaArray
javachk
javaclasspath
javaMethod
javaMethodEDT
javaObject
javaObjectEDT
javarmpath
keyboard
keys (Map)
kron
last (MException)
lastDirectory
lasterr

lasterror
lastwarn
lcm
ldl
ldivide, rdivide
le
legend
legendre
length
length (Map)
length (serial)
length (timeseries)
length (tscollection)
libfunctions
libfunctionsview
libisloaded
libpointer
libstruct
license
light
Light Properties
lightangle
lighting
lin2mu
line
Line Properties
Lineseries Properties
LineSpec (Line Specification)
linkaxes
linkdata
linkprop
linsolve
linspace
list (RandStream)
listdlg
listfonts
load

load (COM)
load (serial)
loadlibrary
loadobj
log
log10
log1p
log2
logical
loglog
logm
logspace
lookfor
lower
ls
lscov
lsqnonneg
lsqr
lt
lu
luinc
magic
makehgtform
containers.Map
mat2cell
mat2str
material
matlabcolon (matlab:)
matlabrc
matlabroot
matlab (UNIX)
matlab (Windows)
max
max (timeseries)
MaximizeCommandWindow
maxNumCompThreads
mean

mean (timeseries)
median
median (timeseries)
memmapfile
memory
menu
mesh, meshc, meshz
meshgrid
meta.class
meta.class.fromName
meta.DynamicProperty
meta.event
meta.method
meta.package
meta.package.fromName
meta.package.getAllPackages
meta.property
metaclass
methods
methodsview
mex
mex.getCompilerConfigurations
MException
mexext
mfilename
mget
min
min (timeseries)
MinimizeCommandWindow
minres
mislocked
mkdir
mkdir (ftp)
mkpp
mldivide \, mrdivide /
mlint
mlintrpt

mlock
mmfileinfo
mmreader
mmreader.isPlatformSupported
mod
mode
more
move
movefile
movegui
movie
movie2avi
mput
msgbox
mtimes
mu2lin
multibandread
multibandwrite
munlock
namelengthmax
NaN
nargchk
nargin, nargout
nargoutchk
native2unicode
nchoosek
ndgrid
ndims
ne
nearestNeighbor
ne (MException)
neighbors
NET
NET.addAssembly
NET.Assembly
NET.convertArray
NET.createArray

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

netcdf.sync
newplot
nextDirectory
nextpow2
nnz
noanimate
nonzeros
norm
normest
not
notebook
notify (handle)
now
nthroot
null
num2cell
num2hex
num2str
numberOfStrips
numberOfTiles
numel
nzmax
ode15i
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
odefile
odeget
odeset
odextend
onCleanup
ones
open
openfig
opengl
openvar
optimget
optimset
or

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

pie3
pinv
planerot
playshow
plot
plot (timeseries)
plot3
plotbrowser
plotedit
plotmatrix
plottools
plotyy
pointLocation
pol2cart
polar
poly
polyarea
polyder
polyeig
polyfit
polyint
polyval
polyvalm
pow2
power
ppval
prefdir
preferences
primes
print, printopt
printdlg
printpreview
prod
profile
profsave
propedit
propedit (COM)

properties
propertyeditor
psi
publish
PutCharArray
PutFullMatrix
PutWorkspaceData
pwd
qmr
qr
qrdelete
qrinsert
qrupdate
quad
quad2d
quadgk
quadl
quadv
questdlg
quit
Quit (COM)
quiver
quiver3
Quivergroup Properties
qz
rand
rand (RandStream)
randi
randi (RandStream)
randn
randn (RandStream)
randperm
randperm (RandStream)
RandStream
RandStream (RandStream)
rank
rat, rats

rbbox
rcond
read (mmreader)
read
readasync
readEncodedStrip
readEncodedTile
real
reallog
realmax
realmin
realpow
realsqrt
record
rectangle
Rectangle Properties
rectint
recycle
reducepatch
reducevolume
refresh
refreshdata
regexp, regexpi
regexprep
regexptranslate
registerevent
rehash
release
relationaloperators (handle)
rem
remove (Map)
removets
rename
repmat
resample (timeseries)
resample (tscollection)
reset

reset (RandStream)
reshape
residue
restoredefaultpath
rethrow
rethrow (MException)
return
rewriteDirectory
rgb2hsv
rgb2ind
rgbplot
ribbon
rmappdata
rmdir
rmdir (ftp)
rmfield
rmpath
rmpref
root object
Root Properties
roots
rose
rosser
rot90
rotate
rotate3d
round
rref
rsf2csf
run
save
save (COM)
save (serial)
saveas
saveobj
savepath
scatter

scatter3
Scattergroup Properties
schur
script
sec
secd
sech
selectmoveresize
semilogx, semilogy
sendmail
serial
serialbreak
set
set (COM)
set (hgsetget)
set (RandStream)
set (serial)
set (timer)
set (timeseries)
set (tscollection)
setabstime (timeseries)
setabstime (tscollection)
setappdata
setDefaultStream (RandStream)
setdiff
setDirectory
setdisp (hgsetget)
setenv
setfield
setinterpmethod
setpixelposition
setpref
setstr
setSubDirectory
setTag
settimeseriesnames
setxor

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

sprand
sprandn
sprandsym
sprank
sprintf
spy
sqrt
sqrtm
squeeze
ss2tf
sscanf
stairs
Stairseries Properties
start
startat
startup
std
std (timeseries)
stem
stem3
Stemseries Properties
stop
stopasync
str2double
str2func
str2mat
str2num
strcat
strcmp, strcmpi
stream2
stream3
streamline
streamparticles
streamribbon
streamslice
streamtube
strfind

strings
strjust
strmatch
strncmp, strncmpi
strread
strrep
strtok
strtrim
struct
struct2cell
structfun
strvcat
sub2ind
subplot
subsasgn
subsindex
subspace
subsref
substruct
subvolume
sum
sum (timeseries)
superclasses
superiorto
support
surf, surfc
surf2patch
surface
Surface Properties
Surfaceplot Properties
surfl
surfnorm
svd
svds
swapbytes
switch
symamd

symbfact
symmlq
symrcm
symvar
synchronize
syntax
system
tan
tand
tanh
tar
tempdir
tempname
tetramesh
texlabel
text
Text Properties
textread
textscan
textwrap
tfqmr
throw (MException)
throwAsCaller (MException)
tic, toc
Tiff
timer
timerfind
timerfindall
timeseries
title
todatenum
toeplitz
toolboxdir
trace
transpose (timeseries)
trapz
treelayout

treeplot
tril
trimesh
triplequad
triplot
TriRep
TriRep
TriScatteredInterp
TriScatteredInterp
trisurf
triu
true
try
tscollection
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

uipushtool
Uipushtool Properties
uiputfile
uiresume
uisave
uisetcolor
uisetfont
uisetpref
uistack
uitable
Uitable Properties
uitoggletool
Uitoggletool Properties
uitoolbar
Uitoolbar Properties
uiwait
undocheckout
unicode2native
union
unique
unix
unloadlibrary
unmesh
unmkpp
unregisterallevents
unregisterevent
untar
unwrap
unzip
upper
urlread
urlwrite
usejava
userpath
validateattributes
validatestring
values (Map)

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

which
while
whitebg
who, whos
wilkinson
winopen
winqueryreg
wk1finfo
wk1read
wk1write
workspace
write
writeDirectory
writeEncodedStrip
writeTile
xlabel, ylabel, zlabel
xlim, ylim, zlim
xlsfinfo
xlsread
xlswrite
xmlread
xmlwrite
xor
xslt
zeros
zip
zoom

TriRep.faceNormals

Purpose

Unit normals to specified triangles

Syntax

`FN = faceNormals(TR, TI)`

Description

`FN = faceNormals(TR, TI)` returns the unit normal vector to each of the specified triangles TI.

Note This query is only applicable to triangular surface meshes.

Inputs

`TR`	Triangulation representation.
`TI`	Column vector of indices that index into the triangulation matrix `TR.Triangulation`.

Outputs

`FN`	m-by-3 matrix. `m = length(TI)`, the number of triangles to be queried. Each row `FN(i,:)` represents the unit normal vector to triangle `TI(i)`. If TI is not specified the unit normal information for the entire triangulation is returned, where the normal associated with triangle i is the `i`'th row of `FN`.

Examples

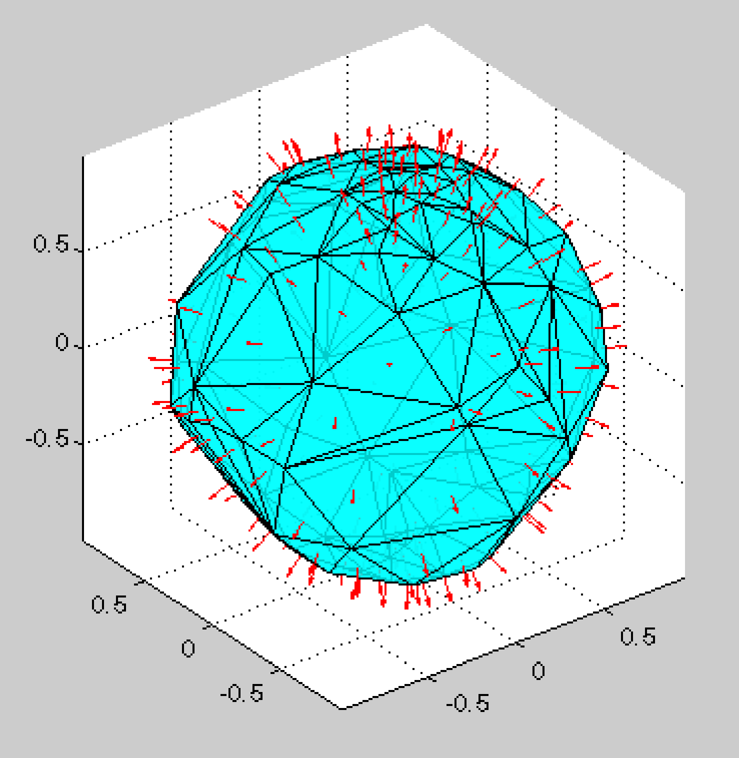
Triangulate a sample of random points on the surface of a sphere and use the `TriRep` to compute the normal to each triangle:

```
numpts = 100;
thetha = rand(numpts,1)*2*pi;
phi = rand(numpts,1)*pi;
x = cos(thetha).*sin(phi);
y = sin(thetha).*sin(phi);
z = cos(phi);
dt = DelaunayTri(x,y,z);
```

```
[tri Xb] = freeBoundary(dt);
tr = TriRep(tri, Xb);
P = incenters(tr);
fn = faceNormals(tr);
trisurf(tri,Xb(:,1),Xb(:,2),Xb(:,3), ...
     'FaceColor', 'cyan', 'faceAlpha', 0.8);
axis equal;
hold on;
```

Display the result using a quiver plot:

```
quiver3(P(:,1),P(:,2),P(:,3), ...
     fn(:,1),fn(:,2),fn(:,3),0.5, 'color','r');
hold off;
```



See Also

`TriRep.freeBoundary`
`DelaunayTri`

Purpose

Prime factors

Syntax

`f = factor(n)`

Description

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

Examples

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

See Also

`isprime`, `primes`

factorial

Purpose Factorial function

Syntax `factorial(N)`

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

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

See Also `prod`

Purpose Logical 0 (false)

Syntax

```
false
false(n)
false(m, n)
false(m, n, p, ...)
false(size(A))
```

Description `false` is shorthand for `logical(0)`.

`false(n)` is an `n`-by-`n` matrix of logical zeros.

`false(m, n)` or `false([m, n])` is an `m`-by-`n` matrix of logical zeros.

`false(m, n, p, ...)` or `false([m n p ...])` is an `m`-by-`n`-by-`p`-by-`...` array of logical zeros.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`false(size(A))` is an array of logical zeros that is the same size as array `A`.

Remarks `false(n)` is much faster and more memory efficient than `logical(zeros(n))`.

See Also `true`, `logical`

fclose

Purpose

Close one or all open files

Syntax

```
fclose(fileID)
fclose('all')
status = fclose(...)
```

Description

`fclose(fileID)` closes an open file. `fileID` is an integer file identifier obtained from `fopen`.

`fclose('all')` closes all open files.

`status = fclose(...)` returns a `status` of `0` when the close operation is successful. Otherwise, it returns `-1`.

See Also

`ferror` | `fopen`

Purpose

Disconnect serial port object from device

Syntax

`fclose(obj)`

Description

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

Remarks

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

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

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

```
help serial/fclose
```

Example

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

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

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

See Also

Functions

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

Properties

`RecordStatus`, `Status`

Purpose

Plot velocity vectors

GUI Alternatives

Use the Plot Selector 

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

Syntax

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

Description

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

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

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

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

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

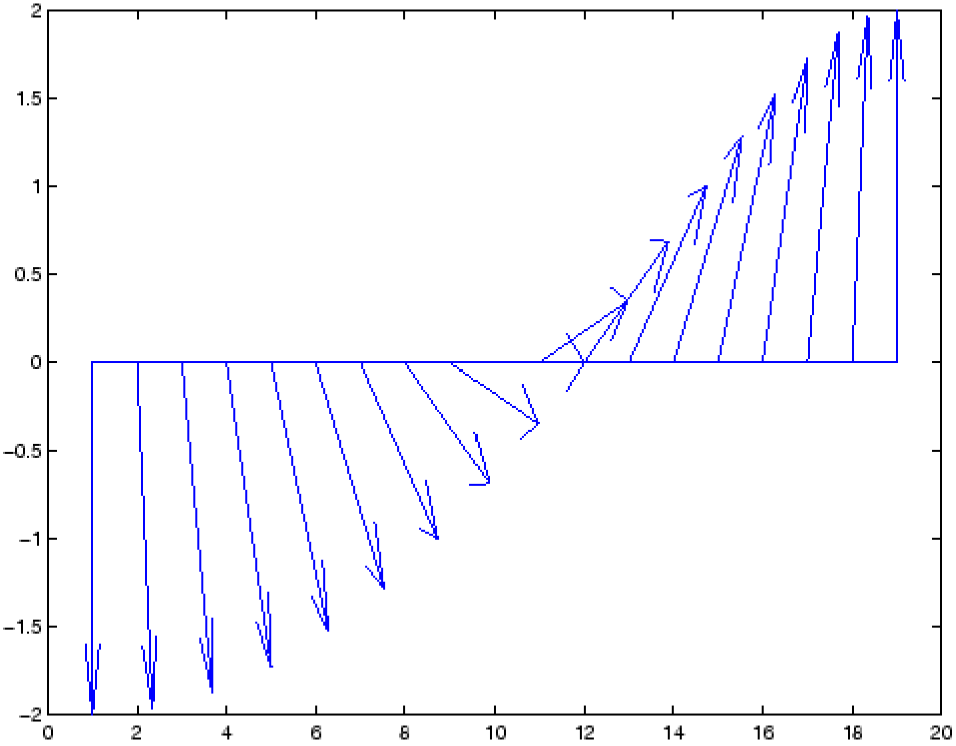
`h = feather(...)` returns the handles to line objects in `h`.

Examples

Create a feather plot showing the direction of `theta`.

```
theta = (-90:10:90)*pi/180;
```

```
r = 2*ones(size(theta));
[u,v] = pol2cart(theta,r);
feather(u,v);
```



See Also

`compass`, `LineSpec`, `rose`

"Direction and Velocity Plots" on page 1-94 for related functions

TriRep.featureEdges

Purpose

Sharp edges of surface triangulation

Syntax

`FE = featureEdges(TR, filterangle)`

Description

`FE = featureEdges(TR, filterangle)` returns an edge matrix `FE`. This method is typically used to extract the sharp edges in the surface mesh for the purpose of display. Edges that are shared by only one triangle and edges that are shared by more than two triangles are considered to be feature edges by default.

Note This query is only applicable to triangular surface meshes.

Inputs

`TR`	Triangulation representation.
`filterangle`	The threshold angle in radians. Must be in the range $(0,\pi)$. `featureEdges` will return adjacent triangles that have a dihedral angle that deviates from π by an angle greater than `filterangle`.

Outputs

`FE`	Edges of the triangulation. `FE` is of size `m`-by-2 where `m` is the number of computed feature edges in the mesh. The vertices of the edges index into the array of points representing the vertex coordinates, `TR.X`.

Examples

Create a surface triangulation:

```
x = [0 0 0 0 0 3 3 3 3 3 3 6 6 6 6 6 9 9 9 9 9]';
y = [0 2 4 6 8 0 1 3 5 7 8 0 2 4 6 8 0 1 3 5 7 8]';
dt = DelaunayTri(x,y);
tri = dt(:,:);
```

Elevate the 2-D mesh to create a surface:

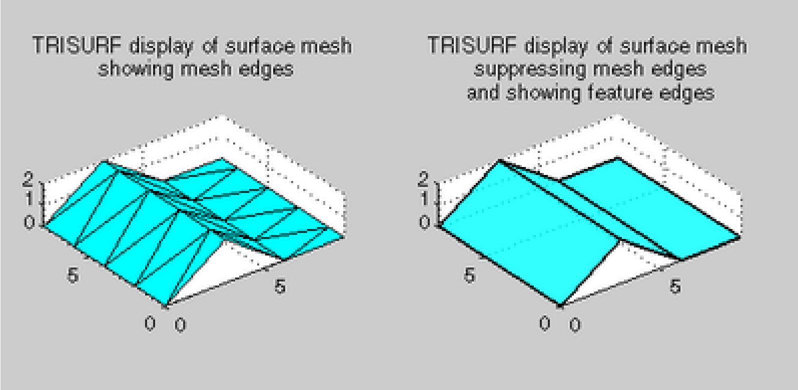
```
z = [0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0]';
subplot(1,2,1);
trisurf(tri,x,y,z, 'FaceColor', 'cyan');
axis equal;
title(sprintf('TRISURF display of surface mesh\n ...
     showing mesh edges\n'));
```

Compute the feature edges using a filter angle of `pi/6`:

```
tr = TriRep(tri, x,y,z);
fe = featureEdges(tr,pi/6)';
subplot(1,2,2);
trisurf(tr, 'FaceColor', 'cyan', 'EdgeColor','none', ...
     'FaceAlpha', 0.8); axis equal;
```

Add the feature edges to the plot:

```
hold on;
plot3(x(fe), y(fe), z(fe), 'k', 'LineWidth',1.5);
hold off;
title(sprintf('TRISURF display of surface mesh\n...
     suppressing mesh edges\n...
     and showing feature edges'));
```



See Also

`edges`
`DelaunayTri`

feof

Purpose

Test for end-of-file

Syntax

`status = feof(fileID)`

Description

`status = feof(fileID)` returns `1` if a previous operation set the end-of-file indicator for the specified file. Otherwise, `feof` returns `0`. *`fileID`* is an integer file identifier obtained from `fopen`.

Opening an empty file does *not* set the end-of-file indicator. Read operations, and the `fseek` and `frewind` functions, move the file position indicator.

Examples

Read `bench.dat`, which contains MATLAB benchmark data, one character at a time:

```
fid = fopen('bench.dat');

k = 0;
while ~feof(fid)
    curr = fscanf(fid,'%c',1);
    if ~isempty(curr)
       k = k+1;
       benchstr(k) = curr;
    end
end

fclose(fid);
```

See Also

`fclose` | `ferror` | `fopen` | `frewind` | `fseek` | `ftell`

How To

-

Purpose

Information about file I/O errors

Syntax

```
message = ferror(fileID)
[message, errnum] = ferror(fileID)
[...] = ferror(fileID, 'clear')
```

Description

`message = ferror(fileID)` returns the error message for the most recent file I/O operation on the specified file. If the operation was successful, `message` is an empty string. `fileID` is an integer file identifier obtained from `fopen`, or an identifier reserved for standard input (`0`), standard output (`1`), or standard error (`2`).

`[message, errnum] = ferror(fileID)` returns the error number. If the most recent file I/O operation was successful, `errnum` is `0`. Negative error numbers correspond to MATLAB error messages. Positive error numbers correspond to C library error messages for your system.

`[...]  = ferror(fileID, 'clear')` clears the error indicator for the specified file.

See Also

`fclose` | `fopen`

feval

Purpose Evaluate function

Syntax

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

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

Note It is not necessary to use `feval` to call a function by means of a function handle. This is explained in in the MATLAB Programming Fundamentals documentation.

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

Remarks The following two statements are equivalent.

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

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

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

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

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

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

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

Feval (COM)

Purpose Evaluate MATLAB function in Automation server

Syntax

MATLAB Client

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

IDL Method Signature

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

Microsoft Visual Basic Client

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

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

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

You can specify as many as 32 input arguments to be passed to the function. These arguments follow `numout` in the `Feval` argument list. The following table shows ways to pass an argument.

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

Passing Mechanism	Description

Pass a client variable	To pass an argument assigned to a variable in the client, specify the variable name alone:
	`x = -pi:0.01:pi;`
	`a = h.Feval('sin', 1, x);`

Reference a server variable	To reference a variable defined in the server, specify the variable name followed by an equals (=) sign:
	`h.PutWorkspaceData('x', 'base', -pi:0.01:pi);`
	`a = h.Feval('sin', 1, 'x=');`
	MATLAB does not reassign the server variable.

Remarks

To display the output from `Feval` in the client window, assign a return value.

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

COM functions are available on Microsoft Windows systems only.

Examples

Passing Arguments — MATLAB Client

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

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

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

  MATLAB displays:

  ```
  a =
  ```

```
        'hello world'
```

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

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

  MATLAB displays:

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

- In this example, the variable `srvstr` is defined in the server, not the client. Putting an equals sign after a variable name (for example, `srvstr=`) indicates it is a server variable, and the variable is not defined in the client:

  ```
  % Define the variable srvstr on the server.
  h.PutCharArray('srvstr', 'base', ' world')

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

  MATLAB displays:

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

Visual Basic .NET Client

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

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

  ```
  Dim Matlab As Object
  Dim out As Object
  out = Nothing
  ```

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

- Define `clistr` locally and pass this variable:

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

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

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

Feval Return Values — MATLAB Client. `Feval` returns data from the evaluated function in a cell array. The cell array has one row for every return value. You control the number of return values using the `numout` argument.

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

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

MATLAB displays:

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

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

```
a{:}
```

MATLAB displays:

```
ans =
d:\work

ans =
ConsoleApp

ans =
.cpp
```

Feval Return Values — Visual Basic .NET Client

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

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

See Also

`Execute`, `PutFullMatrix`, `GetFullMatrix`, `PutCharArray`, `GetCharArray`

Purpose

Discrete Fourier transform

Syntax

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

Definition

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

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

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

where

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

is an Nth root of unity.

Description

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

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

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

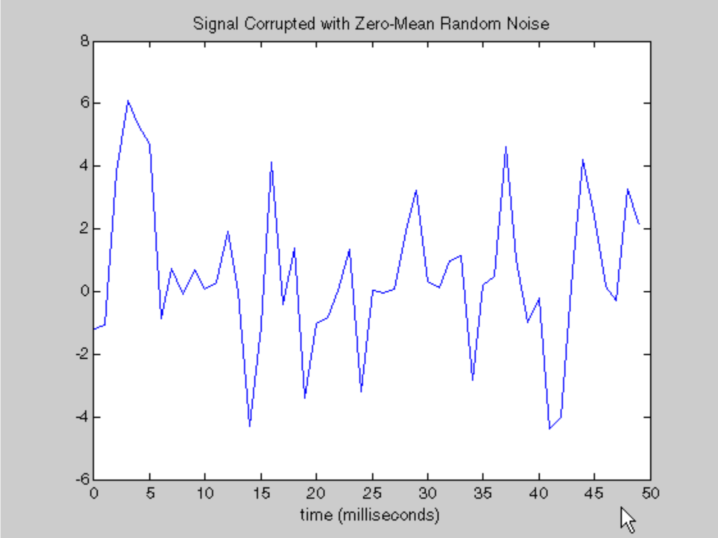
`Y = fft(X,n)` returns the `n`-point DFT. If the length of `X` is less than `n`, `X` is padded with trailing zeros to length `n`. If the length of `X` is greater than `n`, the sequence `X` is truncated. When `X` is a matrix, the length of the columns are adjusted in the same manner.

`Y = fft(X,[],dim)` and `Y = fft(X,n,dim)` applies the FFT operation across the dimension `dim`.

Examples

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

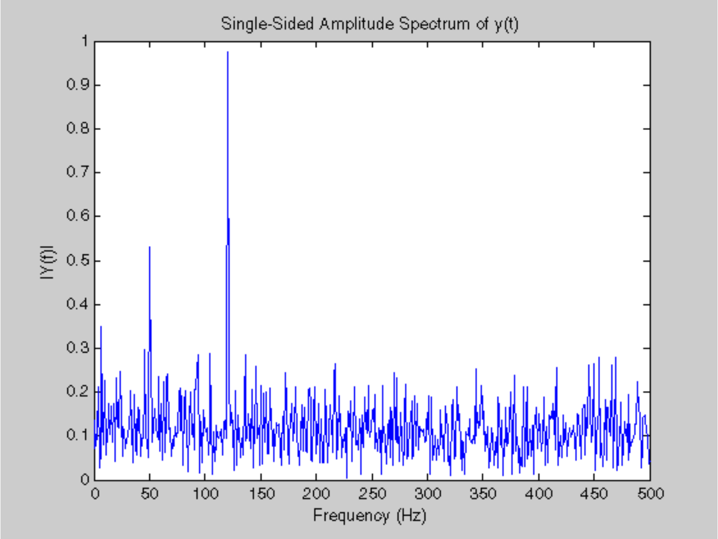
```
Fs = 1000;                    % Sampling frequency
T = 1/Fs;                     % Sample time
L = 1000;                     % Length of signal
t = (0:L-1)*T;                % Time vector
% Sum of a 50 Hz sinusoid and a 120 Hz sinusoid
x = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
y = x + 2*randn(size(t));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



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

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

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



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

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

Algorithm

The FFT functions (`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, `ifftn`) are based on a library called FFTW [3],[4]. To compute an N-point DFT when N is composite (that is, when $N = N_1N_2$), the FFTW library decomposes the problem using the Cooley-Tukey algorithm [1], which first computes N_1 transforms of size N_2, and then computes N_2 transforms of size N_1. The decomposition is applied recursively to both the N_1- and N_2-point DFTs until the problem can be solved using one of several machine-generated fixed-size "codelets." The codelets in turn use several algorithms in combination, including a variation of Cooley-Tukey [5], a prime factor algorithm [6], and a split-radix algorithm [2]. The particular factorization of N is chosen heuristically.

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

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

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

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

Data Type Support

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

See Also

`fft2`, `fftn`, `fftw`, `fftshift`, `ifft`

`dftmtx`, `filter`, and `freqz` in the Signal Processing Toolbox

References

[1] Cooley, J. W. and J. W. Tukey, “An Algorithm for the Machine Computation of the Complex Fourier Series,”*Mathematics of Computation*, Vol. 19, April 1965, pp. 297-301.

[2] Duhamel, P. and M. Vetterli, “Fast Fourier Transforms: A Tutorial Review and a State of the Art,” *Signal Processing*, Vol. 19, April 1990, pp. 259-299.

[3] FFTW (`http://www.fftw.org`)

[4] Frigo, M. and S. G. Johnson, “FFTW: An Adaptive Software Architecture for the FFT,”*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

[5] Oppenheim, A. V. and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 611.

[6] Oppenheim, A. V. and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 619.

[7] Rader, C. M., “Discrete Fourier Transforms when the Number of Data Samples Is Prime,” *Proceedings of the IEEE*, Vol. 56, June 1968, pp. 1107-1108.

Purpose

2-D discrete Fourier transform

Syntax

```
Y = fft2(X)
Y = fft2(X,m,n)
```

Description

`Y = fft2(X)` returns the two-dimensional discrete Fourier transform (DFT) of `X`, computed with a fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`Y = fft2(X,m,n)` truncates `X`, or pads `X` with zeros to create an `m`-by-`n` array before doing the transform. The result is `m`-by-`n`.

Algorithm

`fft2(X)` can be simply computed as

```
fft(fft(X).').'
```

This computes the one-dimensional DFT of each column `X`, then of each row of the result. The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `fft2` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`fft2` supports inputs of data types `double` and `single`. If you call `fft2` with the syntax `y = fft2(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`fft`, `fftn`, `fftw`, `fftshift`, `ifft2`

Purpose

N-D discrete Fourier transform

Syntax

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

Description

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

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

Algorithm

`fftn(X)` is equivalent to

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

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

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

Data Type Support

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

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

Purpose

Shift zero-frequency component to center of spectrum

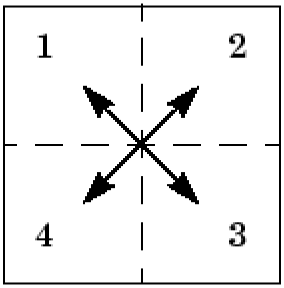
Syntax

```
Y = fftshift(X)
Y = fftshift(X,dim)
```

Description

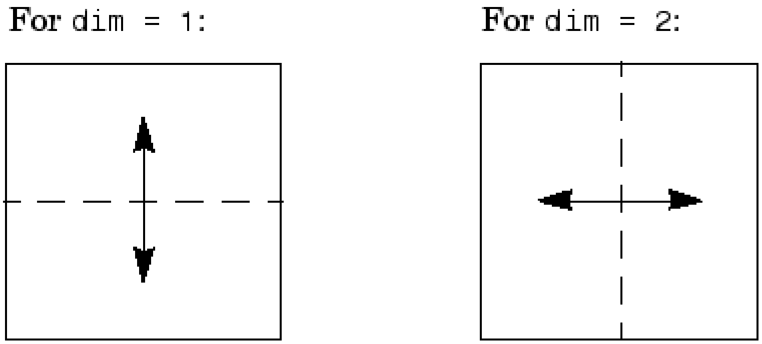
`Y = fftshift(X)` rearranges the outputs of `fft`, `fft2`, and `fftn` by moving the zero-frequency component to the center of the array. It is useful for visualizing a Fourier transform with the zero-frequency component in the middle of the spectrum.

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



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

`Y = fftshift(X,dim)` applies the `fftshift` operation along the dimension `dim`.



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

Examples

For any matrix `X`

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

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

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

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

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

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

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

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

ans =

     0

ans =

     1
```

See Also `circshift`, `fft`, `fft2`, `fftn`, `ifftshift`

Purpose

Interface to FFTW library run-time algorithm tuning control

Syntax

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

Description

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

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

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

When you call `fftw('planner', method)`, the next time you call one of the FFT functions, such as `fft`, the tuning algorithm uses the specified method to optimize the FFT computation. Because the tuning involves trying different algorithms, the first time you call an FFT function, it might run more slowly than if you did not call `fftw`. However, subsequent calls to any of the FFT functions, for a problem of the same size, often run more quickly than they would without using `fftw`.

Note The FFT functions only use the optimal FFT algorithm during the current MATLAB session. "Reusing Optimal FFT Algorithms" on page 2-1221 explains how to reuse the optimal algorithm in a future MATLAB session.

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

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

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

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

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

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

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

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

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

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

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

Example

Comparison of Speed for Different Planner Methods

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

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

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

If you execute the commands

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

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

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

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

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

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

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

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

Reusing Optimal FFT Algorithms

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

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

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

```
save str
```

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

```
load str
```

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

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

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

Purpose

Read line from file, removing newline characters

Syntax

`tline` `= fgetl(`*`fileID`*`)`

Description

`tline` `= fgetl(`*`fileID`*`)` returns the next line of the specified file, removing the newline characters. *`fileID`* is an integer file identifier obtained from `fopen`. *`tline`* is a text string unless the line contains only the end-of-file marker. In this case, *`tline`* is the numeric value `-1`.

`fgetl` reads characters using the encoding scheme associated with the file. To specify the encoding scheme, use `fopen`.

Examples

Read and display the file `fgetl.m` one line at a time:

```
fid = fopen('fgetl.m');

tline = fgetl(fid);
while ischar(tline)
    disp(tline)
    tline = fgetl(fid);
end

fclose(fid);
```

Compare these results to the `fgets` example, which replaces the calls to `fgetl` with `fgets`.

See Also

`fclose` | `ferror` | `fgets` | `fopen` | `fprintf` | `fread` | `fscanf` | `fwrite`

How To

-

fgetl (serial)

Purpose Read line of text from device and discard terminator

Syntax

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

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

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

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

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

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

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

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

```
help serial/fgetl
```

Rules for Completing a Read Operation with fgetl

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

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

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

Example

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

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

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

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

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

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

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

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

See Also

Functions

`fgets`, `fopen`

Properties

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

Purpose

Read line from file, keeping newline characters

Syntax

`tline` `= fgets(`*`fileID`*`)`
`tline` `= fgets(`*`fileID`*`,` *`nchar`*`)`

Description

`tline` `= fgets(`*`fileID`*`)` reads the next line of the specified file, including the newline characters. *`fileID`* is an integer file identifier obtained from `fopen`. *`tline`* is a text string unless the line contains only the end-of-file marker. In this case, *`tline`* is the numeric value `-1`. `fgets` reads characters using the encoding scheme associated with the file. To specify the encoding scheme, use `fopen`.

`tline` `= fgets(`*`fileID`*`,` *`nchar`*`)` returns at most *`nchar`* characters of the next line. *`tline`* does not include any characters after the newline characters or the end-of-file marker.

Examples

Read and display the file `fgets.m`. Because `fgets` keeps newline characters and `disp` adds a newline character, this code displays the file with double-spacing:

```
fid = fopen('fgets.m');

tline = fgets(fid);
while ischar(tline)
    disp(tline)
    tline = fgets(fid);
end

fclose(fid);
```

Compare these results to the `fgetl` example, which replaces the calls to `fgets` with `fgetl`.

See Also

`fclose` | `ferror` | `fgetl` | `fopen` | `fprintf` | `fread` | `fscanf` | `fwrite`

How To

-

fgets (serial)

Purpose

Read line of text from device and include terminator

Syntax

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

Description

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

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

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

Remarks

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

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

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

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

```
help serial/fgets
```

Rules for Completing a Read Operation with fgets

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

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

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

Example

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

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

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

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

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

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

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

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

See Also

Functions

`fgetl`, `fopen`

Properties

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

Purpose

Field names of structure, or public fields of object

Syntax

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

Description

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

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

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

Examples

Given the structure

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

the command `n = fieldnames(mystr)` yields

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

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

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

```
fieldnames(i)
```

MATLAB displays:

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

See Also `setfield`, `getfield`, `isfield`, `orderfields`, `rmfield`, dynamic field names

Purpose

Create figure graphics object

Syntax

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

Description

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

`figure` creates a new figure object using default property values. This automatically becomes the current figure and raises it above all other figures on the screen until a new figure is either created or called.

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

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

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

Remarks

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

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

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

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

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

Making a Figure Current

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

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

  ```
  figure(h)
  ```

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

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

Examples

Specifying Figure Size and Screen Location

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

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

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

Specifying the Figure Window Title

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

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

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

Setting Default Properties

You can set default figure properties only on the level.

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

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

See Also

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

"Object Creation" on page 1-99 for related functions

`Figure Properties` descriptions of all figure properties

See in the MATLAB Graphics User Guide for more information on figures.

Figure Properties

Purpose

Define figure 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 Handle Graphics properties.

To change the default values of properties, see in the Handle Graphics Objects documentation.

Figure Property Descriptions

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

`Alphamap`

m-by-1 matrix of alpha values

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

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

`BeingDeleted`

`on | {off}` read only

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

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

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

`BusyAction`

`cancel | {queue}`

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

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

`ButtonDownFcn`

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

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

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

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

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

Using the ButtonDownFcn

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

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

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

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

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

`Children`

vector of handles

Children of the figure. A vector containing the handles of all axes, user-interface objects displayed within the figure. You can change the order of the handles and thereby change the stacking of the objects on the display.

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

`Clipping`

`{on} | off`

This property has no effect on figures.

`CloseRequestFcn`

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

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

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

The basic mechanism is

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

```
if isempty(gcbf)
   if length(dbstack) == 1
      warning('MATLAB:closereq', ...
         'Calling closereq from the command line ...
    is now obsolete, use close instead');
   end
   close force
else
   delete(gcbf);
end
```

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

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

Redefining the CloseRequestFcn

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

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

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

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

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

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

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

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

Now create a figure using the `CloseRequestFcn`:

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

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

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

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

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

`Color`

`ColorSpec`

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

`Colormap`

m-by-3 matrix of RGB values

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

Number of Colors Allowed

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

Objects That Use Colormaps

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

`CreateFcn`

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

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

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

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

```
function fig_create(src,evnt)
set(src,'Color',[.2 .1 .5],...
   'IntegerHandle','off',...
   'MenuBar','none',...
   'ToolBar','none')
end
```

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

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

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

`CurrentAxes`

handle of current axes

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

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

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

`CurrentCharacter`
single character

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

`CurrentObject`
object handle

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

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

Hidden Handle Objects

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

Mouse Over

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

`CurrentPoint`

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

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

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

CurrentPoint and Cursor Motion

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

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

`CurrentPoint` is updated) anywhere on the screen until the mouse button is released.

- If there is a callback routine defined for the `WindowButtonUpFcn`, MATLAB updates the `CurrentPoint` just before executing the callback. Note that the `WindowButtonUpFcn` executes only while the pointer is within the figure window unless the mouse button is pressed down initially within the window. In this case, releasing the button anywhere on the screen triggers callback execution, which is preceded by an update of the `CurrentPoint`.

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

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

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

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

`DeleteFcn`

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

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

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

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

See also the figure `CloseRequestFcn` property

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

`DockControls`

`{on} | off`

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

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

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

`DoubleBuffer`

`{on} | off`

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

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

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

`FileName`

String

GUI FIG-filename. GUIDE stores the name of the FIG-file used to save the GUI layout in this property. In non-GUIDE GUIs, by default `FileName` is empty. You can `set` the `FileName` property in non-GUIDE GUIs as well, and `get` it to verify what GUI is running or whether it has been previously saved.

`FixedColors`

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

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

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

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

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

Note The FixedColors property is being deprecated and will be removed in a future release

---

`HandleVisibility`

`{on} | callback | off`

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

Handles are always visible when `HandleVisibility` is `on`.

Callback Visibility

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

Visibility Off

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

Visibility and Handles Returned by Other Functions

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

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

Making All Handles Visible

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

Validity of Hidden Handles

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

`HitTest`

`{on} | off`

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

`IntegerHandle`

`{on} | off`

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

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

`Interruptible`

`{on} | off`

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

How Callbacks Are Interrupted

MATLAB checks for queued events that can interrupt a callback function only when it encounters a call to `drawnow`, `figure`, `getframe`, or `pause` in the executing callback function. When executing one of these functions, MATLAB processes all pending events, including executing all waiting callback functions. The interrupted callback then resumes execution.

What Property Callbacks Are Interruptible

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

See the `BusyAction` property for related information.

`InvertHardcopy`

`{on} | off`

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

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

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

See `print` for more information on printing MATLAB figures.

`KeyPressFcn`

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

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

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

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

KeyPressFcn Event Structure

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

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

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

Using the KeyPressFcn

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

```
function figure_keypress
   figure('KeyPressFcn',@printfig);
   function printfig(src,evnt)
      if evnt.Character == 'e'
         print ('-deps',['-f' num2str(src)])
      elseif length(evnt.Modifier) == 1 & strcmp(evnt.Modifier{:},'control') & ...
```

```
            evnt.Key == 't'
                 print ('-dtiff','-r200',['-f' num2str(src)])
           end
        end
```

`KeyReleaseFcn`

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

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

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

KeyReleaseFcn Event Structure

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

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

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

Properties Affected by the KeyReleaseFcn

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

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

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

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

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

Modifier Keys

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

Explore the Results

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

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

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

```
function key_releaseFcn
   figure('KeyReleaseFcn',@cb)
      function cb(src,evnt)
      if ~isempty(evnt.Modifier)
         for ii = 1:length(evnt.Modifier)
            out = sprintf('Character: %c\nModifier: %s\nKey: %s\n',...
     evnt.Character,evnt.Modifier{ii},evnt.Key);
            disp(out)
         end
      else
         out = sprintf('Character: %c\nModifier: %s\nKey: %s\n',...
    evnt.Character,'No modifier key',evnt.Key);
         disp(out)
      end
   end
end
```

`MenuBar`

`none | {figure}`

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

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

`MinColormap`

scalar (default = 64)

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

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

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

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

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

Note The MinColormap property is being deprecated and will be removed in a future release

`Name`

`string`

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

`NextPlot`

`new | {add} | replace | replacechildren`

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

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

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

`NumberTitle`

`{on} | off` (GUIDE default `off`)

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

`OuterPosition`
four-element vector

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

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

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

Position of Docked Figures

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

Moving and Resizing Figures

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

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

`PaperOrientation`

`{portrait} | landscape`

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

`PaperPosition`

four-element rect vector

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

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

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

`PaperPositionMode`

`auto | {manual}`

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

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

`PaperSize`
`[width height]`

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

`PaperType`
Select a value from the following table.

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

Property Value	**Size (Width x Height)**
`usletter (default)`	8.5-by-11 inches
`uslegal`	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

Property Value	Size (Width x Height)
`arch-B`	12-by-18 inches
`arch-C`	18-by-24 inches
`arch-D`	24-by-36 inches
`arch-E`	36-by-48 inches
`A`	8.5-by-11 inches
`B`	11-by-17 inches
`C`	17-by-22 inches
`D`	22-by-34 inches
`E`	34-by-43 inches

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

`PaperUnits`

`normalized | {inches} | centimeters | points`

Hardcopy measurement units. This property specifies the units used to define the `PaperPosition` and `PaperSize` properties. MATLAB measures all units from the lower-left corner of the page. `normalized` units map the lower-left corner of the page to (0, 0) and the upper-right corner to (1.0, 1.0). `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch).

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

`Parent`

`handle`

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

`Pointer`

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

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

`PointerShapeCData`

16-by-16 matrix

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

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

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

`PointerShapeHotSpot`
two-element vector

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

`Position`
four-element vector

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

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

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

Position of Docked Figures

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

Moving and Resizing Figures

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

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

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

`Renderer`

`painters | zbuffer | OpenGL`

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

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

Hardware vs. Software OpenGL Implementations

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

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

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

OpenGL Availability

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

The following software versions are available:

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

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

Selecting Hardware-Accelerated or Software OpenGL

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

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

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

```
opengl software
```

This command forces MATLAB to use software OpenGL. Software OpenGL is useful if your hardware-accelerated version of OpenGL does not function correctly and you want to use image, patch, or surface transparency, which requires the OpenGL renderer. To reenable hardware OpenGL, use the command:

```
opengl hardware
```

on Windows systems or restart MATLAB on UNIX systems.

By default, MATLAB uses hardware-accelerated OpenGL.

See the `opengl` reference page for additional information

Determining What Version You Are Using

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

```
opengl info
```

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

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

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

OpenGL vs. Other MATLAB Renderers

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

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

If You Are Having Problems

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

`RendererMode`

`{auto} | manual`

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

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

For printing, MATLAB switches to `zbuffer` at a greater scene complexity than for screen rendering because printing from a z-buffered figure can be considerably slower than one using the `painters` rendering method, and can result in large PostScript® files. However, the output does always match what is on the screen. The same holds true for OpenGL: the output is the same as that produced by the `zbuffer` renderer — a bitmap with a resolution determined by the `print` command's `-r` option.

Criteria for Autoselection of the OpenGL Renderer

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

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

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

  Or

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

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

`Resize`

`{on} | off`

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

`ResizeFcn`

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

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

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

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

```
u = findobj('Tag','StatusBar');
fig = gcbo;
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 an example of how to implement a resize function for a GUI.

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

`Selected`

`on | off`

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

`SelectionHighlight`

`{on} | off`

Figures do not indicate selection.

`SelectionType`

`{normal} | extend | alt | open`

Mouse selection type. MATLAB maintains this property to provide information about the last mouse button press that occurred within the figure window. This information indicates the type of selection made. Selection types are actions that MATLAB generally associates with particular responses from the user interface software (e.g., single-clicking a graphics object places it in move or resize mode; double-clicking a file name opens it, etc.).

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

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

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

`Tag`

`string`

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

For example, suppose you want to direct all graphics output from an M-file to a particular figure, regardless of user actions that may have changed the current figure. To do this, identify the figure with a `Tag`.

```
figure('Tag','Plotting Figure')
```

Then make that figure the current figure before drawing by searching for the `Tag` with `findobj`.

```
figure(findobj('Tag','Plotting Figure'))
```

`Toolbar`

`none | {auto} | figure`

Control display of figure toolbar. The `Toolbar` property enables you to control whether MATLAB displays the default figure toolbar on figures. There are three possible values:

- `none` — Do not display the figure toolbar.
- `auto` — Display the figure toolbar, but remove it if a uicontrol is added to the figure.
- `figure` — Display the figure toolbar.

Note that this property affects only the figure toolbar; it does not affect other toolbars (e.g., the Camera Toolbar or Plot Edit Toolbar). Selecting **Figure Toolbar** from the figure **View** menu sets this property to `figure`.

If you start MATLAB with the `nojvm` option, figures do not display the toolbar because most tools require Java figures. This option is obsolete and no longer supported in MATLAB

`Type`

`string (read only)`

Object class. This property identifies the kind of graphics object. For figures, `Type` is always the string `'figure'`.

`UIContextMenu`

handle of a uicontextmenu object

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

`Units`

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

Units of measurement. This property specifies the units MATLAB uses to interpret size and location data. All units are measured from the lower-left corner of the window.

- `normalized` units map the lower-left corner of the figure window to (0,0) and the upper-right corner to (1.0,1.0).
- `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch).
- The size of a `pixel` depends on screen resolution.
- `characters` units are defined by characters from the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

This property affects the `CurrentPoint` and `Position` properties. If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

When specifying the units as property/value pairs during object creation, you must set the `Units` property before specifying the properties that you want to use these units.

`UserData`

matrix

User-specified data. You can specify `UserData` as any matrix you want to associate with the figure object. The object does not use this data, but you can access it using the `set` and `get` commands.

`Visible`

`{on} | off`

Object visibility. The `Visible` property determines whether an object is displayed on the screen. If the `Visible` property of a figure is `off`, the entire figure window is invisible.

A Note About Using the Window Button Properties

Your window button callback functions might need to update the display by calling `drawnow` or `pause`, which causes MATLAB to process all events in the queue. Processing the event queue can cause your window button callback functions to be reentered. For example, a `drawnow` in the `WindowButtonDownFcn` might result in the `WindowButtonDownFcn` being called again before the first call has finished. You should design your code to handle reentrancy and you should not depend on global variables that might change state during reentrance.

You can use the `Interruptible` and `BusyAction` figure properties to control how events interact.

`WindowButtonDownFcn`

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

Button press callback function. Use this property to define a callback that MATLAB executes whenever you press a mouse button while the pointer is in the figure window. See the `WindowButtonMotionFcn` property for an example.

Note When using a two- or three-button mouse on Macintosh systems, right-button and middle-button presses are not always reported. This happens *only* when a new figure window appears under the mouse cursor and the mouse is clicked without first moving it. In this circumstance, for the `WindowButtonDownFcn` to work, the user needs to do one of the following:

- Move the mouse after the figure is created, then click any mouse button
- Press **Shift** or **Ctrl** while clicking the left mouse button to perform the Extend and Alternate selection types

Pressing the left mouse button (or single mouse button) works without having to take either of the above actions.

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

`WindowButtonMotionFcn`

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

Mouse motion callback function. Use this property to define a callback that MATLAB executes whenever you move the pointer within the figure window. Define the `WindowButtonMotionFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

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

Example Using All Window Button Properties

Click to view in editor — This example enables you to use mouse motion to draw lines. It uses all three window button functions.

Click to run example — Click the left mouse button in the axes and move the cursor, left-click to define the line end point, right-click to end drawing mode.

Note On some computer systems, the `WindowButtonMotionFcn` is executed when a figure is created even though there has been no mouse motion within the figure.

`WindowButtonUpFcn`

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

Button release callback function. Use this property to define a callback that MATLAB executes whenever you release a mouse button. Define the `WindowButtonUpFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

The button up event is associated with the figure window in which the preceding button down event occurred. Therefore, the pointer need not be in the figure window when you release the button to generate the button up event.

If the callback routines defined by `WindowButtonDownFcn` or `WindowButtonMotionFcn` contain `drawnow` commands or call other functions that contain `drawnow` commands and the `Interruptible` property is set to `off`, the `WindowButtonUpFcn` might not be called. You can prevent this problem by setting `Interruptible` to `on`.

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

`WindowKeyPressFcn`

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

Key press callback function for the figure window. Use this property to define a callback that MATLAB executes whenever a key press occurs. This is a callback function invoked by a key press that occurs while either the figure window or any of its children has focus. Define the `WindowKeyPressFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

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

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

WindowKeyPressFcn Event Structure

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

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

`WindowKeyReleaseFcn`

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

Key release callback function for the figure window. Use this property to define a callback that MATLAB executes whenever a key release occurs. This is a callback function invoked by a key release that occurs while the figure window or any of its children has focus. Define the `WindowKeyReleaseFcn` as a function handle. The function must define at least two input arguments (handle of the figure associated with key release and an event structure).

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

WindowKeyReleaseFcn Event Structure

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

Field	Contents
`Character`	The character corresponding to the key(s) released.
`Modifier`	This field is a cell array that contains the names of one or more modifier keys that the user released (i.e., **control**, **alt**, **shift**). On Macintosh computers, MATLAB can also return 'command' if the user released the **command** modifier key.
`Key`	The key released (lower case label on key).

`WindowScrollWheelFcn`

string, function handle, or cell array containing function handle and additional arguments

Respond to mouse scroll wheel. Use this property to define a callback that MATLAB executes when the mouse wheel is scrolled while the figure has focus. MATLAB executes the callback with each single mouse wheel click.

Note that it is possible for another object to capture the event from MATLAB. For example, if the figure contains Java or ActiveX control objects that are listening for mouse scroll wheel events, then these objects can consume the events and prevent the `WindowScrollWheelFcn` from executing.

There is no default callback defined for this property.

WindowScrollWheelFcn Event Structure

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

Field	Contents
`VerticalScrollCount`	A positive or negative integer that indicates the number of scroll wheel clicks. Positive values indicate clicks of the wheel scrolled in the down direction. Negative values indicate clicks of the wheel scrolled in the up direction.
`VerticalScrollAmount`	The current system setting for the number of lines that are scrolled for each click of the scroll wheel. If the mouse property setting for scrolling is set to `One screen at a time`, `VerticalScrollAmount` returns a value of 1.

Effects on Other Properties

- `CurrentObject` property — Mouse scrolling does not update this figure property.
- `CurrentPoint` property — If there is no callback defined for the `WindowScrollWheelFcn` property, then MATLAB does not update the `CurrentPoint` property as the scroll wheel is turned. However, if there is a callback defined for the `WindowScrollWheelFcn` property, then MATLAB updates the `CurrentPoint` property just before executing the callback. This enables you to determine the point at which the mouse scrolling occurred.
- `HitTest` property — The `WindowScrollWheelFcn` callback executes regardless of the setting of the figure `HitTest` property.
- `SelectionType` property — The `WindowScrollWheelFcn` callback has no effect on this property.

Values Returned by VerticalScrollCount

When a user moves the mouse scroll wheel by one click, MATLAB increments the count by +/- 1, depending on the direction of the scroll (scroll down being positive). When MATLAB calls the `WindowScrollWheelFcn` callback, the counter is reset. In most cases, this means that the absolute value of the returned value is 1. However, if the `WindowScrollWheelFcn` callback takes a long enough time to return and/or the user spins the scroll wheel very fast, then the returned value can have an absolute value greater than one.

The actual value returned by `VerticalScrollCount` is the algebraic sum of all scroll wheel clicks that occurred since last processed. This enables your callback to respond correctly to the user's action.

Example

Click to view in editor — This example creates a graph of a function and enables you to use the mouse scroll wheel to change the range over which a mathematical function is evaluated and update the graph to reflect the new limits as you turn the scroll wheel.

Click to run example — Mouse over the figure and scroll your mouse wheel.

Related Information

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

`WindowStyle`

`{normal} | modal | docked`

Normal, modal, or dockable window behavior. When `WindowStyle` is set to `modal`:

- The figure window traps all keyboard and mouse events over all MATLAB windows as long as they are visible.
- Windows belonging to applications other than MATLAB are unaffected.
- Modal figures remain stacked above all normal figures and the MATLAB Command Window.
- When multiple modal windows exist, the most recently created window keeps focus and stays above all other windows until it becomes invisible, or is returned to `WindowStyle normal` , or is deleted. At that time, focus reverts to the window that last had focus.

Use modal figures to create dialog boxes that force the user to respond without being able to interact with other windows. Typing **Ctrl+C** while the figure has focus causes all figures with

`WindowStyle modal` to revert to `WindowStyle normal`, allowing you to type at the command line.

Invisible Modal Figures

Figures with `WindowStyle modal` and `Visible off` do not behave modally until they are made visible, so it is acceptable to hide a `modal` window for later reuse instead of destroying it.

Stacking Order of Modal Figures

Creating a figure with `WindowStyle modal` stacks it on top of all existing figure windows, making them inaccessible as long as the top figure exists and remains modal. However, any new figures created after a modal figure is displayed (for example, plots created by a modal GUI) stack on top of it and are accessible; they can be modal as well.

Changing Modes

You can change the `WindowStyle` of a figure at any time, including when the figure is visible and contains children. However, on some systems this may cause the figure to flash or disappear and reappear, depending on the windowing system's implementation of normal and modal windows. For best visual results, you should set `WindowStyle` at creation time or when the figure is invisible.

Window Decorations on Modal Figures

Modal figures do not display `uimenu` children, built-in menus, or toolbars but it is not an error to create uimenus in a modal figure or to change `WindowStyle` to `modal` on a figure with uimenu children. The uimenu objects exist and their handles are retained by the figure. If you reset the figure's `WindowStyle` to `normal`, the uimenus are displayed.

Docked WindowStyle

When `WindowStyle` is set to `docked`, the figure is docked in the desktop or a document window. When you issue the following command,

```
set(figure_handle,'WindowStyle','docked')
```

MATLAB docks the figure identified by *`figure_handle`* and sets the `DockControls` property to `on`, if it was `off`.

Note that if `WindowStyle` is `docked`, you cannot set the `DockControls` property to `off`.

The value of the `WindowStyle` property is not changed by calling `reset` on a figure.

`WVisual`

identifier string (Windows only)

Specify pixel format for figure. MATLAB automatically selects a pixel format for figures based on your current display settings, the graphics hardware available on your system, and the graphical content of the figure.

Usually, MATLAB chooses the best pixel format to use in any given situation. However, in cases where graphics objects are not rendered correctly, you might be able to select a different pixel format and improve results. See "Understanding the WVisual String" for more information.

Querying Available Pixel Formats on Window Systems

You can determine what pixel formats are available on your system for use with MATLAB using the following statement:

```
set(gcf,'WVisual')
```

MATLAB returns a list of the currently available pixel formats for the current figure. For example, the following are the first three entries from a typical list:

01 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, OpenGL, GDI, Window)

02 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, OpenGL, Double Buffered, Window)

03 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, OpenGL, Double Buffered, Window)

Use the number at the beginning of the string to specify which pixel format to use. For example,

```
set(gcf,'WVisual','02')
```

specifies the second pixel format in the list above. Note that pixel formats might differ on your system.

Understanding the WVisual String

The string returned by querying the `WVisual` property provides information on the pixel format. For example:

- `RGB 16 bits(05 06 05 00)` — Indicates true color with 16-bit resolution (5 bits for red, 6 bits for green, 5 bits for blue, and 0 for alpha (transparency). MATLAB requires true color.
- `zdepth 24` — Indicates 24-bit resolution for sorting object's front to back position on the screen. Selecting pixel formats with higher (24 or 32) zdepth might solve sorting problems.
- `Hardware Accelerated` — Some graphics functions may be performed by hardware for increased speed. If there are incompatibilities between your particular graphic hardware and MATLAB, select a pixel format in which the term `Generic` appears instead of `Hardware Accelerated`.

- `Opengl` — Supports OpenGL. See the preceding “Pixel Formats and OpenGL” for more information.
- `GDI` — Supports for Windows 2-D graphics interface.
- `Double Buffered` — Support for double buffering with the OpenGL renderer. Note that the figure `DoubleBuffer` property applies only to the painters renderer.
- `Bitmap` — Support for rendering into a bitmap (as opposed to drawing in the window).
- `Window` — Support for rendering into a window.

Pixel Formats and OpenGL

If you are experiencing problems using hardware OpenGL on your system, you can try using generic OpenGL, which is implemented in software. To do this, first instruct MATLAB to use the software version of OpenGL with the following statement:

```
opengl software
```

Then allow MATLAB to select best pixel format to use.

See the `Renderer` property for more information on how MATLAB uses OpenGL.

`WVisualMode`

`auto | manual` (Windows only)

Auto or manual selection of pixel format. `WVisualMode` can take on two values — `auto` (the default) and `manual`. In `auto` mode, MATLAB selects the best pixel format to use based on your computer system and the graphical content of the figure. In `manual` mode, MATLAB does not change the visual from the one currently in use. Setting the `WVisual` property sets this property to `manual`.

`XDisplay`

display identifier (UNIX only)

Contains the display used for MATLAB. You can query this property to determine the name of the display that MATLAB is using. For example, if MATLAB is running on a system called `mycomputer`, querying `XDisplay` returns a string of the following form:

```
get(gcf,'XDisplay')
ans
mycomputer:0.0
```

Setting XDisplay on Motif

If your computer uses Motif-based figures, you can specify the display MATLAB uses for a figure by setting the value of the figure's `XDisplay` property. For example, to display the current figure on a system called `fred`, use the command

```
set(gcf,'XDisplay','fred:0.0')
```

`XVisual`

visual identifier (UNIX only)

Select visual used by MATLAB. You can select the visual used by MATLAB by setting the `XVisual` property to the desired visual ID. This can be useful if you want to test your application on an 8-bit or grayscale visual. To see what visuals are available on your system, use the UNIX `xdpyinfo` command. From MATLAB, type

```
!xdpyinfo
```

The information returned contains a line specifying the visual ID. For example:

```
visual id:    0x23
```

To use this visual with the current figure, set the `XVisual` property to the ID.

```
set(gcf,'XVisual','0x23')
```

To see which of the available visuals MATLAB can use, call `set` on the `XVisual` property:

```
set(gcf,'XVisual')
```

The following typical output shows the visual being used (in curly braces) and other possible visuals. Note that MATLAB requires a TrueColor visual.

```
{ 0x23 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff) }
  0x24 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x25 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x26 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x27 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x28 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x29 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x2a (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
```

You can also use the `glxinfo` UNIX command to see what visuals are available for use with the OpenGL renderer. From MATLAB, type

```
!glxinfo
```

After providing information about the implementation of OpenGL on your system, `glxinfo` returns a table of visuals. The partial listing below shows typical output:

```
visual  x  bf lv rg d st colorbuffer ax dp st accumbuffer  ms  cav
 id dep cl sp sz l  ci b ro   r  g  b  a bf th cl   r  g  b  a ns b eat
---------------------------------------------------------------
-
0x23 24 tc  0 24  0 r  y  .  8  8  8  8  0  0  0  0  0  0  0  0 0 None
0x24 24 tc  0 24  0 r  .  .  8  8  8  8  0  0  0  0  0  0  0  0 0 None
```

```
0x25 24 tc  0 24  0 r  y  .  8  8  8  8  0 24  8  0  0  0  0  0 0 None
0x26 24 tc  0 24  0 r  .  .  8  8  8  8  0 24  8  0  0  0  0  0 0 None
0x27 24 tc  0 24  0 r  y  .  8  8  8  8  0  0  0 16 16 16  0  0 0 Slow
```

The third column is the class of visual. `tc` means a true color visual. Note that some visuals may be labeled `Slow` under the caveat column. Such visuals should be avoided.

To determine which visual MATLAB will use by default with the OpenGL renderer, use the MATLAB `opengl info` command. The returned entry for the visual might look like the following:

```
Visual = 0x23 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
```

Experimenting with a different TrueColor visual may improve certain rendering problems.

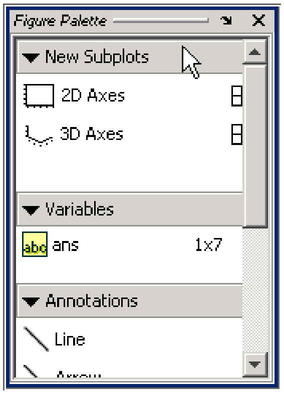
`XVisualMode`

`auto` | `manual` (UNIX only)

Auto or manual selection of visual. `XVisualMode` can take on two values — `auto` (the default) and `manual`. In `auto` mode, MATLAB selects the best visual to use based on the number of colors, availability of the OpenGL extension, etc. In `manual` mode, MATLAB does not change the visual from the one currently in use. Setting the `XVisual` property sets this property to `manual`.

Purpose

Show or hide figure palette



GUI Alternatives

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

Syntax

```
figurepalette('show')
figurepalette('hide')
figurepalette('toggle')
figurepalette(figure_handle,...)
```

Description

`figurepalette('show')` displays the palette on the current figure.

`figurepalette('hide')` hides the palette on the current figure.

`figurepalette('toggle')` or `figurepalette` toggles the visibility of the palette on the current figure.

`figurepalette(figure_handle,...)` shows or hides the palette on the figure specified by `figure_handle`.

See Also

`plottools`, `plotbrowser`, `propertyeditor`

fileattrib

Purpose

Set or get attributes of file or folder

Syntax

```
fileattrib
fileattrib('name')
fileattrib('name','attrib')
fileattrib('name','attrib','users')
fileattrib('name','attrib','users','s')
[status,message,messageid] =  fileattrib(...)
```

Description

`fileattrib` displays the attributes for the current folder. `fileattrib` is like the DOS `attrib` command, or the `chmod` command used on UNIX[4] platforms.

Values are as follows.

Value	Description
`0`	Attribute is off
`1`	Attribute is on
`NaN`	Attribute does not apply

`fileattrib('name')` displays the attributes for `name`, where `name` is the absolute or relative path for a folder or file. Use a wildcard, `*`, at the end of `name` to view attributes for all matching files.

`fileattrib('name','`*`attrib`*`')` sets the attribute for `name`, where `name` is the absolute or relative path for a folder or file. Specify the `+` qualifier before the attribute to set it, and specify the `-` qualifier before the attribute to clear it. Use a wildcard, `*`, at the end of `name` to set attributes for all matches. Values for *`attrib`* are as follows.

Value for attrib	Description
`a`	Archive (Microsoft Windows platform only)

---

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

Value for attrib	**Description**
`h`	Hidden file (Windows platform only)
`s`	System file (Windows platform only)
`w`	Write access (Windows and UNIX platforms). Results differ by platform and application. For example, even though `fileattrib` disables the "write" privilege for a folder, files in the folder could be writable for some platforms or applications.
`x`	Executable (UNIX platform only)

For example, `fileattrib('myfile.m','+w')` makes `myfile.m` a writable file.

`fileattrib('name','`*`attrib`*`','`*`users`*`')` sets the attribute for `name`, where `name` is the absolute or relative path for a folder or file. The *`users`* argument is applicable only for UNIX platforms. For more information about these attributes, see reference information for `chmod` in UNIX operating system documentation. The default value for *`users`* is `u`. The following table lists values for *`users`*.

Value for users	**Description**
`a`	All users
`g`	Group of users
`o`	All other users
`u`	Current user

`fileattrib('name','`*`attrib`*`','`*`users`*`','s')` sets the attribute for `name`, where `name` is the absolute or relative path for a file or a folder and its contents. Here, `s` applies *`attrib`* to all contents of `name`, where `name` is a folder.

`[status,message,messageid] = fileattrib(...)` sets the attribute for `name`, returning the status, a message, and the MATLAB error message ID. Here, `status` is `1` for success and `0` for error. If you do not specify *`attrib`*, *`users`*, and `s`, and the `status` is `1`, then `message` is a structure containing the file attributes and `messageid` is blank. If `status` is `0`, then `messageid` contains the error. To return `message` as a structure, use a wildcard `*` at the end of `name`.

Examples

Get Attributes of File

To view the attributes of `myfile.m`, type:

```
fileattrib('myfile.m')
```

MATLAB returns:

```
           Name: 'd:/work/myfile.m'
        archive: 0
         system: 0
         hidden: 0
      directory: 0
       UserRead: 1
      UserWrite: 0
    UserExecute: 1
      GroupRead: NaN
     GroupWrite: NaN
   GroupExecute: NaN
      OtherRead: NaN
     OtherWrite: NaN
   OtherExecute: NaN
```

`UserWrite` is `0`, meaning `myfile.m` is read only. The `Group` and `Other` values are `NaN` because they do not apply to the current operating system, Windows.

Set File Attribute

To make `myfile.m` become writable, type:

```
fileattrib('myfile.m','+w')
```

Running `fileattrib('myfile.m')` now shows `UserWrite` to be `1`.

Set Attributes for Specified Users

To make the folder `d:/work/results` be a read-only folder for all users, type:

```
fileattrib('d:/work/results','-w','a')
```

The `-` preceding the write attribute, `w`, removes the write status.

Set Multiple Attributes for Folder and Its Contents

To make the folder `d:/work/results` and all its contents read only and hidden, on Windows platforms, type:

```
fileattrib('d:/work/results','+h-w','','s')
```

Because *`users`* is not applicable on Windows systems, the value is empty. Here, `s` applies the attribute to the contents of the specified folder.

Return Status and Structure of Attributes

To return the attributes for the folder `results` to a structure, type:

```
[stat,mess]=fileattrib('results')
```

MATLAB returns

```
stat =
     1

mess =
           Name: 'd:\work\results'
        archive: 0
         system: 0
         hidden: 0
      directory: 1
       UserRead: 1
```

```
      UserWrite: 1
    UserExecute: 1
      GroupRead: NaN
     GroupWrite: NaN
   GroupExecute: NaN
      OtherRead: NaN
     OtherWrite: NaN
   OtherExecute: NaN
```

The operation is successful as indicated by the status, `stat`, being `1`. The structure `mess` contains the file attributes. Access the attribute values in the structure. For example, type:

```
mess.Name
```

MATLAB returns the path for `results`

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

Return Attributes with Wildcard for Name

To return the attributes for all files in the current folder whose names begin with `new`, type:

```
[stat,mess]=fileattrib('new*')
```

MATLAB returns:

```
stat =
     1

mess =
1x3 struct array with fields:
    Name
    archive
    system
    hidden
    directory
```

```
    UserRead
    UserWrite
    UserExecute
    GroupRead
    GroupWrite
    GroupExecute
    OtherRead
    OtherWrite
    OtherExecute
```

The results indicate there are three matching files. To view the file names, type:

```
mess.Name
```

MATLAB returns:

```
ans =
d:\work\results\newname.m

ans =
d:\work\results\newone.m

ans =
d:\work\results\newtest.m
```

To view just the first file name, type:

```
mess(1).Name

ans =
d:\work\results\newname.m
```

See Also

`copyfile`, `cd`, `dir`, `ls`, `mkdir`, `movefile`, `rmdir`

filebrowser

Purpose

Open Current Folder browser, or select it if already open

Graphical Interface

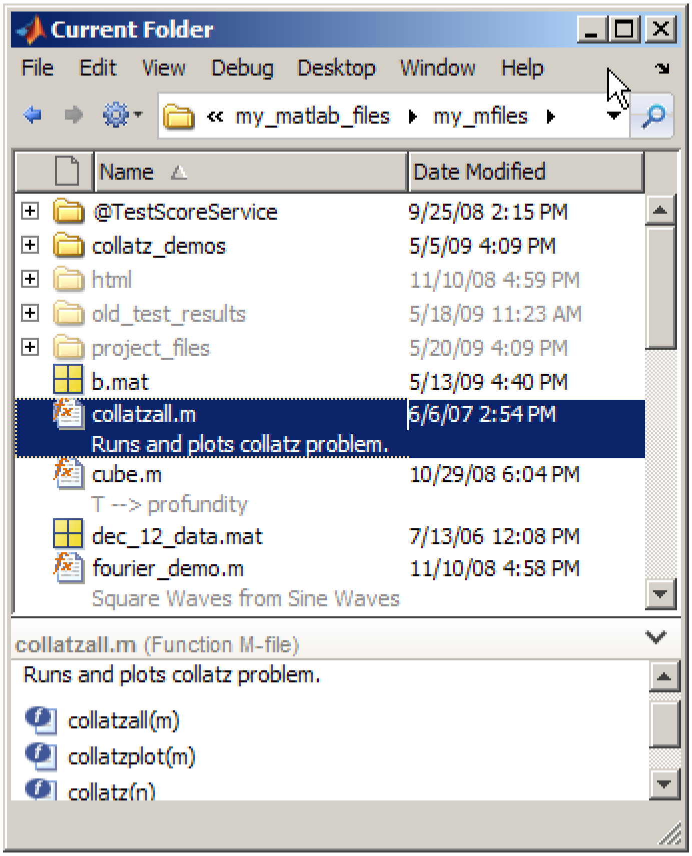
As an alternative to the `filebrowser` function, select **Desktop > Current Folder** in the MATLAB desktop.

Syntax

```
filebrowser
```

Description

`filebrowser` opens the Current Folder Browser, or if it is already open, makes it the selected tool.



Name △	Date Modified
@TestScoreService	9/25/08 2:15 PM
collatz_demos	5/5/09 4:09 PM
html	11/10/08 4:59 PM
old_test_results	5/18/09 11:23 AM
project_files	5/20/09 4:09 PM
b.mat	5/13/09 4:40 PM
collatzall.m Runs and plots collatz problem.	6/6/07 2:54 PM
cube.m T --> profundity	10/29/08 6:04 PM
dec_12_data.mat	7/13/06 12:08 PM
fourier_demo.m Square Waves from Sine Waves	11/10/08 4:58 PM

collatzall.m (Function M-file)

Runs and plots collatz problem.

collatzall(m)

collatzplot(m)

collatz(n)

See Also `cd`, `copyfile`, `fileattrib`, `ls`, `mkdir`, `movefile`, `pwd`, `rmdir`

Purpose

Supported file formats

Description

This table shows the file formats that you can import and export from the MATLAB application.

You can import any of these file formats using the Import Wizard or the `importdata` function, except the following:

- netCDF
- HDF5
- Motion JPEG 2000
- Platform-specific video

File Content	Extension	Description	Import Function	Export Function
MATLAB formatted data	MAT	Saved MATLAB workspace	`load`	`save`
Text	any	White-space delimited numbers	`load`	`save -ascii`
		Delimited numbers	`dlmread`	`dlmwrite`
		Comma delimited numbers	`csvread`	`csvwrite`
		Any of the above text formats, or a mix of strings and numbers	`textscan`	
Spreadsheet	XLS	Microsoft Excel worksheet	`xlsread`	`xlswrite`
	XLSX XLSB XLSM	Formats supported with Excel 2007 for Windows installed		

File Content	Extension	Description	Import Function	Export Function
Extended Markup Language	XML	XML-formatted text	`xmlread`	`xmlwrite`
Data Acquisition Toolbox file	DAQ	Data Acquisition Toolbox	`daqread`	none
Scientific data	CDF	Common Data Format	`cdfread`	`cdfwrite`
	FITS	Flexible Image Transport System	`fitsread`	none
	HDF	Hierarchical Data Format, version 4, or HDF-EOS v. 2	`hdfread`	
	H5	HDF or HDF-EOS, version 5	`hdf5read`	`hdf5write`
	NC	Network Common Data Form (netCDF)	See `netcdf`	See `netcdf`

File Content	Extension	Description	Import Function	Export Function
Image	BMP	Windows Bitmap	`imread`	`imwrite`
	GIF	Graphics Interchange Format		
	HDF	Hierarchical Data Format		
	JPEG JPG	Joint Photographic Experts Group		
	PBM	Portable Bitmap		
	PCX	Paintbrush		
	PGM	Portable Graymap		
	PNG	Portable Network Graphics		
	PNM	Portable Any Map		
	PPM	Portable Pixmap		
	RAS	Sun Raster		
	TIFF TIF	Tagged Image File Format		
	XWD	X Window Dump		
	CUR	Windows Cursor resources	`imread`	none
	FITS FTS	Flexible Image Transport System		
	ICO	Windows Icon resources		
	JP2 JPF JPX J2C J2K	JPEG 2000		

File Content	Extension	Description	Import Function	Export Function
Audio file	AU SND	NeXT/Sun sound	`auread`	`auwrite`
	WAV	Microsoft WAVE sound	`wavread`	`wavwrite`
Video (Windows, Macintosh, and Linux)	AVI	Audio Video Interleave	`mmreader`	`avifile`
	MJ2	Motion JPEG 2000	`mmreader`	none
Video (Windows)	MPG	Motion Picture Experts Group, phases 1 and 2	`mmreader`	none
	ASF ASX WMV	Windows Media®		
	any	Formats supported by Microsoft DirectShow®		
Video (Mac®)	MPG MP4 M4V	MPEG-1 and MPEG-4	`mmreader`	none
	any	Formats supported by QuickTime®, including `.mov`, `.3gp`, `.3g2`, and `.dv`		
Video (Linux)	any	Formats supported by your installed GStreamer plug-ins, including `.ogg`	`mmreader`	none

See Also

`uiimport`, `clipboard`, `fscanf`, `fread`, `fprintf`, `fwrite`, `imformats`, `hdf`, `hdf5`

filemarker

Purpose Character to separate file name and internal function name

Syntax `M = filemarker`

Description `M = filemarker` returns the character that separates a file and a within-file function name.

Examples On the Microsoft Windows platform, for example, `filemarker` returns the '>' character:

```
filemarker
ans =
   >
```

You can use the following command on any platform to get the help text for subfunction `pdeodes` defined in M-file `pdepe.m`:

```
helptext = help(['pdepe' filemarker 'pdeodes'])

helptext =
 PDEODES  Assemble the difference equations and
    evaluate the time derivative for the ODE system.
```

You can use the filemarker character to indicate a location within an M-file where you want to set a breakpoint, for example. On all platforms, if you need to distinguish between two nested functions with the same name, use the forward slash (/) character to indicate the path to a particular instance of a function.

For instance, suppose an M-file, `myfile.m`, contains the following code:

```
function x = A(p1, p2)
...
  function y = B(p3)
    ...
  end
  function m = C(p4)
    ...
```

```
    end
end

function z = C(p5)
...
  function y = D(p6)
    ...
  end
end
```

To indicate that you want to set a breakpoint at `function y` nested within `function x`, use the following command on the Windows platform:

```
dbstop myfile>x/y
```

To indicate that you want to set a breakpoint at `function m` nested within `function x` use the following command on the Windows platform:

```
dbstop myfile>m
```

In the first case, you specify `x/y` because the M-file contains two nested functions named `y`. In the second case, there is no need to specify `x/m` because there is only one function `m` within `myfile.m`.

See Also

`filesep`

Purpose Parts of file name and path

Syntax `[pathstr, name, ext, versn] = fileparts(filename)`

Description `[pathstr, name, ext, versn] = fileparts(filename)` returns the path name, file name, extension, and version for the specified file. `filename` is a string enclosed in single quotes. The returned `ext` field contains a dot (`.`) before the file extension.

The `fileparts` function is platform dependent.

You can reconstruct the file from the parts using

```
fullfile(pathstr,[name ext versn])
```

Examples Return the pieces of a file specification string to the separate string outputs `pathstr`, `name`, `ext`, and `versn`. The full file specification is

```
file = '\home\user4\matlab\classpath.txt';
```

The character used to separate the segments of a path name is dependent on the operating system you are currently running on. In this example, it is the backslash (\) character, which is used as a separator on Microsoft Windows platforms. Use the `filesep` function to insert the correct separator character:

```
sep = filesep;
file = ['' sep 'home' sep 'user4' sep 'matlab' sep ...
        'classpath.txt' ''];
```

Use `fileparts` to return the path name, file name, user name, and file version, if there is one:

```
[pathstr, name, ext, versn] = fileparts(file)

pathstr =
   \home\user4\matlab
```

```
name =
   classpath

ext =
   .txt

versn =
     ''
```

Remarks

On Windows platforms, `C:\` and `C:` are two distinct entities, where `C:\` (with backslash) is the C drive in your computer, and `C:` (without backslash) represents the current working directory.

See Also

`fullfile`, `ver`, `verLessThan`, `version`

fileread

Purpose Read contents of file into string

Syntax *`text`* `= fileread(`*`filename`*`)`

Description *`text`* `= fileread(`*`filename`*`)` returns the contents of the file *`filename`* as a MATLAB string.

Examples Read and search the file `Contents.m` in the MATLAB `iofun` directory for the reference to `fileread`:

```
% find the correct directory and file
io_contents = ...
  fullfile(matlabroot, 'toolbox', 'matlab', 'iofun', 'Contents.m');

% read the file
filetext = fileread(io_contents);

% search for the line of code that includes 'fileread'
% each line is separated by a newline ('\n')

expr = '[^\n]*fileread[^\n]*';
fileread_info = regexp(filetext, expr, 'match');
```

See Also `fgetl` | `fgets` | `fscanf` | `fread` | `importdata` | `textscan` | `type`

Purpose File separator for current platform

Syntax

```
f = filesep
```

Description `f = filesep` returns the platform-specific file separator character. The file separator is the character that separates individual folder and file names in a path string.

Examples Create a path to the `iofun` folder on a Microsoft Windows platform:

```
iofun_dir = ['toolbox' filesep 'matlab' filesep 'iofun']

iofun_dir =
   toolbox\matlab\iofun
```

Create a path to `iofun` on a UNIX[5] platform.

```
iodir = ['toolbox' filesep 'matlab' filesep 'iofun']

iodir =
   toolbox/matlab/iofun
```

See Also `fullfile`, `fileparts`, `pathsep`

---

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

Purpose

Filled 2-D polygons

Syntax

```
fill(X,Y,C)
fill(X,Y,ColorSpec)
fill(X1,Y1,C1,X2,Y2,C2,...)
fill(...,'PropertyName',PropertyValue)
h = fill(...)
```

Description

The `fill` function creates colored polygons.

`fill(X,Y,C)` creates filled polygons from the data in `X` and `Y` with vertex color specified by `C`. `C` is a vector or matrix used as an index into the colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`. If necessary, `fill` closes the polygon by connecting the last vertex to the first.

`fill(X,Y,ColorSpec)` fills two-dimensional polygons specified by `X` and `Y` with the color specified by `ColorSpec`.

`fill(X1,Y1,C1,X2,Y2,C2,...)` specifies multiple two-dimensional filled areas.

`fill(...,'PropertyName',PropertyValue)` allows you to specify property names and values for a patch graphics object.

`h = fill(...)` returns a vector of handles to patch graphics objects, one handle per patch object.

Remarks

If `X` or `Y` is a matrix, and the other is a column vector with the same number of elements as rows in the matrix, `fill` replicates the column vector argument to produce a matrix of the required size. `fill` forms a vertex from corresponding elements in `X` and `Y` and creates one polygon from the data in each column.

The type of color shading depends on how you specify color in the argument list. If you specify color using `ColorSpec`, `fill` generates flat-shaded polygons by setting the patch object's `FaceColor` property to the corresponding RGB triple.

If you specify color using `C`, `fill` scales the elements of `C` by the values specified by the axes property `CLim`. After scaling `C`, `C` indexes the current colormap.

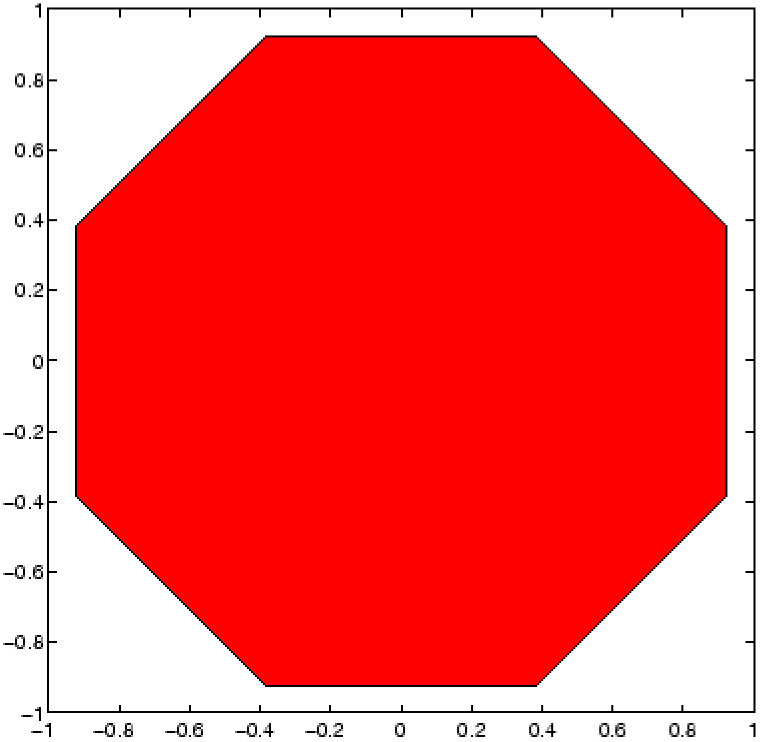
If `C` is a row vector, `fill` generates flat-shaded polygons where each element determines the color of the polygon defined by the respective column of the `X` and `Y` matrices. Each patch object's `FaceColor` property is set to `'flat'`. Each row element becomes the `CData` property value for the `n`th patch object, where *n* is the corresponding column in `X` or `Y`.

If `C` is a column vector or a matrix, `fill` uses a linear interpolation of the vertex colors to generate polygons with interpolated colors. It sets the patch graphics object `FaceColor` property to `'interp'` and the elements in one column become the `CData` property value for the respective patch object. If `C` is a column vector, `fill` replicates the column vector to produce the required sized matrix.

Examples

Create a red octagon.

```
t = (1/16:1/8:1)'*2*pi;
x = sin(t);
y = cos(t);
fill(x,y,'r')
axis square
```



See Also

`axis`, `caxis`, `colormap`, `ColorSpec`, `fill3`, `patch`

“Polygons and Surfaces” on page 1-95 for related functions

Purpose

Filled 3-D polygons

Syntax

```
fill3(X,Y,Z,C)
fill3(X,Y,Z,ColorSpec)
fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)
fill3(...,'PropertyName',PropertyValue)
h = fill3(...)
```

Description

The `fill3` function creates flat-shaded and Gouraud-shaded polygons.

`fill3(X,Y,Z,C)` fills three-dimensional polygons. `X`, `Y`, and `Z` triplets specify the polygon vertices. If `X`, `Y`, or `Z` is a matrix, `fill3` creates *n* polygons, where *n* is the number of columns in the matrix. `fill3` closes the polygons by connecting the last vertex to the first when necessary.

`C` specifies color, where `C` is a vector or matrix of indices into the current colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`.

`fill3(X,Y,Z,ColorSpec)` fills three-dimensional polygons defined by `X`, `Y`, and `Z` with color specified by `ColorSpec`.

`fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)` specifies multiple filled three-dimensional areas.

`fill3(...,'PropertyName',PropertyValue)` allows you to set values for specific patch properties.

`h = fill3(...)` returns a vector of handles to patch graphics objects, one handle per patch.

Algorithm

If `X`, `Y`, and `Z` are matrices of the same size, `fill3` forms a vertex from the corresponding elements of `X`, `Y`, and `Z` (all from the same matrix location), and creates one polygon from the data in each column.

If `X`, `Y`, or `Z` is a matrix, `fill3` replicates any column vector argument to produce matrices of the required size.

If you specify color using `ColorSpec`, `fill3` generates flat-shaded polygons and sets the patch object `FaceColor` property to an RGB triple.

If you specify color using `C`, `fill3` scales the elements of `C` by the axes property `CLim`, which specifies the color axis scaling parameters, before indexing the current colormap.

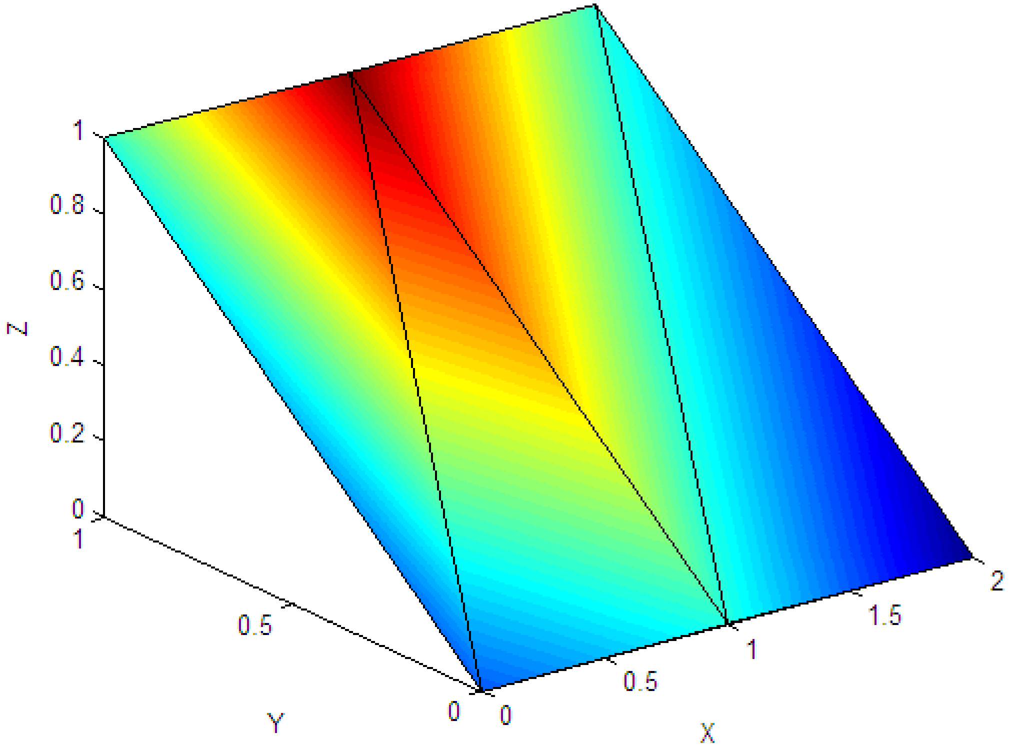
If `C` is a row vector, `fill3` generates flat-shaded polygons and sets the `FaceColor` property of the patch objects to `'flat'`. Each element becomes the `CData` property value for the respective patch object.

If `C` is a column vector or a matrix, `fill3` generates polygons with interpolated colors and sets the patch object `FaceColor` property to `'interp'`. `fill3` uses a linear interpolation of the vertex colormap indices when generating polygons with interpolated colors. The elements in one column become the `CData` property value for the respective patch object. If `C` is a column vector, `fill3` replicates the column vector to produce the required sized matrix.

Examples

Create four triangles with interpolated colors.

```
X = [0 1 1 2;1 1 2 2;0 0 1 1];
Y = [1 1 1 1;1 0 1 0;0 0 0 0];
Z = [1 1 1 1;1 0 1 0;0 0 0 0];
C = [0.5000 1.0000 1.0000 0.5000;
     1.0000 0.5000 0.5000 0.1667;
     0.3330 0.3330 0.5000 0.5000];
fill3(X,Y,Z,C)
```



See Also `axis`, `caxis`, `colormap`, `ColorSpec`, `fill`, `patch`

"Polygons and Surfaces" on page 1-95 for related functions

Purpose

1-D digital filter

Syntax

```
y = filter(b,a,X)
[y,zf] = filter(b,a,X)
[y,zf] = filter(b,a,X,zi)
y = filter(b,a,X,zi,dim)
[...] = filter(b,a,X,[],dim)
```

Description

The `filter` function filters a data sequence using a digital filter which works for both real and complex inputs. The filter is a *direct form II transposed* implementation of the standard difference equation (see “Algorithm”).

`y = filter(b,a,X)` filters the data in vector `X` with the filter described by numerator coefficient vector `b` and denominator coefficient vector `a`. If `a(1)` is not equal to `1`, `filter` normalizes the filter coefficients by `a(1)`. If `a(1)` equals `0`, `filter` returns an error.

If `X` is a matrix, `filter` operates on the columns of `X`. If `X` is a multidimensional array, `filter` operates on the first nonsingleton dimension.

`[y,zf] = filter(b,a,X)` returns the final conditions, `zf`, of the filter delays. If `X` is a row or column vector, output `zf` is a column vector of `max(length(a),length(b))-1`. If `X` is a matrix, `zf` is an array of such vectors, one for each column of `X`, and similarly for multidimensional arrays.

`[y,zf] = filter(b,a,X,zi)` accepts initial conditions, `zi`, and returns the final conditions, `zf`, of the filter delays. Input `zi` is a vector of length `max(length(a),length(b))-1`, or an array with the leading dimension of size `max(length(a),length(b))-1` and with remaining dimensions matching those of `X`.

`y = filter(b,a,X,zi,dim)` and `[...]  = filter(b,a,X,[],dim)` operate across the dimension `dim`.

Example

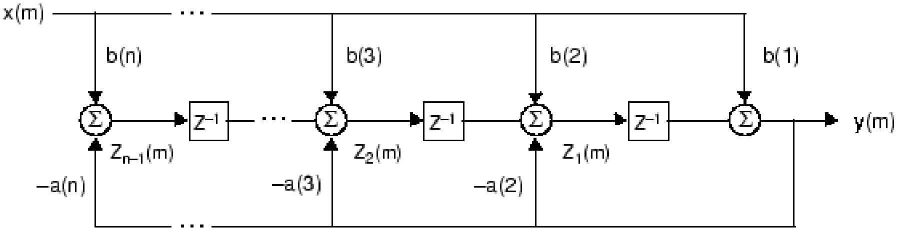
You can use `filter` to find a running average without using a `for` loop. This example finds the running average of a 16-element vector, using a window size of 5.

```
data = [1:0.2:4]';
windowSize = 5;
filter(ones(1,windowSize)/windowSize,1,data)

ans =
    0.2000
    0.4400
    0.7200
    1.0400
    1.4000
    1.6000
    1.8000
    2.0000
    2.2000
    2.4000
    2.6000
    2.8000
    3.0000
    3.2000
    3.4000
    3.6000
```

Algorithm

The `filter` function is implemented as a direct form II transposed structure,



or

```
y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb)
                 - a(2)*y(n-1) - ... - a(na+1)*y(n-na)
```

where `n-1` is the filter order, which handles both FIR and IIR filters [1], `na` is the feedback filter order, and `nb` is the feedforward filter order.

The operation of `filter` at sample m is given by the time domain difference equations

$$\begin{aligned}
y(m) &= b(1)x(m) + z_1(m-1) \\
z_1(m) &= b(2)x(m) + z_2(m-1) - a(2)y(m) \\
\vdots \quad &= \quad \vdots \qquad\qquad\qquad \vdots \\
z_{n-2}(m) &= b(n-1)x(m) + z_{n-1}(m-1) - a(n-1)y(m) \\
z_{n-1}(m) &= b(n)x(m) - a(n)y(m)
\end{aligned}$$

The input-output description of this filtering operation in the z-transform domain is a rational transfer function,

$$Y(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(nb+1)z^{-nb}}{1 + a(2)z^{-1} + \ldots + a(na+1)z^{-na}} X(z)$$

See Also

`filter2`

`filtfilt`, `filtic` in the Signal Processing Toolbox

References

[1] Oppenheim, A. V. and R.W. Schafer. *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989, pp. 311-312.

Purpose

Shape frequency content of time series

Syntax

```
ts2 = filter(ts1,b,a)
ts2 = filter(ts1,b,a,Index)
```

Description

`ts2 = filter(ts1,b,a)` applies the transfer function filter $b(z^{-1})/a(z^{-1})$ to the data in the `timeseries` object `ts1`.

`b` and `a` are the coefficient arrays of the transfer function numerator and denominator, respectively.

`ts2 = filter(ts1,b,a,Index)` uses the optional `Index` integer array to specify the columns or rows to filter. When `ts.IsTimeFirst` is `true`, `Index` specifies one or more data columns. When `ts.IsTimeFirst` is `false`, Index specifies one or more data rows.

Remarks

The time-series data must be uniformly sampled to use this filter.

The following function

```
y = filter(b,a,x)
```

creates filtered data `y` by processing the data in vector `x` with the filter described by vectors `a` and `b`.

The `filter` function is a general tapped delay-line filter, described by the difference equation

$$\begin{aligned} a(1)y(n) = \; & b(1)x(n) + b(2)x(n-1) + \ldots + b(nb)x(n-nb+1) \\ & -a(2)y(n-1) - \ldots - a(N_a)y(n-N_b+1) \end{aligned}$$

Here, n is the index of the current sample, N_a is the order of the polynomial described by vector `a`, and N_b is the order of the polynomial described by vector `b`. The output $y(n)$ is a linear combination of current and previous inputs, $x(n)$ $x(n\text{-}1)$..., and previous outputs, $y(n\text{-}1)$ $y(n\text{-}2)$... .

You use the discrete filter to shape the data by applying a transfer function to the input signal.

Depending on your objectives, the transfer function you choose might alter both the amplitude and the phase of the variations in the data at different frequencies to produce either a smoother or a rougher output.

In digital signal processing (DSP), it is customary to write transfer functions as rational expressions in z^{-1} and to order the numerator and denominator terms in ascending powers of z^{-1}.

Taking the z-transform of the difference equation

$$\begin{aligned} a(1)y(n) = {} & b(1)x(n) + b(2)x(n-1) + \ldots + b(nb)x(n-nb+1) \\ & - a(2)y(n-1) - \ldots - a(na)y(n-na+1) \end{aligned}$$

results in the transfer function

$$Y(z) = H(z^{-1})X(z) = \frac{b(1)+b(2)z^{-1}+\ldots+b(nb)z^{-nb+1}}{a(1)+a(2)z^{-1}+\ldots+a(na)z^{-na+1}}X(z)$$

where $Y(z)$ is the z-transform of the filtered output $y(n)$. The coefficients b and a are unchanged by the z-transform.

Examples

Consider the following transfer function:

$$H(z^{-1}) = \frac{b(z^{-1})}{a(z^{-1})} = \frac{2+3z^{-1}}{1+0.2z^{-1}}$$

You will apply this transfer function to the data in `count.dat`.

1 Load the matrix `count` into the workspace.

```
load count.dat;
```

2 Create a time-series object based on this matrix.

```
count1=timeseries(count(:,1),[1:24]);
```

3 Enter the coefficients of the denominator ordered in ascending powers of z^{-1} to represent $1+0.2z^{-1}$.

```
a = [1 0.2];
```

4 Enter the coefficients of the numerator to represent $2+3z^{-1}$.

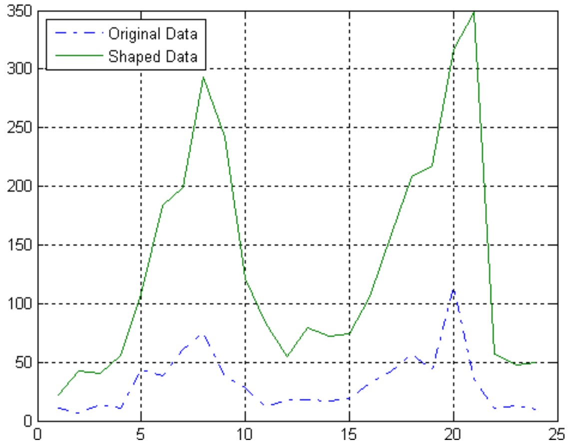
```
b = [2 3];
```

5 Call the filter function.

```
filter_count = filter(count1,b,a)
```

6 Compare the original data and the shaped data with an overlaid plot of the two curves:

```
plot(count1,'-.'), grid on, hold on
plot(filter_count,'-')
legend('Original Data','Shaped Data',2)
```



See Also

`idealfilter (timeseries)`, `timeseries`, `tsprops`

Purpose

2-D digital filter

Syntax

```
Y = filter2(h,X)
Y = filter2(h,X,shape)
```

Description

`Y = filter2(h,X)` filters the data in `X` with the two-dimensional FIR filter in the matrix `h`. It computes the result, `Y`, using two-dimensional correlation, and returns the central part of the correlation that is the same size as `X`.

`Y = filter2(h,X,`*`shape`*`)` returns the part of `Y` specified by the `shape` parameter. *`shape`* is a string with one of these values:

’`full`’	Returns the full two-dimensional correlation. In this case, `Y` is larger than `X`.
’`same`’	(default) Returns the central part of the correlation. In this case, `Y` is the same size as `X`.
’`valid`’	Returns only those parts of the correlation that are computed without zero-padded edges. In this case, `Y` is smaller than `X`.

Remarks

Two-dimensional correlation is equivalent to two-dimensional convolution with the filter matrix rotated 180 degrees. See the Algorithm section for more information about how `filter2` performs linear filtering.

Algorithm

Given a matrix `X` and a two-dimensional FIR filter `h`, `filter2` rotates your filter matrix 180 degrees to create a convolution kernel. It then calls `conv2`, the two-dimensional convolution function, to implement the filtering operation.

`filter2` uses `conv2` to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, `filter2` then extracts the central part of the convolution that is the same size as the input matrix, and returns this as the result. If the `shape` parameter specifies an alternate part of the convolution for the result, `filter2` returns the appropriate part.

See Also `conv2`, `filter`

Purpose

Find indices and values of nonzero elements

Syntax

```
ind = find(X)
ind = find(X, k)
ind = find(X, k, 'first')
ind = find(X, k, 'last')
[row,col] = find(X, ...)
[row,col,v] = find(X, ...)
```

Description

`ind = find(X)` locates all nonzero elements of array `X`, and returns the linear indices of those elements in vector `ind`. If `X` is a row vector, then `ind` is a row vector; otherwise, `ind` is a column vector. If `X` contains no nonzero elements or is an empty array, then `ind` is an empty array.

`ind = find(X, k)` or `ind = find(X, k, 'first')` returns at most the first `k` indices corresponding to the nonzero entries of `X`. `k` must be a positive integer, but it can be of any numeric data type.

`ind = find(X, k, 'last')` returns at most the last `k` indices corresponding to the nonzero entries of `X`.

`[row,col] = find(X, ...)` returns the row and column indices of the nonzero entries in the matrix `X`. This syntax is especially useful when working with sparse matrices. If `X` is an N-dimensional array with N > 2, `col` contains linear indices for the columns. For example, for a 5-by-7-by-3 array `X` with a nonzero element at `X(4,2,3)`, `find` returns 4 in `row` and 16 in `col`. That is, (7 columns in page 1) + (7 columns in page 2) + (2 columns in page 3) = 16.

`[row,col,v] = find(X, ...)` returns a column or row vector `v` of the nonzero entries in `X`, as well as row and column indices. If `X` is a logical expression, then `v` is a logical array. Output `v` contains the non-zero elements of the logical array obtained by evaluating the expression `X`. For example,

```
A= magic(4)
A =
    16     2     3    13
     5    11    10     8
```

```
     9     7     6    12
     4    14    15     1

[r,c,v]= find(A>10);

r', c', v'
ans =
     1     2     4     4     1     3
ans =
     1     2     2     3     4     4
ans =
     1     1     1     1     1     1
```

Here the returned vector `v` is a logical array that contains the nonzero elements of `N` where

```
N=(A>10)
```

Examples

Example 1

```
X = [1 0 4 -3 0 0 0 8 6];
indices = find(X)
```

returns linear indices for the nonzero entries of `X`.

```
indices =
     1     3     4     8     9
```

Example 2

You can use a logical expression to define `X`. For example,

```
find(X > 2)
```

returns linear indices corresponding to the entries of `X` that are greater than 2.

```
ans =
     3     8     9
```

Example 3

The following `find` command

```
X = [3 2 0; -5 0 7; 0 0 1];
[r,c,v] = find(X)
```

returns a vector of row indices of the nonzero entries of `X`

```
r =
     1
     2
     1
     2
     3
```

a vector of column indices of the nonzero entries of `X`

```
c =
     1
     1
     2
     3
     3
```

and a vector containing the nonzero entries of `X`.

```
v =
     3
    -5
     2
     7
     1
```

Example 4

The expression

```
[r,c,v] = find(X>2)
```

returns a vector of row indices of the nonzero entries of `X`

```
r =
     1
     2
```

a vector of column indices of the nonzero entries of `X`

```
c =
     1
     3
```

and a logical array that contains the non zero elements of `N` where `N=(X>2)`.

```
v =
     1
     1
```

Recall that when you use `find` on a logical expression, the output vector `v` does not contain the nonzero entries of the input array. Instead, it contains the nonzero values returned after evaluating the logical expression.

Example 5

Some operations on a vector

```
x = [11  0  33  0  55]';

find(x)
ans =
     1
     3
     5

find(x == 0)
ans =
     2
```

```
     4

find(0 < x & x < 10*pi)
ans =
     1
```

Example 6

For the matrix

```
M = magic(3)
M =
     8     1     6
     3     5     7
     4     9     2

find(M > 3, 4)
```

returns the indices of the first four entries of `M` that are greater than 3.

```
ans =
     1
     3
     5
     6
```

Example 7

If `X` is a vector of all zeros, `find(X)` returns an empty matrix. For example,

```
indices = find([0;0;0])
indices =
   Empty matrix: 0-by-1
```

See Also

`nonzeros`, `sparse`, `colon`, logical operators (elementwise and short-circuit), relational operators, `ind2sub`

findall

Purpose

Find all graphics objects

Syntax

```
object_handles = findall(handle_list)
object_handles = findall(handle_list,'property','value',...)
```

Description

`object_handles = findall(handle_list)` returns the handles, including hidden handles, of all objects in the hierarchy under the objects identified in `handle_list`.

`object_handles = findall(handle_list,'property','value',...)` returns the handles of all objects in the hierarchy under the objects identified in `handle_list` that have the specified properties set to the specified values.

Remarks

`findall` is similar to `findobj`, except that it finds objects even if their `HandleVisibility` is set to `off`.

Examples

```
plot(1:10)
xlabel xlab
a = findall(gcf)
b = findobj(gcf)
c = findall(b,'Type','text') % return the xlabel handle twice
d = findobj(b,'Type','text') % can't find the xlabel handle
```

See Also

`allchild`, `findobj`

Purpose Find visible offscreen figures

Syntax `findfigs`

Description `findfigs` finds all visible figure windows whose display area is off the screen and positions them on the screen.

A window appears to the MATLAB software to be offscreen when its display area (the area not covered by the window's title bar, menu bar, and toolbar) does not appear on the screen.

This function is useful when you are bringing an application from a larger monitor to a smaller one (or one with lower resolution). Windows visible on the larger monitor may appear offscreen on a smaller monitor. Using `findfigs` ensures that all windows appear on the screen.

See Also for related functions.

findobj

Purpose

Locate graphics objects with specific properties

Syntax

```
findobj
h = findobj
h = findobj('PropertyName',PropertyValue,...)
h =
findobj('PropertyName',PropertyValue,'-logicaloperator',
   PropertyName',PropertyValue,...)
h = findobj('-regexp','PropertyName','regexp',...)
h = findobj('-property','PropertyName')
h = findobj(objhandles,...)
h = findobj(objhandles,'-depth',d,...)
h = findobj(objhandles,'flat','PropertyName',PropertyValue,
   ...)
```

Description

`findobj` returns handles of the root object and all its descendants without assigning the result to a variable.

`h = findobj` returns handles of the root object and all its descendants.

`h = findobj('`*`PropertyName`*`',PropertyValue,...)` returns handles of all graphics objects having the property *`PropertyName`*, set to the value `PropertyValue`. You can specify more than one property/value pair, in which case, `findobj` returns only those objects having all specified values.

`h =`
`findobj('`*`PropertyName`*`',PropertyValue,'`*`-logicaloperator`*`',`
`PropertyName``',PropertyValue,...)` applies the logical operator to the property value matching. Possible values for *`-logicaloperator`* are:

- `-and`
- `-or`
- `-xor`
- `-not`

See for an explanation of logical operators.

`h = findobj('-regexp','`*`PropertyName`*`','`*`regexp`*`',...)` matches objects using regular expressions as if the value of you passed the property `PropertyName` to the `regexp` function as

```
regexp(PropertyValue,'regexp')
```

If a match occurs, `findobj` returns the object handle. See the `regexp` function for information on how the MATLAB software uses regular expressions.

`h = findobj('-property','`*`PropertyName`*`')` finds all objects having the specified property.

`h = findobj(objhandles,...)` restricts the search to objects listed in `objhandles` and their descendants.

`h = findobj(objhandles,'-depth',d,...)` specifies the depth of the search. The depth argument `d` controls how many levels under the handles in `objhandles` MATLAB traverses. Specify `d` as `inf` to get the default behavior of all levels. Specify `d` as `0` to get the same behavior as using the `flat` argument.

`h = findobj(objhandles,'flat','`*`PropertyName`*`',PropertyValue,...)` restricts the search to those objects listed in `objhandles` and does not search descendants.

`findobj` returns an error if a handle refers to a nonexistent graphics object.

`findobj` correctly matches any legal property value. For example,

```
findobj('Color','r')
```

finds all objects having a `Color` property set to `red`, `r`, or `[1 0 0]`.

When a graphics object is a descendant of more than one object identified in `objhandles`, MATLAB searches the object each time `findobj` encounters its handle. Therefore, implicit references to a graphics object can result in multiple returns of its handle.

To find handle objects that meet specified conditions, use `handle.findobj`.

Examples

Find all line objects in the current axes:

```
h = findobj(gca,'Type','line')
```

Find all objects having a `Label` set to `'foo'` and a `String` set to `'bar'`:

```
h = findobj('Label','foo','-and','String','bar');
```

Find all objects whose `String` is not `'foo'` and is not `'bar'`:

```
h = findobj('-not','String','foo','-not','String','bar');
```

Find all objects having a `String` set to `'foo'` and a `Tag` set to `'button one'` and whose `Color` is not `'red'` or `'blue'`:

```
h = findobj('String','foo','-and','Tag','button one',...
  '-and','-not',{'Color','red','-or','Color','blue'})
```

Find all objects for which you have assigned a value to the `Tag` property (that is, the value is not the empty string `''`):

```
h = findobj('-regexp','Tag','[^'']')
```

Find all children of the current figure that have their `BackgroundColor` property set to a certain shade of gray (`[.7 .7 .7]`). This statement also searches the current figure for the matching property value pair.

```
h = findobj(gcf,'-depth',1,'BackgroundColor',[.7 .7 .7])
```

See Also `copyobj` | `findall` | `handle.findobj` | `gcf` | `gca` | `gcbo` | `gco` | `get` | `regexp` | `set`

Tutorials

-

findobj (handle)

Purpose Find handle objects matching specified conditions

Syntax `Hmatch = findobj(Hobj,<conditions>)`

Description `Hmatch = findobj(Hobj,<conditions>)` finds `handle` class objects that meet the specified conditions. The `Hobj` argument must be an array of `handle` objects. The returned value, `Hmatch` contains an array of `handles` matching the conditions.

`findobj` has access only to public members of the objects in `Hobj`. See for more information on using `findobj`.

See the Handle Graphics `findobj` function for information on specifying conditions. You cannot use regular expression with `handle` class objects.

See Also `findprop`, `handle`

Purpose

Find `meta.property` object associated with property name

Syntax

```
p = findprop(h,'Name')
```

Description

`p = findprop(h,'Name')` returns the `meta.property` object associated with the property `Name` of the object `h`. `Name` can be a property defined by the class of `h` or a dynamic property defined only for the object `h`.

Examples

Use `findprop` to view property attribute settings:

```
findprop(containers.Map,'Count')

ans =

  meta.property handle
  Package: meta

  Properties:
                   Name: 'Count'
            Description: 'Number of pairs in the collection'
    DetailedDescription: ''
              GetAccess: 'public'
              SetAccess: 'private'
                 Sealed: 0
              Dependent: 1
               Constant: 0
               Abstract: 0
              Transient: 1
                 Hidden: 0
          GetObservable: 0
          SetObservable: 0
               AbortSet: 0
              GetMethod: []
              SetMethod: []
          DefiningClass: [1x1 meta.class]

  Methods, Events, Superclasses
```

See Also `handle`, `findobj (handle)`, `meta.property`

Purpose

Find string within another, longer string

Syntax

`k = findstr(str1, str2)`

Description

`k = findstr(str1, str2)` searches the longer of the two input strings for any occurrences of the shorter string, returning the starting index of each such occurrence in the double array `k`. If no occurrences are found, then `findstr` returns the empty array, `[]`.

The search performed by `findstr` is case sensitive. Any leading and trailing blanks in either input string are explicitly included in the comparison.

Unlike the `strfind` function, the order of the input arguments to `findstr` is not important. This can be useful if you are not certain which of the two input strings is the longer one.

Examples

```
s = 'Find the starting indices of the shorter string.';

findstr(s, 'the')
ans =
     6    30

findstr('the', s)
ans =
     6    30
```

See Also

`strfind`, `strmatch`, `strtok`, `strcmp`, `strncmp`, `strcmpi`, `strncmpi`, `regexp`, `regexpi`, `regexprep`

finish

Purpose

Termination M-file for MATLAB program

Description

When the MATLAB program quits, it runs a script called `finish.m`, if the script exists and is on the search path MATLAB uses or in the current directory. This is a file you create yourself that instructs MATLAB to perform any final tasks just prior to terminating. For example, you might want to save the data in your workspace to a MAT-file before MATLAB exits.

`finish.m` is invoked whenever you do one of the following:

- Click the Close box in the MATLAB desktop on Microsoft Windows platforms or the equivalent on UNIX[6] platforms
- Select **Exit MATLAB** from the desktop **File** menu
- Type `quit` or `exit` at the Command Window prompt

Remarks

When using Handle Graphics features in `finish.m`, use `uiwait`, `waitfor`, or `drawnow` so that figures are visible. See the reference pages for these functions for more information.

Examples

Two sample `finish.m` files are provided with MATLAB in *`matlabroot`*`/toolbox/local`. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` and add it to the path to use it:

- `finishsav.m` — Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m` — Displays a dialog allowing you to cancel quitting and saves the workspace. See also the and the option for exiting MATLAB.

See Also

`quit`, `exit`, `startup`

---

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

in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Information about FITS file

Syntax

```
info = fitsinfo(filename)
```

Description

`info = fitsinfo(filename)` returns the structure, `info`, with fields that contain information about the contents of a Flexible Image Transport System (FITS) file. `filename` is a string enclosed in single quotes that specifies the name of the FITS file.

The `info` structure contains the following fields, listed in the order they appear in the structure. In addition, the info structure can also contain information about any number of optional file components, called *extensions* in FITS terminology. For more information, see "FITS File Extensions" on page 2-1343.

Field Name	Description	Return Type
`Filename`	Name of the file	String
`FileModDate`	File modification date	String
`FileSize`	Size of the file in bytes	Double
`Contents`	List of extensions in the file in the order that they occur	Cell array of strings
`PrimaryData`	Information about the primary data in the FITS file	Structure array

PrimaryData

The `PrimaryData` field is a structure that describes the primary data in the file. The following table lists the fields in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Array containing the size of each dimension	Double array

Field Name	Description	Return Type
`DataSize`	Size of the primary data in bytes	Double
`MissingDataValue`	Value used to represent undefined data	Double
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	Double
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	Double
`Offset`	Number of bytes from beginning of the file to the location of the first data value	Double
`Keywords`	A number-of-keywords-by-3 cell array containing keywords, values, and comments of the header in each column	Cell array of strings

FITS File Extensions

A FITS file can also include optional extensions. If the file contains any of these extensions, the `info` structure can contain these additional fields.

- AsciiTable — Numeric information in tabular format, stored as ASCII characters

- BinaryTable — Numeric information in tabular format, stored in binary representation
- Image — A multidimensional array of pixels
- Unknown — Nonstandard extension

AsciiTable Extension

The `AsciiTable` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`Rows`	Number of rows in the table	Double
`RowSize`	Number of characters in each row	Double
`NFields`	Number of fields in each row	Double array
`FieldFormat`	A 1-by-`NFields` cell containing formats in which each field is encoded. The formats are FORTRAN-77 format codes.	Cell array of strings
`FieldPrecision`	A 1-by-`NFields` cell containing precision of the data in each field	Cell array of strings
`FieldWidth`	A 1-by-`NFields` array containing the number of characters in each field	Double array
`FieldPos`	A 1-by-`NFields` array of numbers representing the starting column for each field	Double array
`DataSize`	Size of the data in the table in bytes	Double
`MissingDataValue`	A 1-by-`NFields` array of numbers used to represent undefined data in each field	Cell array of strings

Field Name	Description	Return Type
`Intercept`	A 1-by-`NFields` array of numbers used along with `Slope` to calculate actual data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	Double array
`Slope`	A 1-by-`NFields` array of numbers used with `Intercept` to calculate true data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	Double array
`Offset`	Number of bytes from beginning of the file to the location of the first data value in the table	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, Values and Comments in the ASCII table header	Cell array of strings

BinaryTable Extension

The `BinaryTable` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`Rows`	Number of rows in the table	Double
`RowSize`	Number of bytes in each row	Double
`NFields`	Number of fields in each row	Double

Field Name	Description	Return Type
`FieldFormat`	A 1-by-`NFields` cell array containing the data type of the data in each field. The data type is represented by a FITS binary table format code.	Cell array of strings
`FieldPrecision`	A 1-by-`NFields` cell containing precision of the data in each field	Cell array of strings
`FieldSize`	A 1-by-`NFields` array, where each element contains the number of values in the Nth field	Double array
`DataSize`	Size of the data in the Binary Table, in bytes. Includes any data past the main table.	Double
`MissingDataValue`	An 1-by-`NFields` array of numbers used to represent undefined data in each field	Cell array of double
`Intercept`	A 1-by-`NFields` array of numbers used along with Slope to calculate actual data values from the array data values using the equation: `actual_value = slope*array_value+Intercept`	Double array
`Slope`	A 1-by-`NFields` array of numbers used with Intercept to calculate true data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	Double array

Field Name	Description	Return Type
`Offset`	Number of bytes from beginning of the file to the location of the first data value	Double
`ExtensionSize`	Size of any data past the main table, in bytes	Double
`ExtensionOffset`	Number of bytes from the beginning of the file to any data past the main table	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Image Extension

The `Image` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Array containing sizes of each dimension	Double array
`DataSize`	Size of the data in the Image extension in bytes	Double
`Offset`	Number of bytes from the beginning of the file to the first data value	Double
`MissingDataValue`	Value used to represent undefined data	Double

Field Name	Description	Return Type
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value+Intercept`	Double
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Unknown Structure

The `Unknown` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Sizes of each dimension	Double array
`DataSize`	Size of the data in nonstandard extensions, in bytes	Double
`Offset`	Number of bytes from beginning of the file to the first data value	Double

Field Name	Description	Return Type
`MissingDataValue`	Representation of undefined data	Double
`Intercept`	Value, used with `Slope`, to calculate actual data values from the array data values, using the equation: `actual_value = Slope*array_value+Intercept`	Double
`Slope`	Value, used with `Intercept`, to calculate actual data values from the array data values, using the equation: `actual_value = Slope*array_value+Intercept`	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Example

Use `fitsinfo` to obtain information about the FITS file `tst0012.fits`. In addition to its primary data, the file also contains an example of the extensions `BinaryTable`, `Unknown`, `Image`, and `AsciiTable`.

```
S = fitsinfo('tst0012.fits');
S =
         Filename: [1x71 char]
      FileModDate: '12-Mar-2001 18:37:46'
         FileSize: 109440
         Contents: {'Primary' 'Binary Table' 'Unknown'
'Image' 'ASCII Table'}
      PrimaryData: [1x1 struct]
      BinaryTable: [1x1 struct]
```

```
          Unknown: [1x1 struct]
            Image: [1x1 struct]
       AsciiTable: [1x1 struct]
```

The `PrimaryData` field describes the data in the file. For example, the `Size` field indicates the data is a 102-by-109 matrix.

```
S.PrimaryData
         DataType: 'single'
             Size: [102 109]
         DataSize: 44472
 MissingDataValue: []
        Intercept: 0
            Slope: 1
           Offset: 2880
         Keywords: {25x3 cell}
```

The `AsciiTable` field describes the `AsciiTable` extension. For example, using the `FieldWidth` and `FieldPos` fields you can determine the length and location of each field within a row.

```
S.AsciiTable
ans =
             Rows: 53
          RowSize: 59
          NFields: 8
      FieldFormat: {'A9' 'F6.2' 'I3' 'E10.4' 'D20.15' 'A5' 'A1' 'I4'}
   FieldPrecision: {1x8 cell}
       FieldWidth: [9 6.2000 3 10.4000 20.1500 5 1 4]
         FieldPos: [1 11 18 22 33 54 54 55]
         DataSize: 3127
 MissingDataValue: {'*' '---.--' '  *' [] '*' '*' '*' ''}
        Intercept: [0 0 -70.2000 0 0 0 0 0]
            Slope: [1 1 2.1000 1 1 1 1 1]
           Offset: 103680
         Keywords: {65x3 cell}
```

See Also

`fitsread`

Purpose

Read data from FITS file

Syntax

```
data = fitsread(filename)
data = fitsread(filename, extname)
data = fitsread(filename, extname, index)
data = fitsread(filename, 'raw')
```

Description

`data = fitsread(filename)` reads the primary data of the Flexible Image Transport System (FITS) file specified by `filename`. Undefined data values are replaced by `NaN`. Numeric data are scaled by the slope and intercept values and are always returned in double precision. The `filename` argument is a string enclosed in single quotes.

`data = fitsread(filename, extname)` reads data from a FITS file according to the data array or extension specified in `extname`. You can specify only one `extname`. The valid choices for `extname` are shown in the following table.

Data Arrays or Extensions

extname	Description
’primary’	Read data from the primary data array.
’table’	Read data from the ASCII Table extension.
’bintable’	Read data from the Binary Table extension.
’image’	Read data from the Image extension.
’unknown’	Read data from the Unknown extension.

`data = fitsread(filename, extname, index)` is the same as the above syntax, except that if there is more than one of the specified extension type `extname` in the file, then only the one at the specified `index` is read.

`data = fitsread(filename, 'raw')` reads the primary or extension data of the FITS file, but, unlike the above syntaxes, does not replace undefined data values with `NaN` and does not scale the data. The `data` returned has the same class as the data stored in the file.

Example

Read FITS file `tst0012.fits` into a 109-by-102 matrix called `data`.

```
data = fitsread('tst0012.fits');

whos data
  Name       Size            Bytes  Class

  data     109x102           88944  double array
```

Here is the beginning of the data read from the file.

```
data(1:5,1:6)
ans =
  135.200  134.9436  134.1752  132.8980  131.1165  128.8378
  137.568  134.9436  134.1752  132.8989  131.1167  126.3343
 135.9946  134.9437  134.1752  132.8989  131.1185  128.1711
 134.0093  134.9440  134.1749  132.8983  131.1201  126.3349
 131.5855  134.9439  134.1749  132.8989  131.1204  126.3356
```

Read only the Binary Table extension from the file.

```
data = fitsread('tst0012.fits', 'bintable')

data =
  Columns 1 through 4
    {11x1 cell}  [11x1 int16]  [11x3 uint8]  [11x2 double]
  Columns 5 through 9
    [11x3 cell]  {11x1 cell}  [11x1 int8]  {11x1 cell}  [11x3 int32]
  Columns 10 through 13
    [11x2 int32]  [11x2 single]  [11x1 double]  [11x1 uint8]
```

See Also

`fitsinfo`

Purpose Round toward zero

Syntax `B = fix(A)`

Description `B = fix(A)` rounds the elements of `A` toward zero, resulting in an array of integers. For complex `A`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000        -0.2000        3.4000        5.6000

  Columns 5 through 6
   7.0000        2.4000 + 3.6000i

fix(a)

ans =
  Columns 1 through 4
  -1.0000        0             3.0000        5.0000

  Columns 5 through 6
   7.0000        2.0000 + 3.0000i
```

See Also `ceil`, `floor`, `round`

flipdim

Purpose

Flip array along specified dimension

Syntax

```
B = flipdim(A,dim)
```

Description

`B = flipdim(A,dim)` returns `A` with dimension `dim` flipped.

When the value of `dim` is `1`, the array is flipped row-wise down. When `dim` is `2`, the array is flipped columnwise left to right. `flipdim(A,1)` is the same as `flipud(A)`, and `flipdim(A,2)` is the same as `fliplr(A)`.

Examples

`flipdim(A,1)` where

```
A =

        1      4
        2      5
        3      6
```

produces

```
        3      6
        2      5
        1      4
```

See Also

`fliplr`, `flipud`, `permute`, `rot90`

Purpose

Flip matrix left to right

Syntax

```
B = fliplr(A)
```

Description

`B = fliplr(A)` returns `A` with columns flipped in the left-right direction, that is, about a vertical axis.

If `A` is a row vector, then `fliplr(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a column vector, then `fliplr(A)` simply returns `A`.

Examples

If `A` is the 3-by-2 matrix,

```
A =
     1     4
     2     5
     3     6
```

`then fliplr(A)` produces

```
     4     1
     5     2
     6     3
```

If `A` is a row vector,

```
A =
     1     3     5     7     9
```

then `fliplr(A)` produces

```
     9     7     5     3     1
```

Limitations

The array being operated on cannot have more than two dimensions. This limitation exists because the axis upon which to flip a multidimensional array would be undefined.

See Also

`flipdim`, `flipud`, `rot90`

flipud

Purpose Flip matrix up to down

Syntax

```
B = flipud(A)
```

Description `B = flipud(A)` returns `A` with rows flipped in the up-down direction, that is, about a horizontal axis.

If `A` is a column vector, then `flipud(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a row vector, then `flipud(A)` simply returns `A`.

Examples If `A` is the 3-by-2 matrix,

```
A =
     1     4
     2     5
     3     6
```

`then flipud(A)` produces

```
     3     6
     2     5
     1     4
```

If `A` is a column vector,

```
A =
     3
     5
     7
```

then `flipud(A)` produces

```
A =
     7
     5
     3
```

Limitations

The array being operated on cannot have more than two dimensions. This limitation exists because the axis upon which to flip a multidimensional array would be undefined.

See Also

`flipdim`, `fliplr`, `rot90`

floor

Purpose Round toward negative infinity

Syntax `B = floor(A)`

Description `B = floor(A)` rounds the elements of `A` to the nearest integers less than or equal to `A`. For complex `A`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000            -0.2000              3.4000              5.6000

  Columns 5 through 6
   7.0000             2.4000 + 3.6000i

floor(a)

ans =
  Columns 1 through 4
  -2.0000            -1.0000              3.0000              5.0000

  Columns 5 through 6
   7.0000             2.0000 + 3.0000i
```

See Also `ceil`, `fix`, `round`

Purpose

Simple function of three variables

Syntax

```
v = flow
v = flow(n)
v = flow(x,y,z)
[x,y,z,v] = flow(...)
```

Description

`flow`, a function of three variables, generates fluid-flow data that is useful for demonstrating `slice`, `interp3`, and other functions that visualize scalar volume data.

`v = flow` produces a 50-by-25-by-25 array.

`v = flow(n)` produces a `n`-by-`2n`-by-`n` array.

`v = flow(x,y,z)` evaluates the speed profile at the points `x`, `y`, and `z`.

`[x,y,z,v] = flow(...)` returns the coordinates as well as the volume data.

See Also

`slice`, `interp3`

"Volume Visualization" on page 1-106 for related functions

See for an example that uses `flow`.

fminbnd

Purpose

Find minimum of single-variable function on fixed interval

Syntax

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
[x,fval] = fminbnd(...)
[x,fval,exitflag] = fminbnd(...)
[x,fval,exitflag,output] = fminbnd(...)
```

Description

`fminbnd` finds the minimum of a function of one variable within a fixed interval.

`x = fminbnd(fun,x1,x2)` returns a value `x` that is a local minimizer of the function that is described in `fun` in the interval `x1 < x < x2`. `fun` is a function handle. See in the MATLAB Programming documentation for more information.

in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`.

`x = fminbnd(fun,x1,x2,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminbnd` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex` or `NaN`. `'off'` displays no error.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.

`OutputFcn`	User-defined function that is called at each iteration. See in MATLAB Mathematics for more information.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`). • `@optimplotx` plots the current point • `@optimplotfval` plots the function value • `@optimplotfunccount` plots the function count See in MATLAB Mathematics for more information.
`TolX`	Termination tolerance on `x`.

`[x,fval] = fminbnd(...)` returns the value of the objective function computed in `fun` at `x`.

`[x,fval,exitflag] = fminbnd(...)` returns a value `exitflag` that describes the exit condition of `fminbnd`:

`1`	`fminbnd` converged to a solution `x` based on `options.TolX`.
`0`	Maximum number of function evaluations or iterations was reached.
`-1`	Algorithm was terminated by the output function.
`-2`	Bounds are inconsistent (`x1 > x2`).

`[x,fval,exitflag,output] = fminbnd(...)` returns a structure `output` that contains information about the optimization in the following fields:

`algorithm`	Algorithm used
`funcCount`	Number of function evaluations
`iterations`	Number of iterations
`message`	Exit message

Arguments

`fun` is the function to be minimized. `fun` accepts a scalar `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fminbnd(@myfun,x1,x2);
```

where `myfun.m` is an M-file function such as

```
function f = myfun(x)
f = ...          % Compute function value at x.
```

or as a function handle for an anonymous function:

```
x = fminbnd(@(x) sin(x*x),x1,x2);
```

Other arguments are described in the syntax descriptions above.

Examples

`x = fminbnd(@cos,3,4)` computes π to a few decimal places and gives a message on termination.

```
[x,fval,exitflag] = ...
   fminbnd(@cos,3,4,optimset('TolX',1e-12,'Display','off'))
```

computes π to about 12 decimal places, suppresses output, returns the function value at `x`, and returns an `exitflag` of 1.

The argument `fun` can also be a function handle for an anonymous function. For example, to find the minimum of the function $f(x) = x^3 - 2x - 5$ on the interval `(0,2)`, create an anonymous function `f`

```
f = @(x)x.^3-2*x-5;
```

Then invoke `fminbnd` with

```
x = fminbnd(f, 0, 2)
```

The result is

```
x =
    0.8165
```

The value of the function at the minimum is

```
y = f(x)

y =
    -6.0887
```

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = (x - a)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminbind`. To optimize for a specific value of `a`, such as `a = 1.5`.

1. Assign the value to `a`.

   ```
   a = 1.5; % define parameter first
   ```

2. Call `fminbnd` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

   ```
   x = fminbnd(@(x) myfun(x,a),0,1)
   ```

Algorithm

`fminbnd` is an M-file. The algorithm is based on golden section search and parabolic interpolation. Unless the left endpoint x_1 is very close to the right endpoint x_2, `fminbnd` never evaluates `fun` at the endpoints, so `fun` need only be defined for x in the interval $x_1 < x < x_2$. If the minimum actually occurs at x_1 or x_2, `fminbnd` returns an interior point at a distance of no more than `2*TolX` from x_1 or x_2, where `TolX` is the termination tolerance. See [1] or [2] for details about the algorithm.

Limitations

The function to be minimized must be continuous. `fminbnd` may only give local solutions.

`fminbnd` often exhibits slow convergence when the solution is on a boundary of the interval.

`fminbnd` only handles real variables.

See Also

`fminsearch`, `fzero`, `optimset`, `function_handle` (`@`), anonymous function

References

[1] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

[2] Brent, Richard. P., *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973

Purpose

Find minimum of unconstrained multivariable function using derivative-free method

Syntax

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,options)
[x,fval] = fminsearch(...)
[x,fval,exitflag] = fminsearch(...)
[x,fval,exitflag,output] = fminsearch(...)
```

Description

`fminsearch` finds the minimum of a scalar function of several variables, starting at an initial estimate. This is generally referred to as *unconstrained nonlinear optimization*.

`x = fminsearch(fun,x0)` starts at the point `x0` and returns a value `x` that is a local minimizer of the function described in `fun`. `x0` can be a scalar, vector, or matrix. `fun` is a function handle. See in the MATLAB Programming documentation for more information.

in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`. See also “Example 2” on page 2-1467 and “Example 3” on page 2-1468 below.

`x = fminsearch(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminsearch` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex`, `Inf` or `NaN`. `'off'` (the default) displays no error.
`MaxFunEvals`	Maximum number of function evaluations allowed

`MaxIter`	Maximum number of iterations allowed
`OutputFcn`	User-defined function that is called at each iteration. See in MATLAB Mathematics for more information.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`). • `@optimplotx` plots the current point • `@optimplotfval` plots the function value • `@optimplotfunccount` plots the function count See in MATLAB Mathematics for more information.
`TolFun`	Termination tolerance on the function value
`TolX`	Termination tolerance on `x`

`[x,fval] = fminsearch(...)` returns in `fval` the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fminsearch(...)` returns a value `exitflag` that describes the exit condition of `fminsearch`:

`1`	`fminsearch` converged to a solution `x`.
`0`	Maximum number of function evaluations or iterations was reached.
`-1`	Algorithm was terminated by the output function.

`[x,fval,exitflag,output] = fminsearch(...)` returns a structure `output` that contains information about the optimization in the following fields:

`algorithm`	Algorithm used
`funcCount`	Number of function evaluations
`iterations`	Number of iterations
`message`	Exit message

Arguments

`fun` is the function to be minimized. It accepts an input `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fminsearch(@myfun, x0)
```

where `myfun` is an M-file function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

or as a function handle for an anonymous function, such as

```
x = fminsearch(@(x)sin(x^2), x0);
```

Other arguments are described in the syntax descriptions above.

Examples

Example 1

A classic test example for multidimensional minimization is the Rosenbrock banana function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2 .$$

The minimum is at `(1,1)` and has the value `0`. The traditional starting point is `(-1.2,1)`. The anonymous function shown here defines the function and returns a function handle called `banana`:

```
banana = @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

Pass the function handle to `fminsearch`:

```
[x,fval] = fminsearch(banana,[-1.2, 1])
```

This produces

```
x =

    1.0000    1.0000

fval =

    8.1777e-010
```

This indicates that the minimizer was found to at least four decimal places with a value near zero.

Example 2

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = x(1)^2 + a*x(2)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminsearch`. To optimize for a specific value of `a`, such as `a = 1.5`.

1. Assign the value to `a`.

   ```
   a = 1.5; % define parameter first
   ```

2. Call `fminsearch` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

   ```
   x = fminsearch(@(x) myfun(x,a),[0,1])
   ```

Example 3

You can modify the first example by adding a parameter a to the second term of the banana function:

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(a - x_1\right)^2.$$

This changes the location of the minimum to the point `[a,a^2]`. To minimize this function for a specific value of `a`, for example a = `sqrt(2)`, create a one-argument anonymous function that captures the value of `a`.

```
a = sqrt(2);
banana = @(x)100*(x(2)-x(1)^2)^2+(a-x(1))^2;
```

Then the statement

```
[x,fval] = fminsearch(banana, [-1.2, 1], ...
   optimset('TolX',1e-8));
```

seeks the minimum `[sqrt(2), 2]` to an accuracy higher than the default on `x`.

Algorithm

fminsearch uses the simplex search method of [1]. This is a direct search method that does not use numerical or analytic gradients.

If `n` is the length of `x`, a simplex in `n`-dimensional space is characterized by the `n+1` distinct vectors that are its vertices. In two-space, a simplex is a triangle; in three-space, it is a pyramid. At each step of the search, a new point in or near the current simplex is generated. The function value at the new point is compared with the function's values at the vertices of the simplex and, usually, one of the vertices is replaced by the new point, giving a new simplex. This step is repeated until the diameter of the simplex is less than the specified tolerance.

For more information, see .

Limitations

`fminsearch` can often handle discontinuity, particularly if it does not occur near the solution. fminsearch may only give local solutions.

`fminsearch` only minimizes over the real numbers, that is, x must only consist of real numbers and $f(x)$ must only return real numbers. When x has complex variables, they must be split into real and imaginary parts.

See Also

`fminbnd`, `optimset`, `function_handle` (@), anonymous function

References

[1] Lagarias, J.C., J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions," *SIAM Journal of Optimization*, Vol. 9 Number 1, pp. 112-147, 1998.

Purpose

Open file, or obtain information about open files

Syntax

```
fileID = fopen(filename)
fileID = fopen(filename, permission)
fileID = fopen(filename, permission, machineformat)
fileID = fopen(filename, permission,
machineformat, encoding)
[fileID, message] = fopen(filename, ...)
fIDs = fopen('all')
[filename, permission, machineformat,
   encoding] = fopen(fileID)
```

Description

`fileID = fopen(filename)` opens the file `filename` for read access, and returns an integer file identifier.

`fileID = fopen(filename, permission)` opens the file with the specified `permission`.

`fileID = fopen(filename, permission, machineformat)` specifies the order for reading or writing bytes or bits in the file.

`fileID = fopen(filename, permission, machineformat, encoding)` specifies the character encoding scheme associated with the file.

`[fileID, message] = fopen(filename, ...)` opens a file. If the operation fails, `message` is a system-dependent error message. Otherwise, `message` is an empty string.

`fIDs = fopen('all')` returns a row vector containing the file identifiers of all open files.

`[filename, permission, machineformat, encoding] = fopen(fileID)` returns the file name, permission, machine format, and encoding that a previous call to `fopen` used when it opened the specified file. `fopen` does not read information from the file to determine these output values. An invalid `fileID` returns empty strings for all output arguments.

Inputs

`filename`

String in single quotation marks that specifies the name of the file to open. Can include a full or partial path.

On UNIX systems, if *`filename`* begins with `'~/'` or `'~`*`username`*`/'`, the `fopen` function expands the path to the current or specified user's home directory, respectively.

If you open a file with read access and `fopen` cannot find *`filename`* in the current folder, `fopen` searches along the MATLAB search path. Otherwise, `fopen` creates a file in the current directory.

`permission`

String that describes the type of access for the file: read, write, append, or update. Also specifies whether to open files in binary or text mode.

To open files in binary mode, specify one of the following:

`'r'`	Open file for reading (default).
`'w'`	Open or create new file for writing. Discard existing contents, if any.
`'a'`	Open or create new file for writing. Append data to the end of the file.
`'r+'`	Open file for reading and writing.
`'w+'`	Open or create new file for reading and writing. Discard existing contents, if any.
`'a+'`	Open or create new file for reading and writing. Append data to the end of the file.
`'A'`	Append without automatic flushing. (Used with tape drives.)
`'W'`	Write without automatic flushing. (Used with tape drives.)

To read and write to the same file:

- Open the file in update mode (with a *`permission`* that includes a plus sign, `'+'`).
- Call `fseek` or `frewind` between read and write operations. For example, do not call `fread` followed by `fwrite`, or `fwrite` followed by `fread`, unless you call `fseek` or `frewind` between them.

To open files in text mode, attach the letter `'t'` to the *`permission`*, such as `'rt'` or `'wt+'`. For better performance, do not use text mode. The following applies on Windows systems, in text mode:

- Read operations that encounter a carriage return followed by a newline character (`'\r\n'`) remove the carriage return from the input.
- Write operations insert a carriage return before any newline character in the output.

This additional processing is unnecessary for most cases. All MATLAB import functions, and most text editors (including Microsoft Word and WordPad), recognize both `'\r\n'` and `'\n'` as newline sequences. However, when you create files for use in Microsoft Notepad, end each line with `'\r\n'`. For an example, see `fprintf`.

`machineformat`

String that specifies the order for reading or writing bytes or bits in the file. Specify *`machineformat`* to:

- Read a file created on a different system.
- Read bits in a particular order.
- Create a file for use on a different system.

Possible values are:

`'n'` or `'native'`	The byte ordering that your system uses (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering
`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit data type

Windows systems use little-endian ordering, and most UNIX systems use big-endian ordering, for both bytes and bits. Solaris systems use big-endian ordering for bytes, but little-endian ordering for bits.

encoding

String that specifies the character encoding scheme to use for subsequent read and write operations, including `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`.

Possible values are:

`'Big5'`
`'EUC-JP'`
`'GBK'`
`'Shift_JIS'`
`'US-ASCII'`
`'UTF-8'`
`'ISO-8859-1'`
`'ISO-8859-2'`
`'ISO-8859-3'`
`'ISO-8859-4'`
`'ISO-8859-9'`
`'ISO-8859-13'`
`'ISO-8859-15'`
`'windows-932'`
`'windows-936'`
`'windows-949'`
`'windows-950'`
`'windows-1250'`
`'windows-1251'`
`'windows-1252'`
`'windows-1253'`
`'windows-1254'`
`'windows-1257'`

Default: system-dependent

Outputs

`fileID`

An integer that identifies the file for all subsequent low-level file I/O operations. If `fopen` cannot open the file, *`fileID`* is `-1`.

MATLAB reserves file identifiers `0`, `1`, and `2` for standard input, standard output (the screen), and standard error, respectively. When `fopen` successfully opens a file, it returns a file identifier greater than or equal to `3`.

`message`

A system-dependent error message when `fopen` cannot open the specified file. Otherwise, an empty string.

`fIDs`

Row vector containing the identifiers for all open files, except the identifiers reserved for standard input, output, and error. The number of elements in the vector is equal to the number of open files.

`filename`

Name of the file associated with the specified *`fileID`*.

`permission`

The *`permission`* that `fopen` assigned to the file specified by *`fileID`*.

`machineformat`

The value of *`machineformat`* that `fopen` used when it opened the file specified by *`fileID`*.

`encoding`

The character encoding scheme that `fopen` associated with the file specified by *`fileID`*.

The value that `fopen` returns for *`encoding`* is a standard character encoding scheme name. It is not always the same as the *`encoding`* argument that you used in the call to `fopen` to open the file.

Examples

Open a file. Pass the file identifier, `fid`, to other file I/O functions to read data and close the file.

```
fid = fopen('fgetl.m');

tline = fgetl(fid);
while ischar(tline)
    disp(tline);
    tline = fgetl(fid);
end

fclose(fid);
```

Create a prompt to request the name of a file to open. If `fopen` cannot open the file, display the relevant error message.

```
fid = -1;
msg = '';
while fid < 0
   disp(msg);
   filename = input('Open file: ', 's');
   [fid,msg] = fopen(filename);
end
```

Open a file to write Unicode characters to a file using the Shift-JIS character encoding scheme:

```
fid = fopen('japanese_out.txt', 'w', 'n', 'Shift_JIS');
```

See Also

`fclose` | `ferror`

Related Links

- http://www.iana.org/assignments/character-sets

fopen (serial)

Purpose

Connect serial port object to device

Syntax

```
fopen(obj)
```

Description

`fopen(obj)` connects the serial port object, `obj` to the device.

Remarks

Before you can perform a read or write operation, `obj` must be connected to the device with the `fopen` function. When `obj` is connected to the device:

- Data remaining in the input buffer or the output buffer is flushed.
- The `Status` property is set to `open`.
- The `BytesAvailable`, `ValuesReceived`, `ValuesSent`, and `BytesToOutput` properties are set to 0.

An error is returned if you attempt to perform a read or write operation while `obj` is not connected to the device. You can connect only one serial port object to a given device.

Some properties are read-only while the serial port object is open (connected), and must be configured before using `fopen`. Examples include `InputBufferSize` and `OutputBufferSize`. Refer to the property reference pages to determine which properties have this constraint.

The values for some properties are verified only after `obj` is connected to the device. If any of these properties are incorrectly configured, then an error is returned when `fopen` is issued and `obj` is not connected to the device. Properties of this type include `BaudRate`, and are associated with device settings.

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

```
help serial/fopen
```

Example

This example creates the serial port object `s`, connects `s` to the device using `fopen`, writes and reads text data, and then disconnects `s` from the device. This example works on a Windows platform.

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

See Also

Functions

`fclose`

Properties

`BytesAvailable`, `BytesToOutput`, `Status`, `ValuesReceived`, `ValuesSent`

Purpose Execute block of code specified number of times

Syntax

```
for x=initval:endval, statements, end
for x=initval:stepval:endval, statements, end
```

Description `for x=initval:endval, statements, end` repeatedly executes one or more MATLAB *`statements`* in a loop. Loop counter variable `x` is initialized to value `initval` at the start of the first pass through the loop, and automatically increments by 1 each time through the loop. The program makes repeated passes through *`statements`* until either `x` has incremented to the value *`endval`*, or MATLAB encounters a `break`, or `return` instruction, thus forcing an immediately exit of the loop. If MATLAB encounters a `continue` statement in the loop code, it immediately exits the current pass at the location of the `continue` statement, skipping any remaining code in that pass, and begins another pass at the start of the loop *`statements`* with the value of the loop counter incremented by one.

The values `initval` and `endval` must be real numbers or arrays of real numbers, or can also be calls to functions that return the same. The value assigned to `x` is often used in the code within the loop, however it is recommended that you do not assign to `x` in the loop code.

`for x=initval:stepval:endval, statements, end` is the same as the above syntax, except that loop counter `x` is incremented (or decremented when *`stepval`* is negative) by the value *`stepval`* on each iteration through the loop. The value `stepval` must be a real number or can also be a call to a function that returns a real number.

The general format is

```
for variable = initval:endval
    statement
    ...
    statement
end
```

If MATLAB encounters a `continue` statement in the loop code, it immediately exits the current pass at the location of the `continue` statement, skipping any remaining code in that pass, and begins another pass at the start of the loop *`statements`* with the value of the loop counter incremented by the appropriate value (either 1 or the specified step value).

`continue` works the same way nested loops. That is, execution continues at the beginning of the loop in which the `continue` statement was encountered.

The scope of the `for` 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

It is recommended that you do not assign to the loop control variable while in the body of a loop. If you do assign to a variable that has the same name as the loop control variable (see `k` in the example below), then the value of that variable alternates between the value assigned by the `for` statement at the start of each loop iteration and the value explicitly assigned to it in the loop code:

```
for k=1:2
   disp(sprintf('  At the start of the loop, k = %d', k))
   k = 10;
   disp(sprintf('  Following the assignment, k = %d\n', k))
end

  At the start of the loop, k = 1
  Following the assignment, k = 10

  At the start of the loop, k = 2
  Following the assignment, k = 10
```

Examples

Assume `k` has already been assigned a value. Create the Hilbert matrix, using `zeros` to preallocate the matrix to conserve memory:

```
a = zeros(k,k)  % Preallocate matrix
for m = 1:k
    for n = 1:k
        a(m,n) = 1/(m+n -1);
    end
end
```

Step `s` with increments of `-0.1`:

```
for s = 1.0: -0.1: 0.0,..., end
```

Step `s` with values `1`, `5`, `8`, and `17`:

```
for s = [1,5,8,17], ..., end
```

Successively set `e` to the unit `n`-vectors:

```
for e = eye(n), ..., end
```

The line

```
for V = A, ..., end
```

has the same effect as

```
for k = 1:n, V = A(:,k); ..., end
```

except `k` is also set here.

See Also

`end`, `while`, `break`, `continue`, `parfor`, `return`, `if`, `switch`, `colon`

Purpose

Set display format for output

Graphical Interface

As an alternative to using the `format` command, you can also use the MATLAB Preferences GUI. Select **File > Preferences > Command Window** and press the **Help** button for more information.

Syntax

```
format
format type
format('type')
```

Description

Use the `format` function to control the output format of numeric values displayed in the Command Window.

Note The `format` function affects only how numbers are displayed, not how MATLAB computes or saves them.

`format` by itself, changes the output format to the default appropriate for the class of the variable currently being used. For floating-point variables, for example, the default is `format short` (i.e., 5-digit scaled, fixed-point values).

`format` *`type`* changes the format to the specified *`type`*. The tables shown below list the allowable values for *`type`*.

`format('`*`type`*`')` is the function form of the syntax.

The tables below show the allowable values for *`type`*, and provides an example for each type using `pi`.

Use these format types to switch between different output display formats for floating-point variables.

Type	Result
`short`	Scaled fixed point format, with 4 digits after the decimal point. For example, `3.1416`.

Type	Result
`long`	Scaled fixed point format with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, `3.141592653589793`.
`short e`	Floating point format, with 4 digits after the decimal point. For example, `3.1416e+000`.
`long e`	Floating point format, with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, `3.141592653589793e+000`.
`short g`	Best of fixed or floating point, with 4 digits after the decimal point. For example, `3.1416`.
`long g`	Best of fixed or floating point, with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, `3.14159265358979`.
`short eng`	Engineering format that has 4 digits after the decimal point, and a power that is a multiple of three. For example, `3.1416e+000`.
`long eng`	Engineering format that has exactly 16 significant digits and a power that is a multiple of three. For example, `3.14159265358979e+000`.

Use these format types to switch between different output display formats for all numeric variables.

Value for type	Result
`+`	+, -, blank
`bank`	Fixed dollars and cents. For example, `3.14`
`hex`	Hexadecimal (hexadecimal representation of a binary double-precision number). For example, `400921fb54442d18`
`rat`	Ratio of small integers. For example, `355/113`

Use these format types to affect the spacing in the display of all variables.

Value for type	Result	Example

`compact`	Suppresses excess line feeds to show more output in a single screen. Contrast with `loose`.	`theta = pi/2`
		`theta =`
		`1.5708`

`loose`	Adds linefeeds to make output more readable. Contrast with `compact`.	`theta = pi/2`
		
		`theta =`
		
		`1.5708`

Remarks

Computations on floating-point variables, namely `single` or `double`, are done in appropriate floating-point precision, no matter how those variables are displayed. Computations on integer variables are done natively in integer.

MATLAB always displays integer variables to the appropriate number of digits for the class. For example, MATLAB uses three digits to display numbers of type `int8` (i.e., -128:127). Setting `format` to `short` or `long` does not affect the display of integer variables.

The specified format applies only to the current MATLAB session. To maintain a format across sessions, use MATLAB preferences.

To see which `type` is currently in use, type

```
get(0,'Format')
```

To see if `compact` or `loose` formatting is currently selected, type

```
get(0,'FormatSpacing').
```

Examples

Example 1

Change the format to `long` by typing

```
format long
```

View the result for the value of `pi` by typing

```
pi
ans =
   3.14159265358979
```

View the current format by typing

```
get(0,'format')
ans =
   long
```

Set the format to `short e` by typing

```
format short e
```

or use the function form of the syntax

```
format('short','e')
```

Example 2

When the format is set to `short`, both `pi` and `single(pi)` display as 5-digit values:

```
format short

pi
ans =
    3.1416

single(pi)
ans =
    3.1416
```

Now set format to `long`, and `pi` displays a 15-digit value while `single(pi)` display an 8-digit value:

```
format long

pi
ans =
    3.14159265358979

single(pi)
ans =
    3.1415927
```

Example 3

Set the format to its default, and display the maximum values for integers and real numbers in MATLAB:

```
format

intmax('uint64')
ans =
   18446744073709551615

realmax
ans =
  1.7977e+308
```

Now change the format to hexadecimal, and display these same values:

```
format hex

intmax('uint64')
ans =
   ffffffffffffffff

realmax
ans =
   7fefffffffffffff
```

The hexadecimal display corresponds to the internal representation of the value. It is not the same as the hexadecimal notation in the C programming language.

Example 4

This example illustrates the `short eng` and `long eng` formats. The value assigned to variable `A` increases by a multiple of 10 each time through the `for` loop.

```
A = 5.123456789;

for k=1:10
   disp(A)
   A = A * 10;
end
```

The values displayed for `A` are shown here. The power of 10 is always a multiple of 3. The value itself is expressed in 5 or more digits for the `short eng` format, and in exactly 15 digits for `long eng`:

```
  format short eng                format long eng

     5.1235e+000                5.12345678900000e+000
    51.2346e+000               51.2345678900000e+000
   512.3457e+000              512.345678900000e+000
     5.1235e+003                5.12345678900000e+003
    51.2346e+003               51.2345678900000e+003
   512.3457e+003              512.345678900000e+003
     5.1235e+006                5.12345678900000e+006
    51.2346e+006               51.2345678900000e+006
   512.3457e+006              512.345678900000e+006
     5.1235e+009                5.12345678900000e+009
```

Algorithms

If the largest element of a matrix is larger than 10^3 or smaller than 10^{-3}, MATLAB applies a common scale factor for the short and long formats. The function `format +` displays +, -, and blank characters for positive, negative, and zero elements. `format hex` displays the hexadecimal representation of a binary double-precision number. `format rat` uses a continued fraction algorithm to approximate floating-point values by ratios of small integers. See `rat.m` for the complete code.

See Also

`disp`, `display`, `isnumeric`, `isfloat`, `isinteger`, `floor`, `sprintf`, `fprintf`, `num2str`, `rat`, `spy`

Purpose

Plot function between specified limits

Syntax

```
fplot(fun,limits)
fplot(fun,limits,LineSpec)
fplot(fun,limits,tol)
fplot(fun,limits,tol,LineSpec)
fplot(fun,limits,n)
fplot(fun,lims,...)
fplot(axes_handle,...)
[X,Y] = fplot(fun,limits,...)
```

Description

`fplot` plots a function between specified limits. The function must be of the form $y = f(x)$, where x is a vector whose range specifies the limits, and `y` is a vector the same size as `x` and contains the function's value at the points in x (see the first example). If the function returns more than one value for a given x, then y is a matrix whose columns contain each component of $f(x)$ (see the second example).

`fplot(fun,limits)` plots `fun` between the limits specified by `limits`. `limits` is a vector specifying the *x*-axis limits (`[xmin xmax]`), or the *x*- and *y*-axes limits, (`[xmin xmax ymin ymax]`).

`fun` must be

- The name of an M-file function
- A string with variable `x` that may be passed to `eval`, such as `'sin(x)'`, `'diric(x,10)'`, or `'[sin(x),cos(x)]'`
- A function handle for an M-file function or an anonymous function (see and for more information)

The function $f(x)$ must return a row vector for each element of vector x. For example, if $f(x)$ returns `[f1(x),f2(x),f3(x)]` then for input `[x1;x2]` the function should return the matrix

```
f1(x1) f2(x1) f3(x1)
f1(x2) f2(x2) f3(x2)
```

`fplot(fun,limits,`*`LineSpec`*`)` plots `fun` using the line specification `LineSpec`.

`fplot(fun,limits,tol)` plots `fun` using the relative error tolerance `tol` (the default is `2e-3`, i.e., 0.2 percent accuracy).

`fplot(fun,limits,tol,`*`LineSpec`*`)` plots `fun` using the relative error tolerance `tol` and a line specification that determines line type, marker symbol, and color. See `LineSpec` for more information.

`fplot(fun,limits,n)` with `n >= 1` plots the function with a minimum of `n+1` points. The default `n` is `1`. The maximum step size is restricted to be `(1/n)*(xmax-xmin)`.

`fplot(fun,lims,...)` accepts combinations of the optional arguments `tol`, `n`, and `LineSpec`, in any order.

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

`[X,Y] = fplot(fun,limits,...)` returns the abscissas and ordinates for `fun` in `X` and `Y`. No plot is drawn on the screen; however, you can plot the function using `plot(X,Y)`.

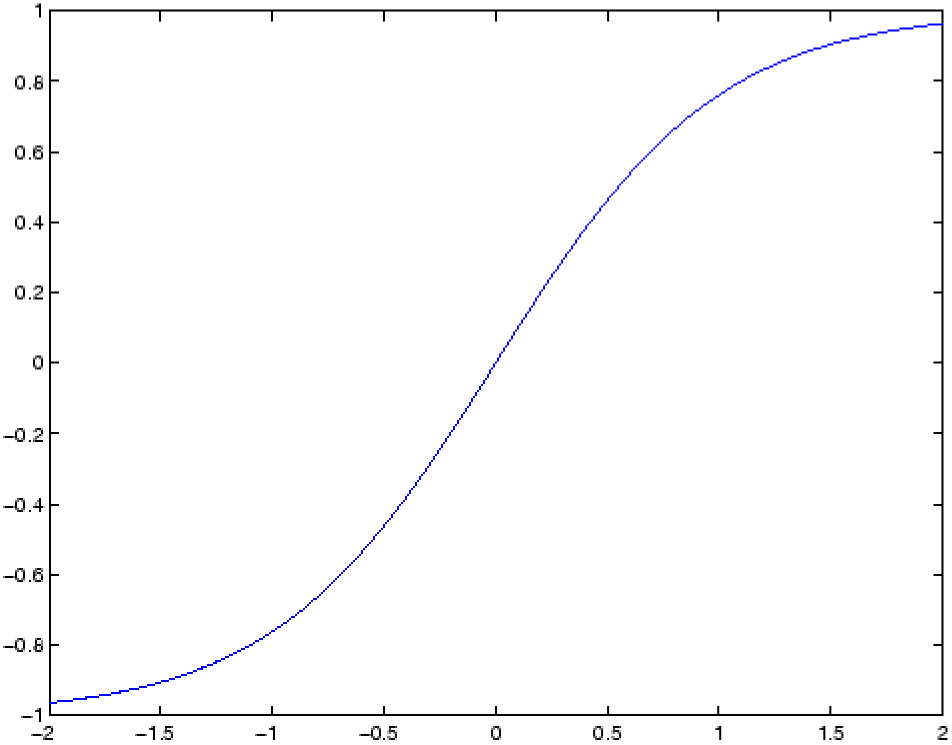
Remarks

`fplot` uses adaptive step control to produce a representative graph, concentrating its evaluation in regions where the function's rate of change is the greatest.

Examples

Plot the hyperbolic tangent function from -2 to 2:

```
fnch = @tanh;
fplot(fnch,[-2 2])
```



Create an M-file, `myfun`, that returns a two-column matrix:

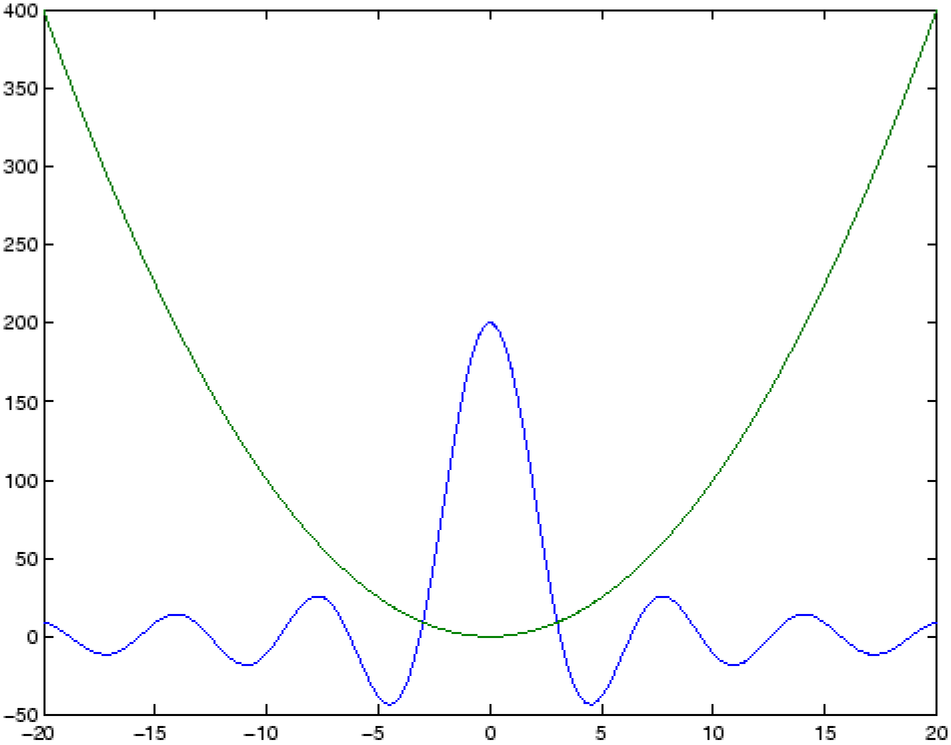
```
function Y = myfun(x)
Y(:,1) = 200*sin(x(:))./x(:);
Y(:,2) = x(:).^2;
```

Create a function handle pointing to `myfun`:

```
fh = @myfun;
```

Plot the function with the statement

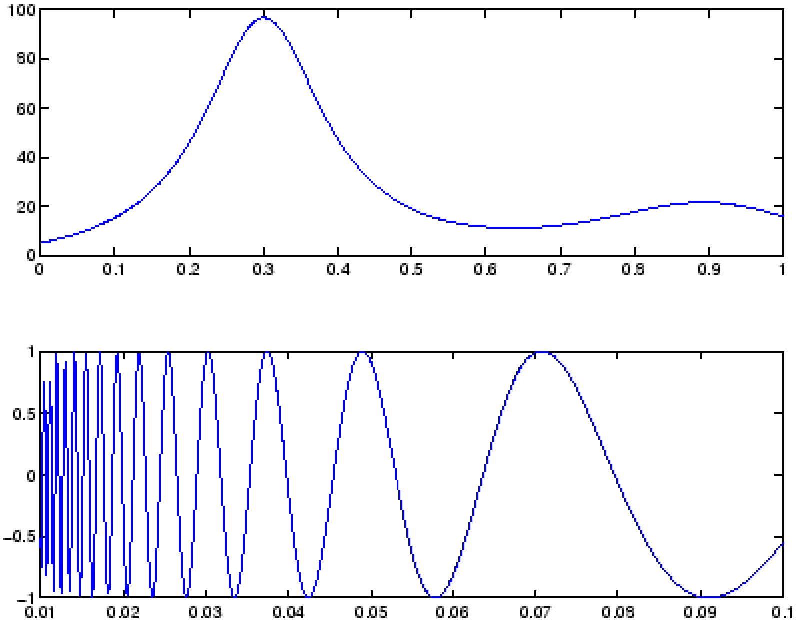
```
fplot(fh,[-20 20])
```



Additional Example

This example passes function handles to `fplot`, one created from a MATLAB function and the other created from an anonymous function.

```
hmp = @humps;
subplot(2,1,1);fplot(hmp,[0 1])
sn = @(x) sin(1./x);
subplot(2,1,2);fplot(sn,[.01 .1])
```



See Also `eval`, `ezplot`, `feval`, `LineSpec`, `plot`

"Function Plots" on page 1-94 for related functions

Purpose

Write data to text file

Syntax

```
fprintf(fileID, format, A, ...)
fprintf(format, A, ...)
count = fprintf(...)
```

Description

`fprintf(`*`fileID`*`, `*`format`*`, `*`A`*`, ...)` applies the *`format`* to array *`A`* and any additional array arguments in column order, and writes the data to a text file. `fprintf` uses the encoding scheme specified in the call to `fopen`.

`fprintf(`*`format`*`, `*`A`*`, ...)` formats data and displays the results on the screen.

`count`` = fprintf(...)` returns the number of bytes that `fprintf` writes.

Inputs

`fileID`

One of the following:

- An integer file identifier obtained from `fopen`.
- `1` for standard output (the screen).
- `2` for standard error.

Default: `1` (the screen)

`format`

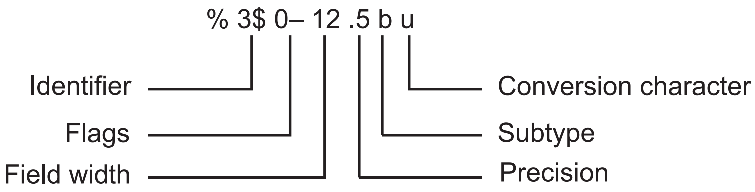
String in single quotation marks that describes the format of the output fields. Can include combinations of the following:

- 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
`\x`*N*	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

Value Type	Conversion	Details
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

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`

Value Type	**Conversion**	**Details**
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

Print multiple values and literal text to the screen:

```
B = [8.8  7.7 ; ...
     8800 7700];
fprintf('X is %4.2f meters or %8.3f mm\n', 9.9, 9900, B)
```

MATLAB displays:

```
X is 9.90 meters or 9900.000 mm
X is 8.80 meters or 8800.000 mm
X is 7.70 meters or 7700.000 mm
```

Explicitly convert double-precision values with fractions to integer values, and print to the screen:

```
a = [1.02 3.04 5.06];
fprintf('%d\n', round(a));
```

Write a short table of the exponential function to a text file called `exp.txt`:

```
x = 0:.1:1;
y = [x; exp(x)];

% open the file with write permission
fid = fopen('exp.txt', 'w');
fprintf(fid, '%6.2f %12.8f\n', y);
fclose(fid);

% view the contents of the file
type exp.txt
```

MATLAB import functions, all UNIX applications, and Microsoft Word and WordPad recognize `'\n'` as a newline indicator. However, if you plan to read the file with Microsoft Notepad, use `'\r\n'` to move to a new line when writing. For example, replace the previous call to `fprintf` with the following:

```
fprintf(fid, '%6.2f %12.8f\r\n', y);
```

On a Windows system, convert PC-style exponential notation (three digits in the exponent) to UNIX-style notation (two digits), and print data to a file:

```
a = [0.06 0.1 5 300]

% use sprintf to convert the numeric data to text, using %e
a_str = sprintf('%e\t',a);

% use strrep to replace exponent prefix with shorter version
a_str = strrep(a_str,'e+0','e+');
a_str = strrep(a_str,'e-0','e-');

% call fprintf to print the updated text strings
fid = fopen('newfile.txt','w');
fprintf(fid, '%s', a_str);
fclose(fid);

% view the contents of the file
type newfile.txt
```

Display a hyperlink (The MathWorks Web Site) on the screen:

```
site = 'http://www.mathworks.com';
title = 'The MathWorks Web Site';

fprintf('<a href = "%s">%s</a>\n', site, title)
```

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

`disp` | `fclose` | `ferror` | `fopen` | `fread` | `fscanf` | `fwrite` | `sprintf`

How To

-
-

Purpose

Write text to device

Syntax

```
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
```

Description

`fprintf(obj,'cmd')` writes the string `cmd` to the device connected to the serial port object, `obj`. The default format is `%s\n`. The write operation is synchronous and blocks the command-line until execution completes.

`fprintf(obj,'format','cmd')` writes the string using the format specified by *`format`*.

`fprintf(obj,'cmd','mode')` writes the string with command line access specified by *`mode`*. *`mode`* specifies if `cmd` is written synchronously or asynchronously.

`fprintf(obj,'format','cmd','mode')` writes the string using the specified format. *`format`* is a C language conversion specification.

You need an open connection from the serial port object, `obj`, to the device before performing read or write operations.

Use the `fopen` function to open a connection to the device. When `obj` has an open connection to the device it has a `Status` property value of `open`. Refer to for `fprintf` errors.

To understand the use of `fprintf` refer to and .

Inputs

`format`

ANSI C conversion specification includes these conversion characters.

Specifier	Description
`%c`	Single character
`%d` or `%i`	Decimal notation (signed)

Specifier	Description
`%e`	Exponential notation (using lowercase `e` as in `3.1415e+00`)
`%E`	Exponential notation (using uppercase `E` as in `3.1415E+00`)
`%f`	Fixed-point notation
`%g`	The more compact of `%e` or `%f`, as defined above. Insignificant zeros do not print.
`%G`	Same as `%g`, but using uppercase `E`
`%o`	Octal notation (unsigned)
`%s`	String of characters
`%u`	Decimal notation (unsigned)
`%x`	Hexadecimal notation (using lowercase letters `a`–`f`)
`%X`	Hexadecimal notation (using uppercase letters `A`–`F`)

`mode`

Specifies whether the string `cmd` is written synchronously or asynchronously:

- `sync`: `cmd` is written synchronously and the command line is blocked.
- `async`: `cmd` is written asynchronously and the command line is not blocked.

If *`mode`* is not specified, the write operation is synchronous.

If you specify asynchronous *`mode`*, when the write operation occurs:

- The `BytesToOutput` property value continuously updates to reflect the number of bytes in the output buffer.

- The MATLAB file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

Use the `TransferStatus` property to determine whether an asynchronous write operation is in progress.

For more information on synchronous and asynchronous write operations, see Controlling Access to the MATLAB Command Line.

Examples

Create a serial port object `s` and connect it to a Tektronix TDS 210 oscilloscope. Write the `RS232?` command with `fprintf`. `RS232?` instructs the scope to return serial port communications settings. This example works on a Windows platform.

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

Specify a format for the data that does not include the terminator, or configure the terminator to empty.

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

The default format for `fprintf` is `%s\n`. Therefore, the terminator specified by the `Terminator` property is automatically written. However, in some cases you might want to suppress writing the terminator.

Specify an array of formats and commands:

```
s = serial('COM1');
fopen(s)
fprintf(s,['ch:%d scale:%d'],[1 20e-3],'sync');
```

See Also

`fopen` | `fwrite` | `stopasync` | `BytesToOutput` | `OutputBufferSize` | `OutputEmptyFcn` | `Status` | `TransferStatus` | `ValuesSent`

Tutorials

- Writing and Reading Data
- Controlling Access to the MATLAB Command Line

Purpose

Return image data associated with movie frame

Syntax

`[X,Map] = frame2im(F)`

Description

`[X,Map] = frame2im(F)` returns the indexed image `X` and associated colormap `Map` from the single movie frame `F`. If the frame contains true-color data, the MxNx3 matrix `Map` is empty. The functions `getframe` and `im2frame` create a movie frame.

Example

Create and capture an image using`getframe` and `frame2im`:

```
peaks                          %Make figure
f = getframe;                  %Capture screen shot
[im,map] = frame2im(f);        %Return associated image data
if isempty(map)                %Truecolor system
  rgb = im;
else                           %Indexed system
  rgb = ind2rgb(im,map);       %Convert image data
end
```

See Also

`getframe`, `im2frame`, `movie`

fread

Purpose

Read data from binary file

Syntax

```
A = fread(fileID)
A = fread(fileID, sizeA)
A = fread(fileID, sizeA, precision)
A = fread(fileID, sizeA, precision, skip)
A = fread(fileID, sizeA, precision, skip, machineformat)
[A, count] = fread(...)
```

Description

`A = fread(fileID)` reads data from a binary file into column vector `A` and positions the file pointer at the end-of-file marker.

`A = fread(fileID, sizeA)` reads `sizeA` elements into `A` and positions the file pointer after the last element read. `sizeA` can be an integer, or can have the form `[m,n]`.

`A = fread(fileID, sizeA, precision)` interprets values in the file according to the form and size described by the `precision`. The `sizeA` parameter is optional.

`A = fread(fileID, sizeA, precision, skip)` skips `skip` bytes after reading each value. If `precision` is `bitn` or `ubitn`, specify `skip` in bits. The `sizeA` parameter is optional.

`A = fread(fileID, sizeA, precision, skip, machineformat)` reads data with the specified `machineformat`. The `sizeA` and `skip` parameters are optional.

`[A, count] = fread(...)` returns the number of elements `fread` reads into `A`.

Inputs

`fileID`

An integer file identifier obtained from `fopen`.

`sizeA`

Dimensions of the output array `A`. Specify in one of the following forms:

`inf`	Column vector with the number of elements in the file. (default)
n	Column vector with *n* elements. If the file has fewer than *n* elements, `fread` pads *A* with zeros.
`[`*m*`,`*n*`]`	*m*-by-*n* matrix, filled in column order. If the file has fewer than *m*n* elements, `fread` pads *A* with zeros. *n* can be `inf`, but *m* cannot.

precision

String that specifies the form and size of the values to read. Optionally includes the class for the output matrix *A*.

Use one of the following forms:

'*source*'	Specifies class of input values. Output matrix *A* is class `double`. Example: `'int16'`
'*source=>output*'	Specifies classes of input and output. Example: `'int8=>char'`
'**source*'	Output has the same class as input. Example: `'*uint8'` For `'bit`*n*`'` or `'ubit`*n*`'` precisions, output has the smallest class that can contain the input. Example: `'*ubit18'` is equivalent to `'ubit18=>uint32`’
'*N*source*' OR '*N*source=>output*'	Valid if you specify a *skip* parameter. Read *N* values before each skip. Example: `'4*int8'`

The following table shows possible values for *source* and *output*.

Value Type	**Precision**	**Bits (Bytes)**

Integers, unsigned	`uint`
	`uint8`
	`uint16`
	`uint32`
	`uint64`
	`uchar`
	`unsigned char`
	`ushort`
	`ulong`
	`ubit`*n*
		32 (4)
		8 (1)
		16 (2)
		32 (4)
		64 (8)
		8 (1)
		8 (1)
		16 (2)
		system-dependent
		$1 \leq n \leq 64$

Integers, signed	`int`
	`int8`
	`int16`
	`int32`
	`int64`
	`integer*1`
	`integer*2`
	`integer*3`
	`integer*4`
	`schar`
	`signed char`
	`short`
	`long`
	`bit`*n*
		32 (4)
		8 (1)
		16 (2)
		32 (4)
		64 (8)
		8 (1)
		16 (2)
		32 (4)
		64 (8)
		8 (1)
		8 (1)
		16 (2)
		system-dependent
		$1 \leq n \leq 64$

Value Type	Precision	Bits (Bytes)

Floating-point numbers	`single`
	`double`
	`float`
	`float32`
	`float64`
	`real*4`
	`real*8`
		`32 (4)`
		`64 (8)`
		`32 (4)`
		`32 (4)`
		`64 (8)`
		`32 (4)`
		`64 (8)`

Characters	`char*1` `char`	`8 (1)` Depends on the encoding scheme associated with the file. Set encoding with `fopen`.

`long` and `ulong` are 32 bits on 32-bit systems, and 64 bits on 64-bit systems.

For most *`source`* precisions, if `fread` reaches the end of the file before reading a complete element, `fread` does not return a value for the final element. However, if *`source`* is `bit`*`n`* or `ubit`*`n`*, then `fread` returns a partial result for the final element.

Default: `'uint8=>double'`

`skip`

Number of bytes to skip after reading each value. If you specify a *`precision`* of `bit`*`n`* or `ubit`*`n`*, specify *`skip`* in bits. Use this parameter to read data from noncontiguous fields in fixed-length records.

Default: 0

`machineformat`

String that specifies the order for reading bytes within the file. For `bit`*`n`* and `ubit`*`n`* precisions, specifies the order for reading bits within a byte. Specify *`machineformat`* to read and write to the file on different systems, or to read parts of a byte in a specific order.

Possible values are:

`'n'` or `'native'`	The byte ordering that your system uses (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering
`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit data type

Windows systems use little-endian ordering, and most UNIX systems use big-endian ordering, for both bytes and bits. Solaris systems use big-endian ordering for bytes, but little-endian ordering for bits.

Outputs

`A`

A column vector, unless you specify *`sizeA`* with the form `[`*`m`*`,`*`n`*`]`. Data in *`A`* is class `double` unless you specify a different class in the *`precision`* argument.

`count`

The number of elements that `fread` successfully reads.

Examples

Read the contents of a file:

```
% Create the file
fid = fopen('magic5.bin', 'w');
fwrite(fid, magic(5));
fclose(fid);

% Read the contents back into an array
fid = fopen('magic5.bin');
```

```
m5 = fread(fid, [5, 5], '*uint8');
fclose(fid);
```

Simulate the `type` function with `fread`, to display the contents of a text file:

```
fid = fopen('fread.m');

% read the entire file as characters
% transpose so that F is a row vector
F = fread(fid, '*char')'

fclose(fid);
```

If you do not specify the precision, `fread` applies the default `uint8=>double`:

```
fid = fopen('fread.m');
F_nums = fread(fid, 6)'  % read the first 6 bytes ('%FREAD')
fclose(fid);
```

This code returns

```
F_nums =
    37    70    82    69    65    68
```

Read selected rows or columns from a file:

```
% Create a file with values from 1 to 9
fid = fopen('nine.bin', 'w');
alldata = reshape([1:9],3,3);
fwrite(fid, alldata);
fclose(fid);

% Read the first six values into two columns
```

```
fid = fopen('nine.bin');
two_cols = fread(fid, [3, 2]);

% Return to the beginning of the file
frewind(fid);

% Read two values at a time, skip one
% Returns six values into two rows
% (first two rows of 'alldata')

two_rows = fread(fid, [2, 3], '2*uint8', 1);

% Close the file
fclose(fid);
```

Specify *`machineformat`* to read separate digits of binary coded decimal (BCD) values correctly:

```
% Create a file with BCD values
str = ['AB'; 'CD'; 'EF'; 'FA'];

fid = fopen('bcd.bin', 'w');
fwrite(fid, hex2dec(str), 'ubit8');
fclose(fid);

% If you read one byte at a time,
% no need to specify machine format
fid = fopen('bcd.bin');

onebyte = fread(fid, 4, '*ubit8');
disp('Correct data, read with ubit8:')
disp(dec2hex(onebyte))

% However, if you read 4 bits on a little-endian
% system, your results appear in the wrong order
frewind(fid);     % return to beginning of file
```

```
part_err = fread(fid, 8, '*ubit4');
disp('Incorrect data on little-endian systems, ubit4:')
disp(dec2hex(part_err))

% Specify a big-endian format for correct results
frewind(fid);

part_corr = fread(fid, 8, '*ubit4', 'ieee-be');
disp('Correct result, ubit4:')
disp(dec2hex(part_corr))

fclose(fid);
```

See Also

`fclose` | `ferror` | `fgetl` | `fgets` | `fopen` | `fscanf` | `fprintf` | `fwrite`

How To

-
-

fread (serial)

Purpose

Read binary data from device

Syntax

```
A = fread(obj)
A = fread(obj,size,'precision')
[A,count] = fread(...)
[A,count,msg] = fread(...)
```

Description

`A = fread(obj)` and A = fread(obj,size) read binary data from the device connected to the serial port object, `obj`, and returns the data to `A`. The maximum number of values to read is specified by `size`. If `size` is not specified, the maximum number of values to read is determined by the object's `InputBufferSize` property. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. A value is defined as a byte multiplied by the *`precision`* (see below).

`A = fread(obj,`*`size`*`,'`*`precision`*`')` reads binary data with precision specified by *`precision`*.

`precision` controls the number of bits read for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified, `uchar` (an 8-bit unsigned character) is used. By default, numeric values are returned in double-precision arrays. The supported values for *`precision`* are listed below in Remarks.

`[A,count] = fread(...)` returns the number of values read to `count`.

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

Remarks

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

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

The `ValuesReceived` property value is increased by the number of values read, each time `fread` is issued.

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

```
help serial/fread
```

Rules for Completing a Binary Read Operation

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

- The specified number of values are read.
- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used for binary read operations.

Supported Precisions

The supported values for *`precision`* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character

Data Type	Precision	Interpretation
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

See Also

Functions

`fgetl`, `fgets`, `fopen`, `fscanf`

Properties

`BytesAvailable`, `BytesAvailableFcn`, `InputBufferSize`, `Status`, `Terminator`, `ValuesReceived`

Purpose

Facets referenced by only one simplex

Syntax

```
FF = freeBoundary(TR)
[FF XF] = freeBoundary(TR)
```

Description

`FF = freeBoundary(TR)` returns a matrix `FF` that represents the free boundary facets of the triangulation. A facet is on the free boundary if it is referenced by only one simplex (triangle/tetrahedron, etc). `FF` is of size `m`-by-`n`, where m is the number of boundary facets and n is the number of vertices per facet. The vertices of the facets index into the array of points representing the vertex coordinates `TR.X`. The array `FF` could be empty as in the case of a triangular mesh representing the surface of a sphere.

`[FF XF] = freeBoundary(TR)` returns a matrix of free boundary facets

Inputs

`TR`	Triangulation representation.

Outputs

`FF`	`FF` that has vertices defined in terms of a compact array of coordinates `XF`.
`XF`	`XF` is of size `m`-by-`ndim` where `m` is the number of free facets, and `ndim` is the dimension of the space where the triangulation resides

Definitions

A *simplex* is a triangle/tetrahedron or higher-dimensional equivalent. A *facet* is an edge of a triangle or a face of a tetrahedron.

Examples

Example 1

Use `TriRep` to compute the boundary triangulation of an imported triangulation.

Load a 3-D triangulation:

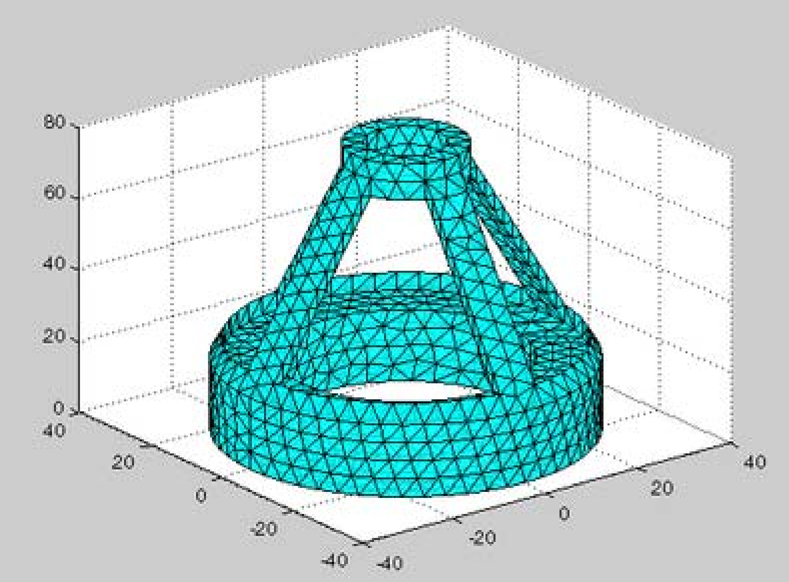
```
load tetmesh;
trep = TriRep(tet, X);
```

Compute the boundary triangulation:

```
[tri xf] = freeBoundary(trep);
```

Plot the boundary triangulation:

```
trisurf(tri, xf(:,1),xf(:,2),xf(:,3), ...
     'FaceColor','cyan', 'FaceAlpha', 0.8);
```



Example 2

Perform a direct query of a 2-D triangulation created with `DelaunayTri`.

Plot the mesh:

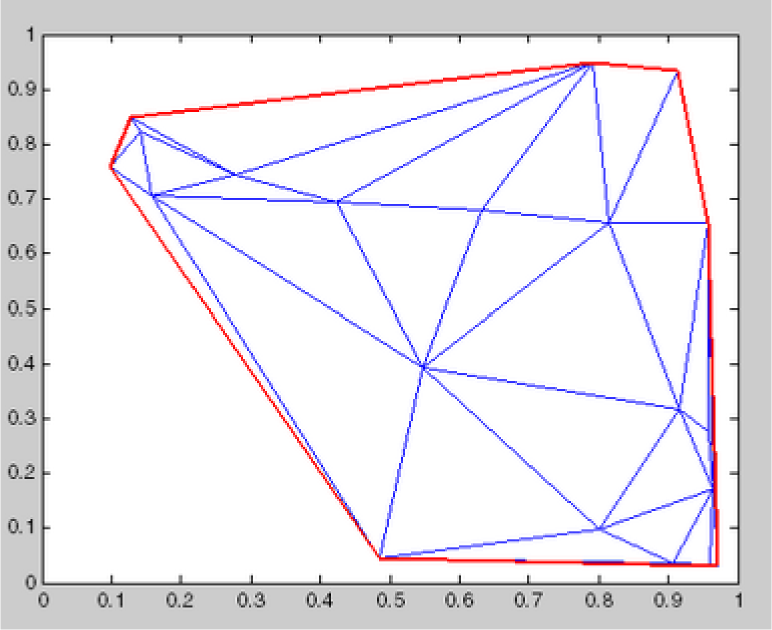
```
x = rand(20,1);
y = rand(20,1);
```

```
dt = DelaunayTri(x,y);
fe = freeBoundary(dt)';
triplot(dt);
hold on;
```

Display the free boundary edges in red:

```
plot(x(fe), y(fe), '-r', 'LineWidth',2) ;
hold off;
```

In this instance the free edges correspond to the convex hull of `(x, y)`.



See Also

`DelaunayTri`
`DelaunayTri.convexHull`
`TriRep.featureEdges`
`TriRep.faceNormals`

freqspace

Purpose

Frequency spacing for frequency response

Syntax

```
[f1,f2] = freqspace(n)
[f1,f2] = freqspace([m n])
[x1,y1] = freqspace(...,'meshgrid')
f = freqspace(N)
f = freqspace(N,'whole')
```

Description

`freqspace` returns the implied frequency range for equally spaced frequency responses. `freqspace` is useful when creating desired frequency responses for various one- and two-dimensional applications.

`[f1,f2] = freqspace(n)` returns the two-dimensional frequency vectors `f1` and `f2` for an `n`-by-`n` matrix.

For `n` odd, both `f1` and `f2` are `[-n+1:2:n-1]/n`.

For `n` even, both `f1` and `f2` are `[-n:2:n-2]/n`.

`[f1,f2] = freqspace([m n])` returns the two-dimensional frequency vectors `f1` and `f2` for an `m`-by-`n` matrix.

`[x1,y1] = freqspace(...,'meshgrid')` is equivalent to

```
[f1,f2] = freqspace(...);
[x1,y1] = meshgrid(f1,f2);
```

`f = freqspace(N)` returns the one-dimensional frequency vector `f` assuming `N` evenly spaced points around the unit circle. For `N` even or odd, `f` is `(0:2/N:1)`. For `N` even, `freqspace` therefore returns `(N+2)/2` points. For `N` odd, it returns `(N+1)/2` points.

`f = freqspace(N,'whole')` returns `N` evenly spaced points around the whole unit circle. In this case, `f` is `0:2/N:2*(N-1)/N`.

See Also

`meshgrid`

Purpose

Move file position indicator to beginning of open file

Syntax

`frewind(fileID)`

Description

`frewind(fileID)` sets the file position indicator to the beginning of a file. *`fileID`* is an integer file identifier obtained from `fopen`.

If the file is on a tape device and the rewind operation fails, `frewind` does not return an error message.

Alternatives

`frewind(fileID)` is equivalent to:

```
fseek(fileID, 0, 'bof');
```

See Also

`fclose` | `feof` | `ferror` | `fopen` | `fseek` | `ftell`

fscanf

Purpose

Read data from a text file

Syntax

```
A = fscanf(fileID, format)
A = fscanf(fileID, format, sizeA)
[A, count] = fscanf(...)
```

Description

`A = fscanf(fileID, format)` reads and converts data from a text file into array *A* in column order. To convert, `fscanf` uses the *format* and the encoding scheme associated with the file. To set the encoding scheme, use `fopen`. The `fscanf` function reapplies the *format* throughout the entire file, and positions the file pointer at the end-of-file marker. If `fscanf` cannot match the *format* to the data, it reads only the portion that matches into *A* and stops processing.

`A = fscanf(fileID, format, sizeA)` reads *sizeA* elements into *A*, and positions the file pointer after the last element read. *sizeA* can be an integer, or can have the form `[m,n]`.

`[A, count] = fscanf(...)` returns the number of elements that `fscanf` successfully reads.

Inputs

fileID

Integer file identifier obtained from `fopen`.

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.

Field Type	Specifier	Details
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 file. (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.

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

The number of elements `fscanf` reads into *A*.

Examples

Read the contents of a file. `fscanf` reuses the *`format`* throughout the file, so you do not need a control loop:

```
% Create a file with an exponential table
x = 0:.1:1;
y = [x; exp(x)];

fid = fopen('exp.txt', 'w');
fprintf(fid, '%6.2f %12.8f\n', y);
fclose(fid);

% Read the data, filling A in column order
% First line of the file:
%    0.00    1.00000000

fid = fopen('exp.txt');
A = fscanf(fid, '%g %g', [2 inf]);
fclose(fid);

% Transpose so that A matches
% the orientation of the file
A = A';
```

Skip specific characters in a file, and return only numeric values:

```
% Create a file with temperatures
tempstr = '78 F 72 F 64 F 66 F 49 F';

fid = fopen('temperature.dat', 'w+');
fprintf(fid, '%s', tempstr);

% Return to the beginning of the file
frewind(fid);

% Read the numbers in the file, skipping the units
% num_temps is a numeric column vector
```

```
degrees = char(176);
num_temps = fscanf(fid, ['%d' degrees 'F']);

fclose(fid);
```

See Also

`fclose` | `ferror` | `fgetl` | `fgets` | `fopen` | `fprintf` | `fread` | `fwrite` | `sscanf` | `textscan`

How To

-
-

fscanf (serial)

Purpose Read data from device, and format as text

Syntax

```
A = fscanf(obj)
A = fscanf(obj,'format')
A = fscanf(obj,'format',size)
[A,count] = fscanf(...)
[A,count,msg] = fscanf(...)
```

Description `A = fscanf(obj)` reads data from the device connected to the serial port object, `obj`, and returns it to `A`. The data is converted to text using the `%c` format.

`A = fscanf(obj,'`*`format`*`')` reads data and converts it according to *`format`*. *`format`* is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sscanf` file I/O format specifications or a C manual for more information.

`A = fscanf(obj,'`*`format`*`',size)` reads the number of values specified by `size`. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. If `size` is not of the form `[m,n]`, and a character conversion is specified, then `A` is returned as a row vector. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. An ASCII value is one byte.

`[A,count] = fscanf(...)` returns the number of values read to `count`.

`[A,count,msg] = fscanf(...)` returns a warning message to `msg` if the read operation did not complete successfully.

Remarks

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

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

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

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

```
help serial/fscanf
```

Rules for Completing a Read Operation with fscanf

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

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The number of values specified by `size` is read.
- The input buffer is filled (unless `size` is specified)

Example

Create the serial port object `s` and connect `s` to a Tektronix TDS 210 oscilloscope, which is displaying sine wave. This example works on a Windows platform.

```
s = serial('COM1');
fopen(s)
```

Use the `fprintf` function to configure the scope to measure the peak-to-peak voltage of the sine wave, return the measurement type, and return the peak-to-peak voltage.

```
fprintf(s,'MEASUREMENT:IMMED:TYPE PK2PK')
fprintf(s,'MEASUREMENT:IMMED:TYPE?')
fprintf(s,'MEASUREMENT:IMMED:VALUE?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data associated with the two query commands is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    21
```

Use `fscanf` to read the measurement type. The operation will complete when the first terminator is read.

```
meas = fscanf(s)
meas =
PK2PK
```

Use `fscanf` to read the peak-to-peak voltage as a floating-point number, and exclude the terminator.

```
pk2pk = fscanf(s,'%e',14)
pk2pk =
    2.0200
```

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

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

See Also

Functions

`fgetl`, `fgets`, `fopen`, `fread`, `strread`

Properties

`BytesAvailable`, `BytesAvailableFcn`, `InputBufferSize`, `Status`, `Terminator`, `Timeout`

Purpose

Move to specified position in file

Syntax

```
fseek(fileID, offset, origin)
status = fseek(fileID, offset, origin)
```

Description

`fseek(fileID, offset, origin)` sets the file position indicator `offset` bytes from `origin` in the specified file.

`status = fseek(fileID, offset, origin)` returns `0` when the operation is successful. Otherwise, it returns `-1`.

Inputs

`fileID`

Integer file identifier obtained from `fopen`.

`offset`

Number of bytes to move from `origin`. Can be positive, negative, or zero. The *n* bytes of a given file are in positions `0` through *n*`-1`.

`origin`

Starting location in the file:

`'bof'` or `-1`	Beginning of file
`'cof'` or `0`	Current position in file
`'eof'` or `1`	End of file

Examples

Copy 5 bytes from the file `test1.dat`, starting at the tenth byte, and append to the end of `test2.dat`:

```
% Create files test1.dat and test2.dat
% Each character uses 8 bits (1 byte)

fid1 = fopen('test1.dat', 'w+');
fwrite(fid1, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ');
```

```
fid2 = fopen('test2.dat', 'w+');
fwrite(fid2, 'Second File');

% Seek to the 10th btye ('J'), read 5
fseek(fid1, 9, 'bof');
A = fread(fid1, 5, 'uint8=>char');
fclose(fid1);

% Append to test2.dat
fseek(fid2, 0, 'eof');
fwrite(fid2, A);
fclose(fid2);
```

Alternatives

To move to the beginning of a file, call

```
frewind(fileID)
```

This call is identical to

```
fseek(fileID, 0, 'bof')
```

See Also

`fclose` | `feof` | `ferror` | `fopen` | `frewind` | `ftell`

How To

-

Purpose

Position in open file

Syntax

`position = ftell(fileID)`

Description

`position = ftell(fileID)` returns the current position in the specified file. `position` is a zero-based integer that indicates the number of bytes from the beginning of the file. If the query is unsuccessful, `position` is `-1`. `fileID` is an integer file identifier obtained from `fopen`.

See Also

`fclose` | `feof` | `ferror` | `fopen` | `frewind` | `fseek`

Purpose

Connect to FTP server, creating FTP object

Syntax

```
f = ftp('host','username','password')
```

Description

`f = ftp('host','username','password')` connects to the FTP server, `host`, creating the FTP object, `f`. If a user name and password are not required for an anonymous connection, only use the `host` argument. Specify an alternate port by separating it from `host` using a colon (:). After running `ftp`, perform file operation functions on the FTP object, `f`, using methods such as `cd` and others listed under "See Also." When you're finished using the server, run `close (ftp)` to close the connection.

FTP is not a secure protocol; others can see your user name and password.

The `ftp` function is based on code from the Apache Jakarta Project.

Examples

Connect Without User Name

Connect to `ftp.mathworks.com`, which does not require a user name or password. Assign the resulting FTP object to `tmw`. You can access this FTP site to experiment with the FTP functions.

```
tmw=ftp('ftp.mathworks.com')


tmw =
 FTP Object
  host: ftp.mathworks.com
  user: anonymous
   dir: /
  mode: binary
```

Connect to Specified Port

To connect to port 34, type:

```
tmw=ftp('ftp.mathworks.com:34')
```

Connect with User Name

Connect to `ftp.testsite.com` and assign the resulting FTP object to `test`.

```
test=ftp('ftp.testsite.com','myname','mypassword')

test =
 FTP Object
  host: ftp.testsite.com
  user: myname
   dir: /
  mode: binary
  myname@ftp.testsite.com
  /
```

See Also

`ascii` | `binary` | `cd (ftp)` | `close (ftp)` | `delete (ftp)` | `dir (ftp)` | `mget` | `mkdir (ftp)` | `mput` | `rename` | `rmdir (ftp)`

How To

-

full

Purpose

Convert sparse matrix to full matrix

Syntax

```
A = full(S)
```

Description

`A = full(S)` converts a sparse matrix `S` to full storage organization. If `S` is a full matrix, it is left unchanged. If `A` is full, `issparse(A)` is `0`.

Remarks

Let `X` be an `m`-by-`n` matrix with `nz = nnz(X)` nonzero entries. Then `full(X)` requires space to store `m*n` real numbers while `sparse(X)` requires space to store `nz` real numbers and `(nz+n)` integers.

On most computers, a real number requires twice as much storage as an integer. On such computers, `sparse(X)` requires less storage than `full(X)` if the density, `nnz/prod(size(X))`, is less than one third. Operations on sparse matrices, however, require more execution time per element than those on full matrices, so density should be considerably less than two-thirds before sparse storage is used.

Examples

Here is an example of a sparse matrix with a density of about two-thirds. `sparse(S)` and `full(S)` require about the same number of bytes of storage.

```
S = sparse(+(rand(200,200) < 2/3));
A = full(S);
whos
Name    Size    Bytes     Class
  A    200X200 320000 double array
  S    200X200 318432 double array (sparse)
```

See Also

`issparse`, `sparse`

Purpose

Build full file name from parts

Syntax

```
f = fullfile(folderName1, folderName2, ..., fileName)
```

Description

`f = fullfile(folderName1, folderName2, ..., fileName)` builds a full file specification `f` from the folders and file name specified. Input arguments `folderName1`, `folderName2`, etc. and `fileName` are each strings enclosed in single quotation marks. The output of `fullfile` is conceptually equivalent to

```
f = [folderName1 filesep folderName2 filesep ... filesep filename]
```

except that care is taken to handle the cases when the folders begin or end with a file separator.

Examples

To create the full file name from a disk name, folders, and filename,

```
f = fullfile('C:', 'Applications', 'matlab', 'myfun.m')
f =
C:\Applications\matlab\myfun.m
```

The following examples both produce the same result on UNIX[7] platforms, but only the second one works on all platforms.

```
fullfile(matlabroot, 'toolbox/matlab/general/Contents.m')
fullfile(matlabroot, 'toolbox', 'matlab', 'general', ...
         'Contents.m')
```

See Also `fileparts`, `filesep`, `path`, `pathsep`, `genpath`

---

[7] UNIX is a registered trademark of The Open Group in the United States and other countries

Purpose

Construct function name string from function handle

Syntax

`func2str(fhandle)`

Description

`func2str(fhandle)` constructs a string `s` that holds the name of the function to which the function handle `fhandle` belongs.

When you need to perform a string operation, such as compare or display, on a function handle, you can use `func2str` to construct a string bearing the function name.

The `func2str` command does not operate on nonscalar function handles. Passing a nonscalar function handle to `func2str` results in an error.

Examples

Example 1

Convert a `sin` function handle to a string:

```
fhandle = @sin;

func2str(fhandle)
ans =
    sin
```

Example 2

The `catcherr` function shown here accepts function handle and data arguments and attempts to evaluate the function through its handle. If the function fails to execute, `catcherr` uses `sprintf` to display an error message giving the name of the failing function. The function name must be a string for `sprintf` to display it. The code derives the function name from the function handle using `func2str`:

```
function catcherr(func, data)
try
    ans = func(data);
    disp('Answer is:');
    ans
catch
```

```
    disp(sprintf('Error executing function ''%s''\n', ...
       func2str(func)))
end
```

The first call to `catcherr` passes a handle to the `round` function and a valid data argument. This call succeeds and returns the expected answer. The second call passes the same function handle and an improper data type (a MATLAB structure). This time, `round` fails, causing `catcherr` to display an error message that includes the failing function name:

```
catcherr(@round, 5.432)
ans =
Answer is 5

xstruct.value = 5.432;
catcherr(@round, xstruct)
Error executing function "round"
```

See Also

`function_handle`, `str2func`, `functions`

Purpose

Declare M-file function

Syntax

```
function [out1, out2, ...] = funname(in1, in2, ...)
```

Description

`function [out1, out2, ...]  = funname(in1, in2, ...)` defines function `funname` that accepts inputs `in1`, `in2`, etc. and returns outputs `out1`, `out2`, etc.

You add new functions to the MATLAB vocabulary by expressing them in terms of existing functions. The existing commands and functions that compose the new function reside in a text file called an *M-file*.

M-files can be either *scripts* or *functions*. Scripts are simply files containing a sequence of MATLAB statements. Functions make use of their own local variables and accept input arguments.

The name of an M-file begins with an alphabetic character and has a filename extension of `.m`. The M-file name, less its extension, is what MATLAB searches for when you try to use the script or function.

A line at the top of a function M-file contains the syntax definition. The name of a function, as defined in the first line of the M-file, should be the same as the name of the file without the `.m` extension.

The variables within the body of the function are all local variables.

A *subfunction*,visible only to the other functions in the same file, is created by defining a new function with the `function` keyword after the body of the preceding function or subfunction. Subfunctions are not visible outside the file where they are defined.

You can terminate any function with an `end` statement but, in most cases, this is optional. `end` statements are required only in M-files that employ one or more nested functions. Within such an M-file, *every* function (including primary, nested, private, and subfunctions) must be terminated with an `end` statement. You can terminate any function type with `end`, but doing so is not required unless the M-file contains a nested function.

Functions normally return when the end of the function is reached. Use a `return` statement to force an early return.

When MATLAB does not recognize a function by name, it searches for a file of the same name on disk. If the function is found, MATLAB compiles it into memory for subsequent use. The section in the MATLAB Programming Fundamentals documentation explains how MATLAB interprets variable and function names that you enter, and also covers the precedence used in function dispatching.

When you call an M-file function from the command line or from within another M-file, MATLAB parses the function and stores it in memory. The parsed function remains in memory until cleared with the `clear` command or you `quit` MATLAB. The `pcode` command performs the parsing step and stores the result on the disk as a P-file to be loaded later.

Examples

Example 1

The existence of a file on disk called `stat.m` containing this code defines a new function called `stat` that calculates the mean and standard deviation of a vector:

```
function [mean,stdev] = stat(x)
n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2/n));
```

Example 2

`avg` is a subfunction within the file `stat.m`:

```
function [mean,stdev] = stat(x)
n = length(x);
mean = avg(x,n);
stdev = sqrt(sum((x-avg(x,n)).^2)/n);

function mean = avg(x,n)
mean = sum(x)/n;
```

See Also

`nargin`, `nargout`, `pcode`, `varargin`, `varargout`, `what`

Purpose

Handle used in calling functions indirectly

Syntax

```
handle = @functionname
handle = @(arglist)anonymous_function
```

Description

`handle = @functionname` returns a handle to the specified MATLAB function.

A function handle is a MATLAB value that provides a means of calling a function indirectly. You can pass function handles in calls to other functions (often called *function functions*). You can also store function handles in data structures for later use (for example, as Handle Graphics callbacks). A function handle is one of the standard MATLAB data types.

At the time you create a function handle, the function you specify must be on the MATLAB path and in the current scope. This condition does not apply when you evaluate the function handle. You can, for example, execute a subfunction from a separate (out-of-scope) M-file using a function handle as long as the handle was created within the subfunction's M-file (in-scope).

`handle = @(arglist)anonymous_function` constructs an anonymous function and returns a `handle` to that function. The body of the function, to the right of the parentheses, is a single MATLAB statement or command. `arglist` is a comma-separated list of input arguments. Execute the function by calling it by means of the function handle, `handle`.

Remarks

The function handle is a standard MATLAB data type. As such, you can manipulate and operate on function handles in the same manner as on other MATLAB data types. This includes using function handles in structures and cell arrays:

```
S.a = @sin;  S.b = @cos;  S.c = @tan;
C = {@sin, @cos, @tan};
```

However, standard matrices or arrays of function handles are not supported:

```
A = [@sin, @cos, @tan];        % This is not supported
```

For nonoverloaded functions, subfunctions, and private functions, a function handle references just the one function specified in the `@functionname` syntax. When you evaluate an overloaded function by means of its handle, the arguments the handle is evaluated with determine the actual function that MATLAB dispatches to.

Use `isa(h, 'function_handle')` to see if variable `h` is a function handle.

Examples

Example 1 — Constructing a Handle to a Named Function

The following example creates a function handle for the `humps` function and assigns it to the variable `fhandle`.

```
fhandle = @humps;
```

Pass the handle to another function in the same way you would pass any argument. This example passes the function handle just created to `fminbnd`, which then minimizes over the interval `[0.3, 1]`.

```
x = fminbnd(fhandle, 0.3, 1)
x =
    0.6370
```

The `fminbnd` function evaluates the `@humps` function handle. A small portion of the `fminbnd` M-file is shown below. In line 1, the `funfcn` input parameter receives the function handle `@humps` that was passed in. The statement, in line 113, evaluates the handle.

```
1    function [xf,fval,exitflag,output] = ...
         fminbnd(funfcn,ax,bx,options,varargin)
            .
            .
            .
```

```
113  fx = funfcn(x,varargin{:});
```

Example 2 — Constructing a Handle to an Anonymous Function

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `sqr`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
   25
```

Because `sqr` is a function handle, you can pass it in an argument list to other functions. The code shown here passes the `sqr` anonymous function to the MATLAB `quad` function to compute its integral from zero to one:

```
quad(sqr, 0, 1)
ans =
   0.3333
```

See Also

`str2func`, `func2str`, `functions`, `isa`

Purpose

Information about function handle

Syntax

```
S = functions(funhandle)
```

Description

`S = functions(funhandle)` returns, in MATLAB structure `S`, the function name, type, filename, and other information for the function handle stored in the variable `funhandle`.

`functions` does not operate on nonscalar function handles. Passing a nonscalar function handle to `functions` results in an error.

Caution The `functions` function is provided for querying and debugging purposes. Because its behavior may change in subsequent releases, you should not rely upon it for programming purposes.

This table lists the standard fields of the return structure.

Field Name	Field Description
`function`	Function name
`type`	Function type (e.g., simple, overloaded)
`file`	The file to be executed when the function handle is evaluated with a nonoverloaded data type

Examples

Example 1

To obtain information on a function handle for the `poly` function, type

```
f = functions(@poly)
f =
    function: 'poly'
        type: 'simple'
        file:  '$matlabroot\toolbox\matlab\polyfun\poly.m'
```

(The term `$matlabroot` used in this example stands for the file specification of the directory in which MATLAB software is installed for your system. Your output will display this file specification.)

Access individual fields of the returned structure using dot selection notation:

```
f.type
ans =
   simple
```

Example 2

The function `get_handles` returns function handles for a subfunction and private function in output arguments `s` and `p` respectively:

```
function [s, p] = get_handles
s = @mysubfun;
p = @myprivatefun;
%
function mysubfun
disp 'Executing subfunction mysubfun'
```

Call `get_handles` to obtain the two function handles, and then pass each to the `functions` function. MATLAB returns information in a structure having the fields `function`, `type`, `file`, and `parentage`. The `file` field contains the file specification for the subfunction or private function:

```
[fsub fprv] = get_handles;

functions(fsub)
ans =
     function: 'mysubfun'
         type: 'scopedfunction'
         file: 'c:\matlab\get_handles.m'
    parentage: {'mysubfun'  'get_handles'}

functions(fprv)
```

```
ans =
     function: 'myprivatefun'
         type: 'scopedfunction'
         file: 'c:\matlab\private\myprivatefun.m'
    parentage: {'myprivatefun'}
```

Example 3

In this example, the function `get_handles_nested.m` contains a nested function `nestfun`. This function has a single output which is a function handle to the nested function:

```
function handle = get_handles_nested(A)
nestfun(A);

   function y = nestfun(x)
   y = x + 1;
   end

handle = @nestfun;
end
```

Call this function to get the handle to the nested function. Use this handle as the input to `functions` to return the information shown here. Note that the `function` field of the return structure contains the names of the nested function and the function in which it is nested in the format. Also note that `functions` returns a `workspace` field containing the variables that are in context at the time you call this function by its handle:

```
fh = get_handles_nested(5);

fhinfo = functions(fh)
fhinfo =
     function: 'get_handles_nested/nestfun'
         type: 'nested'
         file: 'c:\matlab\get_handles_nested.m'
    workspace: [1x1 struct]
```

```
fhinfo.workspace
ans = 
    handle: @get_handles_nested/nestfun
         A: 5
```

See Also `function_handle`

Purpose

Evaluate general matrix function

Syntax

```
F = funm(A,fun)
F = funm(A, fun, options)
F=funm(A, fun, options, p1, p2,...)
[F, exitflag] = funm(...)
[F, exitflag, output] = funm(...)
```

Description

`F = funm(A,fun)` evaluates the user-defined function `fun` at the square matrix argument `A`. `F = fun(x, k)` must accept a vector `x` and an integer `k`, and return a vector `f` of the same size of `x`, where `f(i)` is the `k`th derivative of the function `fun` evaluated at `x(i)`. The function represented by fun must have a Taylor series with an infinite radius of convergence, except for `fun = @log`, which is treated as a special case.

You can also use `funm` to evaluate the special functions listed in the following table at the matrix `A`.

Function	Syntax for Evaluating Function at Matrix A
`exp`	`funm(A, @exp)`
`log`	`funm(A, @log)`
`sin`	`funm(A, @sin)`
`cos`	`funm(A, @cos)`
`sinh`	`funm(A, @sinh)`
`cosh`	`funm(A, @cosh)`

For matrix square roots, use `sqrtm(A)` instead. For matrix exponentials, which of `expm(A)` or `funm(A, @exp)` is the more accurate depends on the matrix `A`.

The function represented by `fun` must have a Taylor series with an infinite radius of convergence. The exception is `@log`, which is treated as a special case. , in the online MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`F = funm(A, fun, options)` sets the algorithm's parameters to the values in the structure `options`.

The following table lists the fields of `options`.

Field	Description	Values
`options.Display`	Level of display	`'off'` (default), `'on'`, `'verbose'`
`options.TolBlk`	Tolerance for blocking Schur form	Positive scalar. The default is `0.1`.
`options.TolTay`	Termination tolerance for evaluating the Taylor series of diagonal blocks	Positive scalar. The default is `eps`.
`options.MaxTerms`	Maximum number of Tayor series terms	Positive integer. The default is `250`.
`options.MaxSqrt`	When computing a logarithm, maximum number of square roots computed in inverse scaling and squaring method.	Positive integer. The default is `100`.
`options.Ord`	Specifies the ordering of the Schur form `T`.	A vector of length `length(A)`. `options.Ord(i)` is the index of the block into which `T(i,i)` is placed. The default is `[]`.

`F=funm(A, fun, options, p1, p2,...)` passes extra inputs `p1, p2,...` to the function.

`[F, exitflag] = funm(...)` returns a scalar `exitflag` that describes the exit condition of `funm`. `exitflag` can have the following values:

- `0` — The algorithm was successful.
- `1` — One or more Taylor series evaluations did not converge, or, in the case of a logarithm, too many square roots are needed. However, the computed value of `F` might still be accurate. This is different from R13 and earlier versions that returned an expensive and often inaccurate error estimate as the second output argument.

`[F, exitflag, output] = funm(...)` returns a structure `output` with the following fields:

Field	Description
`output.terms`	Vector for which `output.terms(i)` is the number of Taylor series terms used when evaluating the *i*th block, or, in the case of the logarithm, the number of square roots of matrices of dimension greater than 2.
`output.ind`	Cell array for which the `(i,j)` block of the reordered Schur factor `T` is `T(output.ind{i}, output.ind{j})`.
`output.ord`	Ordering of the Schur form, as passed to `ordschur`
`output.T`	Reordered Schur form

If the Schur form is diagonal then `output = struct('terms',ones(n,1),'ind',{1:n})`.

Examples

Example 1

The following command computes the matrix sine of the 3-by-3 magic matrix.

```
F=funm(magic(3), @sin)

F =
```

```
   -0.3850    1.0191    0.0162
    0.6179    0.2168   -0.1844
    0.4173   -0.5856    0.8185
```

Example 2

The statements

```
S = funm(X,@sin);
C = funm(X,@cos);
```

produce the same results to within roundoff error as

```
E = expm(i*X);
C = real(E);
S = imag(E);
```

In either case, the results satisfy `S*S+C*C = I`, where `I = eye(size(X))`.

Example 3

To compute the function `exp(x) + cos(x)` at `A` with one call to `funm`, use

```
F = funm(A,@fun_expcos)
```

where `fun_expcos` is the following M-file function.

```
function f = fun_expcos(x, k)
% Return kth derivative of exp + cos at X.
        g = mod(ceil(k/2),2);
        if mod(k,2)
           f = exp(x) + sin(x)*(-1)^g;
        else
           f = exp(x) + cos(x)*(-1)^g;
        end
```

Algorithm

The algorithm `funm` uses is described in [1].

See Also

`expm`, `logm`, `sqrtm`, `function_handle` (`@`)

References

[1] Davies, P. I. and N. J. Higham, “A Schur-Parlett algorithm for computing matrix functions,” *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Third Edition, Johns Hopkins University Press, 1996, p. 384.

[3] Moler, C. B. and C. F. Van Loan, “Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later” *SIAM Review 20*, Vol. 45, Number 1, pp. 1-47, 2003.

Purpose

Write data to binary file

Syntax

```
fwrite(fileID, A)
fwrite(fileID, A, precision)
fwrite(fileID, A, precision, skip)
fwrite(fileID, A, precision, skip, machineformat)
count = fwrite(...)
```

Description

`fwrite(`*`fileID`*`, `*`A`*`)` writes the elements of array *`A`* to a binary file in column order.

`fwrite(`*`fileID`*`, `*`A`*`, `*`precision`*`)` translates the values of *`A`* according to the form and size described by the *`precision`*.

`fwrite(`*`fileID`*`, `*`A`*`, `*`precision`*`, `*`skip`*`)` skips *`skip`* bytes before writing each value. If *`precision`* is `bit`*`n`* or `ubit`*`n`*, specify *`skip`* in bits.

`fwrite(`*`fileID`*`, `*`A`*`, `*`precision`*`, `*`skip`*`, `*`machineformat`*`)` writes data with the specified *`machineformat`*. The *`skip`* parameter is optional.

`count`` = fwrite(...)` returns the number of elements of *`A`* that `fwrite` successfully writes to the file.

Inputs

`fileID`

One of the following:

- An integer file identifier obtained from `fopen`.
- `1` for standard output (the screen).
- `2` for standard error.

`A`

Numeric or character array.

`precision`

String in single quotation marks that controls the form and size of the output. The following table shows possible values for *`precision`*.

Value Type	**Precision**	**Bits (Bytes)**
Integers, unsigned	`uint`	`32 (4)`
	`uint8`	`8 (1)`
	`uint16`	`16 (2)`
	`uint32`	`32 (4)`
	`uint64`	`64 (8)`
	`uchar`	`8 (1)`
	`unsigned char`	`8 (1)`
	`ushort`	`16 (2)`
	`ulong`	system-dependent
	`ubit`*n*	$1 \le n \le 64$
Integers, signed	`int`	`32 (4)`
	`int8`	`8 (1)`
	`int16`	`16 (2)`
	`int32`	`32 (4)`
	`int64`	`64 (8)`
	`integer*1`	`8 (1)`
	`integer*2`	`16 (2)`
	`integer*3`	`32 (4)`
	`integer*4`	`64 (8)`
	`schar`	`8 (1)`
	`signed char`	`8 (1)`
	`short`	`16 (2)`
	`long`	system-dependent
	`bit`*n*	$1 \le n \le 64$

Value Type	Precision	Bits (Bytes)

Floating-point numbers	`single`
	`double`
	`float`
	`float32`
	`float64`
	`real*4`
	`real*8`
		`32 (4)`
		`64 (8)`
		`32 (4)`
		`32 (4)`
		`64 (8)`
		`32 (4)`
		`64 (8)`

Characters	`char*1` `char`	`8 (1)` Depends on the encoding scheme associated with the file. Set encoding with `fopen`.

`long` and `ulong` are 32 bits on 32-bit systems, and 64 bits on 64-bit systems.

If you specify a precision of `bit`*`n`* or `ubit`*`n`*, and a value is out of range, `fwrite` sets all bits for that value.

Default: `uint8`

`skip`

Number of bytes to skip before writing each value. If you specify a *`precision`* of `bit`*`n`* or `ubit`*`n`*, specify *`skip`* in bits. Use this parameter to insert data into noncontiguous fields in fixed-length records.

Default: `0`

`machineformat`

String that specifies the order for writing bytes within the file. For `bit`*`n`* and `ubit`*`n`* precisions, specifies the order for writing bits within a byte. Use this parameter when you plan to read and write to the same file on different systems.

Possible values are:

`'n'` or `'native'`	The byte ordering that your system uses (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering
`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit data type

Windows systems use little-endian ordering, and most UNIX systems use big-endian ordering, for both bytes and bits. Solaris systems use big-endian ordering for bytes, but little-endian ordering for bits.

Examples

Create a 100-byte binary file containing the 25 elements of the 5-by-5 magic square, stored as 4-byte integers:

```
fid = fopen('magic5.bin', 'w');
fwrite(fid, magic(5), 'integer*4');
fclose(fid);
```

See Also

`fclose` | `ferror` | `fopen` | `fprintf` | `fscanf` | `fread`

How To

-
-
-

Purpose

Write binary data to device

Syntax

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A,'mode')
fwrite(obj,A,'precision','mode')
```

Description

`fwrite(obj,A)` writes the binary data `A` to the device connected to the serial port object, `obj`.

`fwrite(obj,A,'precision')` writes binary data with precision specified by *`precision`*.

`precision` controls the number of bits written for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified, `uchar` (an 8-bit unsigned character) is used. The supported values for *`precision`* are listed below in Remarks.

`fwrite(obj,A,'mode')` writes binary data with command line access specified by *`mode`*. If *`mode`* is `sync`, `A` is written synchronously and the command line is blocked. If *`mode`* is `async`, `A` is written asynchronously and the command line is not blocked. If *`mode`* is not specified, the write operation is synchronous.

`fwrite(obj,A,'precision','mode')` writes binary data with precision specified by *`precision`* and command line access specified by *`mode`*.

Remarks

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

The `ValuesSent` property value is increased by the number of values written each time `fwrite` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

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

```
help serial/fwrite
```

`fwrite` will return an error message if you set the `FlowControl` property to `hardware` on a serial object, and a hardware connection is not detected. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check you remote device's status and flow control settings to see if hardware flow control is causing errors in MATLAB.

Note If you want to check to see if the device is asserting that it is ready to receive data, set the `FlowControl` to `none`. Once you connect to the device check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is `off`, there is a problem on the remote device side. If `ClearToSend` is `on`, there is a hardware `FlowControl` device prepared to receive data and you can execute `fwrite`.

Synchronous Versus Asynchronous Write Operations

By default, data is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *`mode`* input argument to be `async`. For asynchronous writes:

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The M-file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in Writing Data.

Rules for Completing a Write Operation with fwrite

A binary write operation using `fwrite` completes when:

- The specified data is written.
- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used with binary write operations.

Supported Precisions

The supported values for *`precision`* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character

Data Type	Precision	Interpretation
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

See Also

Functions

`fopen`, `fprintf`

Properties

`BytesToOutput`, `OutputBufferSize`, `OutputEmptyFcn`, `Status`, `Timeout`, `TransferStatus`, `ValuesSent`

Purpose

Find root of continuous function of one variable

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)
[x,fval] = fzero(...)
[x,fval,exitflag] = fzero(...)
[x,fval,exitflag,output] = fzero(...)
```

Description

`x = fzero(fun,x0)` tries to find a zero of `fun` near `x0`, if `x0` is a scalar. `fun` is a function handle. See in the MATLAB Programming documentation for more information. The value `x` returned by `fzero` is near a point where `fun` changes sign, or `NaN` if the search fails. In this case, the search terminates when the search interval is expanded until an `Inf`, `NaN`, or complex value is found.

in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`. See also “Example 2” on page 2-1467 and “Example 3” on page 2-1468 below.

If `x0` is a vector of length two, `fzero` assumes `x0` is an interval where the sign of `fun(x0(1))` differs from the sign of `fun(x0(2))`. An error occurs if this is not true. Calling `fzero` with such an interval guarantees `fzero` will return a value near a point where `fun` changes sign.

`x = fzero(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fzero` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
FunValCheck	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex` or `NaN`. `'off'` (the default) displays no error.

OutputFcn	User-defined function that is called at each iteration. See in MATLAB Mathematics for more information.
PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`). • `@optimplotx` plots the current point • `@optimplotfval` plots the function value See in MATLAB Mathematics for more information.
`TolX`	Termination tolerance on `x`

`[x,fval] = fzero(...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fzero(...)` returns a value `exitflag` that describes the exit condition of `fzero`:

`1`	Function converged to a solution `x`.
`-1`	Algorithm was terminated by the output function.
`-3`	`NaN` or `Inf` function value was encountered during search for an interval containing a sign change.
`-4`	Complex function value was encountered during search for an interval containing a sign change.
`-5`	`fzero` might have converged to a singular point.
`-6`	`fzero` can not detect a change in sign of the function.

`[x,fval,exitflag,output] = fzero(...)` returns a structure `output` that contains information about the optimization in the following fields:

`algorithm`	Algorithm used
`funcCount`	Number of function evaluations

`intervaliterations`	Number of iterations taken to find an interval
`iterations`	Number of zero-finding iterations
`message`	Exit message

Note For the purposes of this command, zeros are considered to be points where the function actually crosses, not just touches, the *x*-axis.

Arguments

`fun` is the function whose zero is to be computed. It accepts a vector `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fzero(@myfun,x0);
```

where `myfun` is an M-file function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

or as a function handle for an anonymous function:

```
x = fzero(@(x)sin(x*x),x0);
```

Other arguments are described in the syntax descriptions above.

Examples

Example 1

Calculate π by finding the zero of the sine function near 3.

```
x = fzero(@sin,3)
x =
    3.1416
```

Example 2

To find the zero of cosine between 1 and 2

```
x = fzero(@cos,[1 2])
x =
    1.5708
```

Note that `cos(1)` and `cos(2)` differ in sign.

Example 3

To find a zero of the function $f(x) = x^3 - 2x - 5$, write an anonymous function `f`:

```
f = @(x)x.^3-2*x-5;
```

Then find the zero near 2:

```
z = fzero(f,2)
z =
    2.0946
```

Because this function is a polynomial, the statement `roots([1 0 -2 -5])` finds the same real zero, and a complex conjugate pair of zeros.

```
    2.0946
  -1.0473 + 1.1359i
  -1.0473 - 1.1359i
```

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = cos(a*x);
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fzero`. To optimize for a specific value of `a`, such as `a = 2`.

1. Assign the value to `a`.

   ```
   a = 2; % define parameter first
   ```

2 Call `fzero` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

```
x = fzero(@(x) myfun(x,a),0.1)
```

Algorithm

The `fzero` command is an M-file. The algorithm, which was originated by T. Dekker, uses a combination of bisection, secant, and inverse quadratic interpolation methods. An Algol 60 version, with some improvements, is given in [1]. A Fortran version, upon which the `fzero` M-file is based, is in [2].

Limitations

The `fzero` command finds a point where the function changes sign. If the function is *continuous*, this is also a point where the function has a value near zero. If the function is not continuous, `fzero` may return values that are discontinuous points instead of zeros. For example, `fzero(@tan,1)` returns `1.5708`, a discontinuous point in `tan`.

Furthermore, the `fzero` command defines a *zero* as a point where the function crosses the x-axis. Points where the function touches, but does not cross, the x-axis are not valid zeros. For example, `y = x.^2` is a parabola that touches the x-axis at 0. Because the function never crosses the x-axis, however, no zero is found. For functions with no valid zeros, `fzero` executes until `Inf`, `NaN`, or a complex value is detected.

See Also

`roots`, `fminbnd`, `optimset`, `function_handle` (@),

References

[1] Brent, R., *Algorithms for Minimization Without Derivatives*, Prentice-Hall, 1973.

[2] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

Purpose

Test matrices

Syntax

```
[A,B,C,...] = gallery(matname,P1,P2,...)
[A,B,C,...] = gallery(matname,P1,P2,...,classname)
gallery(3)
gallery(5)
```

Description

`[A,B,C,...] = gallery(matname,P1,P2,...)` returns the test matrices specified by the quoted string `matname`. The `matname` input is the name of a matrix family selected from the table below. `P1,P2,...` are input parameters required by the individual matrix family. The number of optional parameters `P1,P2,...` used in the calling syntax varies from matrix to matrix. The exact calling syntaxes are detailed in the individual matrix descriptions below.

`[A,B,C,...] = gallery(matname,P1,P2,...,classname)` produces a matrix of class `classname`. The `classname` input is a quoted string that must be either `'single'` or `'double'` (unless `matname` is `'integerdata'`, in which case `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, and `'uint32'` are also allowed). If `classname` is not specified, then the class of the matrix is determined from those arguments among `P1,P2,...` that do not specify dimensions or select an option. If any of these arguments is of class `single` then the matrix is `single`; otherwise the matrix is `double`.

`gallery(3)` is a badly conditioned 3-by-3 matrix and `gallery(5)` is an interesting eigenvalue problem.

The gallery holds over fifty different test matrix functions useful for testing algorithms and other purposes.

- `binomial`
- `cauchy`
- `chebspec`
- `chebvand`
- `chow`

- `circul`
- `clement`
- `compar`
- `condex`
- `cycol`
- `dorr`
- `dramadah`
- `fiedler`
- `forsythe`
- `frank`
- `gearmat`
- `gcdmat`
- `grcar`
- `hanowa`
- `house`
- `integerdata`
- `invhess`
- `invol`
- `ipjfact`
- `jordbloc`
- `kahan`
- `kms`
- `krylov`
- `lauchli`
- `lehmer`

- `leslie`
- `lesp`
- `lotkin`
- `minij`
- `moler`
- `neumann`
- `normaldata`
- `orthog`
- `parter`
- `pei`
- `poisson`
- `prolate`
- `randcolu`
- `randcorr`
- `randhess`
- `randjorth`
- `rando`
- `randsvd`
- `redheff`
- `riemann`
- `ris`
- `sampling`
- `smoke`
- `toeppd`
- `tridiag`

- `triw`
- `uniformdata`
- `wathen`
- `wilk`

binomial — Multiple of involutory matrix

`A = gallery('binomial',n)` returns an `n`-by-`n` matrix,with integer entries such that `A^2 = 2^(n-1)*eye(n)`.

Thus, `B = A*2^((1-n)/2)` is involutory, that is, `B^2 = eye(n)`.

cauchy — Cauchy matrix

`C = gallery('cauchy',x,y)` returns an `n`-by-`n` matrix, `C(i,j) = 1/(x(i)+y(j))`. Arguments `x` and `y` are vectors of length `n`. If you pass in scalars for `x` and `y`, they are interpreted as vectors `1:x` and `1:y`.

`C = gallery('cauchy',x)` returns the same as above with `y = x`. That is, the command returns `C(i,j) = 1/(x(i)+x(j))`.

Explicit formulas are known for the inverse and determinant of a Cauchy matrix. The determinant `det(C)` is nonzero if `x` and `y` both have distinct elements. `C` is totally positive if `0 < x(1) <...  < x(n)` and `0 < y(1) < ...  < y(n)`.

chebspec — Chebyshev spectral differentiation matrix

`C = gallery('chebspec',n,switch)` returns a Chebyshev spectral differentiation matrix of order `n`. Argument `switch` is a variable that determines the character of the output matrix. By default, `switch` = `0`.

For `switch` = `0` (“no boundary conditions”), `C` is nilpotent (`C^n = 0`) and has the null vector `ones(n,1)`. The matrix `C` is similar to a Jordan block of size `n` with eigenvalue zero.

For `switch` = `1`, `C` is nonsingular and well-conditioned, and its eigenvalues have negative real parts.

The eigenvector matrix of the Chebyshev spectral differentiation matrix is ill-conditioned.

chebvand — Vandermonde-like matrix for the Chebyshev polynomials

`C = gallery('chebvand',p)` produces the (primal) Chebyshev Vandermonde matrix based on the vector of points `p`, which define where the Chebyshev polynomial is calculated.

`C = gallery('chebvand',m,p)` where `m` is scalar, produces a rectangular version of the above, with `m` rows.

If `p` is a vector, then $C(i, j) = T_{i-1}(p(j))$ where T_{i-1} is the Chebyshev polynomial of degree i-1. If `p` is a scalar, then `p` equally spaced points on the interval `[0,1]` are used to calculate `C`.

chow — Singular Toeplitz lower Hessenberg matrix

`A = gallery('chow',n,alpha,delta)` returns `A` such that `A = H(alpha) + delta*eye(n)`, where $H_{i,j}(\alpha) = \alpha^{(i-j+1)}$ and argument `n` is the order of the Chow matrix. Default value for scalars `alpha` and `delta` are `1` and `0`, respectively.

`H(alpha)` has `p = floor(n/2)` eigenvalues that are equal to zero. The rest of the eigenvalues are equal to `4*alpha*cos(k*pi/(n+2))^2`, `k=1:n-p`.

circul — Circulant matrix

`C = gallery('circul',v)` returns the circulant matrix whose first row is the vector `v`.

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. It is a special Toeplitz matrix in which the diagonals “wrap around.”

If `v` is a scalar, then `C = gallery('circul',1:v)`.

The eigensystem of `C` (`n`-by-`n`) is known explicitly: If `t` is an `n`th root of unity, then the inner product of `v` and $w = [1\ t\ t^2 \ldots t^{(n-1)}]$ is an eigenvalue of `C` and `w(n:-1:1)` is an eigenvector.

clement — Tridiagonal matrix with zero diagonal entries

`A = gallery('clement',n,k)` returns an `n`-by-`n` tridiagonal matrix with zeros on its main diagonal and known eigenvalues. It is singular if `n` is odd. About 64 percent of the entries of the inverse are zero. The eigenvalues include plus and minus the numbers `n-1`, `n-3`, `n-5`, `...`, (1 or 0).

For `k=0` (the default), `A` is nonsymmetric. For `k=1`, `A` is symmetric.

`gallery('clement',n,1)` is diagonally similar to `gallery('clement',n)`.

For odd `N = 2*M+1, M+1` of the singular values are the integers `sqrt((2*M+1)^2 - (2*K+1).^2), K = 0:M`.

Note Similar properties hold for `gallery('tridiag',x,y,z)` where `y = zeros(n,1)`. The eigenvalues still come in plus/minus pairs but they are not known explicitly.

---

compar — Comparison matrices

`A = gallery('compar',A,1)` returns `A` with each diagonal element replaced by its absolute value, and each off-diagonal element replaced by minus the absolute value of the largest element in absolute value in its row. However, if `A` is triangular `compar(A,1)` is too.

`gallery('compar',A)` is `diag(B) - tril(B,-1) - triu(B,1)`, where `B = abs(A)`. `compar(A)` is often denoted by $M(A)$ in the literature.

`gallery('compar',A,0)` is the same as `gallery('compar',A)`.

condex — Counter-examples to matrix condition number estimators

`A = gallery('condex',n,k,theta)` returns a "counter-example" matrix to a condition estimator. It has order `n` and scalar parameter `theta` (default 100).

The matrix, its natural size, and the estimator to which it applies are specified by `k`:

`k = 1`	4-by-4	LINPACK
`k = 2`	3-by-3	LINPACK
`k = 3`	arbitrary	LINPACK (`rcond`) (independent of `theta`)
`k = 4`	`n >= 4`	LAPACK (RCOND) (default). It is the inverse of this matrix that is a counter-example.

If `n` is not equal to the natural size of the matrix, then the matrix is padded out with an identity matrix to order `n`.

cycol — Matrix whose columns repeat cyclically

`A = gallery('cycol',[m n],k)` returns an `m`-by-`n` matrix with cyclically repeating columns, where one "cycle" consists of `randn(m,k)`. Thus, the rank of matrix `A` cannot exceed `k`, and `k` must be a scalar.

Argument `k` defaults to `round(n/4)`, and need not evenly divide `n`.

`A = gallery('cycol',n,k)`, where `n` is a scalar, is the same as `gallery('cycol',[n n],k)`.

dorr — Diagonally dominant, ill-conditioned, tridiagonal matrix

`[c,d,e] = gallery('dorr',n,theta)` returns the vectors defining an `n`-by-`n`, row diagonally dominant, tridiagonal matrix that is ill-conditioned for small nonnegative values of `theta`. The default value of `theta` is `0.01`. The Dorr matrix itself is the same as `gallery('tridiag',c,d,e)`.

`A = gallery('dorr',n,theta)` returns the matrix itself, rather than the defining vectors.

dramadah — Matrix of zeros and ones whose inverse has large integer entries

`A = gallery('dramadah',n,k)` returns an `n`-by-`n` matrix of `0`'s and `1`'s for which `mu(A) = norm(inv(A),'fro')` is relatively large, although not necessarily maximal. An anti-Hadamard matrix `A` is a matrix with elements `0` or `1` for which `mu(A)` is maximal.

`n` and `k` must both be scalars. Argument `k` determines the character of the output matrix:

k = 1	Default. `A` is Toeplitz, with `abs(det(A)) = 1`, and `mu(A) > c(1.75)^n`, where `c` is a constant. The inverse of `A` has integer entries.
k = 2	`A` is upper triangular and Toeplitz. The inverse of `A` has integer entries.
k = 3	`A` has maximal determinant among lower Hessenberg (0,1) matrices. `det(A)` = the `n`th Fibonacci number. `A` is Toeplitz. The eigenvalues have an interesting distribution in the complex plane.

fiedler — Symmetric matrix

`A = gallery('fiedler',c)`, where `c` is a length `n` vector, returns the `n`-by-`n` symmetric matrix with elements `abs(n(i)-n(j))`. For scalar `c`, `A = gallery('fiedler',1:c)`.

Matrix `A` has a dominant positive eigenvalue and all the other eigenvalues are negative.

Explicit formulas for `inv(A)` and `det(A)` are given in [Todd, J., *Basic Numerical Mathematics*, Vol. 2: Numerical Algebra, Birkhauser, Basel, and Academic Press, New York, 1977, p. 159] and attributed to Fiedler. These indicate that `inv(A)` is tridiagonal except for nonzero `(1,n)` and `(n,1)` elements.

forsythe — Perturbed Jordan block

`A = gallery('forsythe',n,alpha,lambda)` returns the `n`-by-`n` matrix equal to the Jordan block with eigenvalue `lambda`, excepting that `A(n,1) = alpha`. The default values of scalars `alpha` and `lambda` are `sqrt(eps)` and `0`, respectively.

The characteristic polynomial of `A` is given by:

```
det(A-t*I) = (lambda-t)^N - alpha*(-1)^n.
```

frank — Matrix with ill-conditioned eigenvalues

`F = gallery('frank',n,k)` returns the Frank matrix of order `n`. It is upper Hessenberg with determinant `1`. If `k = 1`, the elements are reflected about the anti-diagonal `(1,n)` — `(n,1)`. The eigenvalues of `F` may be obtained in terms of the zeros of the Hermite polynomials. They are positive and occur in reciprocal pairs; thus if `n` is odd, `1` is an eigenvalue. `F` has `floor(n/2)` ill-conditioned eigenvalues — the smaller ones.

gcdmat — Greatest common divisor matrix

`A = gallery('gcdmat',n)` returns the `n`-by-`n` matrix with `(i,j)` entry `gcd(i,j).  Matrix` A is symmetric positive definite, and A.^r is symmetric positive semidefinite for all nonnegative `r`.

gearmat — Gear matrix

`A = gallery('gearmat',n,i,j)` returns the `n`-by-`n` matrix with ones on the sub- and super-diagonals, `sign(i)` in the `(1,abs(i))` position, `sign(j)` in the `(n,n+1-abs(j))` position, and zeros everywhere else. Arguments `i` and `j` default to `n` and `-n`, respectively.

Matrix `A` is singular, can have double and triple eigenvalues, and can be defective.

All eigenvalues are of the form `2*cos(a)` and the eigenvectors are of the form `[sin(w+a), sin(w+2*a), ..., sin(w+n*a)]`, where `a` and `w` are given in Gear, C. W., “A Simple Set of Test Matrices for Eigenvalue Programs,” *Math. Comp.*, Vol. 23 (1969), pp. 119-125.

grcar — Toeplitz matrix with sensitive eigenvalues

`A = gallery('grcar',n,k)` returns an `n`-by-`n` Toeplitz matrix with `-1`s on the subdiagonal, `1`s on the diagonal, and `k` superdiagonals of `1`s. The default is `k = 3`. The eigenvalues are sensitive.

hanowa — Matrix whose eigenvalues lie on a vertical line in the complex plane

`A = gallery('hanowa',n,d)` returns an `n`-by-`n` block `2`-by-`2` matrix of the form:

```
[d*eye(m) -diag(1:m)
diag(1:m) d*eye(m)]
```

Argument `n` is an even integer `n=2*m`. Matrix `A` has complex eigenvalues of the form `d` ± `k*i`, for `1 <= k <= m`. The default value of `d` is `-1`.

house — Householder matrix

`[v,beta,s] = gallery('house',x,k)` takes `x`, an `n`-element column vector, and returns `V` and `beta` such that `H*x = s*e1`. In this expression, `e1` is the first column of `eye(n)`, `abs(s) = norm(x)`, and `H = eye(n) - beta*V*V'` is a Householder matrix.

`k` determines the sign of `s`:

`k = 0`	`sign(s) = -sign(x(1))` (default)
`k = 1`	`sign(s) = sign(x(1))`
`k = 2`	`sign(s) = 1` (x must be real)

If `x` is complex, then `sign(x) = x./abs(x)` when `x` is nonzero.

If `x = 0`, or if `x = alpha*e1` (`alpha >= 0`) and either `k = 1` or `k = 2`, then `V = 0`, `beta = 1`, and `s = x(1)`. In this case, `H` is the identity matrix, which is not strictly a Householder matrix.

`[v, beta] = gallery('house',x)` takes `x`, a scalar or `n`-element column vector, and returns `v` and `beta` such that `eye(n,n) -`

`beta*v*v'` is a Householder matrix. A Householder matrix `H` satisfies the relationship

```
H*x = -sign(x(1))*norm(x)*e1
```

where `e1` is the first column of `eye(n,n)`. Note that if `x` is complex, then `sign(x) exp(i*arg(x))` (which equals `x./abs(x)` when `x` is nonzero).

If `x = 0`, then `v = 0` and `beta = 1`.

integerdata — Array of arbitrary data from uniform distribution on specified range of integers

`A = gallery('integerdata',imax,[m,n,...],j)` returns an `m`-by-`n`-by-... array `A` whose values are a sample from the uniform distribution on the integers `1:imax`. `j` must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('integerdata', ...)` with different values of `J` will return different arrays. Repeated calls to `gallery('integerdata',...)` with the same `imax`, size vector and `j` inputs will always return the same array.

In any call to `gallery('integerdata', ...)` you can substitute individual inputs `m`,`n`,... for the size vector input `[m,n,...]`. For example, `gallery('integerdata',7,[1,2,3,4],5)` is equivalent to `gallery('integerdata',7,1,2,3,4,5)`.

`A = gallery('integerdata',[imin imax],[m,n,...],j)` returns an `m`-by-`n`-by-... array `A` whose values are a sample from the uniform distribution on the integers `imin:imax`.

`[A,B,...]  = gallery('integerdata',[imin imax],[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('integerdata',[imin imax],[m,n,...],j,classname)` produces an array of class `classname`. `classname` must be `'uint8'`, `'uint16'`, `'uint32'`, `'int8'`, `'int16'`, `int32'`, `'single'` or `'double'`.

invhess — Inverse of an upper Hessenberg matrix

`A = gallery('invhess',x,y)`, where `x` is a length `n` vector and `y` is a length `n-1` vector, returns the matrix whose lower triangle agrees with that of `ones(n,1)*x'` and whose strict upper triangle agrees with that of `[1 y]*ones(1,n)`.

The matrix is nonsingular if `x(1) ~= 0` and `x(i+1) ~= y(i)` for all `i`, and its inverse is an upper Hessenberg matrix. Argument `y` defaults to `-x(1:n-1)`.

If `x` is a scalar, `invhess(x)` is the same as `invhess(1:x)`.

invol — Involutory matrix

`A = gallery('invol',n)` returns an `n`-by-`n` involutory `(A*A = eye(n))` and ill-conditioned matrix. It is a diagonally scaled version of `hilb(n)`.

`B = (eye(n)-A)/2` and `B = (eye(n)+A)/2` are idempotent `(B*B = B)`.

ipjfact — Hankel matrix with factorial elements

`[A,d] = gallery('ipjfact',n,k)` returns `A`, an `n`-by-`n` Hankel matrix, and `d`, the determinant of `A`, which is known explicitly. If `k = 0` (the default), then the elements of `A` are `A(i,j) = (i+j)!` If `k = 1`, then the elements of `A` are `A(i,j) 1/(i+j)`.

Note that the inverse of `A` is also known explicitly.

jordbloc — Jordan block

`A = gallery('jordbloc',n,lambda)` returns the `n`-by-`n` Jordan block with eigenvalue `lambda`. The default value for `lambda` is `1`.

kahan — Upper trapezoidal matrix

`A = gallery('kahan',n,theta,pert)` returns an upper trapezoidal matrix that has interesting properties regarding estimation of condition and rank.

If `n` is a two-element vector, then `A` is `n(1)`-by-`n(2)`; otherwise, `A` is `n`-by-`n`. The useful range of `theta` is `0 < theta < pi`, with a default value of `1.2`.

To ensure that the QR factorization with column pivoting does not interchange columns in the presence of rounding errors, the diagonal is perturbed by `pert*eps*diag([n:-1:1])`. The default `pert` is `25`, which ensures no interchanges for `gallery('kahan',n)` up to at least `n = 90` in IEEE arithmetic.

kms — Kac-Murdock-Szego Toeplitz matrix

`A = gallery('kms',n,rho)` returns the `n`-by-`n` Kac-Murdock-Szego Toeplitz matrix such that `A(i,j) = rho^(abs(i-j))`, for real `rho`.

For complex `rho`, the same formula holds except that elements below the diagonal are conjugated. `rho` defaults to 0.5.

The KMS matrix `A` has these properties:

- An LDL' factorization with `L inv(gallery('triw',n,-rho,1))'`, and `D(i,i) (1-abs(rho)^2)*eye(n)`, except `D(1,1) = 1`.
- Positive definite if and only if `0 < abs(rho) < 1`.
- The inverse `inv(A)` is tridiagonal.

krylov — Krylov matrix

`B = gallery('krylov',A,x,j)` returns the Krylov matrix

```
[x, Ax, A^2x, ..., A^(j-1)x]
```

where `A` is an `n`-by-`n` matrix and `x` is a length `n` vector. The defaults are `x ones(n,1)`, and `j = n`.

`B = gallery('krylov',n)` is the same as `gallery('krylov',(randn(n))`.

lauchli — Rectangular matrix

`A = gallery('lauchli',n,mu)` returns the `(n+1)`-by-`n` matrix

```
[ones(1,n); mu*eye(n)]
```

The Lauchli matrix is a well-known example in least squares and other problems that indicates the dangers of forming `A'*A`. Argument `mu` defaults to `sqrt(eps)`.

lehmer — Symmetric positive definite matrix

`A = gallery('lehmer',n)` returns the symmetric positive definite `n`-by-`n` matrix such that `A(i,j) = i/j` for `j >= i`.

The Lehmer matrix `A` has these properties:

- `A` is totally nonnegative.
- The inverse `inv(A)` is tridiagonal and explicitly known.
- The order `n <= cond(A) <= 4*n*n`.

leslie — Matrix of birth numbers and survival rates

`L = gallery('leslie',a,b)` is the `n`-by-`n` matrix from the Leslie population model with average birth numbers `a(1:n)` and survival rates `b(1:n-1)`. It is zero, apart from the first row (which contains the `a(i)`) and the first subdiagonal (which contains the `b(i)`). For a valid model, the `a(i)` are nonnegative and the `b(i)` are positive and bounded by 1, i.e., `0 < b(i) <= 1`.

`L = gallery('leslie',n)` generates the Leslie matrix with `a = ones(n,1)`, `b = ones(n-1,1)`.

lesp — Tridiagonal matrix with real, sensitive eigenvalues

`A = gallery('lesp',n)` returns an `n`-by-`n` matrix whose eigenvalues are real and smoothly distributed in the interval approximately `[-2*N-3.5, -4.5]`.

The sensitivities of the eigenvalues increase exponentially as the eigenvalues grow more negative. The matrix is similar to the symmetric tridiagonal matrix with the same diagonal entries and with off-diagonal entries 1, via a similarity transformation with `D = diag(1!,2!,...,n!)`.

lotkin — Lotkin matrix

`A = gallery('lotkin',n)` returns the Hilbert matrix with its first row altered to all ones. The Lotkin matrix `A` is nonsymmetric, ill-conditioned, and has many negative eigenvalues of small magnitude. Its inverse has integer entries and is known explicitly.

minij — Symmetric positive definite matrix

`A = gallery('minij',n)` returns the `n`-by-`n` symmetric positive definite matrix with `A(i,j) = min(i,j)`.

The `minij` matrix has these properties:

- The inverse `inv(A)` is tridiagonal and equal to `-1` times the second difference matrix, except its `(n,n)` element is `1`.
- Givens' matrix, `2*A-ones(size(A))`, has tridiagonal inverse and eigenvalues `0.5*sec((2*r-1)*pi/(4*n))^2`, where `r=1:n`.
- `(n+1)*ones(size(A))-A` has elements that are `max(i,j)` and a tridiagonal inverse.

moler — Symmetric positive definite matrix

`A = gallery('moler',n,alpha)` returns the symmetric positive definite `n`-by-`n` matrix `U'*U`, where `U = gallery('triw',n,alpha)`.

For the default `alpha` = `-1`, `A(i,j) = min(i,j)-2`, and `A(i,i) = i`. One of the eigenvalues of `A` is small.

neumann — Singular matrix from the discrete Neumann problem (sparse)

`C = gallery('neumann',n)` returns the sparse `n`-by-`n` singular, row diagonally dominant matrix resulting from discretizing the Neumann problem with the usual five-point operator on a regular mesh. Argument `n` is a perfect square integer $n = m^2$ or a two-element vector. `C` is sparse and has a one-dimensional null space with null vector `ones(n,1)`.

normaldata — Array of arbitrary data from standard normal distribution

`A = gallery('normaldata',[m,n,...],j)` returns an `m`-by-`n`-by-... array `A`. The values of `A` are a random sample from the standard normal distribution. `j` must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('normaldata', ...)` with different values of `j` will return different arrays. Repeated calls to `gallery('normaldata',...)`

with the same size vector and `j` inputs will always return the same array.

In any call to `gallery('normaldata', ...)` you can substitute individual inputs `m,n,...` for the size vector input `[m,n,...]`. For example, `gallery('normaldata',[1,2,3,4],5)` is equivalent to `gallery('normaldata',1,2,3,4,5)`.

`[A,B,...]  = gallery('normaldata',[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('normaldata',[m,n,...],j, classname)` produces a matrix of class `classname`. `classname` must be either `'single'` or `'double'`.

Generate the arbitrary 6-by-4 matrix of data from the standard normal distribution `N(0, 1)` corresponding to `j = 2`:.

```
x = gallery('normaldata', [6, 4], 2);
```

Generate the arbitrary 1-by-2-by-3 single array of data from the standard normal distribution `N(0, 1)` corresponding to `j = 17`:.

```
y = gallery('normaldata', 1, 2, 3, 17, 'single');
```

orthog — Orthogonal and nearly orthogonal matrices

`Q = gallery('orthog',n,k)` returns the kth type of matrix of order `n`, where `k > 0` selects exactly orthogonal matrices, and `k < 0` selects diagonal scalings of orthogonal matrices. Available types are:

`k = 1`	`Q(i,j) = sqrt(2/(n+1)) * sin(i*j*pi/(n+1))` Symmetric eigenvector matrix for second difference matrix. This is the default.
`k = 2`	`Q(i,j) = 2/(sqrt(2*n+1)) * sin(2*i*j*pi/(2*n+1))` Symmetric.

`k = 3`	`Q(r,s) = exp(2*pi*i*(r-1)*(s-1)/n) / sqrt(n)` Unitary, the Fourier matrix. `Q^4` is the identity. This is essentially the same matrix as `fft(eye(n))/sqrt(n)`!
`k = 4`	Helmert matrix: a permutation of a lower Hessenberg matrix, whose first row is `ones(1:n)/sqrt(n)`.
`k = 5`	`Q(i,j) = sin(2*pi*(i-1)*(j-1)/n) + cos(2*pi*(i-1)*(j-1)/n)` Symmetric matrix arising in the Hartley transform.
`k = 6`	`Q(i,j) = sqrt(2/n)*cos((i-1/2)*(j-1/2)*pi/n)` Symmetric matrix arising as a discrete cosine transform.
`k = -1`	`Q(i,j) = cos((i-1)*(j-1)*pi/(n-1))` Chebyshev Vandermonde-like matrix, based on extrema of `T(n-1)`.
`k = -2`	`Q(i,j) = cos((i-1)*(j-1/2)*pi/n))` Chebyshev Vandermonde-like matrix, based on zeros of `T(n)`.

parter — Toeplitz matrix with singular values near pi

`C = gallery('parter',n)` returns the matrix `C` such that `C(i,j) = 1/(i-j+0.5)`.

`C` is a Cauchy matrix and a Toeplitz matrix. Most of the singular values of `C` are very close to `pi`.

pei — Pei matrix

`A = gallery('pei',n,alpha)`, where `alpha` is a scalar, returns the symmetric matrix `alpha*eye(n) + ones(n)`. The default for `alpha` is 1. The matrix is singular for `alpha` equal to either `0` or `-n`.

poisson — Block tridiagonal matrix from Poisson's equation (sparse)

`A = gallery('poisson',n)` returns the block tridiagonal (sparse) matrix of order `n^2` resulting from discretizing Poisson's equation with the 5-point operator on an `n`-by-`n` mesh.

prolate — Symmetric, ill-conditioned Toeplitz matrix

`A = gallery('prolate',n,w)` returns the `n`-by-`n` prolate matrix with parameter `w`. It is a symmetric Toeplitz matrix.

If `0 < w < 0.5` then `A` is positive definite

- The eigenvalues of `A` are distinct, lie in `(0,1)`, and tend to cluster around `0` and `1`.
- The default value of `w` is 0.25.

randcolu — Random matrix with normalized cols and specified singular values

`A = gallery('randcolu',n)` is a random `n`-by-`n` matrix with columns of unit 2-norm, with random singular values whose squares are from a uniform distribution.

`A'*A` is a correlation matrix of the form produced by `gallery('randcorr',n)`.

`gallery('randcolu',x)` where `x` is an `n`-vector (`n` > 1), produces a random `n`-by-`n` matrix having singular values given by the vector `x`. The vector `x` must have nonnegative elements whose sum of squares is `n`.

`gallery('randcolu',x,m)` where `m >= n`, produces an `m`-by-`n` matrix.

`gallery('randcolu',x,m,k)` provides a further option:

`k = 0`	`diag(x)` is initially subjected to a random two-sided orthogonal transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have zero entries.

For more information, see:

[1] Davies, P. I. and N. J. Higham, “Numerically Stable Generation of Correlation Matrices and Their Factors,” *BIT*, Vol. 40, 2000, pp. 640-651.

randcorr — Random correlation matrix with specified eigenvalues

`gallery('randcorr',n)` is a random `n`-by-`n` correlation matrix with random eigenvalues from a uniform distribution. A correlation matrix is a symmetric positive semidefinite matrix with `1`s on the diagonal (see `corrcoef`).

`gallery('randcorr',x)` produces a random correlation matrix having eigenvalues given by the vector `x`, where `length(x) > 1`. The vector `x` must have nonnegative elements summing to `length(x)`.

`gallery('randcorr',x,k)` provides a further option:

`k = 0`	The diagonal matrix of eigenvalues is initially subjected to a random orthogonal similarity transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have some zero entries.

For more information, see:

[1] Bendel, R. B. and M. R. Mickey, "Population Correlation Matrices for Sampling Experiments," *Commun. Statist. Simulation Comput.*, B7, 1978, pp. 163-182.

[2] Davies, P. I. and N. J. Higham, "Numerically Stable Generation of Correlation Matrices and Their Factors," *BIT*, Vol. 40, 2000, pp. 640-651.

randhess — Random, orthogonal upper Hessenberg matrix

`H = gallery('randhess',n)` returns an `n`-by-`n` real, random, orthogonal upper Hessenberg matrix.

`H = gallery('randhess',x)` if `x` is an arbitrary, real, length `n` vector with `n > 1`, constructs `H` nonrandomly using the elements of `x` as parameters.

Matrix `H` is constructed via a product of `n-1` Givens rotations.

randjorth — Random J-orthogonal matrix

`A = gallery('randjorth', n)`, for a positive integer `n`, produces a random `n`-by-`n` `J`-orthogonal matrix `A`, where

- `J = blkdiag(eye(ceil(n/2)),-eye(floor(n/2)))`
- `cond(A) = sqrt(1/eps)`

`J`-orthogonality means that A'*J*A = J. Such matrices are sometimes called *hyperbolic*.

`A = gallery('randjorth', n, m)`, for positive integers `n` and `m`, produces a random `(n+m)`-by-`(n+m)` `J`-orthogonal matrix `A`, where

- `J = blkdiag(eye(n),-eye(m))`
- `cond(A) = sqrt(1/eps)`

`A = gallery('randjorth',n,m,c,symm,method)`

uses the following optional input arguments:

- `c` — Specifies `cond(A)` to be the scalar `c`.
- `symm` — Enforces symmetry if the scalar `symm` is nonzero.
- `method` — calls `qr` to perform the underlying orthogonal transformations if the scalar `method` is nonzero. A call to `qr` is much faster than the default method for large dimensions

rando — Random matrix composed of elements -1, 0 or 1

`A = gallery('rando',n,k)` returns a random `n`-by-`n` matrix with elements from one of the following discrete distributions:

`k = 1`	`A(i,j)` = `0` or `1` with equal probability (default).
`k = 2`	`A(i,j)` = `-1` or `1` with equal probability.
`k = 3`	`A(i,j)` = `-1`, `0` or `1` with equal probability.

Argument `n` may be a two-element vector, in which case the matrix is `n(1)`-by-`n(2)`.

randsvd — Random matrix with preassigned singular values

`A = gallery('randsvd',n,kappa,mode,kl,ku)` returns a banded (multidiagonal) random matrix of order `n` with `cond(A) = kappa` and singular values from the distribution mode. If `n` is a two-element vector, `A` is `n(1)`-by-`n(2)`.

Arguments `kl` and `ku` specify the number of lower and upper off-diagonals, respectively, in `A`. If they are omitted, a full matrix is produced. If only `kl` is present, `ku` defaults to `kl`.

Distribution `mode` can be:

1	One large singular value.
2	One small singular value.
3	Geometrically distributed singular values (default).
4	Arithmetically distributed singular values.

`5`	Random singular values with uniformly distributed logarithm.
`< 0`	If `mode` is `-1`, `-2`, `-3`, `-4`, or `-5`, then `randsvd` treats `mode` as `abs(mode)`, except that in the original matrix of singular values the order of the diagonal entries is reversed: small to large instead of large to small.

Condition number `kappa` defaults to `sqrt(1/eps)`. In the special case where `kappa < 0`, `A` is a random, full, symmetric, positive definite matrix with `cond(A) = -kappa` and eigenvalues distributed according to `mode`. Arguments `kl` and `ku`, if present, are ignored.

`A = gallery('randsvd',n,kappa,mode,kl,ku,method)` specifies how the computations are carried out. `method = 0` is the default, while `method = 1` uses an alternative method that is much faster for large dimensions, even though it uses more flops.

redheff — Redheffer's matrix of 1s and 0s

`A = gallery('redheff',n)` returns an `n`-by-`n` matrix of `0`'s and `1`'s defined by `A(i,j) = 1`, if `j = 1` or if `i` divides `j`, and `A(i,j) = 0` otherwise.

The Redheffer matrix has these properties:

- `(n-floor(log2(n)))-1` eigenvalues equal to `1`
- A real eigenvalue (the spectral radius) approximately `sqrt(n)`
- A negative eigenvalue approximately `-sqrt(n)`
- The remaining eigenvalues are provably "small."
- The Riemann hypothesis is true if and only if $\det(A) = O(n^{\frac{1}{2} + \varepsilon})$ for every epsilon `> 0`.

Barrett and Jarvis conjecture that "the small eigenvalues all lie inside the unit circle `abs(Z) = 1`," and a proof of this conjecture, together with a proof that some eigenvalue tends to zero as `n` tends to infinity, would yield a new proof of the prime number theorem.

riemann — Matrix associated with the Riemann hypothesis

`A = gallery('riemann',n)` returns an `n`-by-`n` matrix for which the Riemann hypothesis is true if and only if

$$\det(A) = O(n!n^{-\frac{1}{2}+\varepsilon})$$

for every $\varepsilon > 0$.

The Riemann matrix is defined by:

```
A = B(2:n+1,2:n+1)
```

where `B(i,j) = i-1` if `i` divides `j`, and `B(i,j) = -1` otherwise.

The Riemann matrix has these properties:

- Each eigenvalue `e(i)` satisfies `abs(e(i)) <= m-1/m`, where `m = n+1`.
- `i <= e(i) <= i+1` with at most `m-sqrt(m)` exceptions.
- All integers in the interval `(m/3, m/2]` are eigenvalues.

ris — Symmetric Hankel matrix

`A = gallery('ris',n)` returns a symmetric `n`-by-`n` Hankel matrix with elements

```
A(i,j) = 0.5/(n-i-j+1.5)
```

The eigenvalues of `A` cluster around $\pi/2$ and $-\pi/2$. This matrix was invented by F.N. Ris.

sampling — Nonsymmetric matrix with ill-conditioned integer eigenvalues.

`A = gallery('sampling',x)`, where `x` is an `n`-vector, is the `n`-by-`n` matrix with `A(i,j) = X(i)/(X(i)-X(j))` for `i ~= j` and `A(j,j)` the sum of the off-diagonal elements in column `j`. A has eigenvalues `0:n-1`. For the eigenvalues 0 and `n`–1, corresponding eigenvectors are `X` and `ones(n,1)`, respectively.

The eigenvalues are ill-conditioned. `A` has the property that `A(i,j) + A(j,i) = 1` for `i ~= j`.

Explicit formulas are available for the left eigenvectors of `A`. For scalar `n`, `sampling(n)` is the same as `sampling(1:n)`. A special case of this matrix arises in sampling theory.

smoke — Complex matrix with a 'smoke ring' pseudospectrum

`A = gallery('smoke',n)` returns an `n`-by-`n` matrix with `1`'s on the superdiagonal, `1` in the `(n,1)` position, and powers of roots of unity along the diagonal.

`A = gallery('smoke',n,1)` returns the same except that element `A(n,1)` is zero.

The eigenvalues of `gallery('smoke',n,1)` are the `n`th roots of unity; those of `gallery('smoke',n)` are the `n`th roots of unity times `2^(1/n)`.

toeppd — Symmetric positive definite Toeplitz matrix

`A = gallery('toeppd',n,m,w,theta)` returns an `n`-by-`n` symmetric, positive semi-definite (SPD) Toeplitz matrix composed of the sum of `m` rank 2 (or, for certain `theta`, rank 1) SPD Toeplitz matrices. Specifically,

```
T = w(1)*T(theta(1)) + ... + w(m)*T(theta(m))
```

where `T(theta(k))` has `(i,j)` element `cos(2*pi*theta(k)*(i-j))`.

By default: `m = n`, `w = rand(m,1)`, and `theta = rand(m,1)`.

toeppen — Pentadiagonal Toeplitz matrix (sparse)

`P = gallery('toeppen',n,a,b,c,d,e)` returns the `n`-by-`n` sparse, pentadiagonal Toeplitz matrix with the diagonals: `P(3,1) = a`, `P(2,1) = b`, `P(1,1) = c`, `P(1,2) = d`, and `P(1,3) = e`, where `a`, `b`, `c`, `d`, and `e` are scalars.

By default, `(a,b,c,d,e) = (1,-10,0,10,1)`, yielding a matrix of Rutishauser. This matrix has eigenvalues lying approximately on the line segment `2*cos(2*t) + 20*i*sin(t)`.

tridiag — Tridiagonal matrix (sparse)

`A = gallery('tridiag',c,d,e)` returns the tridiagonal matrix with subdiagonal `c`, diagonal `d`, and superdiagonal `e`. Vectors `c` and `e` must have `length(d)-1`.

`A = gallery('tridiag',n,c,d,e)`, where `c`, `d`, and `e` are all scalars, yields the Toeplitz tridiagonal matrix of order `n` with subdiagonal elements `c`, diagonal elements `d`, and superdiagonal elements `e`. This matrix has eigenvalues

```
d + 2*sqrt(c*e)*cos(k*pi/(n+1))
```

where `k = 1:n`. (see [1].)

`A = gallery('tridiag',n)` is the same as `A = gallery('tridiag',n,-1,2,-1)`, which is a symmetric positive definite M-matrix (the negative of the second difference matrix).

triw — Upper triangular matrix discussed by Wilkinson and others

`A = gallery('triw',n,alpha,k)` returns the upper triangular matrix with ones on the diagonal and `alpha`s on the first `k >= 0` superdiagonals.

Order `n` may be a 2-element vector, in which case the matrix is `n(1)`-by-`n(2)` and upper trapezoidal.

Ostrowski ["On the Spectrum of a One-parametric Family of Matrices," *J. Reine Angew. Math.*, 1954] shows that

```
cond(gallery('triw',n,2)) = cot(pi/(4*n))^2,
```

and, for large `abs(alpha)`, `cond(gallery('triw',n,alpha))` is approximately `abs(alpha)^n*sin(pi/(4*n-2))`.

Adding `-2^(2-n)` to the `(n,1)` element makes `triw(n)` singular, as does adding `-2^(1-n)` to all the elements in the first column.

uniformdata — Array of arbitrary data from standard uniform distribution

`A = gallery('uniformdata',[m,n,...],j)` returns an `m`-by-`n`-by-... array `A`. The values of `A` are a random sample from the standard uniform distribution. j must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('uniformdata', ...)` with different values of `j` will return different arrays. Repeated calls to `gallery('uniformdata',...)` with the same size vector and `j` inputs will always return the same array.

In any call to `gallery('uniformdata', ...)` you can substitute individual inputs `m`,`n`,... for the size vector input `[m,n,...]`. For example, `gallery('uniformdata',[1,2,3,4],5)` is equivalent to `gallery('uniformdata',1,2,3,4,5)`.

`[A,B,...] = gallery('uniformdata',[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('uniformdata',[m,n,...],j, classname)` produces a matrix of class `classname`. `classname` must be either `'single'` or `'double'`.

Generate the arbitrary 6-by-4 matrix of data from the uniform distribution on [0, 1] corresponding to `j = 2`.

```
x = gallery('uniformdata', [6, 4], 2);
```

Generate the arbitrary 1-by-2-by-3 single array of data from the uniform distribution on [0, 1] corresponding to `j = 17`.

```
y = gallery('uniformdata', 1, 2, 3, 17, 'single');
```

wathen — Finite element matrix (sparse, random entries)

`A = gallery('wathen',nx,ny)` returns a sparse, random, `n`-by-`n` finite element matrix where `n = 3*nx*ny + 2*nx + 2*ny + 1`.

Matrix `A` is precisely the “consistent mass matrix” for a regular `nx`-by-`ny` grid of 8-node (serendipity) elements in two dimensions. `A` is symmetric, positive definite for any (positive) values of the “density,” `rho(nx,ny)`, which is chosen randomly in this routine.

`A = gallery('wathen',nx,ny,1)` returns a diagonally scaled matrix such that

```
0.25 <= eig(inv(D)*A) <= 4.5
```

where `D = diag(diag(A))` for any positive integers `nx` and `ny` and any densities `rho(nx,ny)`.

wilk — Various matrices devised or discussed by Wilkinson

`gallery('wilk',n)` returns a different matrix or linear system depending on the value of `n`.

`n = 3`	Upper triangular system `Ux=b` illustrating inaccurate solution.
`n = 4`	Lower triangular system `Lx=b`, ill-conditioned.
`n = 5`	`hilb(6)(1:5,2:6)*1.8144`. A symmetric positive definite matrix.
`n = 21`	`W21+`, a tridiagonal matrix. eigenvalue problem. For more detail, see [2].

See Also

`hadamard`, `hilb`, `invhilb`, `magic`, `wilkinson`

References

[1] The MATLAB gallery of test matrices is based upon the work of Nicholas J. Higham at the Department of Mathematics, University of Manchester, Manchester, England. Further background can be found in the books *MATLAB Guide, Second Edition*, Desmond J. Higham and Nicholas J. Higham, SIAM, 2005, and *Accuracy and Stability of Numerical Algorithms,* Nicholas J. Higham, SIAM, 1996.

[2] Wilkinson, J. H., *The Algebraic Eigenvalue Problem*, Oxford University Press, London, 1965, p.308.

Purpose Gamma functions

Syntax

```
Y = gamma(A)
Y = gammainc(X,A)
Y = gammainc(X,A,tail)
Y = gammaln(A)
```

Definition The `gamma` function is defined by the integral:

$$\Gamma(a) = \int_0^\infty e^{-t}\, t^{a-1} dt$$

The `gamma` function interpolates the `factorial` function. For integer `n`:

```
gamma(n+1) = n! = prod(1:n)
```

The incomplete `gamma` function is:

$$P(x, a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

For any `a>=0`, `gammainc(x,a)` approaches 1 as `x` approaches `infinity`. For small `x` and `a`, `gammainc(x,a)` is approximately equal to `x^a`, so `gammainc(0,0) = 1`.

Description `Y = gamma(A)` returns the `gamma` function at the elements of `A`. `A` must be real.

`Y = gammainc(X,A)` returns the incomplete `gamma` function of corresponding elements of `X` and `A`. Arguments `X` and `A` must be real and the same size (or either can be scalar).

`Y = gammainc(X,A,tail)` specifies the tail of the incomplete `gamma` function when `X` is non-negative. The choices are for `tail` are `'lower'` (the default) and `'upper'`. The upper incomplete `gamma` function is defined as

```
1 - gammainc(x,a)
```

Note When `X` is negative, `Y` can be inaccurate for `abs(X)>A+1`.

`Y = gammaln(A)` returns the logarithm of the `gamma` function, `gammaln(A) = log(gamma(A))`. The `gammaln` command avoids the underflow and overflow that may occur if it is computed directly using `log(gamma(A))`.

Algorithm

The computations of `gamma` and `gammaln` are based on algorithms outlined in [1]. Several different minimax rational approximations are used depending upon the value of `A`. Computation of the incomplete `gamma` function is based on the algorithm in [2].

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

Purpose

Inverse incomplete gamma function

Syntax

```
x = gammaincinv(y,a)
y = gammaincinv(x,a,tail)
```

Description

`x = gammaincinv(y,a)` evaluates the inverse incomplete gamma function for corresponding elements of `y` and `a`, such that `y = gammainc(x,a)`. The elements of `y` must be in the closed interval `[0,1]`, and those of `a` must be nonnegative. `y` and `a` must be real and the same size (or either can be a scalar).

`y = gammaincinv(x,a,tail)` specifies the tail of the incomplete gamma function. Choices are `lower` (the default) to use the integral from 0 to `x`, or `upper` to use the integral from `x` to infinity. These two choices are related as `gammaincinv(y,a,'upper') = gammaincinv(1-y,a,'lower')`. When `y` is close to 0, the `upper` option provides a way to compute `x` more accurately than by subtracting `y` from 1.

Definition

The lower incomplete gamma function is defined as:

$$\text{gammainc}(x,a) = \frac{1}{\Gamma(a)}\int_0^x t^{(a-1)}e^{-t}dt$$

The upper incomplete gamma function is defined as:

$$\text{gammainc}(x,a) = \frac{1}{\Gamma(a)}\int_x^\infty t^{(a-1)}e^{-t}dt$$

`gammaincinv` computes the inverse of the incomplete gamma function with respect to the integration limit `x` using Newton's method.

For any `a>0`, as `y` approaches 1, `gammaincinv(y,a)` approaches infinity.

For small `x` and `a`, `gammainc(x,a)` $\cong x^a$, so `gammaincinv(1,0) = 0`.

See Also

`gamma`, `gammainc`, `gammaln`, `psi`

gca

Purpose Current axes handle

Syntax `h = gca`

Description `h = gca` returns the handle to the current axes for the current figure. If no axes exists, the MATLAB software creates one and returns its handle. You can use the statement

```
get(gcf,'CurrentAxes')
```

if you do not want MATLAB to create an axes if one does not already exist.

Current Axes

The current axes is the target for graphics output when you create axes children. The current axes is typically the last axes used for plotting or the last axes clicked on by the mouse. Graphics commands such as `plot`, `text`, and `surf` draw their results in the current axes. Changing the current figure also changes the current axes.

See Also `axes`, `cla`, `gcf`, `findobj`

figure `CurrentAxes` property

"Graphics Object Identification" on page 1-98 for related functions

Purpose Handle of figure containing object whose callback is executing

Syntax `fig = gcbf`

Description `fig = gcbf` returns the handle of the figure that contains the object whose callback is currently executing. This object can be the figure itself, in which case, `gcbf` returns the figure's handle.

When no callback is executing, `gcbf` returns the empty matrix, `[]`.

The value returned by `gcbf` is identical to the `figure` output argument returned by `gcbo`.

See Also `gcbo`, `gco`, `gcf`, `gca`

Purpose

Handle of object whose callback is executing

Syntax

```
h = gcbo
[h,figure] = gcbo
```

Description

`h = gcbo` returns the handle of the graphics object whose callback is executing.

`[h,figure] = gcbo` returns the handle of the current callback object and the handle of the figure containing this object.

Remarks

The MATLAB software stores the handle of the object whose callback is executing in the root `CallbackObject` property. If a callback interrupts another callback, MATLAB replaces the `CallbackObject` value with the handle of the object whose callback is interrupting. When that callback completes, MATLAB restores the handle of the object whose callback was interrupted.

The root `CallbackObject` property is read only, so its value is always valid at any time during callback execution. The root `CurrentFigure` property, and the figure `CurrentAxes` and `CurrentObject` properties (returned by `gcf`, `gca`, and `gco`, respectively) are user settable, so they can change during the execution of a callback, especially if that callback is interrupted by another callback. Therefore, those functions are not reliable indicators of which object's callback is executing.

When you write callback routines for the `CreateFcn` and `DeleteFcn` of any object and the figure `ResizeFcn`, you must use `gcbo` since those callbacks do not update the root's `CurrentFigure` property, or the figure's `CurrentObject` or `CurrentAxes` properties; they only update the root's `CallbackObject` property.

When no callbacks are executing, `gcbo` returns `[]` (an empty matrix).

See Also

`gca`, `gcf`, `gco`, `rootobject`

for related functions.

Purpose

Greatest common divisor

Syntax

```
G = gcd(A,B)
[G,C,D] = gcd(A,B)
```

Description

`G = gcd(A,B)` returns an array containing the greatest common divisors of the corresponding elements of integer arrays `A` and `B`. By convention, `gcd(0,0)` returns a value of `0`; all other inputs return positive integers for `G`.

`[G,C,D] = gcd(A,B)` returns both the greatest common divisor array `G`, and the arrays `C` and `D`, which satisfy the equation: `A(i).*C(i) + B(i).*D(i) = G(i)`. These are useful for solving Diophantine equations and computing elementary Hermite transformations.

Examples

The first example involves elementary Hermite transformations.

For any two integers `a` and `b` there is a `2`-by-`2` matrix `E` with integer entries and determinant = `1` (a *unimodular* matrix) such that:

```
E * [a;b] = [g,0],
```

where `g` is the greatest common divisor of `a` and `b` as returned by the command `[g,c,d] = gcd(a,b)`.

The matrix `E` equals:

```
c       d
-b/g    a/g
```

In the case where `a = 2` and `b = 4`:

```
[g,c,d] = gcd(2,4)
g =
     2
c =
     1
d =
     0
```

So that

```
E =
     1     0
    -2     1
```

In the next example, we solve for `x` and `y` in the Diophantine equation `30x + 56y = 8`.

```
[g,c,d] = gcd(30,56)
g =
      2
c =
    -13
d =
      7
```

By the definition, for scalars `c` and `d`:

```
30(-13) + 56(7) = 2,
```

Multiplying through by 8/2:

```
30(-13*4) + 56(7*4) = 8
```

Comparing this to the original equation, a solution can be read by inspection:

```
x = (-13*4) = -52; y = (7*4) = 28
```

See Also

`lcm`

References

[1] Knuth, Donald, *The Art of Computer Programming*, Vol. 2, Addison-Wesley: Reading MA, 1973. Section 4.5.2, Algorithm X.

Purpose

Current figure handle

Syntax

```
h = gcf
```

Description

`h = gcf` returns the handle of the current figure. The current figure is the figure window in which graphics commands such as `plot`, `title`, and `surf` draw their results. If no figure exists, the MATLAB software creates one and returns its handle. You can use the statement

```
get(0,'CurrentFigure')
```

if you do not want MATLAB to create a figure if one does not already exist.

See Also

`clf`, `figure`, `gca`

Root `CurrentFigure` property

"Graphics Object Identification" on page 1-98 for related functions

Purpose

Handle of current object

Syntax

```
h = gco
h = gco(figure_handle)
```

Description

`h = gco` *returns* the handle of the current object.

`h = gco(figure_handle)` returns the value of the current object for the figure specified by `figure_handle`.

Remarks

The current object is the last object clicked on, excluding `uimenus`. If the mouse click did not occur over a figure child object, the figure becomes the current object. The MATLAB software stores the handle of the current object in the figure's `CurrentObject` property.

The `CurrentObject` of the `CurrentFigure` does not always indicate the object whose callback is being executed. Interruptions of callbacks by other callbacks can change the `CurrentObject` or even the `CurrentFigure`. Some callbacks, such as `CreateFcn` and `DeleteFcn`, and uimenu `Callback`, intentionally do not update `CurrentFigure` or `CurrentObject`.

`gcbo` provides the only completely reliable way to retrieve the handle to the object whose callback is executing, at any point in the `callback` function, regardless of the type of callback or of any previous interruptions.

Examples

This statement returns the handle to the current object in figure window 2:

```
h = gco(2)
```

See Also

`gca`, `gcbo`, `gcf`

The `root` object description

"Graphics Object Identification" on page 1-98 for related functions

Purpose

Test for greater than or equal to

Syntax

```
A >= B
ge(A, B)
```

Description

`A >= B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is greater than or equal to `B`, or set to logical 0 (`false`) where `A` is less than `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLAB software returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`ge(A, B)` is called for the syntax `A >= B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are greater than or equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A >= B
ans =
     1     0     0     1     1     1
     0     1     0     1     1     1
     1     0     0     1     1     1
     0     1     1     0     1     0
```

```
     1     0     1     1     0     0
     0     1     1     1     0     1
```

See Also `gt`, `eq`, `le`, `lt`, `ne`,

genpath

Purpose

Generate path string

Syntax

```
genpath
genpath folderName
p = genpath('folderName')
```

Description

`genpath` returns a path string that includes all the folders and subfolders below *`matlabroot`*`/toolbox`, including empty subfolders.

`genpath folderName` returns a path string that includes `folderName` and multiple levels of subfolders below `folderName`. The path string does not include folders named `private` or folders that begin with the `@` character (class folders) or the `+` character (package folders).

`p = genpath('folderName')` returns the path string to variable, `p`.

Examples

Generate a path that includes *`matlabroot`*`/toolbox/images` and all folders below it:

```
p = genpath(fullfile(matlabroot,'toolbox','images'))
```

```
p =

   C:\Program Files\MATLAB\R20nn\toolbox\images;C:\Program Files\MATLA
   R20nn\toolbox\images\images;C:\Program Files\MATLAB\R200nn\toolbox\
   images\images\ja;C:\Program Files\MATLAB\R200nn\toolbox\images\
   imdemos;C:\Program Files\MATLAB\R200nn\toolbox\images\imdemos\ja;
```

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

Use `genpath` in conjunction with `addpath` to add a folder and its subfolders to the search path. Add `mymfiles` and its subfolders to the search path:

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

See Also

`addpath`, `path`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `rmpath`, `savepath`

genvarname

Purpose

Construct valid variable name from string

Syntax

```
varname = genvarname(str)
varname = genvarname(str, exclusions)
```

Description

`varname = genvarname(str)` constructs a string `varname` that is similar to or the same as the `str` input, and can be used as a valid variable name. `str` can be a single character array or a cell array of strings. If `str` is a cell array of strings, `genvarname` returns a cell array of strings in `varname`. The strings in a cell array returned by `genvarname` are guaranteed to be different from each other.

`varname = genvarname(str, exclusions)` returns a valid variable name that is different from any name listed in the `exclusions` input. The `exclusions` input can be a single character array or a cell array of strings. Specify the function `who` in the `exclusions` character array to create a variable name that will be unique in the current MATLAB workspace (see "Example 4" on page 2-1512, below).

Note `genvarname` returns a string that can be used as a variable name. It does not create a variable in the MATLAB workspace. You cannot, therefore, assign a value to the output of `genvarname`.

Remarks

A valid MATLAB variable name is a character string of letters, digits, and underscores, such that the first character is a letter, and the length of the string is less than or equal to the value returned by the `namelengthmax` function. Any string that exceeds `namelengthmax` is truncated in the `varname` output. See "Example 6" on page 2-1513, below.

The variable name returned by `genvarname` is not guaranteed to be different from other variable names currently in the MATLAB workspace unless you use the `exclusions` input in the manner shown in "Example 4" on page 2-1512, below.

If you use `genvarname` to generate a field name for a structure, MATLAB does create a variable for the structure and field in the MATLAB workspace. See "Example 3" on page 2-1511, below.

If the `str` input contains any whitespace characters, `genvarname` removes then and capitalizes the next alphabetic character in `str`. If `str` contains any nonalphanumeric characters, `genvarname` translates these characters into their hexadecimal value.

Examples

Example 1

Create four similar variable name strings that do not conflict with each other:

```
v = genvarname({'A', 'A', 'A', 'A'})
v =
    'A'    'A1'    'A2'    'A3'
```

Example 2

Read a column header `hdr` from worksheet `trial2` in Excel spreadsheet myproj_apr23:

```
[data hdr] = xlsread('myproj_apr23.xls', 'trial2');
```

Make a variable name from the text of the column header that will not conflict with other names:

```
v = genvarname(['Column ' hdr{1,3}]);
```

Assign data taken from the spreadsheet to the variable in the MATLAB workspace:

```
eval([v '= data(1:7, 3);']);
```

Example 3

Collect readings from an instrument once every minute over the period of an hour into different fields of a structure. `genvarname` not only generates unique fieldname strings, but also creates the structure and fields in the MATLAB workspace:

```
for k = 1:60
record.(genvarname(['reading' datestr(clock, 'HHMMSS')])) = takeReading;
pause(60)
end
```

After the program ends, display the recorded data from the workspace:

```
record
record =
    reading090446: 27.3960
    reading090546: 23.4890
    reading090646: 21.1140
    reading090746: 23.0730
    reading090846: 28.5650
         .
         .
         .
```

Example 4

Generate variable names that are unique in the MATLAB workspace by putting the output from the `who` function in the `exclusions` list.

```
for k = 1:5
   t = clock;
   pause(uint8(rand * 10));
   v = genvarname('time_elapsed', who);
   eval([v ' = etime(clock,t)'])
   end
```

As this code runs, you can see that the variables created by `genvarname` are unique in the workspace:

```
time_elapsed =
    5.0070
time_elapsed1 =
    2.0030
time_elapsed2 =
    7.0010
```

```
time_elapsed3 =
    8.0010
time_elapsed4 =
    3.0040
```

After the program completes, use the `who` function to view the workspace variables:

```
who

k           time_elapsed   time_elapsed2  time_elapsed4
t           time_elapsed1  time_elapsed3  v
```

Example 5

If you try to make a variable name from a MATLAB keyword, `genvarname` creates a variable name string that capitalizes the keyword and precedes it with the letter `x`:

```
v = genvarname('global')
v =
   xGlobal
```

Example 6

If you enter a string that is longer than the value returned by the `namelengthmax` function, `genvarname` truncates the resulting variable name string:

```
namelengthmax
ans =
    63

vstr = genvarname(sprintf('%s%s', ...
   'This name truncates because it contains ', ...
   'more than the maximum number of characters'))
vstr =
ThisNameTruncatesBecauseItContainsMoreThanTheMaximumNumberOfCha
```

See Also

`isvarname`, `iskeyword`, `isletter`, `namelengthmax`, `who`, `regexp`

Purpose

Query Handle Graphics object properties

Syntax

```
get(h)
get(h,'PropertyName')
<m-by-n value cell array> = get(H,pn)
a = get(h)
a = get(0)
a = get(0,'Factory')
a = get(0,'FactoryObjectTypePropertyName')
a = get(h,'Default')
a = get(h,'DefaultObjectTypePropertyName')
```

Description

Note Do not use the `get` function on Java objects as it will cause a memory leak. For more information, see

---

`get(h)` returns all properties of the graphics object identified by the handle `h` and their current values. For this syntax, `h` must be a scalar.

`get(h,'PropertyName')` returns the value of the property `'PropertyName'` of the graphics object identified by `h`.

`<m-by-n value cell array> = get(H,pn)` returns *n* property values for *m* graphics objects in the *m*-by-*n* cell array, where `m = length(H)` and *n* is equal to the number of property names contained in `pn`.

`a = get(h)` returns a structure whose field names are the object's property names and whose values are the current values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen. For this syntax, `h` may be a scalar or a `m`-by-`n` array of handles. If `h` is a vector, `a` will be a `(m*n)`-by-1 struct array.

`a = get(0)` returns the current values of all user-settable properties. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0,'Factory')` returns the factory-defined values of all user-settable properties. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0,'Factory`*`ObjectTypePropertyName`*`')` returns the factory-defined value of the named property for the specified object type. The argument *`FactoryObjectTypePropertyName`* is the word `Factory` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`)`FactoryFigureColor`.

`a = get(h,'Default')` returns all default values currently defined on object `h`. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(h,'Default`*`ObjectTypePropertyName`*`')` returns the factory-defined value of the named property for the specified object type. The argument *`DefaultObjectTypePropertyName`* is the word `Default` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`).

```
DefaultFigureColor
```

Examples

You can obtain the default value of the `LineWidth` property for line graphics objects defined on the root level with the statement

```
get(0,'DefaultLineLineWidth')
ans =
  0.5000
```

To query a set of properties on all axes children, define a cell array of property names:

```
props = {'HandleVisibility', 'Interruptible';
  'SelectionHighlight', 'Type'};
output = get(get(gca,'Children'),props);
```

The variable `output` is a cell array of dimension `length(get(gca,'Children')`-by-4.

For example, type

```
patch;surface;text;line
output = get(get(gca,'Children'),props)
output =
 'on'    'on'    'on'    'line'
 'on'    'off'   'on'    'text'
 'on'    'on'    'on'    'surface'
 'on'    'on'    'on'    'patch'
```

See Also

`findobj`, `gca`, `gcf`, `gco`, `set`

Handle Graphics Properties

“Graphics Object Identification” on page 1-98 for related functions

Purpose Get property value from interface, or display properties

Syntax

```
V = h.get
V = h.get('propertyname')
V = get(h, ...)
```

Description `V = h.get` returns a list of all properties and their values for the object or interface, `h`.

If `V` is empty, either there are no properties in the object, or the MATLAB software cannot read the object's type library. Refer to the COM vendor's documentation. For Automation objects, if the vendor provides documentation for a specific property, use the `V = get(h, ...)` syntax to call it.

`V = h.get('propertyname')` returns the value of the property specified in the string, `propertyname`.

`V = get(h, ...)` is an alternate syntax for the same operation.

Remarks The meaning and type of the return value is dependent upon the specific property being retrieved. The object's documentation should describe the specific meaning of the return value. MATLAB may convert the data type of the return value. For a description of how MATLAB converts COM data types, see in the External Interfaces documentation.

COM functions are available on Microsoft Windows systems only.

Examples Create a COM server running Microsoft Excel software:

```
e = actxserver ('Excel.Application');
```

Retrieve a single property value:

```
e.Path
```

Depending on your spreadsheet program, MATLAB software displays:

```
ans =
```

```
    C:\Program Files\MSOffice\OFFICE11
```

Retrieve a list of all properties for the `CommandBars` interface:

```
c = e.CommandBars.get
```

MATLAB displays information similar to the following:

```
c =
                    Application: [1x1
Interface.Microsoft_Excel_11.0_Object_Library._Application]
                        Creator: 1.4808e+009
                  ActionControl: []
                  ActiveMenuBar: [1x1
Interface.Microsoft_Office_12.0_Object_Library.CommandBar]
                          Count: 129
                DisplayTooltips: 1
          DisplayKeysInTooltips: 0
                   LargeButtons: 0
             MenuAnimationStyle: 'msoMenuAnimationNone'
                         Parent: [1x1
Interface.Microsoft_Excel_11.0_Object_Library._Application]
                  AdaptiveMenus: 0
                   DisplayFonts: 1
               DisableCustomize: 0
    DisableAskAQuestionDropdown: 0
```

See Also

`set (COM)`, `inspect`, `isprop`, `addproperty`, `deleteproperty`

Purpose

Query property values of handle objects derived from `hgsetget` class

Syntax

```
CV = get(H,'PropertyName')
SV = get(h)
get(h)
```

Description

`CV = get(H,'PropertyName')` returns the value of the named property from the objects in the handle array `H`. If `H` is scalar, `get` returns a single value; if `H` is an array, `get` returns a cell array of property values. If you specify a cell array of property names, then `get` returns a cell array of values, where each row in the cell corresponds to an element in `H` and each column in the cell corresponds to an element in the property name cell array.

If `H` is nonscalar and `PropertyName` is the name of a dynamic property, `get` returns a value only if the property exists in all objects referenced in `H`.

`SV = get(h)` returns a `struct` array in which the field names are the object's property names and the values are the current values of the corresponding properties. If `h` is an array, then `SV` is a `numel(h)`-by-`1` array of `structs`.

`get(h)` displays all property names and their current values for the MATLAB objects with handle `h`.

Your subclass can override the `hgsetget` `getdisp` method to control how MATLAB displays this information.

See Also

See

`get`, `handle`, `hgsetget`, `set (hgsetget)`

get (memmapfile)

Purpose Memmapfile object properties

Syntax

```
s = get(obj)
val = get(obj, prop)
```

Description `s = get(obj)` returns the values of all properties of the `memmapfile` object `obj` in structure array `s`. Each property retrieved from the object is represented by a field in the output structure. The name and contents of each field are the same as the name and value of the property it represents.

Note Although property names of a `memmapfile` object are not case sensitive, field names of the output structure returned by `get` (named the same as the properties they represent) are case sensitive.

`val = get(obj, prop)` returns the value(s) of one or more properties specified by `prop`. The `prop` input can be a quoted string or a cell array of quoted strings, each containing a property name. If the latter is true, `get` returns the property values in a cell array.

Examples You can use the `get` method of the `memmapfile` class to return information on any or all of the object's properties. Specify one or more property names to get the values of specific properties.

This example returns the values of the `Offset`, `Repeat`, and `Format` properties for a `memmapfile` object. Start by constructing the object:

```
m = memmapfile('records.dat',                 ...
               'Offset', 2048,                ...
               'Format', {                    ...
                  'int16'  [2 2] 'model';     ...
                  'uint32' [1 1] 'serialno';  ...
                  'single' [1 3] 'expenses'});
```

Use the `get` method to return the specified property values in a 1-by-3 cell array `m_props`:

```
m_props = get(m, {'Offset', 'Repeat', 'Format'})
m_props =
    [2048]    [Inf]    {3x3 cell}

m_props{3}
ans =
    'int16'     [1x2 double]    'model'
    'uint32'    [1x2 double]    'serialno'
    'single'    [1x2 double]    'expenses'
```

Another way to return the same information is to use the `objname.property` syntax:

```
m_props = {m.Offset, m.Repeat, m.Format}
m_props =
    [2048]    [Inf]    {3x3 cell}
```

To return the values for all properties with `get`, pass just the object name:

```
s = get(m)
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: 0
      Offset: 2048
      Format: {3x3 cell}
      Repeat: Inf
        Data: [753 1]
```

To see just the `Format` field of the returned structure, type

```
s.Format
ans =
    'int16'     [1x2 double]    'model'
    'uint32'    [1x2 double]    'serialno'
    'single'    [1x2 double]    'expenses'
```

See Also `memmapfile`, `disp(memmapfile)`

get (RandStream)

Purpose Random stream properties

Class `@RandStream`

Syntax

```
get(s)
P = get(s)
P = get(s,'PropertyName')
```

Description `get(s)` prints the list of properties for the random stream `s`.

`P = get(s)` returns all properties of `s` in a scalar structure.

`P = get(s,'PropertyName')` returns the property `'PropertyName'`.

See Also `@RandStream`, `set (RandStream)`

get (serial)

Purpose

Serial port object properties

Syntax

```
get(obj)
out = get(obj)
out = get(obj,'PropertyName')
```

Description

`get(obj)` returns all property names and their current values to the command line for the serial port object, `obj`.

`out = get(obj)` returns the structure `out` where each field name is the name of a property of `obj`, and each field contains the value of that property.

`out = get(obj,'PropertyName')` returns the value `out` of the property specified by *`PropertyName`* for `obj`. If *`PropertyName`* is replaced by a 1-by-n or n-by-1 cell array of strings containing property names, then `get` returns a 1-by-n cell array of values to `out`. If `obj` is an array of serial port objects, then `out` will be a m-by-n cell array of property values where m is equal to the length of `obj` and n is equal to the number of properties specified.

Remarks

Refer to Displaying Property Names and Property Values for a list of serial port object properties that you can return with `get`.

When you specify a property name, you can do so without regard to case, and you can make use of property name completion. For example, if `s` is a serial port object, then these commands are all valid.

```
out = get(s,'BaudRate');
out = get(s,'baudrate');
out = get(s,'BAUD');
```

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

```
help serial/get
```

Example

This example illustrates some of the ways you can use `get` to return property values for the serial port object `s` on a Windows platform.

```
s = serial('COM1');
out1 = get(s);
out2 = get(s,{'BaudRate','DataBits'});
get(s,'Parity')
ans =
none
```

See Also

Functions

`set`

get (timer)

Purpose Timer object properties

Syntax

```
get(obj)
V = get(obj)
V = get(obj,'PropertyName')
```

Description `get(obj)` displays all property names and their current values for the timer object `obj`. `obj` must be a single timer object.

`V = get(obj)` returns a structure, `V`, where each field name is the name of a property of `obj` and each field contains the value of that property. If `obj` is an M-by-1 vector of timer objects, V is an M-by-1 array of structures.

`V = get(obj,'PropertyName')` returns the value, `V`, of the timer object property specified in *`PropertyName`*.

If *`PropertyName`* is a 1-by-N or N-by-1 cell array of strings containing property names, `V` is a 1-by-N cell array of values. If `obj` is a vector of timer objects, `V` is an M-by-N cell array of property values where M is equal to the length of `obj` and N is equal to the number of properties specified.

Examples

```
t = timer;
get(t)
       AveragePeriod: NaN
            BusyMode: 'drop'
            ErrorFcn: ''
       ExecutionMode: 'singleShot'
       InstantPeriod: NaN
                Name: 'timer-1'
    ObjectVisibility: 'on'
              Period: 1
             Running: 'off'
          StartDelay: 1
            StartFcn: ''
             StopFcn: ''
                 Tag: ''
```

```
          TasksExecuted: 0
         TasksToExecute: Inf
               TimerFcn: ''
                   Type: 'timer'
               UserData: []
get(t, {'StartDelay','Period'})
ans =

    [0]    [1]
```

See Also `timer`, `set(timer)`

get (timeseries)

Purpose Query `timeseries` object property values

Syntax

```
value = get(ts,'PropertyName')
get(ts)
```

Description `value = get(ts,'PropertyName')` returns the value of the specified property of the `timeseries` object. The following syntax is equivalent:

```
value = ts.PropertyName
```

`get(ts)` displays all properties and values of the time series `ts`.

See Also `set (timeseries)`, `timeseries`, `tsprops`

Purpose

Query `tscollection` object property values

Syntax

```
value = get(tsc,'PropertyName')
```

Description

`value = get(tsc,'PropertyName')` returns the value of the specified property of the `tscollection` object tsc. The following syntax is equivalent:

```
value = tsc.PropertyName
```

`get(tsc)` displays all properties and values of the `tscollection` object tsc.

See Also

`set (tscollection)`, `tscollection`

getabstime (timeseries)

Purpose

Extract date-string time vector into cell array

Syntax

```
getabstime(ts)
```

Description

`getabstime(ts)` extracts the time vector from the `timeseries` object `ts` as a cell array of date strings. To define the time vector relative to a calendar date, set the `TimeInfo.StartDate` property of the `timeseries` object. When the `TimeInfo.StartDate` format is a valid `datestr` format, the output strings from `getabstime` have the same format.

Examples

The following example shows how to extract a time vector as a cell array of date strings from a `timeseries` object.

1. Create a `timeseries` object.

   ```
   ts = timeseries([3 6 8 0 10]);
   ```

   The default time vector for `ts` is [0 1 2 3 4], which starts at 0 and increases in 1-second increments. The length of the time vector is equal to the length of the data.

2. Set the `StartDate` property.

   ```
   ts.TimeInfo.StartDate = '10/27/2005 07:05:36';
   ```

3. Extract the time vector.

   ```
   getabstime(ts)

   ans =

       '27-Oct-2005 07:05:36'
       '27-Oct-2005 07:05:37'
       '27-Oct-2005 07:05:38'
       '27-Oct-2005 07:05:39'
       '27-Oct-2005 07:05:40'
   ```

4. Change the date-string format of the time vector.

   ```
   ts.TimeInfo.Format = 'mm/dd/yy'
   ```

5. Extract the time vector with the new date-string format.

   ```
   getabstime(ts)

   ans =

       '10/27/05'
       '10/27/05'
       '10/27/05'
       '10/27/05'
       '10/27/05'
   ```

See Also

`setabstime (timeseries)`, `timeseries`, `tsprops`

getabstime (tscollection)

Purpose

Extract date-string time vector into cell array

Syntax

```
getabstime(tsc)
```

Description

`getabstime(tsc)` extracts the time vector from the `tscollection` object tsc as a cell array of date strings. To define the time vector relative to a calendar date, set the `TimeInfo.StartDate` property of the time-series collection. When the `TimeInfo.StartDate` format is a valid `datestr` format, the output strings from `getabstime` have the same format.

Examples

1. Create a `tscollection` object.

   ```
   tsc = tscollection(timeseries([3 6 8 0 10]));
   ```

2. Set the `StartDate` property.

   ```
   tsc.TimeInfo.StartDate = '10/27/2005 07:05:36';
   ```

3. Extract a vector of absolute time values.

   ```
   getabstime(tsc)

   ans =

       '27-Oct-2005 07:05:36'
       '27-Oct-2005 07:05:37'
       '27-Oct-2005 07:05:38'
       '27-Oct-2005 07:05:39'
       '27-Oct-2005 07:05:40'
   ```

4. Change the date-string format of the time vector.

   ```
   tsc.TimeInfo.Format = 'mm/dd/yy';
   ```

5. Extract the time vector with the new date-string format.

   ```
   getabstime(tsc)
   ```

```
ans = 

    '10/27/05'
    '10/27/05'
    '10/27/05'
    '10/27/05'
    '10/27/05'
```

See Also

`datestr`, `setabstime (tscollection)`, `tscollection`

Purpose Value of application-defined data

Syntax

```
value = getappdata(h,name)
values = getappdata(h)
```

Description

`value = getappdata(h,name)` gets the value of the application-defined data with the name specified by `name`, in the object with handle `h`. If the application-defined data does not exist, the MATLAB software returns an empty matrix in `value`.

`values = getappdata(h)` returns all application-defined data for the object with handle `h`.

Remarks

Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property.

See Also `setappdata`, `rmappdata`, `isappdata`

GetCharArray

Purpose

Character array from Automation server

Syntax

MATLAB Client

```
str = h.GetCharArray('varname', 'workspace')
str = GetCharArray(h, 'varname', 'workspace')
```

IDL Method Signature

```
HRESULT GetCharArray([in] BSTR varName, [in] BSTR Workspace,  [out,
```

Microsoft Visual Basic Client

```
GetCharArray(varname As String, workspace As String) As String
```

Description

`str = h.GetCharArray('varname', 'workspace')` gets the character array stored in `varname` from the specified `workspace` of the server attached to handle `h` and returns it in `str`. The values for *`workspace`* are`base` or `global`.

`str = GetCharArray(h, 'varname', 'workspace')` is an alternate syntax.

Examples

This example uses a MATLAB client.

```
h = actxserver('matlab.application');
%Assign a string to variable 'str' in the base workspace of the ser
h.PutCharArray('str', 'base', ...
    'He jests at scars that never felt a wound.');
%Read 'str' back in the client
S = h.GetCharArray('str', 'base')
```

This example uses a Visual Basic client.

1. Create the Visual Basic application. Use the `MsgBox` command to control flow between MATLAB and the application.

   ```
   Dim Matlab As Object
   Dim S As String
   ```

```
Matlab = CreateObject("matlab.application")
MsgBox("In MATLAB, type" & vbCrLf _
    & "str='new string';")
```

2 Open the MATLAB window, then type:

```
str='new string';
```

3 Click **Ok**.

```
Try
    S = Matlab.GetCharArray("str", "base")
    MsgBox("str = " & S)
Catch ex As Exception
    MsgBox("You did not set 'str' in MATLAB")
End Try
```

The Visual Basic `MsgBox` displays what you typed in MATLAB.

See Also

`PutCharArray` | `GetWorkspaceData` | `GetVariable`

Purpose

Size of data sample in `timeseries` object

Syntax

```
getdatasamplesize(ts)
```

Description

`getdatasamplesize(ts)` returns the size of each data sample in a `timeseries` object.

Remarks

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

Examples

The following example shows how to get the size of a data sample in a `timeseries` object.

1. Load a 24-by-3 data array.

   ```
   load count.dat
   ```

2. Create a `timeseries` object with 24 time values.

   ```
   count_ts = timeseries(count,[1:24],'Name','VehicleCount')
   ```

3. Get the size of the data sample for this `timeseries` object.

   ```
   getdatasamplesize(count_ts)

   ans =

          1     3
   ```

The size of each data sample in `count_ts` is 1-by-3, which means that each data sample is stored as a row with three values.

See Also

`addsample`, `size (timeseries)`, `tsprops`

getDefaultStream (RandStream)

Purpose Default random number stream

Class `@RandStream`

Syntax

```
stream = RandStream.getDefaultStream
```

Description `stream = RandStream.getDefaultStream` returns the default random number stream. The MATLAB functions `rand`, `randi`, and `randn` use the default stream to generate values.

`rand`, `randi`, and `randn` all rely on the default stream of uniform pseudorandom numbers. `randi` uses one uniform value from the default stream to generate each integer value; `randn` uses one or more uniform values from the default stream to generate each normal value. Note that there are also `rand`, `randi`, and `randn` methods for which you specify a specific random stream from which to draw values.

See Also `@RandStream`, `setDefaultStream (RandStream)`, `rand`, `randi`, `randn`

Purpose

Override to change command window display

Syntax

```
getdisp(H)
```

Description

`getdisp(H)` called by `get` when `get` 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`, `get (hgsetget)`

Purpose Environment variable

Syntax

```
getenv 'name'
N = getenv('name')
```

Description `getenv 'name'` searches the underlying operating system's environment list for a string of the form `name=value`, where `name` is the input string. If found, the MATLAB software returns the string `value`. If the specified name cannot be found, an empty matrix is returned.

`N = getenv('name')` returns `value` to the variable `N`.

Examples

```
os = getenv('OS')

os =
Windows_NT
```

See Also `setenv`, `computer`, `pwd`, `ver`, `path`

Purpose

Field of structure array

Syntax

```
f = getfield(s,'field')
f = getfield(s, {i,j}, 'field', {k})
```

Description

`f = getfield(s,'field')`, where `s` is a 1-by-1 structure, returns the contents of the specified field. This is equivalent to the syntax `f = s.field`.

If `s` is a structure having dimensions greater than 1-by-1, `getfield` returns the first of all output values requested in the call. That is, for structure array `s(m,n)`, `getfield` returns `f = s(1,1).field`.

`f = getfield(s, {i,j}, 'field', {k})` returns the contents of the specified field. This is equivalent to the syntax `f = s(i,j).field(k)`. All subscripts must be passed as cell arrays — that is, they must be enclosed in curly braces (similar to`{i,j}` and `{k}` above). Pass field references as strings.

Remarks

In many cases, you can use dynamic field names in place of the `getfield` and `setfield` functions. Dynamic field names express structure fields as variable expressions that the MATLAB software evaluates at run-time. See Solution 1-19QWG for information about using dynamic field names versus the `getfield` and `setfield` functions.

Examples

Given the structure

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

Then the command `f = getfield(mystr, {2,1}, 'name')` yields

```
f =
   gertrude
```

To list the contents of all name (or other) fields, embed `getfield` in a loop.

```
for k = 1:2
    name{k} = getfield(mystr, {k,1}, 'name');
end
name

name =

    'alice'    'gertrude'
```

The following example starts out by creating a structure using the standard structure syntax. It then reads the fields of the structure, using `getfield` with variable and quoted field names and additional subscripting arguments.

```
class = 5;     student = 'John_Doe';
grades(class).John_Doe.Math(10,21:30) = ...
             [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];
```

Use `getfield` to access the structure fields.

```
getfield(grades, {class}, student, 'Math', {10,21:30})

ans =
   85   89   76   93   85   91   68   84   95   73
```

See Also

`setfield`, `fieldnames`, `isfield`, `orderfields`, `rmfield`, dynamic field names

Purpose

Capture movie frame

Syntax

```
getframe
F = getframe
F = getframe(h)
F = getframe(h,rect)
```

Description

`getframe` returns a movie frame. The frame is a snapshot (pixmap) of the current axes or figure.

`F = getframe` gets a frame from the current axes.

`F = getframe(h)` gets a frame from the figure or axes identified by handle `h`.

`F = getframe(h,rect)` specifies a rectangular area from which to copy the pixmap. `rect` is relative to the lower left corner of the figure or axes `h`, in pixel units. `rect` is a four-element vector in the form `[left bottom width height]`, where `width` and `height` define the dimensions of the rectangle.

`getframe` returns a movie frame, which is a structure having two fields:

- `cdata` — The image data stored as a matrix of `uint8` values. The dimensions of `F.cdata` are height-by-width-by-3.
- `colormap` — The colormap stored as an n-by-3 matrix of doubles. `F.colormap` is empty on true color systems.

To capture an image, use this approach:

```
F = getframe(gcf);
image(F.cdata)
colormap(F.colormap)
```

Remarks

`getframe` is usually used in a `for` loop to assemble an array of movie frames for playback using `movie`. For example,

```
for j = 1:n plotting commands
 F(j) = getframe;
```

```
end
movie(F)
```

If you are capturing frames of a plot that takes a long time to generate or are repeatedly calling `getframe` in a loop, make sure that your computer's screen saver does not activate and that your monitor does not turn off for the duration of the capture; otherwise one or more of the captured frames can contain graphics from your screen saver or nothing at all.

Note In situations where MATLAB software is running on a virtual desktop that is not currently visible on your monitor, calls to `getframe` will complete, but will capture a region on your monitor that corresponds to the position occupied by the figure or axes on the hidden desktop. Therefore, make sure that the window to be captured by `getframe` exists on the currently active desktop.

Capture Regions

Note that `F = getframe` returns the contents of the current axes, exclusive of the axis labels, title, or tick labels. `F = getframe(gcf)` captures the entire interior of the current figure window. To capture the figure window menu, use the form `F = getframe(h,rect)` with a rectangle sized to include the menu.

Resolution of Captured Frames

The resolution of the framed image depends on the size of the axes in pixels when `getframe` is called. As the `getframe` command takes a snapshot of the screen, if the axes is small in size (e.g., because you have restricted the view to a window within the axes), `getframe` will capture fewer screen pixels, and the captured image might have poor resolution if enlarged for display.

Capturing UIControls and Information Bars

If your figure contains uicontrols or displays the linking and brushing message bar along its top, `F = getframe(figure_handle)` captures them, along with the axes and any annotations displayed on the plot. `F = getframe` does not capture the message bar or uicontrols outside of the current axes. To avoid including the message bar when capturing the entire figure, click the **X** button on the message bar to dismiss it before running `getframe`. Once you do this, the message bar does not appear on subsequent figures unless you reset a preference to show it.

Examples

Make the `peaks` function vibrate.

```
Z = peaks; surf(Z)
axis tight
set(gca,'nextplot','replacechildren');
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    F(j) = getframe;
end
movie(F,20) % Play the movie twenty times
```

See Also

`frame2im`, `image`, `im2frame`, `movie`

"Bit-Mapped Images" on page 1-96 for related functions

GetFullMatrix

Purpose

Matrix from Automation server workspace

Syntax

MATLAB Client

```
[xreal ximag] = h.GetFullMatrix('varname',
'workspace', zreal,
   zimag)
[xreal ximag] = GetFullMatrix(h, 'varname', 'workspace',
   zreal, zimag)
```

IDL Method Signature

```
GetFullMatrix([in] BSTR varname,  [in] BSTR
workspace, [in, out] SAFEARRAY(double) *pr, [in,
out] SAFEARRAY(double) *pi)
```

Microsoft Visual Basic Client

```
GetFullMatrix(varname As String, workspace As String,
[out] XReal As Double, [out] XImag As Double
```

Description

`[xreal ximag] = h.GetFullMatrix('varname', 'workspace', zreal, zimag)` gets matrix stored in variable `varname` from the specified `workspace` of the server attached to handle `h`. The function returns the real part in `xreal` and the imaginary part in `ximag`. The values for *`workspace`* are `base` or `global`.

`[xreal ximag] = GetFullMatrix(h, 'varname', 'workspace', zreal, zimag)` is an alternate syntax.

The `zreal` and `zimag` arguments are matrices of the same size as the real and imaginary matrices (`xreal` and `ximag`) returned from the server. The `zreal` and `zimag` matrices are commonly set to zero.

Use `GetFullMatrix` for values of type `double` only. Use `GetVariable` or `GetWorkspaceData` for other types.

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 the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`. VBScript does not support `safearray`.

Examples

Use a MATLAB client to read data from a MATLAB Automation server:

```
%Create the MATLAB server
h = actxserver('matlab.application');
%Create variable M in the base workspace of the server
h.PutFullMatrix('M','base',rand(5),zeros(5));
MReal = h.GetFullMatrix('M','base',zeros(5),zeros(5))
```

Use a Visual Basic client to read data from a 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 Result As String
   Dim XReal(4, 4) As Double
   Dim XImag(4, 4) As Double
   Dim i, j As Integer

   MatLab = CreateObject("matlab.application")
   Result = MatLab.Execute("M = rand(5);")
   MsgBox("In MATLAB, type" & vbCrLf _
       & "M(3,4)")
   ```

2. Open the MATLAB window and type:

   ```
   M(3,4)
   ```

3. Click **Ok**.

4. At the MATLAB command line, type:

   ```
   MatLab.GetFullMatrix("M", "base", XReal, XImag)
   ```

```
i = 2   %0-based array
j = 3

MsgBox("XReal(" & i + 1 & "," & j + 1 & ")" & _
    " = " & XReal(i, j))
```

5. Click **Ok** to close and terminate MATLAB.

See Also

`PutFullMatrix` | `GetVariable` | `GetWorkspaceData` | `Execute`

How To

-
-

Purpose

Interpolation method for `timeseries` object

Syntax

```
getinterpmethod(ts)
```

Description

`getinterpmethod(ts)` returns the interpolation method as a string that is used by the `timeseries` object `ts`. Predefined interpolation methods are `'zoh'` (zero-order hold) and `'linear'` (linear interpolation). The method strings are case sensitive.

Examples

1. Create a `timeseries` object.

```
ts = timeseries(rand(5));
```

2. Get the interpolation method for this object.

```
getinterpmethod(ts)

ans =

     linear
```

See Also

`setinterpmethod`, `timeseries`, `tsprops`

getpixelposition

Purpose

Get component position in pixels

Syntax

```
position = getpixelposition(handle)
position = getpixelposition(handle,recursive)
```

Description

`position = getpixelposition(handle)` gets the position, in pixel units, of the component with handle `handle`. The `position` is returned as a four-element vector that specifies the location and size of the component: [distance from left, distance from bottom, width, height].

`position = getpixelposition(handle,recursive)` gets the position as above. If `recursive` is true, the returned position is relative to the parent figure of `handle`.

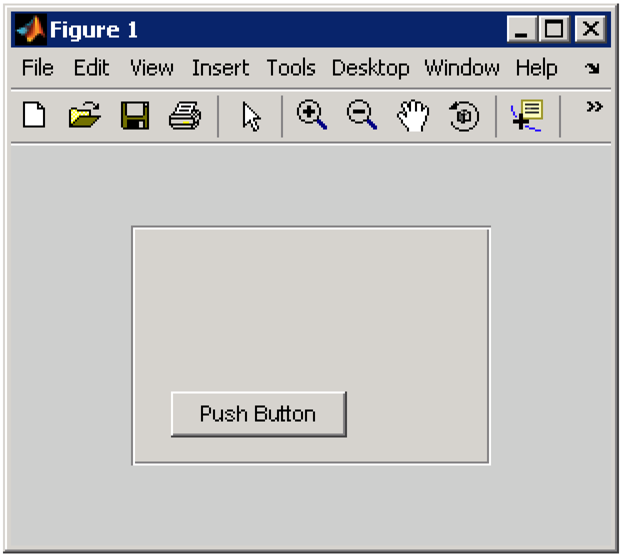
Use the `getpixelposition` function only to obtain coordinates for children of figures and container components (uipanels, or uibuttongroups). Results are not reliable for children of axes or other Handle Graphics objects.

Example

This example creates a push button within a panel, and then retrieves its position, in pixels, relative to the panel.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6]);
h1 = uicontrol(p,'Style','PushButton','Units','Normalized',...
               'String','Push Button','Position',[.1 .1 .5 .2]);
pos1 = getpixelposition(h1)

pos1 =
     18.6000   12.6000   88.0000   23.2000
```



The following statement retrieves the position of the push button, in pixels, relative to the figure.

```
pos1 = getpixelposition(h1,true)

pos1 =
      79.6000   53.6000   88.0000   23.2000
```

See Also

`setpixelposition`, `uicontrol`, `uipanel`

getpref

Purpose

Preference

Syntax

```
getpref('group','pref')
getpref('group','pref',default)
getpref('group',{'pref1','pref2',...'prefn'})
getpref('group',{'pref1',...'prefn'},{default1,...defaultn})
getpref('group')
getpref
```

Description

`getpref('group','pref')` returns the value for the preference specified by `group` and `pref`. It is an error to get a preference that does not exist.

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

`getpref('group','pref',default)` returns the current value if the preference specified by `group` and `pref` exists. Otherwise creates the preference with the specified default value and returns that value.

`getpref('group',{'pref1','pref2',...'prefn'})` returns a cell array containing the values for the preferences specified by `group` and the cell array of preference names. The return value is the same size as the input cell array. It is an error if any of the preferences do not exist.

`getpref('group',{'pref1',...'prefn'},{default1,...defaultn})` returns a cell array with the current values of the preferences specified by `group` and the cell array of preference names. Any preference that does not exist is created with the specified default value and returned.

`getpref('group')` returns the names and values of all preferences in the `group` as a structure.

`getpref` returns all groups and preferences as a structure.

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

Examples

Example 1

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

ans =
     1.0
```

Example 2

```
rmpref('mytoolbox','version')
getpref('mytoolbox','version','1.0');
getpref('mytoolbox','version')

ans =
     1.0
```

See Also

`addpref`, `ispref`, `rmpref`, `setpref`, `uigetpref`, `uisetpref`

getqualitydesc

Purpose Data quality descriptions

Syntax `getqualitydesc(ts)`

Description `getqualitydesc(ts)` returns a cell array of data quality descriptions based on the `Quality` values you assigned to a `timeseries` object `ts`.

Examples

1. Create a `timeseries` object with `Data`, `Time`, and `Quality` values, respectively.

   ```
   ts = timeseries([3; 4.2; 5; 6.1; 8], 1:5, [1; 0; 1; 0; 1]);
   ```

2. Set the `QualityInfo` property, consisting of `Code` and `Description`.

   ```
   ts.QualityInfo.Code = [0 1];
   ts.QualityInfo.Description = {'good' 'bad'};
   ```

3. Get the data quality description strings for `ts`.

   ```
   getqualitydesc(ts)

   ans =

       'bad'
       'good'
       'bad'
       'good'
       'bad'
   ```

See Also `tsprops`

Purpose

Get error message for exception

Syntax

```
msgString = getReport(errRecord)
msgString = getReport(errRecord, type)
msgString = getReport(errRecord, type, 'hyperlinks', value)
```

Description

`msgString = getReport(errRecord)` returns a formatted message string, `msgString`, that is based on the current error (or *exception*). This exception is represented by an object `errRecord` of the `MException` class. The message string returned by `getReport` is the same as the error message displayed by MATLAB when it throws this same exception.

`msgString = getReport(errRecord, type)` returns a message string that either describes just the highest level error (**`basic`** type), or shows the error and the stack as well (**`extended`** type). The `type` argument, when used, must be the second argument in the input argument list. See "Examples" on page 2-1556 , below.

`type` Option	**Displayed Text**
`'extended'`	Display line number, error message, and cause and stack summary (default)
`'basic'`	Display line number and error message

`msgString = getReport(errRecord, type, 'hyperlinks', value)` returns a message string that either does or does not include active hyperlinks to the failing lines in the code. See the table below for the valid choices for `value`. The **`hyperlinks`** and `value` arguments, when used, must be the third and fourth arguments in the input argument list.

`value` Option	**Action**
`'on'`	Display hyperlinks to failing lines (default)

`value` Option	Action
`'off'`	Do not display hyperlinks to failing lines
`'default'`	Use the default for the Command Window to determine whether or not to use hyperlinks in the error message

Examples

Try calling the MATLAB `surf` function without the required input argument. In the `catch` statement, capture the error in an MException object, `errRecord`. Then, use this object with `getReport` to retrieve a basic error string:

```
try
   surf
catch errRecord
   rep = getReport(errRecord, 'basic')
end

rep =
Error using ==> surf at 50
Not enough input arguments.
```

Run the try-catch again, this time replacing `'basic'` with `'extended'`: In this case, the error message includes information from the stack:

```
   rep = getReport(errRecord, 'extended')

rep =
Error using ==> surf at 50
Not enough input arguments.

Error in ==> getRep>getRep3 at 9
   surf

Error in ==> getRep>getRep2 at 5
getRep3(option, state)
```

```
Error in ==> getRep at 2
getRep2(option, state)
```

See Also `try`, `catch`, `error`, `assert`, `MException`, `throw(MException)`, `rethrow(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `last(MException)`,

getsampleusingtime (timeseries)

Purpose Extract data samples into new `timeseries` object

Syntax

```
ts2 = getsampleusingtime(ts1,Time)
ts2 = getsampleusingtime(ts1,StartTime,EndTime)
```

Description `ts2 = getsampleusingtime(ts1,Time)` returns a new `timeseries` object `ts2` with a single sample corresponding to the time Time in ts1.

`ts2 = getsampleusingtime(ts1,StartTime,EndTime)` returns a new `timeseries` object `ts2` with samples between the times `StartTime` and `EndTime` in ts1.

Remarks When the time vector in ts1 is numeric, `StartTime` and `EndTime` must also be numeric. When the times in ts1 are date strings and the `StartTime` and `EndTime` values are numeric, then the `StartTime` and `EndTime` values are treated as `datenum` values.

See Also `timeseries`

Purpose

Extract data samples into new `tscollection` object

Syntax

```
tsc2 = getsampleusingtime(tsc1,Time)
tsc2 = getsampleusingtime(tsc1,StartTime,EndTime)
```

Description

`tsc2 = getsampleusingtime(tsc1,Time)` returns a new `tscollection` tsc2 with a single sample corresponding to Time in tsc1.

`tsc2 = getsampleusingtime(tsc1,StartTime,EndTime)` returns a new `tscollection` tsc2 with samples between the times StartTime and EndTime in tsc1.

Remarks

When the time vector in ts1 is numeric, `StartTime` and `EndTime` must also be numeric. When the times in ts1 are date strings and the `StartTime` and `EndTime` values are numeric, then the `StartTime` and `EndTime` values are treated as `datenum` values.

See Also

`tscollection`

Tiff.getTag

Purpose

Value of specified tag

Syntax

`tagValue = getTag(tagId)`

Description

`tagValue = getTag(tagId)` retrieves the value of the TIFF tag specified by `tagId`. 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.

Examples

Open a `Tiff` object, and get the value of a tag. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path:

```
t = Tiff('myfile.tif', 'r');
% Specify tag by name.
tagval = t.getTag('ImageWidth');
%
% Specify tag by numeric identifier.
tagval1 = t.getTag(256);
%
% Specify tag by name.
tagval2 = t.getTag('t.TagID.ImageWidth');
```

References

This method corresponds to the `TIFFGetField` 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.setTag`

Tutorials

•

•

Purpose

List of recognized TIFF tags

Syntax

`tagNames = Tiff.getTagNames()`

Description

`tagNames = Tiff.getTagNames()` returns a cell array of TIFF tags recognized by the `Tiff` object.

Examples

Retrieve a list of TIFF tags recognized by the `Tiff` object.

```
Tiff.getTagNames

ans =

    'SubFileType'
    'ImageWidth'
    'ImageLength'
    'BitsPerSample'
    'Compression'
    'Photometric'
    'Thresholding'
    'FillOrder'
    'DocumentName'
    'ImageDescription'
     .
     .
     .
```

See Also

`Tiff.getTag`

Tutorials

-
-

gettimeseriesnames

Purpose

Cell array of names of `timeseries` objects in `tscollection` object

Syntax

```
names = gettimeseriesnames(tsc)
```

Description

`names = gettimeseriesnames(tsc)` returns `names` of `timeseries` objects in a `tscollection` object `tsc`. `names` is a cell array of strings.

Examples

1. Create `timeseries` objects `a` and `b`.

```
a = timeseries(rand(1000,1),'name','position');
b = timeseries(rand(1000,1),'name','response');
```

2. Create a `tscollection` object that includes these two time series.

```
tsc = tscollection({a,b});
```

3. Get the names of the `timeseries` objects in `tsc`.

```
names = gettimeseriesnames(tsc)

names = 

    'position'    'response'
```

See Also

`timeseries`, `tscollection`, `tsprops`

Purpose

New `timeseries` object with samples occurring at or after event

Syntax

```
ts1 = gettsafteratevent(ts,event)
ts1 = gettsafteratevent(ts,event,n)
```

Description

`ts1 = gettsafteratevent(ts,event)` returns a new `timeseries` object ts1 with samples occurring at and after an event in ts, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of the time series ts that matches the event name specifies the time.

`ts1 = gettsafteratevent(ts,event,n)` returns a new `timeseries` object ts1 with samples at and after an event in ts, where n is the number of the event occurrence with a matching event name.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

gettsafterevent

Purpose

New `timeseries` object with samples occurring after event

Syntax

```
ts1 = gettsafterevent(ts,event)
ts1 = ttsafterevent(ts,event,n)
```

Description

`ts1 = gettsafterevent(ts,event)` returns a new `timeseries` object ts1 with samples occurring after an event in ts, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = ttsafterevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring after an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafteratevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

Purpose

New `timeseries` object with samples occurring at event

Syntax

```
ts1 = gettsatevent(ts,event)
ts1 = gettsatevent(ts,event,n)
```

Description

`ts1 = gettsatevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring at an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsatevent(ts,event,n)` returns a new time series `ts1` with samples occurring at an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in the `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent`, `gettsafteratevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

gettsbeforeatevent

Purpose

New `timeseries` object with samples occurring before or at event

Syntax

```
ts1 = gettsbeforeatevent(ts,event)
ts1 = gettsbeforeatevent(ts,event,n)
```

Description

`ts1 = gettsbeforeatevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring at and before an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsbeforeatevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring at and before an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

Purpose New `timeseries` object with samples occurring before event

Syntax

```
ts1 = gettsbeforeevent(ts,event)
ts1 = gettsbeforeevent(ts,event,n)
```

Description

`ts1 = gettsbeforeevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring before an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsbeforeevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring before an event in `ts`, where n is the number of the event occurrence with a matching event name.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent`, `gettsbeforeatevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

gettsbetweenevents

Purpose

New `timeseries` object with samples occurring between events

Syntax

```
ts1 = gettsbetweenevents(ts,event1,event2)
ts1 = gettsbetweenevents(ts,event1,event2,n1,n2)
```

Description

`ts1 = gettsbetweenevents(ts,event1,event2)` returns a new `timeseries` object ts1 with samples occurring between events in `ts`, where `event1` and `event2` can be either a `tsdata.event` object or a string. When `event1` and `event2` are `tsdata.event` objects, the time defined by the events is used. When `event1` and `event2` are strings, the first `tsdata.event` object in the `Events` property of `ts` that matches the event names specifies the time.

`ts1 = gettsbetweenevents(ts,event1,event2,n1,n2)` returns a new `timeseries` object ts1 with samples occurring between events in `ts`, where `n1` and `n2` are the `n`th occurrences of the events with matching event names.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent`, `gettsbeforeevent`, `tsdata.event`, `tsprops`

Purpose Data from variable in Automation server workspace

Syntax

MATLAB Client

```
D = h.GetVariable('varname', 'workspace')
D = GetVariable(h, 'varname', 'workspace')
```

IDL Method Signature

```
HRESULT GetVariable([in] BSTR varname, [in] BSTR
workspace, [out, retval] VARIANT* pdata)
```

Microsoft Visual Basic Client

```
GetVariable(varname As String, workspace As
String) As Object
```

Description `D = h.GetVariable('varname', 'workspace')` gets data stored in variable `varname` from the specified `workspace` of the server attached to handle `h` and returns it in output argument `D`. The values for *`workspace`* are `base` or `global`.

`D = GetVariable(h, 'varname', 'workspace')` is an alternate syntax.

Do *not* use `GetVariable` on sparse arrays, structures, or function handles.

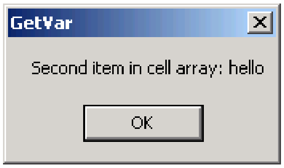
If your scripting language requires a result be returned explicitly, use the `GetVariable` function in place of `GetWorkspaceData`, `GetFullMatrix` or `GetCharArray`.

Examples Use a MATLAB client to read data from a MATLAB Automation server:

```
%Create the MATLAB server
h = actxserver('matlab.application');
%Create variable C1 in the base workspace of the server
h.PutWorkspaceData('C1', 'base', {25.72, 'hello', rand(4)});
%The client reads the data
C2 = h.GetVariable('C1','base')
```

Use a Visual Basic client to read data from a MATLAB Automation server:

```
Dim Matlab As Object
Dim Result As String
Dim C2 As Object
Matlab = CreateObject("matlab.application")
Result = Matlab.Execute("C1 = {25.72, 'hello', rand(4)};")
C2 = Matlab.GetVariable("C1", "base")
MsgBox("Second item in cell array: " & C2(0, 1))
```



Second item in cell array: hello

See Also

`GetWorkspaceData` | `GetFullMatrix` | `GetCharArray` | `Execute`

How To

-
-

Purpose

LibTIFF library version

Syntax

`versionString = Tiff.getVersion()`

Description

`versionString = Tiff.getVersion()` returns the version number and other information about the LibTIFF library.

Examples

Display version of LibTIFF library:

```
Tiff.getVersion

ans =

LIBTIFF, Version 3.7.1
Copyright (c) 1988-1996 Sam Leffler
Copyright (c) 1991-1996 Silicon Graphics, Inc.
```

References

This method corresponds to the `TIFFGetVersion` 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`.

GetWorkspaceData

Purpose Data from Automation server workspace

Syntax

MATLAB Client

D = h.GetWorkspaceData('varname', '*workspace*')
D = GetWorkspaceData(h, 'varname', '*workspace*')

IDL Method Signature

```
HRESULT GetWorkspaceData([in] BSTR varname, [in] BSTR
workspace, [out] VARIANT* pdata)
```

Microsoft Visual Basic Client

```
GetWorkspaceData(varname As String, workspace
As String) As Object
```

Description

`D = h.GetWorkspaceData('varname', 'workspace')` gets data stored in variable `varname` from the specified `workspace` of the server attached to handle `h` and returns it in output argument `D`. The values for *`workspace`* are `base` or `global`.

`D = GetWorkspaceData(h, 'varname', 'workspace')` is an alternate syntax.

Use `GetWorkspaceData` instead of `GetFullMatrix` and `GetCharArray` to get numeric and character array data, respectively. Do *not* use `GetWorkspaceData` on sparse arrays, structures, or function handles.

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 the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`. VBScript does not support `safearray`.

Examples

Use a MATLAB client to read data from a MATLAB Automation server:

```
%Create the MATLAB server
h = actxserver('matlab.application');
%Create cell array C1 in the base workspace of the server
h.PutWorkspaceData('C1', 'base', ...
```

```
    {25.72, 'hello', rand(4)});
C2 = h.GetWorkspaceData('C1', 'base')
```

Use a Visual Basic client to read data from a MATLAB Automation server:

```
Dim Matlab, C2 As Object
Dim Result As String
Matlab = CreateObject("matlab.application")
Result = MatLab.Execute("C1 = {25.72, 'hello', rand(4)};")
MsgBox("In MATLAB, type" & vbCrLf & "C1")
Matlab.GetWorkspaceData("C1", "base", C2)
MsgBox("second value of C1 = " & C2(0, 1))
```

See Also

`PutWorkspaceData` | `GetFullMatrix` | `GetCharArray` | `GetVariable` | `Execute`

How To

-
-

Purpose

Graphical input from mouse or cursor

Syntax

```
[x,y] = ginput(n)
[x,y] = ginput
[x,y,button] = ginput(...)
```

Description

`ginput` raises crosshairs in the current axes to for you to identify points in the figure, positioning the cursor with the mouse. The figure must have focus before `ginput` can receive input. If it has no axes, one is created upon the first click or keypress.

`[x,y] = ginput(n)` enables you to identify `n` points from the current axes and returns their *x*- and *y*-coordinates in the `x` and `y` column vectors. Press the **Return** key to terminate the input before entering `n` points.

`[x,y] = ginput` gathers an unlimited number of points until you press the **Return** key.

`[x,y,button] = ginput(...)` returns the *x*-coordinates, the *y*-coordinates, and the button or key designation. `button` is a vector of integers indicating which mouse buttons you pressed (1 for left, 2 for middle, 3 for right), or ASCII numbers indicating which keys on the keyboard you pressed.

Clicking an axes makes that axes the current axes. Even if you set the current axes before calling `ginput`, whichever axes you click becomes the current axes and `ginput` returns points relative to that axes. If you select points from multiple axes, the results returned are relative to the coordinate system of the axes they come from.

Definitions

Coordinates returned by `ginput` are scaled to the `XLim` and `YLim` bounds of the axes you click (data units). Setting the axes or figure `Units` property has no effect on the output from `ginput`. You can click anywhere within the figure canvas to obtain coordinates. If you click outside the axes limits, `ginput` extrapolates coordinate values so they are still relative to the axes origin.

The figure `CurrentPoint` property, by contrast, is always returned in figure Units, irrespective of axes Units or limits.

Examples

Pick 4 two-dimensional points from the figure window.

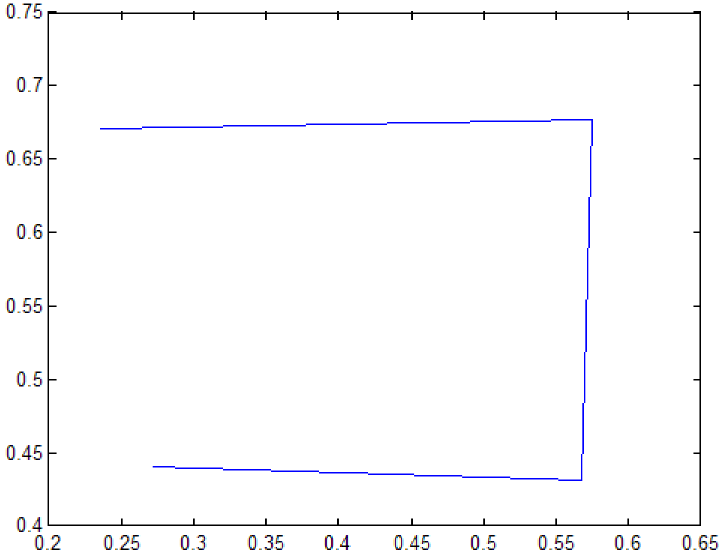
```
[x,y] = ginput(4)
```

Position the cursor with the mouse. Enter data points by pressing a mouse button or a key on the keyboard. To terminate input before entering 4 points, press the **Return** key.

```
x =
    0.2362
    0.5749
    0.5680
    0.2707

y =
    0.6711
    0.6769
    0.4313
    0.4401

plot(x,y)
```



In this example, `plot` rescaled the axes *x*-limits and *y*-limits from `[0 1]` and `[0 1]` to `[0.20 0.65]` and `[0.40 0.75]`. The rescaling occured because the axes `XLimMode` and `YLimMode` are set to `'auto'` (the default). Consider setting `XLimMode` and `YLimMode` to `'manual'` if you want to maintain consistency when you gather results from `ginput` and plot them together.

See Also

`gtext`

Tutorials

-
-

Purpose

Declare global variables

Syntax

```
global X Y Z
```

Description

`global X Y Z` defines `X`, `Y`, and `Z` as global in scope.

Ordinarily, each MATLAB function, defined by an M-file, has its own local variables, which are separate from those of other functions, and from those of the base workspace. However, if several functions, and possibly the base workspace, all declare a particular name as global, they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the functions declaring it global.

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

If a variable with the same name as the global variable already exists in the current workspace, MATLAB issues a warning and changes the value of that variable to match the global.

Remarks

Use `clear global` *`variable`* to clear a global variable from the global workspace. Use `clear` *`variable`* to clear the global link from the current workspace without affecting the value of the global.

To use a global within a callback, declare the global, use it, then clear the global link from the workspace. This avoids declaring the global after it has been referenced. For example,

```
cbstr = sprintf('%s, %s, %s, %s, %s', ...
   'global MY_GLOBAL',               ...
   'MY_GLOBAL = 100',                ...
   'disp(MY_GLOBAL)',                ...
   'MY_GLOBAL = MY_GLOBAL+1',        ...
   'clear MY_GLOBAL');

uicontrol('style', 'pushbutton', 'CallBack', cbstr, ...
   'string', 'count')
```

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

Examples

Here is the code for the functions `tic` and `toc` (some comments abridged). These functions manipulate a stopwatch-like timer. The global variable `TICTOC` is shared by the two functions, but it is invisible in the base workspace or in any other functions that do not declare it.

```
function tic
%    TIC Start a stopwatch timer.
%        TIC; any stuff; TOC
%    prints the time required.
%    See also: TOC, CLOCK.
global TICTOC
TICTOC = clock;

function t = toc
%    TOC Read the stopwatch timer.
%    TOC prints the elapsed time since TIC was used.
%    t = TOC; saves elapsed time in t, does not print.
%    See also: TIC, ETIME.
global TICTOC
if nargout < 1
    elapsed_time = etime(clock, TICTOC)
else
    t = etime(clock, TICTOC);
end
```

See Also

`clear`, `isglobal`, `who`

Purpose

Generalized minimum residual method (with restarts)

Syntax

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

Description

`x = gmres(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x)` returns `A*x`. See in the MATLAB Programming documentation for more information. For this syntax, `gmres` does not restart; the maximum number of iterations is `min(n,10)`.

, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

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

`gmres(A,b,restart)` restarts the method every `restart` inner iterations. The maximum number of outer iterations is `min(n/restart,10)`. The maximum number of total iterations is `restart*min(n/restart,10)`. If `restart` is `n` or `[]`, then `gmres` does not restart and the maximum number of total iterations is `min(n,10)`.

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

`gmres(A,b,restart,tol,maxit)` specifies the maximum number of outer iterations, i.e., the total number of iterations does not exceed `restart*maxit`. If `maxit` is `[]` then `gmres` uses the default, `min(n/restart,10)`. If `restart` is `n` or `[]`, then the maximum number of total iterations is `maxit` (instead of `restart*maxit`).

`gmres(A,b,restart,tol,maxit,M)` and `gmres(A,b,restart,tol,maxit,M1,M2)` use preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `gmres` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

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

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

`flag = 0`	`gmres` converged to the desired tolerance `tol` within `maxit` outer iterations.
`flag = 1`	`gmres` iterated `maxit` times but did not converge.
`flag = 2`	Preconditioner `M` was ill-conditioned.
`flag = 3`	`gmres` stagnated. (Two consecutive iterates were the same.)

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = gmres(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = gmres(A,b,...)` also returns both the outer and inner iteration numbers at which `x` was computed, where `0 <= iter(1) <= maxit` and `0 <= iter(2) <= restart`.

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

Examples

Example 1

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = gmres(A,b,10,tol,maxit,M1);
```

displays the following message:

```
gmres(10) converged at outer iteration 2 (inner iteration 9) to
a solution with relative residual 3.3e-013
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in an M-file `run_gmres` that

- Calls `gmres` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in `run_gmres` are available to `afun` and `mfun`.

The following shows the code for `run_gmres`:

```
function x1 = run_gmres
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
tol = 1e-12;  maxit = 15;
x1 = gmres(@afun,b,10,tol,maxit,@mfun);
```

```
    function y = afun(x)
        y = [0; x(1:n-1)] + ...
              [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
              [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_gmres;
```

MATLAB software displays the message

```
gmres(10) converged at outer iteration 2 (inner iteration 9) to
a solution with relative residual 3.3e-013
```

Example 3

```
load west0479
A = west0479
b = sum(A,2)
[x,flag] = gmres(A,b,5)
```

`flag` is 1 because `gmres` does not converge to the default tolerance `1e-6` within the default 10 outer iterations.

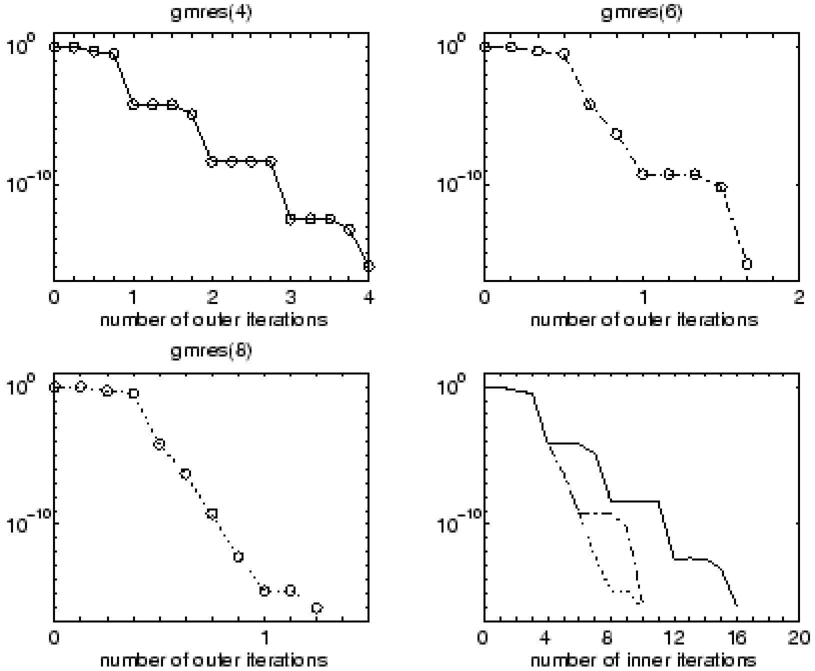
```
[L1,U1] = luinc(A,1e-5);
[x1,flag1] = gmres(A,b,5,1e-6,5,L1,U1);
```

`flag1` is `2` because the upper triangular `U1` has a zero on its diagonal, and `gmres` fails in the first iteration when it tries to solve a system such as `U1*y = r` for `y` using backslash.

```
[L2,U2] = luinc(A,1e-6);
```

```
tol = 1e-15;
[x4,flag4,relres4,iter4,resvec4] = gmres(A,b,4,tol,5,L2,U2);
[x6,flag6,relres6,iter6,resvec6] = gmres(A,b,6,tol,3,L2,U2);
[x8,flag8,relres8,iter8,resvec8] = gmres(A,b,8,tol,3,L2,U2);
```

`flag4`, `flag6`, and `flag8` are all `0` because `gmres` converged when restarted at iterations 4, 6, and 8 while preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. This is verified by the plots of outer iteration number against relative residual. A combined plot of all three clearly shows the restarting at iterations 4 and 6. The total number of iterations computed may be more for lower values of `restart`, but the number of length `n` vectors stored is fewer, and the amount of work done in the method decreases proportionally.



See Also

`bicg`, `bicgstab`, `cgs`, `lsqr`, `ilu`, `luinc`, `minres`, `pcg`, `qmr`, `symmlq` `function_handle` (@), `mldivide` (\)

References

Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

Saad, Youcef and Martin H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, July 1986, Vol. 7, No. 3, pp. 856-869.

Purpose

Plot nodes and links representing adjacency matrix

Syntax

`gplot(A,Coordinates)`
`gplot(A,Coordinates,`*`LineSpec`*`)`

Description

The `gplot` function graphs a set of coordinates using an adjacency matrix.

`gplot(A,Coordinates)` plots a graph of the nodes defined in `Coordinates` according to the *n*-by-*n* adjacency matrix `A`, where *n* is the number of nodes. `Coordinates` is an *n*-by-2 matrix, where *n* is the number of nodes and each coordinate pair represents one node.

`gplot(A,Coordinates,`*`LineSpec`*`)` plots the nodes using the line type, marker symbol, and color specified by *`LineSpec`*.

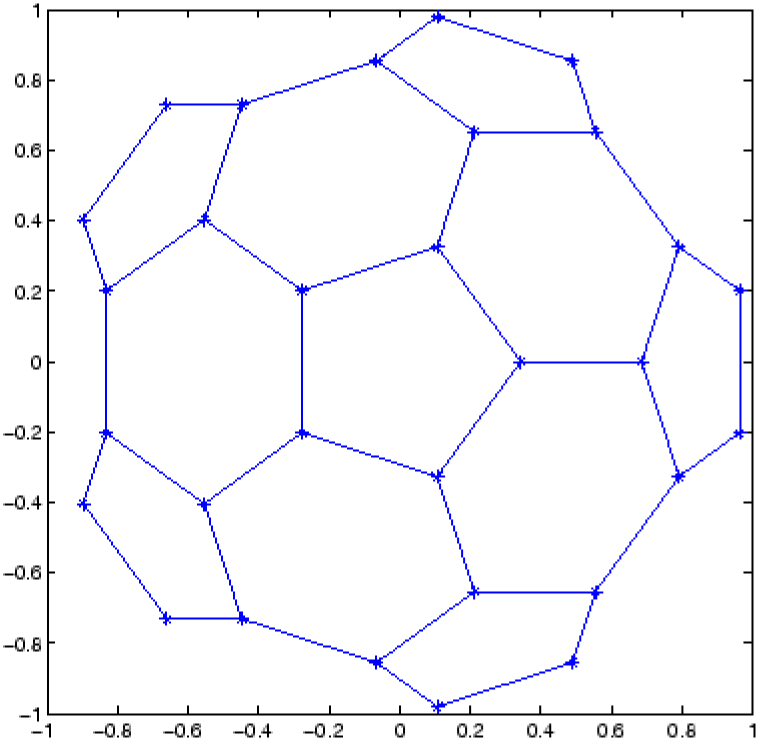
Remarks

For two-dimensional data, `Coordinates(i,:)  = [x(i) y(i)]` denotes node `i`, and `Coordinates(j,:)  = [x(j)y(j)]` denotes node `j`. If node `i` and node `j` are connected, `A(i,j)` or `A(j,i)` is nonzero; otherwise, `A(i,j)` and `A(j,i)` are zero.

Examples

To draw half of a Bucky ball with asterisks at each node,

```
k = 1:30;
[B,XY] = bucky;
gplot(B(k,k),XY(k,:),'-*')
axis square
```



See Also

`LineSpec`, `sparse`, `spy`

"Tree Operations" on page 1-51 for related functions

Purpose MATLAB code from M-files published to HTML

Syntax

```
grabcode('name.html')
grabcode('urlname')
codeString = grabcode('name.html')
```

Description

`grabcode('name.html')` copies MATLAB code from the file `name.html` and pastes it into an untitled document in the Editor. Use `grabcode` to get MATLAB code from demos or other published M-files when the M-file source code is not readily available. The file `name.html` was created by publishing `name.m`, an M-file containing cells. The MATLAB code from `name.m` is included at the end of `name.html` as HTML comments.

`grabcode('urlname')` copies MATLAB code from the `urlname` location and pastes it into an untitled document in the Editor.

`codeString = grabcode('name.html')` get MATLAB code from the file `name.html` and assigns it the variable `codeString`.

Examples

Run

```
sineWaveString = grabcode('d:/mymfiles/sine_wave_.html')
```

and MATLAB displays

```
sineWaveString =

%% Simple Sine Wave Plot

%% Part One: Calculate Sine Wave
% Define the range |x|.
% Calculate the sine |y| over that range.
x = 0:.01:6*pi;
y = sin(x);

%% Part Two: Plot Sine Wave
% Graph the result.
```

```
plot(x,y)
```

See Also `demo`, `publish`

Purpose

Numerical gradient

Syntax

```
FX = gradient(F)
[FX,FY] = gradient(F)
[FX,FY,FZ,...] = gradient(F)
[...] = gradient(F,h)
[...] = gradient(F,h1,h2,...)
```

Definition

The *gradient* of a function of two variables, $F(x, y)$, is defined as

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j}$$

and can be thought of as a collection of vectors pointing in the direction of increasing values of F. In MATLAB software, numerical gradients (differences) can be computed for functions with any number of variables. For a function of N variables, $F(x, y, z, \ldots)$,

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j} + \frac{\partial F}{\partial z}\hat{k} + \ldots$$

Description

`FX = gradient(F)` where `F` is a vector returns the one-dimensional numerical gradient of `F`. `FX` corresponds to $\partial F/\partial x$, the differences in x (horizontal) direction.

`[FX,FY] = gradient(F)` where `F` is a matrix returns the x and y components of the two-dimensional numerical gradient. `FX` corresponds to $\partial F/\partial x$, the differences in x (horizontal) direction. `FY` corresponds to $\partial F/\partial y$, the differences in the y (vertical) direction. The spacing between points in each direction is assumed to be one.

`[FX,FY,FZ,...]  = gradient(F)` where `F` has `N dimensions` returns the `N` components of the gradient of `F`. There are two ways to control the spacing between values in `F`:

- A single spacing value, `h`, specifies the spacing between points in every direction.

- `N` spacing values (`h1,h2,...`) specifies the spacing for each dimension of `F`. Scalar spacing parameters specify a constant spacing for each dimension. Vector parameters specify the coordinates of the values along corresponding dimensions of `F`. In this case, the length of the vector must match the size of the corresponding dimension.

Note The first output `FX` is always the gradient along the 2nd dimension of `F`, going across columns. The second output `FY` is always the gradient along the 1st dimension of `F`, going across rows. For the third output `FZ` and the outputs that follow, the Nth output is the gradient along the Nth dimension of `F`.

`[...] = gradient(F,h)` where `h` is a scalar uses `h` as the spacing between points in each direction.

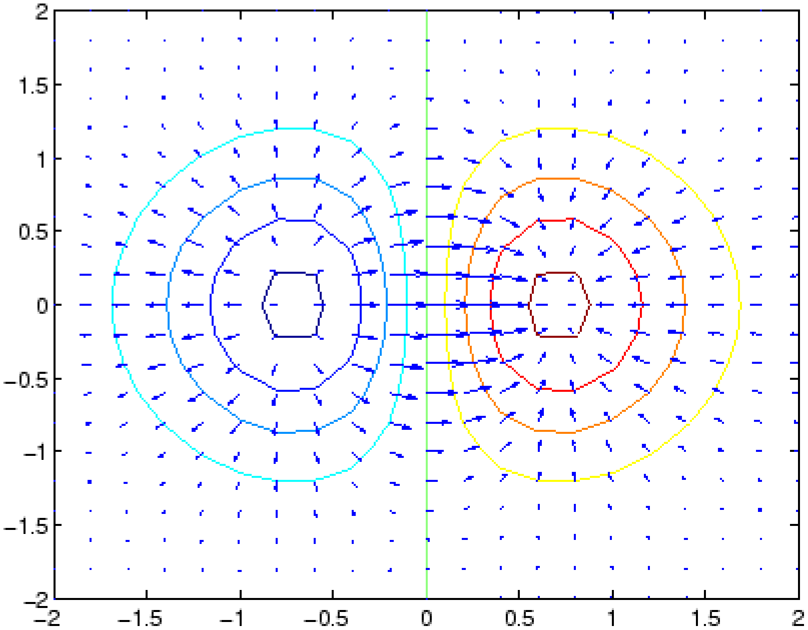
`[...] = gradient(F,h1,h2,...)` with `N` spacing parameters specifies the spacing for each dimension of `F`.

Examples

The statements

```
v = -2:0.2:2;
[x,y] = meshgrid(v);
z = x .* exp(-x.^2 - y.^2);
[px,py] = gradient(z,.2,.2);
contour(v,v,z), hold on, quiver(v,v,px,py), hold off
```

produce



Given,

```
F(:,:,1) = magic(3); F(:,:,2) = pascal(3);
gradient(F)
```

takes `dx` = `dy` = `dz` = `1`.

```
[PX,PY,PZ] = gradient(F,0.2,0.1,0.2)
```

takes `dx = 0.2`, `dy = 0.1`, and `dz = 0.2`.

See Also `del2`, `diff`

graymon

Purpose Set default figure properties for grayscale monitors

Syntax `graymon`

Description `graymon` sets defaults for graphics properties to produce more legible displays for grayscale monitors.

See Also `axes`, `figure`

"Color Operations" on page 1-103 for related functions

Purpose

Grid lines for 2-D and 3-D plots

GUI Alternative

To control the presence and appearance of grid lines on a graph, use the Property Editor, one of the plotting tools. For details, see The Property Editor in the MATLAB Graphics documentation.

Syntax

```
grid on
grid off
grid
grid(axes_handle,...)
grid minor
```

Description

The `grid` function turns the current axes' grid lines on and off.

`grid on` adds major grid lines to the current axes.

`grid off` removes major and minor grid lines from the current axes.

`grid` toggles the major grid visibility state.

`grid(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

Algorithm

`grid` sets the `XGrid`, `YGrid`, and `ZGrid` properties of the axes.

`grid minor` sets the `XMinorGrid`, `YMinorGrid`, and `ZMinorGrid` properties of the axes.

You can set the grid lines for just one axis using the `set` command and the individual property. For example,

```
set(axes_handle,'XGrid','on')
```

turns on only *x*-axis grid lines.

You can set grid line width with the axes `LineWidth` property.

See Also

`box`, `axes`, `set`

The properties of axes objects

"Axes Operations" on page 1-101 for related functions

Purpose

Data gridding

`griddata` is not recommended. Use `TriScatteredInterp` instead

Syntax

```
ZI = griddata(x,y,z,XI,YI)
[XI,YI,ZI] = griddata(x,y,z,XI,YI)
[...] = griddata(...,method)
```

Description

`ZI = griddata(x,y,z,XI,YI)` fits a surface of the form `z = f(x,y)` to the data in the (usually) nonuniformly spaced vectors `(x,y,z)`. `griddata` interpolates this surface at the points specified by `(XI,YI)` to produce `ZI`. The surface always passes through the data points. `XI` and `YI` usually form a uniform grid (as produced by `meshgrid`).

`XI` can be a row vector, in which case it specifies a matrix with constant columns. Similarly, `YI` can be a column vector, and it specifies a matrix with constant rows.

`[XI,YI,ZI] = griddata(x,y,z,XI,YI)` returns the interpolated matrix `ZI` as above, and also returns the matrices `XI` and `YI` formed from row vector `XI` and column vector `yi`. These latter are the same as the matrices returned by `meshgrid`.

`[...]  = griddata(...,method)` uses the specified interpolation method:

’`linear`’	Triangle-based linear interpolation (default)
’`cubic`’	Triangle-based cubic interpolation
’`nearest`’	Nearest neighbor interpolation
’`v4`’	MATLAB 4 `griddata` method

The `method` defines the type of surface fit to the data. The `'cubic'` and `'v4'` methods produce smooth surfaces while `'linear'` and `'nearest'` have discontinuities in the first and zero’th derivatives, respectively. All the methods except `'v4'` are based on a Delaunay triangulation of the data. If `method` is `[]`, then the default `'linear'` method is used.

`[...] = griddata(...,options)` specifies a cell array of strings options 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.

Occasionally, `griddata` might return points on or very near the convex hull of the data as `NaN`s. This is because roundoff in the computations sometimes makes it difficult to determine if a point near the boundary is in the convex hull.

`griddata` uses CGAL, see http://www.cgal.org.

Remarks

`XI` and `YI` can be matrices, in which case `griddata` returns the values for the corresponding points `(XI(i,j),YI(i,j))`. Alternatively, you can pass in the row and column vectors `xi` and `yi`, respectively. In this case, `griddata` interprets these vectors as if they were matrices produced by the command `meshgrid(xi,yi)`.

Examples

Sample a function at 100 random points between ±`2.0`:

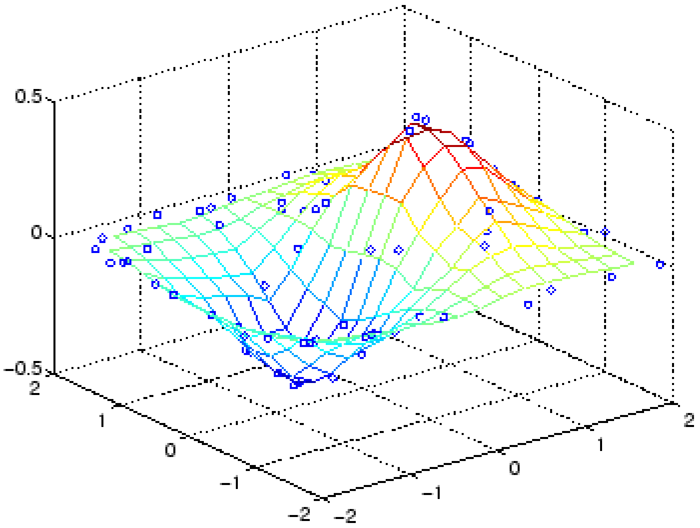
```
rand('seed',0)
x = rand(100,1)*4-2;  y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

`x`, `y`, and `z` are now vectors containing nonuniformly sampled data. Define a regular grid, and grid the data to it:

```
ti = -2:.25:2;
[XI,YI] = meshgrid(ti,ti);
ZI = griddata(x,y,z,XI,YI);
```

Plot the gridded data along with the nonuniform data points used to generate it:

```
mesh(XI,YI,ZI), hold
plot3(x,y,z,'o'), hold off
```



See Also `delaunay`, `griddatan`, `interp2`, `meshgrid`

griddata3

Purpose

Data gridding and hypersurface fitting for 3-D data

`griddata3` will be removed in a future release. Use `TriScatteredInterp` instead.

Syntax

```
w = griddata3(x,y,z,v,xi,yi,zi)
w = griddata3(x,y,z,v,xi,yi,zi,method)
```

Description

`w = griddata3(x,y,z,v,xi,yi,zi)` fits a hypersurface of the form $w = f(x, y, z)$ to the data in the (usually) nonuniformly spaced vectors (`x`, `y`, `z`, `v`). `griddata3` interpolates this hypersurface at the points specified by (`xi`,`yi`,`zi`) to produce `w`. `w` is the same size as `xi`, `yi`, and `zi`.

(`xi`,`yi`,`zi`) is usually a uniform grid (as produced by `meshgrid`) and is where `griddata3` gets its name.

`w = griddata3(x,y,z,v,xi,yi,zi,method)` defines the type of surface that is fit to the data, where `method` is either:

`'linear'`	Tesselation-based linear interpolation (default)
`'nearest'`	Nearest neighbor interpolation

If `method` is `[]`, the default `'linear'` method is used.

`w = griddata3(...,options)` specifies a cell array of strings `options` that were previously in Qhull. Qhull-specific `options` are no longer required and are currently ignored.

`griddata3` uses CGAL, see http://www.cgal.org.

Examples

Create vectors `x`, `y`, and `z` containing nonuniformly sampled data:

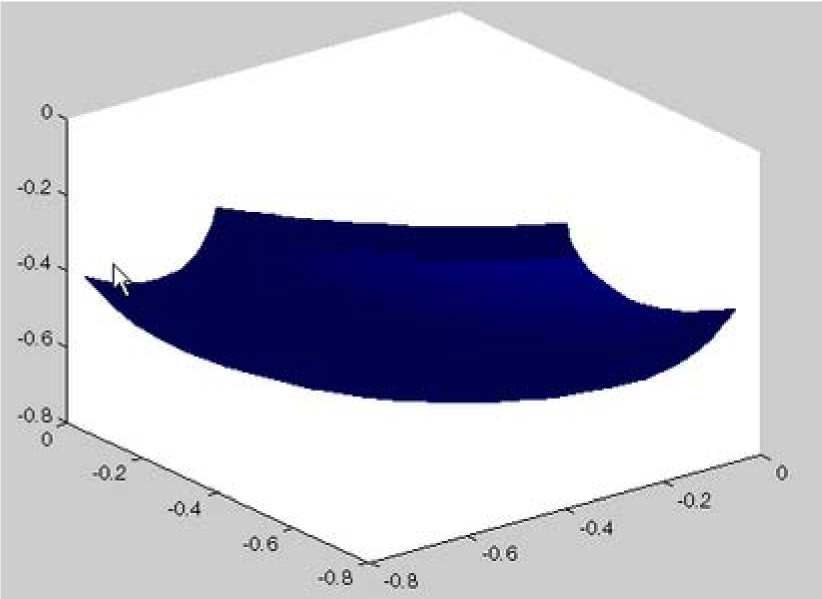
```
x = gallery('uniformdata',[5000 1],0)-1;
y = gallery('uniformdata',[5000 1],1)-1;
z = gallery('uniformdata',[5000 1],2)-1;
v = x.^2 + y.^2 + z.^2;
```

Define a regular grid, and grid the data to it:

```
d = -0.8:0.05:0.8;
[xi,yi,zi] = meshgrid(d,d,d);
w = griddata3(x,y,z,v,xi,yi,zi);
```

Since it is difficult to visualize 4-D data sets, use isosurface at 0.8:

```
p = patch(isosurface(xi,yi,zi,w,0.8));
isonormals(xi,yi,zi,w,p);
set(p,'FaceColor','blue','EdgeColor','none');
view(3), axis equal, axis off, camlight, lighting phong
```



griddatan

Purpose

Data gridding and hypersurface fitting (dimension >= 2)

Syntax

```
yi = griddatan(X,y,xi)
yi = griddatan(x,y,z,v,xi,yi,zi,method)
```

Description

`yi = griddatan(X,y,xi)` fits a hyper-surface of the form $y = f(X)$ to the data in the (usually) nonuniformly-spaced vectors (`X`, `y`). `griddatan` interpolates this hyper-surface at the points specified by `xi` to produce `yi`. `xi` can be nonuniform.

`X` is of dimension `m`-by-`n`, representing `m` points in n-dimensional space. `y` is of dimension `m`-by-`1`, representing `m` values of the hyper-surface f(`X`). `xi` is a vector of size `p`-by-`n`, representing `p` points in the n-dimensional space whose surface value is to be fitted. `yi` is a vector of length `p` approximating the values f(`xi`). The hypersurface always goes through the data points (`X`,`y`). `xi` is usually a uniform grid (as produced by `meshgrid`).

`yi = griddatan(x,y,z,v,xi,yi,zi,method)` defines the type of surface fit to the data, where `'method'` is one of:

`'linear'`	Tessellation-based linear interpolation (default)
`'nearest'`	Nearest neighbor interpolation

All the methods are based on a Delaunay tessellation of the data.

If `method` is `[]`, the default `'linear'` method is used.

`yi = griddatan(x,y,z,v,xi,yi,zi,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default.

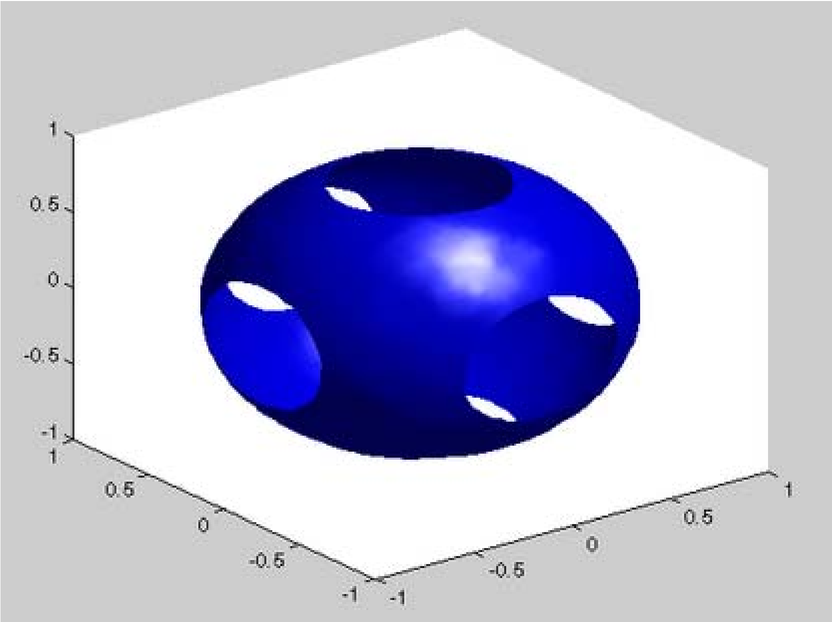
Examples

```
X=2*gallery('uniformdata',[5000 3],0)-1;;
Y = sum(X.^2,2);
d = -0.8:0.05:0.8;
[y0,x0,z0] = ndgrid(d,d,d);
```

```
XI = [x0(:) y0(:) z0(:)];
YI = griddatan(X,Y,XI);
```

Since it is difficult to visualize 4-D data sets, use isosurface at 0.8:

```
YI = reshape(YI, size(x0));
p = patch(isosurface(x0,y0,z0,YI,0.8));
isonormals(x0,y0,z0,YI,p);
set(p,'FaceColor','blue','EdgeColor','none');
view(3), axis equal, axis off, camlight, lighting phong
```



Algorithm

The `griddatan` methods are based on a Delaunay triangulation of the data that uses Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`delaunayn`, `griddata`, `griddata3`, `meshgrid`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, “The Quickhull Algorithm for Convex Hulls,” ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469-483.

Purpose

Generalized singular value decomposition

Syntax

```
[U,V,X,C,S] = gsvd(A,B)
sigma = gsvd(A,B)
```

Description

`[U,V,X,C,S] = gsvd(A,B)` returns unitary matrices `U` and `V`, a (usually) square matrix `X`, and nonnegative diagonal matrices `C` and `S` so that

```
A = U*C*X'
B = V*S*X'
C'*C + S'*S = I
```

`A` and `B` must have the same number of columns, but may have different numbers of rows. If `A` is `m`-by-`p` and `B` is `n`-by-`p`, then `U` is `m`-by-`m`, `V` is `n`-by-`n` and `X` is `p`-by-`q` where `q = min(m+n,p)`.

`sigma = gsvd(A,B)` returns the vector of generalized singular values, `sqrt(diag(C'*C)./diag(S'*S))`.

The nonzero elements of `S` are always on its main diagonal. If `m >= p` the nonzero elements of `C` are also on its main diagonal. But if `m < p`, the nonzero diagonal of `C` is `diag(C,p-m)`. This allows the diagonal elements to be ordered so that the generalized singular values are nondecreasing.

`gsvd(A,B,0)`, with three input arguments and either `m` or `n >= p`, produces the “economy-sized“decomposition where the resulting `U` and `V` have at most `p` columns, and `C` and `S` have at most `p` rows. The generalized singular values are `diag(C)./diag(S)`.

When `B` is square and nonsingular, the generalized singular values, `gsvd(A,B)`, are equal to the ordinary singular values, `svd(A/B)`, but they are sorted in the opposite order. Their reciprocals are `gsvd(B,A)`.

In this formulation of the `gsvd`, no assumptions are made about the individual ranks of `A` or `B`. The matrix `X` has full rank if and only if the matrix `[A;B]` has full rank. In fact, `svd(X)` and `cond(X)` are equal to `svd([A;B])` and `cond([A;B])`. Other formulations, eg. G. Golub and

C. Van Loan [1], require that `null(A)` and `null(B)` do not overlap and replace `X` by `inv(X)` or `inv(X')`.

Note, however, that when `null(A)` and `null(B)` do overlap, the nonzero elements of `C` and `S` are not uniquely determined.

Examples

Example 1

The matrices have at least as many rows as columns.

```
A = reshape(1:15,5,3)
B = magic(3)
A =
          1     6    11
          2     7    12
          3     8    13
          4     9    14
          5    10    15
B =
          8     1     6
          3     5     7
          4     9     2
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 5-by-5 orthogonal `U`, a 3-by-3 orthogonal `V`, a 3-by-3 nonsingular `X`,

```
X =
         2.8284   -9.3761   -6.9346
        -5.6569   -8.3071  -18.3301
         2.8284   -7.2381  -29.7256
```

and

```
C =
         0.0000         0         0
```

```
                 0    0.3155         0
                 0         0    0.9807
                 0         0         0
                 0         0         0
S =
            1.0000         0         0
                 0    0.9489         0
                 0         0    0.1957
```

Since `A` is rank deficient, the first diagonal element of `C` is zero.

The economy sized decomposition,

```
[U,V,X,C,S] = gsvd(A,B,0)
```

produces a 5-by-3 matrix `U` and a 3-by-3 matrix `C`.

```
U =
            0.5700   -0.6457   -0.4279
           -0.7455   -0.3296   -0.4375
           -0.1702   -0.0135   -0.4470
            0.2966    0.3026   -0.4566
            0.0490    0.6187   -0.4661

C =
            0.0000         0         0
                 0    0.3155         0
                 0         0    0.9807
```

The other three matrices, `V`, `X`, and `S` are the same as those obtained with the full decomposition.

The generalized singular values are the ratios of the diagonal elements of `C` and `S`.

```
sigma = gsvd(A,B)
sigma =
            0.0000
            0.3325
```

```
          5.0123
```

These values are a reordering of the ordinary singular values

```
svd(A/B)
ans =
          5.0123
          0.3325
          0.0000
```

Example 2

The matrices have at least as many columns as rows.

```
A = reshape(1:15,3,5)
B = magic(5)
A =

      1      4      7     10     13
          2      5      8     11     14
          3      6      9     12     15
B =
     17     24      1      8     15
         23      5      7     14     16
          4      6     13     20     22
         10     12     19     21      3
         11     18     25      2      9
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 3-by-3 orthogonal `U`, a 5-by-5 orthogonal `V`, a 5-by-5 nonsingular `X` and

```
C =
               0         0    0.0000         0         0
               0         0         0    0.0439         0
               0         0         0         0    0.7432
```

```
S =
        1.0000         0         0         0         0
             0    1.0000         0         0         0
             0         0    1.0000         0         0
             0         0         0    0.9990         0
             0         0         0         0    0.6690
```

In this situation, the nonzero diagonal of `C` is `diag(C,2)`. The generalized singular values include three zeros.

```
sigma = gsvd(A,B)
sigma =
             0
             0
        0.0000
        0.0439
        1.1109
```

Reversing the roles of `A` and `B` reciprocates these values, producing two infinities.

```
gsvd(B,A)
ans =
      1.0e+016 *

        0.0000
        0.0000
        4.4126
           Inf
           Inf
```

Algorithm

The generalized singular value decomposition uses the C-S decomposition described in [1], as well as the built-in `svd` and `qr` functions. The C-S decomposition is implemented in a subfunction in the `gsvd` M-file.

Diagnostics

The only warning or error message produced by `gsvd` itself occurs when the two input arguments do not have the same number of columns.

See Also `qr`, `svd`

References [1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

Purpose

Test for greater than

Syntax

```
A > B
gt(A, B)
```

Description

`A > B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is greater than `B`, or set to logical 0 (`false`) where `A` is less than or equal to `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLAB software returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`gt(A, B)` is called for the syntax `A>B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are greater than the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A > B
ans =
     1     0     0     1     1     1
     0     1     0     1     1     1
     1     0     0     1     0     1
     0     1     1     0     1     0
```

```
     1     0     1     1     0     0
     0     1     1     1     0     1
```

See Also `lt`, `ge`, `le`, `ne`, `eq`,

Purpose

Mouse placement of text in 2-D view

Syntax

```
gtext('string')
gtext({'string1','string2','string3',...})
gtext({'string1';'string2';'string3';...})
h = gtext(...)
```

Description

`gtext` displays a text string in the current figure window after you select a location with the mouse.

`gtext('string')` waits for you to press a mouse button or keyboard key while the pointer is within a figure window. Pressing a mouse button or any key places `'string'` on the plot at the selected location.

`gtext({'string1','string2','string3',...})` places all strings with one click, each on a separate line.

`gtext({'string1';'string2';'string3';...})` places one string per click, in the sequence specified.

`h = gtext(...)` returns the handle to a text graphics object that is placed on the plot at the location you select.

Remarks

As you move the pointer into a figure window, the pointer becomes crosshairs to indicate that `gtext` is waiting for you to select a location. `gtext` uses the functions `ginput` and `text`.

Examples

Place a label on the current plot:

```
gtext('Note this divergence!')
```

See Also

`ginput`, `text`

"Annotating Plots" on page 1-92 for related functions

guidata

Purpose

Store or retrieve GUI data

Syntax

```
guidata(object_handle,data)
data = guidata(object_handle)
```

Description

`guidata(object_handle,data)` stores the variable `data` as GUI data. If `object_handle` is not a figure handle, then the object's parent figure is used. `data` can be any MATLAB variable, but is typically a structure, which enables you to add new fields as required.

`guidata` can manage only one variable at any time. Subsequent calls to `guidata(object_handle,data)` overwrite the previously created version of GUI `data`.

GUIDE Uses guidata

GUIDE uses `guidata` to store and maintain the `handles` structure. From a GUIDE-generated GUI M-file, do *not* use `guidata` to store any data other than `handles`. If you do, you may overwrite the `handles` structure and your GUI will not work. If you need to store other data with your GUI, you can add new fields to the `handles` structure and place your data there. See GUI Data in the MATLAB documentation.

`data = guidata(object_handle)` returns previously stored data, or an empty matrix if nothing is stored.

To change the data managed by `guidata`:

1. Get a copy of the data with the command `data = guidata(object_handle)`.

2. Make the desired changes to `data`.

3. Save the changed version of `data` with the command `guidata(object_handle,data)`.

`guidata` provides application developers with a convenient interface to a figure's application data:

- You do not need to create and maintain a hard-coded property name for the application data throughout your source code.
- You can access the data from within a subfunction callback routine using the component's handle (which is returned by `gcbo`), without needing to find the figure's handle.

If you are not using GUIDE, `guidata` is particularly useful in conjunction with `guihandles`, which creates a structure containing the handles of all the components in a GUI.

Examples

This example calls `guidata` to save a structure containing a GUI figure's application data from within the initialization section of the application M-file. The first section shows how to do this within a GUI you create manually. The second section shows how the code differs when you use GUIDE to create a template M-file. GUIDE provides a `handles` structure as an argument to all subfunction callbacks, so you do not need to call `guidata` to obtain it. You do, however, need to call `guidata` to save changes you make to the structure.

Using guidata in a Programmed GUI

Calling the `guihandles` function creates the structure into which your code places additional data. It contains all handles used by the figure at the time it is called, generating field names based on each object's `Tag` property.

```
% Create figure to use as GUI in your main function or a subfunction
figure_handle = figure('Toolbar','none');
% create structure of handles
myhandles = guihandles(figure_handle);
% Add some additional data as a new field called numberOfErrors
myhandles.numberOfErrors = 0;
% Save the structure
guidata(figure_handle,myhandles)
```

You can recall the data from within a subfunction callback, modify it, and then replace the structure in the figure:

```
function My_Callback()
% ...
% Get the structure using guidata in the subfunction
myhandles = guidata(gcbo);
% Modify the value of your counter
myhandles.numberOfErrors = myhandles.numberOfErrors + 1;
% Save the change you made to the structure
guidata(gcbo,myhandles)
```

Using guidata in a GUIDE GUI

If you use GUIDE, you do not need to call `guihandles` to create a structure, because GUIDE generates a `handles` structure that contains the GUI's handles. You can add your own data to it, for example from within the `OpeningFcn` template that GUIDE creates:

```
% --- Executes just before simple_gui_tab is made visible.
function my_GUIDE_GUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to simple_gui_tab (see VARARGIN)
% ...

% add some additional data as a new field called numberOfErrors
handles.numberOfErrors = 0;
% Save the change you made to the structure
guidata(hObject,handles)
```

Notice that you use the input argument `hObject` in place of `gcbo` to refer to the object whose callback is executing.

Suppose you needed to access the `numberOfErrors` field in a push button callback. Your callback code now looks something like this:

```
% --- Executes on button press in pushbutton1.
function my_GUIDE_GUI_pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% ...

% No need to call guidata to obtain a structure;
% it is provided by GUIDE via the handles argument
handles.numberOfErrors = handles.numberOfErrors + 1;
% save the changes to the structure
guidata(hObject,handles)
```

See Also

`guide`, `guihandles`, `getappdata`, `setappdata`

guide

Purpose Open GUI Layout Editor

Syntax

```
guide
guide('filename.fig')
guide('fullpath')
guide(HandleList)
```

Description `guide` initiates the GUI design environment (GUIDE) tools that allow you to create or edit GUIs interactively.

`guide` opens the GUIDE Quick Start dialog where you can choose to open a previously created GUI or create a new one using one of the provided templates.

`guide('filename.fig')` opens the FIG-file named `filename.fig` for editing if it is on the MATLAB path.

`guide('fullpath')` opens the FIG-file at `fullpath` even if it is not on the MATLAB path.

`guide(HandleList)` opens the content of each of the figures in `HandleList` in a separate copy of the GUIDE design environment.

See Also `inspect`

Creating GUIs

Purpose

Create structure of handles

Syntax

```
handles = guihandles(object_handle)
handles = guihandles
```

Description

`handles = guihandles(object_handle)` returns a structure containing the handles of the objects in a figure, using the value of their `Tag` properties as the fieldnames, with the following caveats:

- Objects are excluded if their `Tag` properties are empty, or are not legal variable names.
- If several objects have the same `Tag`, that field in the structure contains a vector of handles.
- Objects with hidden handles are included in the structure.

`handles = guihandles` returns a structure of handles for the current figure.

See Also

`guidata`, `guide`, `getappdata`, `setappdata`

Purpose

Uncompress GNU zip files

Syntax

```
gunzip(files)
gunzip(files,outputdir)
gunzip(url, ...)
filenames = gunzip(...)
```

Description

`gunzip(files)` uncompresses GNU zip files from the list of files specified in `files`. Directories recursively `gunzip` all of their content. The output files have the same name, excluding the extension `.gz`, and are written to the same directory as the input files.

`files` is a string or cell array of strings containing a list of files or directories. Individual files that are on the MATLAB path can be specified as partial path names. Otherwise, an individual file can be specified relative to the current directory or with an absolute path.

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.

`gunzip(files,outputdir)` writes the gunzipped file into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`gunzip(url, ...)` extracts the GNU zip contents from an Internet universal resource locator (URL). The URL must include the protocol type (for example, `'http://'`). MATLAB downloads the URL to the temp directory, and then deletes it.

`filenames = gunzip(...)` gunzips the files and returns the relative path names of the gunzipped files in the string cell array `filenames`.

Examples

To `gunzip` all `.gz` files in the current directory, type:

```
gunzip('*.gz');
```

To `gunzip` Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory `ncm`, type:

```
url ='http://www.mathworks.com/moler/ncm.tar.gz';
gunzip(url,'ncm')
untar('ncm/ncm.tar','ncm')
```

See Also

`gzip`, `tar`, `untar`, `unzip`, `zip`

gzip

Purpose

Compress files into GNU zip files

Syntax

```
gzip(files)
gzip(files,outputdir)
filenames = gzip(...)
```

Description

`gzip(files)` creates GNU zip files from the list of files specified in `files`. Directories recursively `gzip` all their contents. Each output gzipped file is written to the same directory as the input file and with the file extension `.gz`.

`files` is a string or cell array of strings containing a list of files or directories to `gzip`. Individual files that are on the MATLAB path can be specified as partial path names. Otherwise, an individual file can be specified relative to the current directory or with an absolute path.

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.

`gzip(files,outputdir)` writes the gzipped files into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`filenames = gzip(...)` gzips the files and returns the relative path names of all gzipped files in the string cell array `filenames`.

Example

To `gzip` all `.m` and `.mat` files in the current directory and store the results in the directory `archive`, type:

```
gzip({'*.m','*.mat'},'archive');
```

See Also

`gunzip`, `tar`, `untar`, `unzip`, `zip`

Purpose Hadamard matrix

Syntax

```
H = hadamard(n)
```

Description `H = hadamard(n)` returns the Hadamard matrix of order `n`.

Definition Hadamard matrices are matrices of `1`'s and `-1`'s whose columns are orthogonal,

```
H'*H = n*I
```

where `[n n]=size(H)` and `I = eye(n,n)` ,.

They have applications in several different areas, including combinatorics, signal processing, and numerical analysis, [1], [2].

An `n`-by-`n` Hadamard matrix with `n > 2` exists only if `rem(n,4) = 0`. This function handles only the cases where `n`, `n/12`, or `n/20` is a power of 2.

Examples The command `hadamard(4)` produces the 4-by-4 matrix:

```
1     1     1     1
1    -1     1    -1
1     1    -1    -1
1    -1    -1     1
```

See Also `compan`, `hankel`, `toeplitz`

References

[1] Ryser, H. J., *Combinatorial Mathematics*, John Wiley and Sons, 1963.

[2] Pratt, W. K., *Digital Signal Processing*, John Wiley and Sons, 1978.

Purpose

Abstract class for deriving handle classes

Syntax

`classdef myclass < handle`

Description

The `handle` class is the superclass for all classes that follow handle semantics. A handle is a reference to an object. If you copy an object's handle, MATLAB copies only the handle and both the original and copy refer to the same object data.

This behavior is equivalent to that of Handle Graphics objects, where the handle of a graphics object always refers to a particular object regardless of whether you save the handle when you create the object, store it in another variable, or obtain it with convenience functions like `findobj` or `gca`.

If you want to create a class the defines events, you must derive that class from the `handle` class.

The `handle` class is an abstract class so you cannot create an instance of this class directly. You use the `handle` class to derive other classes, which can be concrete classes whose instances are handle objects. See for information on using handle classes.

`classdef myclass < handle` makes *`myclass`* a subclass of the `handle` class.

Handle Class Methods

When you derive a class from the `handle` class, your class inherits the following methods.

Method	Purpose
`addlistener`	Creates a listener for the specified event and assigns a callback function to execute when the event occurs.
`notify`	Broadcast a notice that a specific event is occurring on a specified handle object or array of handle objects.

Method	Purpose
`delete`	Handle object destructor method that is called when the object's lifecycle ends.
`disp`	Handle object `disp` method which is called by the `display` method. See the MATLAB `disp` function.
`display`	Handle object `display` method called when MATLAB software interprets an expression returning a handle object that is not terminated by a semicolon. See the MATLAB `display` function.
`findobj`	Finds objects matching the specified conditions from the input array of handle objects .
`findprop`	Returns a `meta.property` objects associated with the specified property name.
`fields`	Returns a cell array of string containing the names of public properties.
`fieldnames`	Returns a cell array of string containing the names of public properties. See the MATLAB `fieldnames` function.
`isvalid`	Returns a logical array in which elements are true if the corresponding elements in the input array are valid handles. This method is `Sealed` so you cannot override it in a handle subclass.

Method	Purpose

`eq`
`ne`
`lt`
`le`
`gt`
`ge`
`relationaloperators`
	Relational functions return a logical array of the same size as the pair of input handle object arrays. Comparisons use a number associated with each handle. You can assume that the same two handles will compare as equal and the repeated comparison of any two handles will yield the same result in the same MATLAB session. Different handles are always not-equal. The order of handles is purely arbitrary, but consistent.

`ctranspose` `transpose`	Transposes the elements of the handle object array.
`permute`	Rearranges the dimensions of the handle object array. See the MATLAB `permute` function.
`reshape`	Changes the dimensions of the handle object array to the specified dimensions. See the MATLAB `reshape` function.
`sort`	Sort the handle objects in any array in ascending or descending order. The order of handles is purely arbitrary, but reproducible in a given MATLAB session. See the MATLAB `sort` function.

Handle Class Events

The handle class defines one event:

```
ObjectBeingDestroyed
```

This event is triggered when the handle object is about to be destroyed. If you define a listener for this event, its callback executes before the handle object is destroyed.

You can add a listener for this event using the `addlistener` method. See for more information on using events and listeners.

Handle Subclasses

There are two abstract `handle` subclasses that you can use to derive `handle` classes:

- `hgsetget` — use when you want to create a handle class that inherits set and get methods having the same behavior as Handle Graphics `set` and `get` functions.
- `dynamicprops` — use when you want to create a handle class that allows you to add instance data (dynamically defined properties) to objects.

Useful Functions

- `properties` — list the class public properties
- `methods` — list the class methods
- `events` — list the events defined by the class

Note that `ishandle` does not test for handle class objects. Use `isa` instead.

hankel

Purpose Hankel matrix

Syntax

```
H = hankel(c)
H = hankel(c,r)
```

Description

`H = hankel(c)` returns the square Hankel matrix whose first column is `c` and whose elements are zero below the first anti-diagonal.

`H = hankel(c,r)` returns a Hankel matrix whose first column is `c` and whose last row is `r`. If the last element of `c` differs from the first element of `r`, the last element of `c` prevails.

Definition

A Hankel matrix is a matrix that is symmetric and constant across the anti-diagonals, and has elements `h(i,j) = p(i+j-1)`, where vector `p = [c r(2:end)]` completely determines the Hankel matrix.

Examples

A Hankel matrix with anti-diagonal disagreement is

```
c = 1:3; r = 7:10;
h = hankel(c,r)
h =
    1    2    3    8
    2    3    8    9
    3    8    9   10

p = [1 2 3 8 9 10]
```

See Also

`hadamard`, `toeplitz`, `kron`

Purpose

Summary of MATLAB HDF4 capabilities

Description

The MATLAB software provides a set of low-level functions that enable you to access the HDF4 library developed by the National Center for Supercomputing Applications (NCSA). For information about HDF4, go to the HDF Web page at `http://www.hdfgroup.org`.

Note For information about MATLAB HDF5 capabilities, which is a completely separate, incompatible format, see `hdf5`.

The following table lists all the HDF4 application programming interfaces (APIs) supported by MATLAB with the name of the MATLAB function used to access the API. To use these functions, you must be familiar with the HDF library. For more information about using these MATLAB functions, see and .

Application Programming Interface	Description	MATLAB Function
Annotations	Stores, manages, and retrieves text used to describe an HDF file or any of the data structures contained in the file.	`hdfan`
General Raster Images	Stores, manages, and retrieves raster images, their dimensions and palettes. It can also manipulate unattached palettes. Note: Use the MATLAB functions `imread` and `imwrite` with HDF raster image formats.	`hdfdf24`, `hdfdfr8`

Application Programming Interface	Description	MATLAB Function
HDF-EOS	Provides functions to read HDF-EOS grid (GD), point (PT), and swath (SW) data.	`hdfgd`, `hdfpt`, `hdfsw`
HDF Utilities	Provides functions to open and close HDF files and handle errors.	`hdfh`, `hdfhd`, `hdfhe`
MATLAB HDF Utilities	Provides utility functions that help you work with HDF files in the MATLAB environment.	`hdfml`
Scientific Data	Stores, manages, and retrieves multidimensional arrays of character or numeric data, along with their dimensions and attributes.	`hdfsd`
V Groups	Creates and retrieves groups of other HDF data objects, such as raster images or V data.	`hdfv`
V Data	Stores, manages, and retrieves multivariate data stored as records in a table.	`hdfvf`, `hdfvh`, `hdfvs`

See Also

`hdfinfo`,`hdfread`, `hdftool`, `imread`

hdf5

Purpose Summary of MATLAB HDF5 capabilities

Description The MATLAB software provides both high-level and low-level access to HDF5 files. The high-level access functions make it easy to read a data set from an HDF5 file or write a variable from the MATLAB workspace into an HDF5 file. The MATLAB low-level interface provides direct access to the more than 200 functions in the HDF5 library. MATLAB currently supports version HDF5-1.8.1 of the library.

Note For information about MATLAB HDF4 capabilities, which is a completely separate, incompatible format, see `hdf`.

The following sections provide an overview of both this high- and low-level access. To use these MATLAB functions, you must be familiar with HDF5 programming concepts and, when using the low-level functions, details about the functions in the library. To get this information, go to the HDF Web page at `http://www.hdfgroup.org`.

High-level Access

MATLAB includes three functions that provide high-level access to HDF5 files:

- `hdf5info`
- `hdf5read`
- `hdf5write`

Using these functions you can read data and metadata from an HDF5 file and write data from the MATLAB workspace to a file in HDF5 format. For more information about these functions, see their individual reference pages.

Low-level Access

MATLAB provides direct access to the over 200 functions in the HDF5 Library. Using these functions, you can read and write complex datatypes, utilize HDF5 data subsetting capabilities, and take advantage of other features present in the HDF5 library.

The HDF5 library organizes the routines in the library into interfaces. MATLAB organizes the corresponding MATLAB functions into class directories that match these HDF5 library interfaces. For example, the MATLAB functions for the HDF5 Attribute Interface are in the @H5A class directory.

The following table lists all the HDF5 library interfaces in alphabetical order by name. The table includes the name of the associated MATLAB class directory.

HDF5 Library Interface	MATLAB Class Directory	Description
Attribute	`@H5A`	Manipulate metadata associated with data sets or groups
Dataset	`@H5D`	Manipulate multidimensional arrays of data elements, together with supporting metadata
Dataspace	`@H5S`	Define and work with data spaces, which describe the dimensionality of a data set
Datatype	`@H5T`	Define the type of variable that is stored in a data set
Error	`@H5E`	Handle errors
File	`@H5F`	Access files
Filters and Compression	`@H5Z`	Create inline data filters and data compression
Group	`@H5G`	Organize objects in a file; analogous to a directory structure
Identifier	`@H5I`	Manipulate HDF5 object identifiers

HDF5 Library Interface	MATLAB Class Directory	Description
Library	`@H5`	General-purpose functions for use with the entire HDF5 library, such as initialization
MATLAB	`@H5ML`	MATLAB utility functions that are not part of the HDF5 library itself.
Property	`@H5P`	Manipulate object property lists
Reference	`@H5R`	Manipulate HDF5 references, which are like UNIX links or Windows shortcuts

In most cases, the syntax of the MATLAB function is identical to the syntax of the HDF5 library function. To get detailed information about the MATLAB syntax of an HDF5 library function, view the help for the individual MATLAB function, as follows:

```
help @H5F/open
```

To view a list of all the MATLAB HDF5 functions in a particular interface, type:

```
help imagesci/@H5F
```

See Also

`hdf`, `hdf5info`, `hdf5read`, `hdf5write`

Purpose

Information about HDF5 file

Syntax

```
fileinfo = hdf5info(filename)
fileinfo = hdf5info(...,'ReadAttributes',BOOL)
[...] = hdf5info(..., 'V71Dimensions', BOOL)
```

Description

`fileinfo = hdf5info(filename)` returns a structure `fileinfo` whose fields contain information about the contents of the HDF5 file `filename`. `filename` is a string that specifies the name of the HDF5 file.

`fileinfo = hdf5info(...,'ReadAttributes',BOOL)` specifies whether `hdf5info` returns the values of the attributes or just information describing the attributes. By default, `hdf5info` reads in attribute values (`BOOL` = `true`).

`[...]  = hdf5info(..., 'V71Dimensions', BOOL)` specifies whether to report the dimensions of data sets and attributes as they were returned in previous versions of `hdf5info` (MATLAB 7.1 [R14SP3] and earlier). If `BOOL` is true, `hdf5info` swaps the first two dimensions of the data set. This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimenions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, swapping these dimensions may not correctly reflect the intent of the data in the file and may invalidate metadata. When `BOOL` is false (the default), `hdf5info` returns data dimensions that correctly reflect the data ordering as it is written in the file—each dimension in the output variable matches the same dimension in the file.

Note If you use the `'V71Dimensions'` parameter and intend on passing the `fileinfo` structure returned to the `hdf5read` function, you should also specify the `'V71Dimensions'` parameters with `hdf5read`. If you do not, `hdf5read` uses the new behavior when reading the data set and certain metadata returned by `hdf5info` does not match the actual data returned by `hdf5read`.

Examples

```
fileinfo = hdf5info('example.h5')

fileinfo =

          Filename: 'example.h5'
        LibVersion: '1.4.5'
            Offset: 0
          FileSize: 8172
    GroupHierarchy: [1x1 struct]
```

To get more information about the contents of the HDF5 file, look at the `GroupHierarchy` field in the `fileinfo` structure returned by `hdf5info`.

```
toplevel = fileinfo.GroupHierarchy

toplevel =

      Filename: [1x64 char]
          Name: '/'
        Groups: [1x2 struct]
      Datasets: []
     Datatypes: []
         Links: []
    Attributes: [1x2 struct]
```

To probe further into the file hierarchy, keep examining the `Groups` field.

See also

`hdf5read`, `hdf5write`

hdf5read

Purpose Read HDF5 file

Syntax

```
data = hdf5read(filename,datasetname)
attr = hdf5read(filename,attributename)
[data, attr] = hdf5read(...,'ReadAttributes',BOOL)
data = hdf5read(hinfo)
[...] = hdf5read(..., 'V71Dimensions', BOOL)
```

Description

`data = hdf5read(filename,datasetname)` reads all the data in the data set `datasetname` that is stored in the HDF5 file `filename` and returns it in the variable `data`. To determine the names of data sets in an HDF5 file, use the `hdf5info` function.

The return value, `data`, is a multidimensional array. `hdf5read` maps HDF5 data types to native MATLAB data types, whenever possible. If it cannot represent the data using MATLAB data types, `hdf5read` uses one of the HDF5 data type objects. For example, if an HDF5 file contains a data set made up of an enumerated data type, `hdf5read` uses the `hdf5.h5enum` object to represent the data in the MATLAB workspace. The `hdf5.h5enum` object has data members that store the enumerations (names), their corresponding values, and the enumerated data. For more information about the HDF5 data type objects, see the `hdf5` reference page.

`attr = hdf5read(filename,attributename)` reads all the metadata in the attribute `attributename`, stored in the HDF5 file `filename`, and returns it in the variable `attr`. To determine the names of attributes in an HDF5 file, use the `hdf5info` function.

`[data, attr] = hdf5read(...,'ReadAttributes',BOOL)` reads all the data, as well as all of the associated attribute information contained within that data set. By default, `BOOL` is false.

`data = hdf5read(hinfo)` reads all of the data in the data set specified in the structure `hinfo` and returns it in the variable `data`. The `hinfo` structure is extracted from the output returned by `hdf5info`, which specifies an HDF5 file and a specific data set.

`[...]  = hdf5read(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets read from the file. If `BOOL` is true, `hdf5read` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data dimensions correctly reflect the data ordering as it is written in the file — each dimension in the output variable matches the same dimension in the file.

Examples

Use `hdf5info` to get information about an HDF5 file and then use `hdf5read` to read a data set, using the information structure (`hinfo`) returned by `hdf5info` to specify the data set.

```
hinfo = hdf5info('example.h5');
dset = hdf5read(hinfo.GroupHierarchy.Groups(2).Datasets(1));
```

See Also

`hdf5`, `hdf5info`, `hdf5write`

hdf5write

Purpose

Write data to file in HDF5 format

Syntax

```
hdf5write(filename,location,dataset)
hdf5write(filename,details,dataset)
hdf5write(filename,details,attribute)
hdf5write(filename, details1, dataset1, details2, dataset2,
   ...)
hdf5write(filename,...,'WriteMode',mode,...)
hdf5write(..., 'V71Dimensions', BOOL)
```

Description

`hdf5write(filename,location,dataset)` writes the data `dataset` to the HDF5 file, `filename`. If `filename` does not exist, `hdf5write` creates it. If `filename` exists, `hdf5write` overwrites the existing file, by default, but you can also append data to an existing file using an optional syntax.

`location` defines where to write the data set in the file. HDF5 files are organized in a hierarchical structure similar to a UNIX directory structure. `location` is a string that resembles a UNIX path.

`hdf5write` maps the data in `dataset` to HDF5 data types according to rules outlined below.

`hdf5write(filename,details,dataset)` writes `dataset` to `filename` using the values in the `details` structure. For a data set, the `details` structure can contain the following fields.

Field Name	Description	Data Type
`Location`	Location of the data set in the file	Character array
`Name`	Name to attach to the data set	Character array

`hdf5write(filename,details,attribute)` writes the metadata `attribute` to `filename` using the values in the `details` structure. For an attribute, the `details` structure can contain following fields.

Field Name	Description	Data Type
`AttachedTo`	Location of the object this attribute modifies	Structure array
`AttachType`	Identifies what kind of object this attribute modifies; possible values are `'group'` and `'dataset'`	Character array
`Name`	Name to attach to the data set	Character array

`hdf5write(filename, details1, dataset1, details2, dataset2,...)` writes multiple data sets and associated attributes to `filename` in one operation. Each data set and attribute must have an associated `details` structure.

`hdf5write(filename,...,'WriteMode',mode,...)` specifies whether `hdf5write` overwrites the existing file (the default) or appends data sets and attributes to the file. Possible values for mode are `'overwrite'` and `'append'`.

`hdf5write(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets written to the file. If `BOOL` is true, `hdf5write` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data written to the file correctly reflects the data ordering of the data sets — each dimension in the file's data sets matches the same dimension in the corresponding MATLAB variable.

Data Type Mappings

The following table lists how `hdf5write` maps the data type from the workspace into an HDF5 file. If the data in the workspace that is being written to the file is a MATLAB data type, `hdf5write` uses the following rules when translating MATLAB data into HDF5 data objects.

MATLAB Data Type	HDF5 Data Set or Attribute
Numeric	Corresponding HDF5 native data type. For example, if the workspace data type is `uint8`, the `hdf5write` function writes the data to the file as 8-bit integers. The size of the HDF5 dataspace is the same size as the MATLAB array.
String	Single, null-terminated string
Cell array of strings	Multiple, null-terminated strings, each the same length. Length is determined by the length of the longest string in the cell array. The size of the HDF5 dataspace is the same size as the cell array.
Cell array of numeric data	Numeric array, the same dimensions as the cell array. The elements of the array must all have the same size and type. The data type is determined by the first element in the cell array.
Structure array	HDF5 compound type. Individual fields in the structure employ the same data translation rules for individual data types. For example, a cell array of strings becomes a multiple, null-terminated strings.
HDF5 objects	If the data being written to the file is composed of HDF5 objects, `hdf5write` uses the same data type when writing to the file. For all HDF5 objects, except `HDF5.h5enum` objects, the dataspace has the same dimensions as the array of HDF5 objects passed to the function. For `HDF5.h5enum` objects, the size and dimensions of the data set in the HDF5 file is the same as the object's Data field.

Examples

Write a 5-by-5 data set of `uint8` values to the root group.

```
hdf5write('myfile.h5', '/dataset1', uint8(magic(5)))
```

Write a 2-by-2 string data set in a subgroup.

```
dataset = {'north', 'south'; 'east', 'west'};
hdf5write('myfile2.h5', '/group1/dataset1.1', dataset);
```

Write a data set and attribute to an existing group.

```
dset = single(rand(10,10));
dset_details.Location = '/group1/dataset1.2';
dset_details.Name = 'Random';

attr = 'Some random data';
attr_details.Name = 'Description';
attr_details.AttachedTo = '/group1/dataset1.2/Random';
attr_details.AttachType = 'dataset';

hdf5write('myfile2.h5', dset_details, dset, ...
          attr_details, attr, 'WriteMode', 'append');
```

Write a data set using objects.

```
dset = hdf5.h5array(magic(5));
hdf5write('myfile3.h5', '/g1/objects', dset);
```

See Also

`hdf5`, `hdf5read`, `hdf5info`

Purpose

Information about HDF4 or HDF-EOS file

Syntax

```
S = hdfinfo(filename)
S = hdfinfo(filename,mode)
```

Description

`S = hdfinfo(filename)` returns a structure `S` whose fields contain information about the contents of an HDF4 or HDF-EOS file. `filename` is a string that specifies the name of the HDF4 file.

`S = hdfinfo(filename,mode)` reads the file as an HDF4 file, if `mode` is `'hdf'`, or as an HDF-EOS file, if `mode` is `'eos'`. If `mode` is `'eos'`, only HDF-EOS data objects are queried. To retrieve information on the entire contents of a file containing both HDF4 and HDF-EOS objects, `mode` must be `'hdf'`.

Note `hdfinfo` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To get information about an HDF5 file, use `hdf5info`.

The set of fields in the returned structure `S` depends on the individual file. Fields that can be present in the `S` structure are shown in the following table.

Mode	Field Name	Description	Return Type
`HDF`	`Attributes`	Attributes of the data set	Structure array
	`Description`	Annotation description	Cell array
	`Filename`	Name of the file	String
	`Label`	Annotation label	Cell array
	`Raster8`	Description of 8-bit raster images	Structure array

Mode	Field Name	Description	Return Type
	`Raster24`	Description of 24-bit raster images	Structure array
	`SDS`	Description of scientific data sets	Structure array
	`Vdata`	Description of Vdata sets	Structure array
	`Vgroup`	Description of Vgroups	Structure array
`EOS`	`Filename`	Name of the file	String
	`Grid`	Grid data	Structure array
	`Point`	Point data	Structure array
	`Swath`	Swath data	Structure array

Those fields in the table above that contain structure arrays are further described in the tables shown below.

Fields Common to Returned Structure Arrays

Structure arrays returned by `hdfinfo` contain some common fields. These are shown in the table below. Not all structure arrays will contain all of these fields.

Field Name	Description	Data Type
`Attributes`	Data set attributes. Contains fields `Name` and `Value`.	Structure array
`Description`	Annotation description	Cell array
`Filename`	Name of the file	String
`Label`	Annotation label	Cell array

Field Name	Description	Data Type
`Name`	Name of the data set	String
`Rank`	Number of dimensions of the data set	Double
`Ref`	Data set reference number	Double
`Type`	Type of HDF or HDF-EOS object	String

Fields Specific to Certain Structures

Structure arrays returned by `hdfinfo` also contain fields that are unique to each structure. These are shown in the tables below.

Fields of the Attribute Structure

Field Name	Description	Data Type
`Name`	Attribute name	String
`Value`	Attribute value or description	Numeric or string

Fields of the Raster8 and Raster24 Structures

Field Name	Description	Data Type
`HasPalette`	`1` (`true`) if the image has an associated palette, otherwise `0` (`false`) (8-bit only)	Logical
`Height`	Height of the image, in pixels	Number
`Interlace`	Interlace mode of the image (24-bit only)	String
`Name`	Name of the image	String
`Width`	Width of the image, in pixels	Number

Fields of the SDS Structure

Field Name	Description	Data Type
`DataType`	Data precision	String
`Dims`	Dimensions of the data set. Contains fields `Name`, `DataType`, `Size`, `Scale`, and `Attributes`. `Scale` is an array of numbers to place along the dimension and demarcate intervals in the data set.	Structure array
`Index`	Index of the SDS	Number

Fields of the Vdata Structure

Field Name	Description	Data Type
`DataAttributes`	Attributes of the entire data set. Contains fields `Name` and `Value`.	Structure array
`Class`	Class name of the data set	String
`Fields`	Fields of the Vdata. Contains fields `Name` and `Attributes`.	Structure array
`NumRecords`	Number of data set records	Double
`IsAttribute`	`1` (`true`) if Vdata is an attribute, otherwise `0` (`false`)	Logical

Fields of the Vgroup Structure

Field Name	Description	Data Type
`Class`	Class name of the data set	String

Fields of the Vgroup Structure (Continued)

Field Name	Description	Data Type
`Raster8`	Description of the 8-bit raster image	Structure array
`Raster24`	Description of the 24-bit raster image	Structure array
`SDS`	Description of the Scientific Data sets	Structure array
`Tag`	Tag of this Vgroup	Number
`Vdata`	Description of the Vdata sets	Structure array
`Vgroup`	Description of the Vgroups	Structure array

Fields of the Grid Structure

Field Name	Description	Data Type
`Columns`	Number of columns in the grid	Number
`DataFields`	Description of the data fields in each `Grid` field of the grid. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, `FillValue`, and `TileDims`.	Structure array
`LowerRight`	Lower right corner location, in meters	Number
`Origin Code`	Origin code for the grid	Number
`PixRegCode`	Pixel registration code	Number

Fields of the Grid Structure (Continued)

Field Name	Description	Data Type
`Projection`	Projection code, zone code, sphere code, and projection parameters of the grid. Contains fields `ProjCode`, `ZoneCode`, `SphereCode`, and `ProjParam`.	Structure
`Rows`	Number of rows in the grid	Number
`UpperLeft`	Upper left corner location, in meters	Number

Fields of the Point Structure

Field Name	Description	Data Type
`Level`	Description of each level of the point. Contains fields `Name`, `NumRecords`, `FieldNames`, `DataType`, and `Index`.	Structure

Fields of the Swath Structure

Field Name	Description	Data Type
`DataFields`	Data fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array

Fields of the Swath Structure (Continued)

Field Name	Description	Data Type
`GeolocationFields`	Geolocation fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array
`IdxMapInfo`	Relationship between indexed elements of the geolocation mapping. Contains fields `Map` and `Size`.	Structure
`MapInfo`	Relationship between data and geolocation fields. Contains fields `Map`, `Offset`, and `Increment`.	Structure

Examples

To retrieve information about the file `example.hdf`,

```
fileinfo = hdfinfo('example.hdf')

fileinfo =
    Filename: 'example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

And to retrieve information from this about the scientific data set in `example.hdf`,

```
sds_info = fileinfo.SDS

sds_info =
       Filename: 'example.hdf'
           Type: 'Scientific Data Set'
           Name: 'Example SDS'
```

```
        Rank: 2
    DataType: 'int16'
  Attributes: []
        Dims: [2x1 struct]
       Label: {}
 Description: {}
       Index: 0
```

See Also

`hdfread`, `hdf`

hdfread

Purpose

Read data from HDF4 or HDF-EOS file

Syntax

```
data = hdfread(filename, datasetname)
data = hdfread(hinfo.fieldname)
data = hdfread(...,param1,value1,param2,value2,...)
[data,map] = hdfread(...)
```

Description

`data = hdfread(filename, datasetname)` returns all the data in the data set specified by `datasetname` from the HDF4 or HDF-EOS file specified by `filename`. To determine the name of a data set in an HDF4 file, use the `hdfinfo` function.

Note `hdfread` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To read data from and HDF5 file, use `hdf5read`.

`data = hdfread(hinfo.`*`fieldname`*`)` returns all the data in the data set specified by `hinfo.`*`fieldname`*, where `hinfo` is the structure returned by the `hdfinfo` function and *`fieldname`* is the name of a field in the structure that relates to a particular type of data set. For example, to read an HDF scientific data set, specify the `SDS` field, as in `hinfo.SDS`. To read HDF V data, specify the `Vdata` field, as in `hinfo.Vdata`. `hdfread` can get the name of the HDF file from these structures.

`data = hdfread(...,param1,value1,param2,value2,...)` returns subsets of the data according to the specified parameter and value pairs. See the tables below to find the valid parameters and values for different types of data sets.

`[data,map] = hdfread(...)` returns the image `data` and the colormap `map` for an 8-bit raster image.

Subsetting Parameters

The following tables show the subsetting parameters that can be used with the `hdfread` function for certain types of HDF4 data. These data types are

- HDF Scientific Data (SD)
- HDF Vdata (V)
- HDF-EOS Grid Data
- HDF-EOS Point Data
- HDF-EOS Swath Data

Note the following:

- If a parameter requires multiple values, the values must be stored in a cell array. For example, the `'Index'` parameter requires three values: `start`, `stride`, and `edge`. Enclose these values in curly braces as a cell array.

  ```
  hdfread(dataset_name, 'Index', {start,stride,edge})
  ```

- All values that are indices are 1-based.

Subsetting Parameters for HDF Scientific Data (SD) Data Sets

When you are working with HDF SD files, `hdfread` supports the parameters listed in this table.

Parameter	Description
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set • `start` — A 1-based array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values specified must not exceed the size of any dimension of the data set. • `stride` — A 1-based array specifying the interval between the values to read Default: `1`, read every element of the data set. • `edge` — A 1-based array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions

For example, this code reads the data set `Example SDS` from the HDF file `example.hdf`. The `'Index'` parameter specifies that `hdfread` start reading data at the beginning of each dimension, read until the end of each dimension, but only read every other data value in the first dimension.

```
data = hdfread('example.hdf','Example SDS','Index',{[], [2 1], []})
```

Subsetting Parameters for HDF Vdata Sets

When you are working with HDF Vdata files, `hdfread` supports these parameters.

Parameter	Description
`'Fields'`	Text string specifying the name of the field to be read. When specifying multiple field names, use a cell array.

Parameter	Description
`'FirstRecord'`	1-based number specifying the record from which to begin reading
`'NumRecords'`	Number specifying the total number of records to read

For example, this code reads the Vdata set `Example Vdata` from the HDF file `example.hdf`.

```
data = hdfread('example.hdf','Example Vdata','FirstRecord', 2,'NumRecords', 5)
```

Subsetting Parameters for HDF-EOS Grid Data

When you are working with HDF-EOS grid data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive parameters — You can only specify one of these parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
`'Fields'`	String specifying the field to be read. You can specify only one field name for a Grid data set.
Mutually Exclusive Optional Parameters	

Parameter	Description
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set `start` — An array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. `stride` — An array specifying the interval between the values to read Default: `1`, read every element of the data set. `edge` — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions
`'Interpolate'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude points that define a region for bilinear interpolation. Each element is an N-length vector specifying longitude and latitude coordinates.
`'Pixels'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. Each element is an N-length vector specifying longitude and latitude coordinates. This region is converted into pixel rows and columns with the origin in the upper left corner of the grid. Note: This is the pixel equivalent of reading a `'Box'` region.
`'Tile'`	Vector specifying the coordinates of the tile to read, for HDF-EOS Grid files that support tiles
Optional Parameters	
`'Box'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are each two-element vectors specifying longitude and latitude coordinates.

Parameter	Description
`'Time'`	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and end-point for a period of time
`'Vertical'`	Two-element cell array, `{dimension, range}` `dimension` — String specifying the name of the data set field to be read from. You can specify only one field name for a Grid data set. `range` — Two-element array specifying the minimum and maximum range for the subset. If `dimension` is a dimension name, then `range` specifies the range of elements to extract. If `dimension` is a field name, then `range` specifies the range of values to extract. `'Vertical'` subsetting can be used alone or in conjunction with `'Box'` or `'Time'`. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to `hdfread`.

For example,

```
hdfread(grid_dataset, 'Fields', fieldname, 'Vertical', {dimension, [min, max]})
```

Subsetting Parameters for HDF-EOS Point Data

When you are working with HDF-EOS Point data, `hdfread` has two required parameters and three optional parameters.

Parameter	Description
Required Parameters	
`'Fields'`	String naming the data set field to be read. For multiple field names, use a comma-separated list.
`'Level'`	1-based number specifying which level to read from in an HDF-EOS Point data set
Optional Parameters	

Parameter	Description
`'Box'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are each two-element vectors specifying longitude and latitude coordinates.
`'RecordNumbers'`	Vector specifying the record numbers to read
`'Time'`	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and endpoint for a period of time

For example,

```
hdfread(point_dataset, 'Fields', {field1, field2},...
                       'Level', level, 'RecordNumbers', [1:50, 200:250])
```

Subsetting Parameters for HDF-EOS Swath Data

When you are working with HDF-EOS Swath data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive

You can only use one of the mutually exclusive parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
`'Fields'`	String naming the data set field to be read. You can specify only one field name for a Swath data set.
Mutually Exclusive Optional Parameters	

Parameter	Description
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set • `start` — An array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. • `stride` — An array specifying the interval between the values to read Default: `1`, read every element of the data set. • `edge` — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions
`'Time'`	Three-element cell array, `{start, stop, mode}`, where `start` and `stop` specify the beginning and the endpoint for a period of time, and `mode` is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (`mode='midpoint'`). • Either endpoint is within the box (`mode='endpoint'`). • Any point is within the box (`mode='anypoint'`).
Optional Parameters	

Parameter	Description
`'Box'`	Three-element cell array, `{longitude, latitude, mode}` specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are two-element vectors that specify longitude and latitude coordinates. `mode` is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (`mode='midpoint'`). • Either endpoint is within the box (`mode='endpoint'`). • Any point is within the box (`mode='anypoint'`).
`'ExtMode'`	String specifying whether geolocation fields and data fields must be in the same swath (`mode='internal'`), or can be in different swaths (`mode='external'`) Note: `mode` is only used when extracting a time period or a region.
`'Vertical'`	Two-element cell array, `{dimension, range}` • `dimension` is a string specifying either a dimension name or field name to subset the data by. • `range` is a two-element vector specifying the minimum and maximum range for the subset. If `dimension` is a dimension name, then `range` specifies the range of elements to extract. If `dimension` is a field name, then `range` specifies the range of values to extract. `'Vertical'` subsetting can be used alone or in conjunction with `'Box'` or `'Time'`. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to `hdfread`.

For example,

```
hdfread('example.hdf', swath_dataset, 'Fields', fieldname, ...
```

```
'Time', {start, stop, 'midpoint'})
```

Examples

Example 1

Specify the name of the HDF file and the name of the data set. This example reads a data set named `'Example SDS'` from a sample HDF file.

```
data = hdfread('example.hdf', 'Example SDS')
```

Example 2

Use data returned by `hdfinfo` to specify the data set to read.

1 Call `hdfinfo` to retrieve information about the contents of the HDF file.

```
fileinfo = hdfinfo('example.hdf')
fileinfo =

    Filename: 'N:\toolbox\matlab\demos\example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

2 Extract the structure containing information about the particular data set you want to import from the data returned by `hdfinfo`. The example uses the structure in the `SDS` field to retrieve a scientific data set.

```
sds_info = fileinfo.SDS
sds_info =

       Filename: 'N:\toolbox\matlab\demos\example.hdf'
           Type: 'Scientific Data Set'
           Name: 'Example SDS'
           Rank: 2
       DataType: 'int16'
     Attributes: []
           Dims: [2x1 struct]
          Label: {}
```

```
    Description: {}
          Index: 0
```

3 You can pass this structure to `hdfread` to import the data in the data set.

```
data = hdfread(sds_info)
```

Example 3

You can use the information returned by `hdfinfo` to check the size of the data set.

```
sds_info.Dims.Size
ans =
    16
ans =
     5
```

Using the `'index'` parameter with `hdfread`, you can read a subset of the data in the data set. This example specifies a starting index of `[3 3]`, an interval of 1 between values (`[]` meaning the default value of 1), and a length of 10 rows and 2 columns.

```
data = hdfread(sds_info, 'Index', {[3 3],[],[10 2]});

data(:,1)
ans =
     7
     8
     9
    10
    11
    12
    13
    14
    15
    16
```

```
data(:,2)
ans =
      8
      9
     10
     11
     12
     13
     14
     15
     16
     17
```

Example 4

This example uses the `Vdata` field from the information returned by `hdfinfo` to read two fields of the data, `Idx` and `Temp`.

```
s = hdfinfo('example.hdf');
data1 = hdfread(s.Vdata, 'Fields', {'Idx', 'Temp', 'Dewpt'});
data1{1}

ans =

       1       2       3       4       5       6       7       8       9

data1{2}

ans =

       0      12       3       5      10      -1       3       0       2

data1{3}

ans =

       5       5       7      11       7      10       4      14       4
```

See Also `hdfinfo`, `hdf`

Purpose

Browse and import data from HDF4 or HDF-EOS files

Syntax

```
hdftool
hdftool(filename)
h = hdftool(...)
```

Description

`hdftool` starts the HDF Import Tool, a graphical user interface used to browse the contents of HDF4 and HDF-EOS files and import data and subsets of data from these files. To open an HDF4 or HDF-EOS file, select **Open** from the **File** menu. You can open multiple files in the HDF Import Tool by selecting **Open** from the **File** menu.

`hdftool(filename)` opens the HDF4 or HDF-EOS file specified by `filename` in the HDF Import Tool.

`h = hdftool(...)` returns a handle `h` to the HDF Import Tool. To close the tool from the command line, use `close(h)`.

Example

```
hdftool('example.hdf');
```

See Also

`hdf`, `hdfinfo`, `hdfread`, `uiimport`

help

Purpose

Help for functions in Command Window

GUI Alternatives

Use the Function Browser by clicking its button, *fx*, or run `doc functionname` to view more extensive help for a function in the Help browser.

Syntax

```
help
help /
help functionname
help modelname.mdl
help methodname
help classname
help packagename
help classname.name
help packagename.classname.name
help toolboxname
help syntax
t = help('topic')
```

Description

`help` lists all primary help topics in the Command Window. Each main help topic corresponds to a folder name on the search path the MATLAB software uses.

`help /` lists all operators and special characters, along with their descriptions.

`help functionname` displays a brief description and the syntax for `functionname` in the Command Window. It is called M-file help because the help information is contained at the top of the M-file. For more information or related help, use the links in the help output. If `functionname` is overloaded, that is, appears in multiple folders on the search path, `help` displays the M-file help for the first `functionname` found on the search path, and displays a hyperlinked list of the overloaded functions and their folders. If `functionname` is also the name of a toolbox, `help` also displays a list of subfolders and hyperlinked list of functions in the toolbox, as defined in the `Contents.m` file for the toolbox.

`help modelname.mdl` displays the complete description for the MDL-file `modelname` as defined in **Model Properties > Description**. If the Simulink product is installed, you do not need to specify the `.mdl` extension.

`help methodname` displays help for the method `methodname`. You may need to qualify `methodname` with its class.

`help classname` displays help for the class `classname`. You may need to qualify `classname` with its package.

`help packagename` displays help for the package `packagename`.

`help classname.name` displays help for the method, property, or event `name` in `classname`. You may need to qualify `classname` with its package.

`help packagename.classname.name` displays help for the method, property, or event `name` in `classname`, which is part of `packagename`. If you do not know the `packangename`, create an instance of `classname` and then run `class(obj)`.

`help toolboxname` displays the `Contents.m` file for the specified folder named `toolboxname`, where `Contents.m` contains a list and corresponding description of M-files in `toolboxname`. `toolboxname` can be a partial path. If `toolboxname` is also a function name, help also displays the M-file help for the function.

`help` **`syntax`** displays M-file help describing the syntax used in MATLAB functions.

`t = help('topic')` returns the help text for `topic` as a string, with each line separated by `/n`, where `topic` is any allowable argument for `help`.

Note Some M-file help displayed in the Command Window uses all uppercase characters for the names to make them stand out from the rest of the text. When typing these names, use lowercase characters.

Some names use mixed case. The M-file help accurately reflects that. Use mixed case when typing these names. For example, the `javaObject` function uses mixed case.

Remarks

Prevent Scrolling of Long Help Pages

To prevent long descriptions from scrolling off the screen before you have time to read them, enter `more on`, and then enter the `help` statement.

Examples

`help close` displays help for the `close` function. It lists

`help database.close` displays help for the `close` function in the Database Toolbox™ product.

`help throwAsCaller` displays help for the `MException.throwAsCaller` method.

`help MException` displays help for the `MException` class.

`help MException.cause` displays help for the `cause` property of the `MException` class.

`help containers` displays help for the `containers` package.

`help containers.Map` displays help for the `Map` class in the `containers` package. Note that `help Map` does not provide help for the `Map` class, so include the packagename in the syntax.

`help containers.Map.KeyType` displays help for the `KeyType` property. Note that `help Map.KeyType` do not provide help for `KeyType` , so include the `packagename` in the syntax.

`help datafeed` displays help for the Datafeed Toolbox™ product.

`help database` lists the functions in the Database Toolbox product and displays help for the `database` function, because there is both a function and a toolbox called `database`.

`help general` lists all functions in the folder *`matlabroot`*`/toolbox/matlab/general`. This illustrates how to specify a partial path name rather than a full path name.

`help f14_dap` displays the description of the Simulink `f14_dap.mdl` model file (the Simulink product must be installed).

`t = help('close')` gets help for the function `close` and stores it as a string in `t`.

See Also

`class`, `dbtype`,`doc`, `docsearch`, `helpbrowser`, `helpwin`, `lookfor`, `more`, `path`, `what`, `which`, `whos`

Related topics in the MATLAB Desktop Tools and Development Environment documentation:

-
-
-

helpbrowser

Purpose

Open Help browser to access online documentation and demos

GUI Alternatives

As an alternative to the `helpbrowser` function, select **Desktop > Help** or click the Help button on the toolbar in the MATLAB desktop.

Syntax

`helpbrowser`

Description

`helpbrowser` displays the Help browser, open to its default startup page, providing direct access to a comprehensive library of demos and online documentation, including reference pages and user guides.

See Also

`demo`, `doc`, `docsearch`, `help`, `helpwin`, `web`

Purpose Open Help browser

Syntax `helpdesk`

Description `helpdesk` opens the Help browser to the default startup page. In previous releases, `helpdesk` displayed the Help Desk, which was the precursor to the Help browser. In a future release, the `helpdesk` function will be phased out — use the `doc` or `helpbrowser` function instead.

See Also `doc`, `helpbrowser`

Purpose

Create and open help dialog box

Syntax

```
helpdlg
helpdlg('helpstring')
helpdlg('helpstring','dlgname')
h = helpdlg(...)
```

Description

`helpdlg` creates a nonmodal help dialog box or brings the named help dialog box to the front.

Note A nonmodal dialog box enables the user to interact with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`helpdlg` displays a dialog box named `'Help Dialog'` containing the string `'This is the default help string.'`

`helpdlg('helpstring')` displays a dialog box named `'Help Dialog'` containing the string specified by `'helpstring'`.

`helpdlg('helpstring','dlgname')` displays a dialog box named `'dlgname'` containing the string `'helpstring'`.

`h = helpdlg(...)` returns the handle of the dialog box.

Remarks

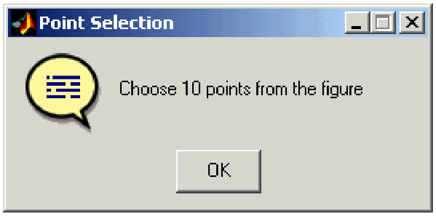
MATLAB wraps the text in `'helpstring'` to fit the width of the dialog box. The dialog box remains on your screen until you press the **OK** button or the **Enter** key. After either of these actions, the help dialog box disappears.

Examples

The statement

```
helpdlg('Choose 10 points from the figure','Point Selection');
```

displays this dialog box:



Choose 10 points from the figure

See Also

`dialog`, `errordlg`, `inputdlg`, `listdlg`, `msgbox`, `questdlg`, `warndlg`

`figure`, `uiwait`, `uiresume`

for related functions

helpwin

Purpose Provide access to M-file help for all functions

Syntax

```
helpwin
helpwin topic
```

Description `helpwin` lists topics for groups of functions in the Help browser. It shows brief descriptions of the topics and provides links to display M-file help for the functions in the Help browser. You cannot follow links in the `helpwin` list of functions if the MATLAB software is busy (for example, running a program).

`helpwin topic` displays help information for the topic in the Help browser. If `topic` is a folder, it displays all functions in the folder. If `topic` is a function, `helpwin` displays M-file help for that function in the Help browser. From the page, you can access a list of folders (**Default Topics** link) as well as the reference page help for the function (**Go to online doc** link). You cannot follow links in the `helpwin` list of functions if MATLAB is busy (for example, running a program).

Examples Typing

```
helpwin datafun
```

displays the functions in the `datafun` folder and a brief description of each.

Typing

```
helpwin fft
```

displays the M-file help for the `fft` function in the Help browser.

See Also `doc`, `help`, `helpbrowser`, `web`

Purpose

Hessenberg form of matrix

Syntax

```
H = hess(A)
[P,H] = hess(A)
[AA,BB,Q,Z] = hess(A,B)
```

Description

`H = hess(A)` finds `H`, the Hessenberg form of matrix `A`.

`[P,H] = hess(A)` produces a Hessenberg matrix `H` and a unitary matrix `P` so that `A = P*H*P'` and `P'*P = eye(size(A))` .

`[AA,BB,Q,Z] = hess(A,B)` for square matrices `A` and `B`, produces an upper Hessenberg matrix `AA`, an upper triangular matrix `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA` and `Q*B*Z = BB`.

Definition

A Hessenberg matrix is zero below the first subdiagonal. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original, but less computation is needed to reveal them.

Examples

`H` is a 3-by-3 eigenvalue test matrix:

```
H =
    -149     -50    -154
     537     180     546
     -27      -9     -25
```

Its Hessenberg form introduces a single zero in the (3,1) position:

```
hess(H) =
    -149.0000     42.2037   -156.3165
    -537.6783    152.5511   -554.9272
            0      0.0728      2.4489
```

Algorithm

Inputs of Type Double

For inputs of type double, `hess` uses the following LAPACK routines to compute the Hessenberg form of a matrix:

Matrix A	Routine
Real symmetric	`DSYTRD` `DSYTRD`, `DORGTR`, (with output `P`)
Real nonsymmetric	`DGEHRD` `DGEHRD`, `DORGHR` (with output `P`)
Complex Hermitian	`ZHETRD` `ZHETRD`, `ZUNGTR` (with output `P`)
Complex non-Hermitian	`ZGEHRD` `ZGEHRD`, `ZUNGHR` (with output `P`)

Inputs of Type Single

For inputs of type `single`, `hess` uses the following LAPACK routines to compute the Hessenberg form of a matrix:

Matrix A	Routine
Real symmetric	`SSYTRD` `SSYTRD`, `DORGTR`, (with output `P`)
Real nonsymmetric	`SGEHRD` `SGEHRD`, `SORGHR` (with output `P`)
Complex Hermitian	`CHETRD` `CHETRD`, `CUNGTR` (with output `P`)
Complex non-Hermitian	`CGEHRD` `CGEHRD`, `CUNGHR` (with output `P`)

See Also

`eig`, `qz`, `schur`

References

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling,

A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose Convert hexadecimal number string to decimal number

Syntax `d = hex2dec('*hex_value*')`

Description `d = hex2dec('hex_value')` converts *`hex_value`* to its floating-point integer representation. The argument *`hex_value`* is a hexadecimal integer stored in a MATLAB string. The value of *`hex_value`* must be smaller than hexadecimal `10,000,000,000,000`.

If *`hex_value`* is a character array, each row is interpreted as a hexadecimal string.

Examples

```
hex2dec('3ff')

ans =

        1023
```

For a character array `S`,

```
S =
0FF
2DE
123
hex2dec(S)

ans =

255
734
291
```

See Also `dec2hex`, `format`, `hex2num`, `sprintf`

Purpose

Convert hexadecimal number string to double-precision number

Syntax

```
n = hex2num(S)
```

Description

`n = hex2num(S)`, where `S` is a 16 character string representing a hexadecimal number, returns the IEEE double-precision floating-point number `n` that it represents. Fewer than 16 characters are padded on the right with zeros. If `S` is a character array, each row is interpreted as a double-precision number.

`NaN`s, infinities and denorms are handled correctly.

Example

```
hex2num('400921fb54442d18')
```

returns `Pi`.

```
hex2num('bff')
```

returns

```
ans =

    -1
```

See Also

`num2hex`, `hex2dec`, `sprintf`, `format`

hgexport

Purpose

Export figure

GUI Alternative

Use the **File —> Saveas** on the figure window menu to access the **Export Setup** GUI. Use **Edit —> Copy Figure** to copy the figure's contents to your system's clipboard. For details, see How to Print or Export in the MATLAB Graphics documentation.

Syntax

```
hgexport(h,filename)
hgexport(h,'-clipboard')
```

Description

`hgexport(h,filename)` writes figure `h` to the file `filename`.

`hgexport(h,'-clipboard')` writes figure `h` to the Microsoft Windows clipboard.

The format in which the figure is exported is determined by which renderer you use. The Painters renderer generates a metafile. The ZBuffer and OpenGL renderers generate a bitmap.

See Also

`print`

Purpose

Create hggroup object

Syntax

Description

An hggroup object can be the parent of any axes children except light objects, as well as other hggroup objects. You can use hggroup objects to form a group of objects that can be treated as a single object with respect to the following cases:

- Visible — Setting the hggroup object's `Visible` property also sets each child object's `Visible` property to the same value.
- Selectable — Setting each hggroup child object's `HitTest` property to `off` enables you to select all children by clicking any child object.
- Current object — Setting each hggroup child object's `HitTest` property to `off` enables the hggroup object to become the current object when any child object is picked. See the next section for an example.

Examples

This example defines a callback for the `ButtonDownFcn` property of an hggroup object. In order for the hggroup to receive the mouse button down event that executes the `ButtonDownFcn` callback, the `HitTest` properties of all the line objects must be set to `off`. The event is then passed up the hierarchy to the hggroup.

The following function creates a random set of lines that are parented to an hggroup object. The subfunction `set_lines` defines a callback that executes when the mouse button is pressed over any of the lines. The callback simply increases the widths of all the lines by 1 with each button press.

Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

```
function doc_hggroup
```

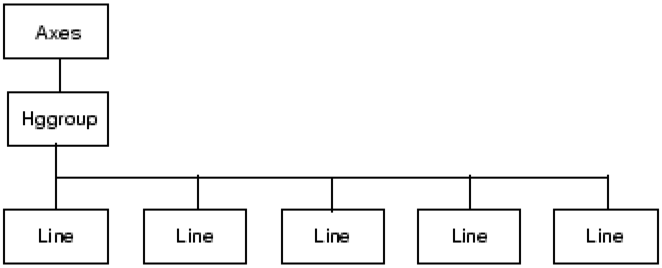
```
hg = hggroup('ButtonDownFcn',@set_lines);
hl  = line(randn(5),randn(5),'HitTest','off','Parent',hg);

function set_lines(cb,eventdata)
hl = get(cb,'Children');% cb is handle of hggroup object
lw = get(hl,'LineWidth');% get current line widths
set(hl,{'LineWidth'},num2cell([lw{:}]+1,[5,1])')
```

Note that selecting any one of the lines selects all the lines. (To select an object, enable plot edit mode by selecting **Plot Edit** from the **Tools** menu.)

Instance Diagram for This Example

The following diagram shows the object hierarchy created by this example.



Hggroup Properties

Setting Default Properties

You can set default hggroup properties on the axes, figure, and root levels.

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

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

See Also

`hgtransform`

for more information and examples.

for information on how to use function handles to define callbacks.

`Hggroup Properties` for property descriptions

Hggroup Properties

Purpose

Hggroup properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands.

To change the default values of properties, see .

See for general information on this type of object.

Hggroup Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of hggroup objects in legends. The `Annotation` property enables you to specify whether this hggroup 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 hggroup object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the hggroup object in a legend as one entry, but not its children objects
`off`	Do not include the hggroup or its children in a legend (default)
`children`	Include only the children of the hggroup 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 whether objects are in the process of being deleted. The MATLAB software sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the `delete` function executes, after which the object no longer exists.

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

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

`ButtonDownFcn`

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

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is over the children of the hggroup object. Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with the mouse button press and an empty event structure).

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

`Children`

array of graphics object handles

Children of the hggroup object. An array containing the handles of all objects parented to the hggroup object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not appear in the hggroup

`Children` property unless you set the Root `ShowHiddenHandles` property to `on`:

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

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips stairs plots to the axes plot box by default. If you set `Clipping` to `off`, lines might be displayed outside the axes plot box.

`CreateFcn`

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

Callback executed during object creation. This property defines a callback function that executes when MATLAB creates an hggroup object. You must define this property as a default value for hggroup objects or in a call to the `hggroup` function to create a new hggroup object. For example, the statement

```
set(0,'DefaulthggroupCreateFcn',@myCreateFcn)
```

defines a default value on the root level that applies to every hggroup object created in that MATLAB session. Whenever you create an hggroup object, the function associated with the function handle `@myCreateFcn` executes.

MATLAB executes the callback after setting all the hggroup object's properties. Setting the `CreateFcn` property on an existing hggroup object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

`DeleteFcn`

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

Callback executed during object deletion. A callback function that executes when the hggroup object is deleted (e.g., this might happen when you issue a `delete` command on the hggroup object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See 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 hggroup object. The `legend` function uses the string defined by the `DisplayName` property to label this hggroup object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this hggroup 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 hggroup child objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

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

if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.

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

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

Printing with Nonnormal Erase Modes

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

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

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the hggroup object.

- `on` — Handles are always visible when `HandleVisibility` is `on`.

- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

`HitTest`

`{on} | off`

Pickable by mouse click. `HitTest` determines whether the hggroup 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 hggroup child objects. Note that to pick the hggroup object, its children must have their `HitTest` property set to `off`.

If the hggroup object's `HitTest` is `off`, clicking it picks the object behind it.

`Interruptible`

`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an hggroup object callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from an hggroup property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`Parent`

axes handle

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

See for more information on parenting graphics objects.

`Selected`

`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints of hggroup child objects if the `SelectionHighlight` property is also `on` (the default).

`SelectionHighlight`

`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing selection handles on the hggroup child objects. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`

string

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

For example, you might create an hggroup object and set the `Tag` property:

```
t = hggroup('Tag','group1')
```

When you want to access the object, you can use `findobj` to find its handle. For example,

```
h = findobj('Tag','group1');
```

`Type`

string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For hggroup 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 the hggroup object. Assign this property the handle of a uicontextmenu object created in the hggroup object's figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click the hggroup object.

`UserData`

array

User-specified data. This property can be any data you want to associate with the hggroup object (including cell arrays and structures). The hggroup object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`

`{on} | off`

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

Purpose

Load Handle Graphics object hierarchy from file

GUI Alternative

Use the **File —> Open** on the figure window menu to access figure files with the **Open** dialog.

Syntax

```
h = hgload('filename')
[h,old_prop_values] = hgload(...,property_structure)
hgload(...,'all')
```

Description

`h = hgload('filename')` loads Handle Graphics objects and its children (if any) from the FIG-file specified by `filename` and returns handles to the top-level objects. If `filename` contains no extension, then the MATLAB software adds the `.fig` extension.

`[h,old_prop_values] = hgload(...,property_structure)` overrides the properties on the top-level objects stored in the FIG-file with the values in `property_structure`, and returns their previous values in `old_prop_values`.

`property_structure` must be a structure having field names that correspond to property names and values that are the new property values.

`old_prop_values` is a cell array equal in length to `h`, containing the old values of the overridden properties for each object. Each cell contains a structure having field names that are property names, each of which contains the original value of each property that has been changed. Any property specified in `property_structure` that is not a property of a top-level object in the FIG-file is not included in `old_prop_values`.

`hgload(...,'all')` overrides the default behavior, which does not reload nonserializable objects saved in the file. These objects include the default toolbars and default menus.

Nonserializable objects (such as the default toolbars and the default menus) are normally not reloaded because they are loaded from different files at figure creation time. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files.

Passing the string `all` to `hgload` ensures that any nonserializable objects contained in the file are also reloaded.

Note that, by default, `hgsave` excludes nonserializable objects from the FIG-file unless you use the `all` flag.

See Also

`hgsave`, `open`

“Figure Windows” on page 1-100 for related functions

Purpose

Save Handle Graphics object hierarchy to file

GUI Alternative

Use the **File —> Saveas** on the figure window menu to access the Export Setup GUI. For details, see How to Print or Export in the MATLAB Graphics documentation.

Syntax

```
hgsave('filename')
hgsave(h,'filename')
hgsave(...,'all')
hgsave(...,'-v6')
hgsave(...,'-v7.3')
```

Description

`hgsave('filename')` saves the current figure to a file named `filename`.

`hgsave(h,'filename')` saves the objects identified by the array of handles `h` to a file named `filename`. If you do not specify an extension for `filename`, then the extension `.fig` is appended. If `h` is a vector, none of the handles in `h` may be ancestors or descendents of any other handles in `h`.

`hgsave(...,'all')` overrides the default behavior, which does not save nonserializable objects. Nonserializable objects include the default toolbars and default menus. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files and also reduces the size of FIG-files. Passing the string `all` to `hgsave` ensures that nonserializable objects are also saved.

Note: the default behavior of `hgload` is to ignore nonserializable objects in the file at load time. This behavior can be overwritten using the `all` argument with `hgload`.

`hgsave(...,'-v6')` saves the FIG-file in a format that can be loaded by versions prior to MATLAB 7.

`hgsave(...,'-v7.3')` saves the FIG-file in a format that can be loaded only by MATLAB versions 7.3 and above. This format, based on HDF5 files, is intended for saving FIG-files larger than 2 GB.

You can make `-v6` or `-v7.3` your default format for saving MAT-files and FIG-files by setting a preference, which will eliminate the need to specify the flag each time you save. See in the MATLAB Desktop Tools and Development Environment documentation.

Full Backward Compatibility

When creating a figure you want to save and use in a MATLAB version prior to MATLAB 7, use the `'v6'` option with the plotting function and the `'-v6'` option for `hgsave`. Check the reference page for the plotting function you are using for more information.

See for more information.

See Also

`hgload`, `open`, `save`

“Figure Windows” on page 1-100 for related functions

Purpose

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

Syntax

`classdef *myclass* < hgsetget`

Description

`classdef` *`myclass`* `< hgsetget` makes *`myclass`* a subclass of the `hgsetget` class, which is a subclass of the `handle` class.

Use the `hgsetget` class to derive classes that inherit `set` and `get` methods that behave like Handle Graphics `set` and `get` functions.

hgsetget Class Methods

When you derive a class from the `hgsetget` class, your class inherits the following methods.

Method	Purpose
`set`	Assigns values to the specified properties or returns a cell array of possible values for writable properties.
`get`	Returns value of specified property or a `struct` with all property values.
`setdisp`	Called when `set` is called with no output arguments and a handle array, but no property name. Override this method to change what set displays.
`getdisp`	Called when `get` is called with no output arguments and handle array, but no property name. Override this method to change what get displays.

See Also

See

`handle`, `set (hgsetget)` , `get (hgsetget)`, `set`, `get`

Purpose Create hgtransform graphics object

Syntax

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

Description

`h = hgtransform` creates an hgtransform object and returns its handle.

`h = hgtransform('PropertyName',propertyvalue,...)` creates an hgtransform object with the property value settings specified in the argument list.

hgtransform objects can contain other objects, which lets you treat the hgtransform and its children as a single entity with respect to visibility, size, orientation, etc. You can group objects by parenting them to a single hgtransform object (i.e., setting the object's `Parent` property to the hgtransform object's handle):

```
h = hgtransform;
surface('Parent',h,...)
```

The primary advantage of parenting objects to an hgtransform object is that you can perform *transforms* (e.g., translation, scaling, rotation, etc.) on the child objects in unison.

The parent of an hgtransform object is either an axes object or another hgtransform.

Although you cannot see an hgtransform object, setting its `Visible` property to `off` makes all its children invisible as well.

Exceptions and Limitations

- An hgtransform object can be the parent of any number of axes child objects belonging to the same axes, except for light objects.
- hgtransform objects can never be the parent of axes objects and therefore can contain objects only from a single axes.
- hgtransform objects can be the parent of other hgtransform objects within the same axes.

- You cannot transform image objects because images are not true 3-D objects. Texture mapping the image data to a surface `CData` enables you to produce the effect of transforming an image in 3-D space.

Note Many plotting functions clear the axes (i.e., remove axes children) before drawing the graph. Clearing the axes also deletes any hgtransform objects in the axes.

More Information

- References in "See Also" on page 2-1707 provide information on types of transforms
- "Examples" on page 2-1697 provide examples that illustrate the use of transforms.

Examples

Transforming a Group of Objects

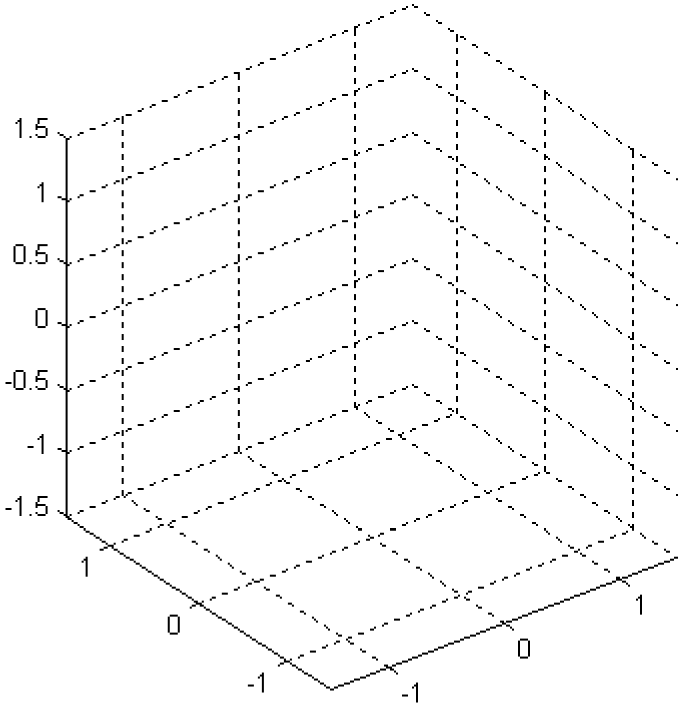
This example shows how to create a 3-D star with a group of surface objects parented to a single hgtransform object. The hgtransform then rotates the object about the *z*-axis while scaling its size.

Tip If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB Editor.

---

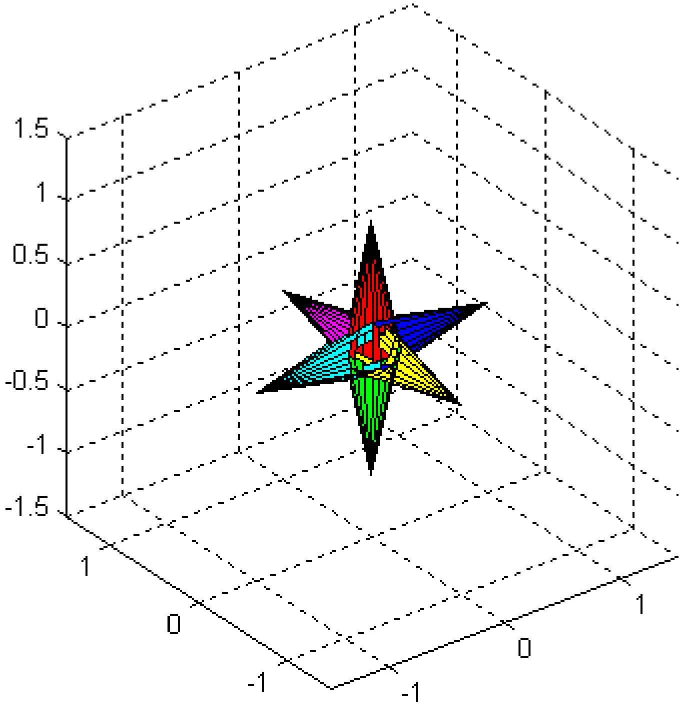
1. Create an axes and adjust the view. Set the axes limits to prevent auto limit selection during scaling.

```
ax = axes('XLim',[-1.5 1.5],'YLim',[-1.5 1.5],...
 'ZLim',[-1.5 1.5]);
view(3); grid on; axis equal
```



2 Create the objects you want to parent to the hgtransform object.

```
[x y z] = cylinder([.2 0]);
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```

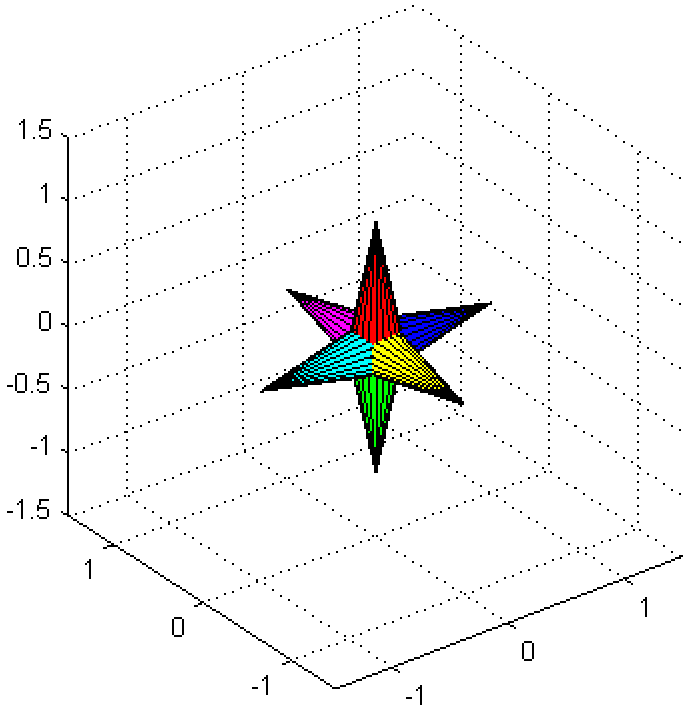


3 Create an hgtransform object and parent the surface objects to it. The figure should not change from the image above.

```
t = hgtransform('Parent',ax);
set(h,'Parent',t)
```

4 Select a renderer and show the objects.

```
set(gcf,'Renderer','opengl')
drawnow
```



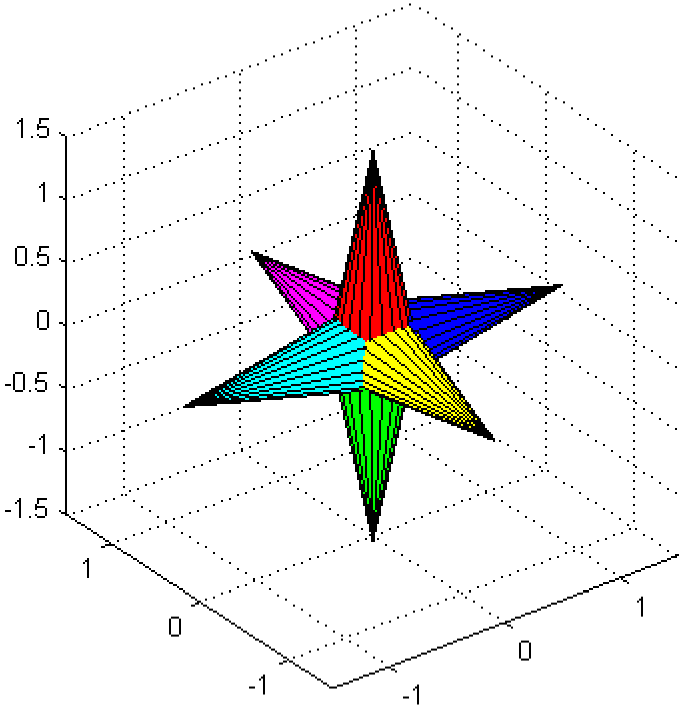
5 Initialize the rotation and scaling matrix to the identity matrix (`eye`). Again, the image should not change.

```
Rz = eye(4);
Sxy = Rz;
```

6 Form the z-axis rotation matrix and the scaling matrix. Rotate 360 degrees (2*pi radians) and scale by using the increasing values of `r`.

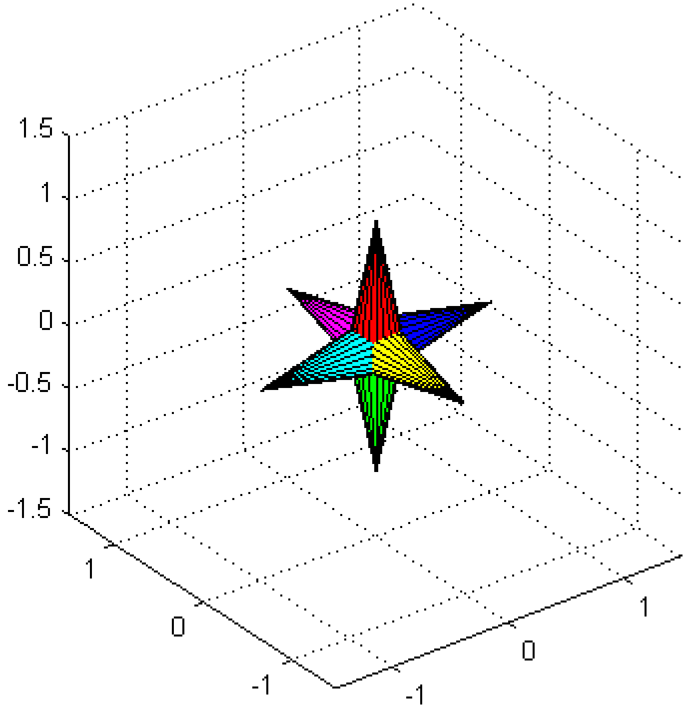
```
for r = 1:.1:2*pi
 % Z-axis rotation matrix
 Rz = makehgtform('zrotate',r);
 % Scaling matrix
 Sxy = makehgtform('scale',r/4);
```

```
   % Concatenate the transforms and
   % set the hgtransform Matrix property
      set(t,'Matrix',Rz*Sxy)
      drawnow
end
pause(1)
```



7 Reset to the original orientation and size using the identity matrix.

```
set(t,'Matrix',eye(4))
```



Transforming Objects Independently

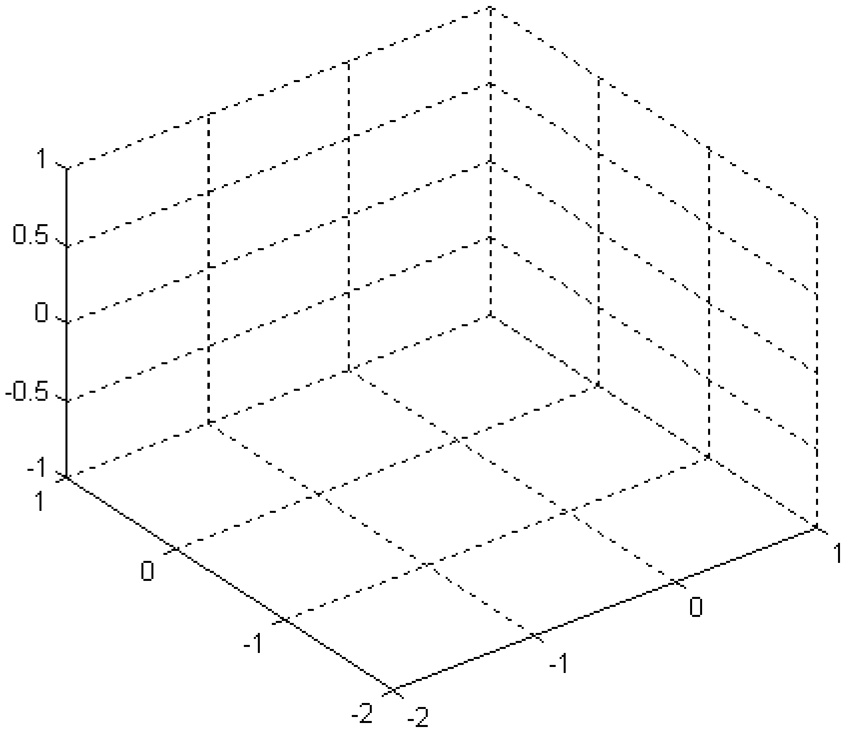
This example creates two hgtransform objects to illustrate how to transform each independently within the same axes. A translation transformation moves one hgtransform object away from the origin.

Tip If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB Editor.

---

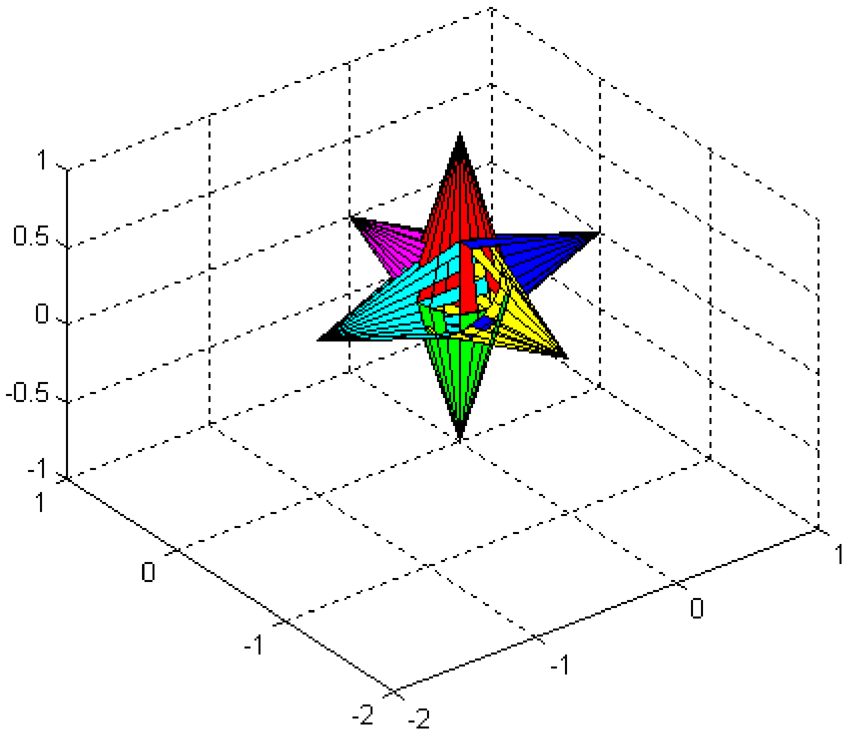
1. Create and set up the axes object that will be the parent of both hgtransform objects. Set the limits to accommodate the translated object.

```
ax = axes('XLim',[-2 1],'YLim',[-2 1],'ZLim',[-1 1]);
view(3); grid on; axis equal
```



2 Create the surface objects to group.

```
[x y z] = cylinder([.3 0]);
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```

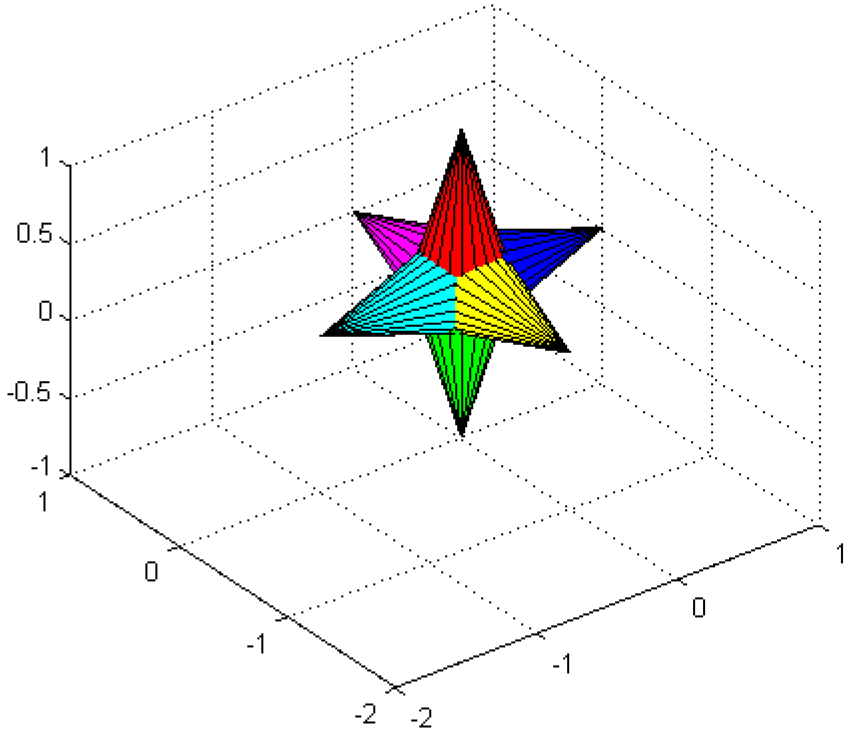


3 Create the hgtransform objects and parent them to the same axes. The figure should not change.

```
t1 = hgtransform('Parent',ax);
t2 = hgtransform('Parent',ax);
```

4 Set the renderer to use OpenGL.

```
set(gcf,'Renderer','opengl')
```

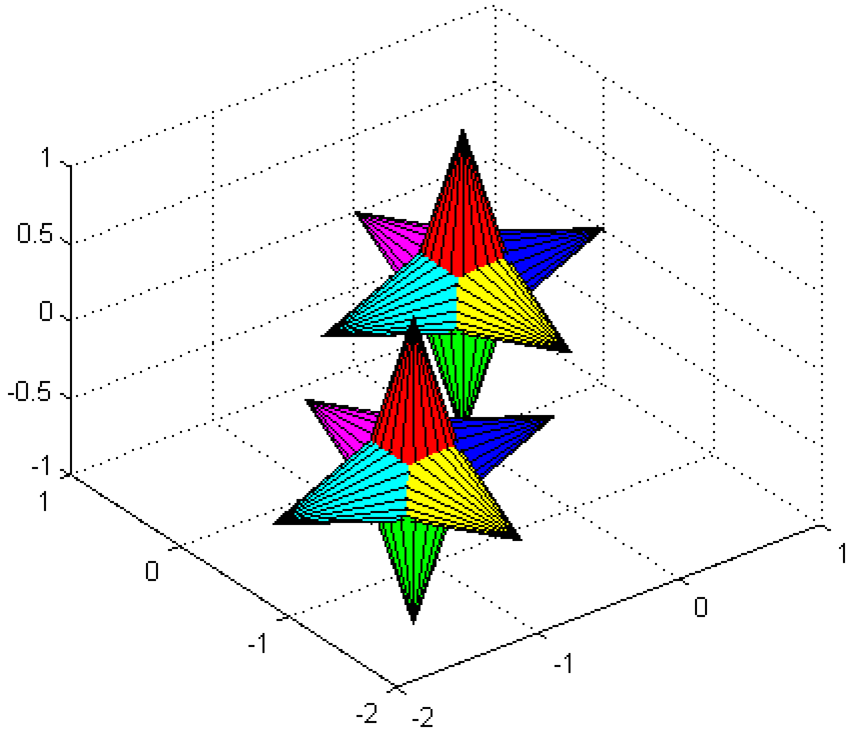


5 Parent the surfaces to hgtransform `t1`, then copy the surface objects and parent the copies to hgtransform `t2`. This figure should not change.

```
set(h,'Parent',t1)
h2 = copyobj(h,t2);
```

6 Translate the second hgtransform object away from the first hgtransform object and display the result.

```
Txy = makehgtform('translate',[-1.5 -1.5 0]);
set(t2,'Matrix',Txy)
drawnow
```



7 Rotate both hgtransform objects in opposite directions. The final image for this step is the same as for step 6. However, you should run the code to see the rotations.

```
% Rotate 10 times (2pi radians = 1 rotation)
for r = 1:.1:20*pi
 % Form z-axis rotation matrix
 Rz = makehgtform('zrotate',r);
 % Set transforms for both hgtransform objects
 set(t1,'Matrix',Rz)
 set(t2,'Matrix',Txy*inv(Rz))
 drawnow
end
```

Setting Default Properties

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

```
set(0,'DefaultHgtransformPropertyName',propertyvalue,...)
set(gcf,'DefaultHgtransformPropertyName',propertyvalue,...)
set(gca,'DefaultHgtransformPropertyName',propertyvalue,...)
```

`PropertyName` is the name of the hgtransform property and `propertyvalue` is the specified value. Use `set` and `get` to access hgtransform properties.

See Also

`hggroup`, `makehgtform`

Tomas Moller and Eric Haines, *Real-Time Rendering*, A K Peters, Ltd., 1999 for more information about transforms.

in MATLAB Graphics documentation for more information and examples.

`Hgtransform Properties` for property descriptions.

Hgtransform Properties

Purpose

Hgtransform properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands.

To change the default values of properties, see .

See for general information on this type of object.

Hgtransform Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

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

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

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

IconDisplayStyle Value	Purpose
`on`	Include the hgtransform object in a legend as one entry, but not its children objects
`off`	Do not include the hgtransform or its children in a legend (default)
`children`	Include only the children of the hgtransform as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

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

Using the IconDisplayStyle Property

See for more information and examples.

`BeingDeleted`

`on | {off}` Read Only

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

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

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback functions. If there is a callback executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

`ButtonDownFcn`

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

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is within the extent of the hgtransform object, but not over another graphics object. The extent of an hgtransform object is the smallest rectangle that encloses all the children. Note that you cannot execute the hgtransform object's button down function if it has no children.

Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with the mouse button press and an empty event structure).

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

`Children`

array of graphics object handles

Children of the hgtransform object. An array containing the handles of all graphics objects parented to the hgtransform object (whether visible or not).

The graphics objects that can be children of an hgtransform are images, lights, lines, patches, rectangles, surfaces, and text. You can change the order of the handles and thereby change the stacking of the objects on the display.

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in the hgtransform `Children` property unless you set the Root `ShowHiddenHandles` property to `on`.

`Clipping`

`{on} | off`

This property has no effect on hgtransform objects.

`CreateFcn`

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

Callback executed during object creation. This property defines a callback function that executes when MATLAB creates an hgtransform object. You must define this property as a default value for hgtransform objects. For example, the statement

```
set(0,'DefaultHgtransformCreateFcn',@myCreateFcn)
```

defines a default value on the root level that applies to every hgtransform object created in a MATLAB session. Whenever you create an hgtransform object, the function associated with the function handle `@myCreateFcn` executes.

MATLAB executes the callback after setting all the hgtransform object's properties. Setting the `CreateFcn` property on an existing hgtransform object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

`DeleteFcn`

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

Callback executed during object deletion. A callback function that executes when the hgtransform object is deleted (e.g., this might happen when you issue a `delete` command on the hgtransform object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which can be queried using `gcbo`.

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

See the `BeingDeleted` property for related information.

`DisplayName`

string (default is empty string)

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

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

See for more examples.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase hgtransform child objects (light objects have no erase mode). Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing

with `EraseMode` `none`, you cannot print these objects because MATLAB stores no information about their former locations.

- `xor`— Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

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

Printing with Nonnormal Erase Modes

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

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

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the hgtransform object.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

`HitTest`

`{on} | off`

Pickable by mouse click. `HitTest` determines whether the hgtransform object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click within the limits of the hgtransform object. If `HitTest` is `off`, clicking the hgtransform picks the object behind it.

`Interruptible`

`{on} | off`

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

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from an hgtransform property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`Matrix`

4-by-4 matrix

Transformation matrix applied to hgtransform object and its children. The hgtransform object applies the transformation matrix to all its children.

See for more information and examples.

`Parent`

figure handle

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

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 on all child objects of the hgtransform if the `SelectionHighlight` property is also `on` (the default).

`SelectionHighlight`

`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing selection handles on the objects parented to the hgtransform. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`

string

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

For example, you might create an hgtransform object and set the `Tag` property:

```
t = hgtransform('Tag','subgroup1')
```

When you want to access the hgtransform object to add another object, you can use `findobj` to find the hgtransform object's handle. The following statement adds a line to `subgroup1` (assuming `x` and `y` are defined).

```
line('XData',x,'YData',y,'Parent',findobj('Tag','subgroup1'))
```

`Type`
string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For hgtransform objects, `Type` is set to '`hgtransform`'. The following statement finds all the hgtransform objects in the current axes.

```
t = findobj(gca,'Type','hgtransform');
```

`UIContextMenu`
handle of a uicontextmenu object

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

`UserData`
array

User-specified data. This property can be any data you want to associate with the hgtransform object (including cell arrays and structures). The hgtransform object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`

`{on} | off`

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

hidden

Purpose

Remove hidden lines from mesh plot

Syntax

```
hidden on
hidden off
hidden
```

Description

Hidden line removal draws only those lines that are not obscured by other objects in the field of view.

`hidden on` turns on hidden line removal for the current graph so lines in the back of a mesh are hidden by those in front. This is the default behavior.

`hidden off` turns off hidden line removal for the current graph.

`hidden` toggles the hidden line removal state.

Algorithm

`hidden on` sets the `FaceColor` property of a surface graphics object to the background `Color` of the axes (or of the figure if axes `Color` is `none`).

Examples

Set hidden line removal `off` and `on` while displaying the `peaks` function.

```
mesh(peaks)
hidden off
hidden on
```

See Also

`shading`, `mesh`

The surface properties `FaceColor` and `EdgeColor`

"Surface and Mesh Creation" on page 1-102 for related functions

Purpose

Hilbert matrix

Syntax

```
H = hilb(n)
```

Description

`H = hilb(n)` returns the Hilbert matrix of order `n`.

Definition

The Hilbert matrix is a notable example of a poorly conditioned matrix [1]. The elements of the Hilbert matrices are $H(i, j) = 1/(i + j - 1)$.

Examples

Even the fourth-order Hilbert matrix shows signs of poor conditioning.

```
cond(hilb(4)) =
    1.5514e+04
```

See Also

`invhilb`

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

hist

Purpose

Histogram plot

GUI Alternatives

To graph selected variables, use the Plot Selector 

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

Syntax

```
n = hist(Y)
n = hist(Y,x)
n = hist(Y,nbins)
[n,xout] = hist(...)
hist(...)
hist(axes_handle,...)
```

Description

A histogram shows the distribution of data values.

`n = hist(Y)` bins the elements in vector `Y` into 10 equally spaced containers and returns the number of elements in each container as a row vector. If `Y` is an `m`-by-`p` matrix, `hist` treats the columns of `Y` as vectors and returns a 10-by-`p` matrix `n`. Each column of `n` contains the results for the corresponding column of `Y`. No elements of `Y` can be complex or of type `integer`.

`n = hist(Y,x)` where `x` is a vector, returns the distribution of `Y` among `length(x)` bins with centers specified by `x`. For example, if `x` is a 5-element vector, `hist` distributes the elements of `Y` into five bins centered on the *x*-axis at the elements in `x`, none of which can be complex. Note: use `histc` if it is more natural to specify bin edges instead of centers.

`n = hist(Y,nbins)` where `nbins` is a scalar, uses `nbins` number of bins.

`[n,xout] = hist(...)` returns vectors `n` and `xout` containing the frequency counts and the bin locations. You can use `bar(xout,n)` to plot the histogram.

`hist(...)` without output arguments produces a histogram plot of the output described above. `hist` distributes the bins along the *x*-axis between the minimum and maximum values of `Y`.

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

Remarks

All elements in vector `Y` or in one column of matrix `Y` are grouped according to their numeric range. Each group is shown as one bin. If `Y` is a matrix, `hist` scales the current colormap and uses each column number (`1:n`) to assign each a unique color.

The histogram's *x*-axis reflects the range of values in `Y`. The histogram's *y*-axis shows the number of elements that fall within the groups; therefore, the *y*-axis ranges from `0` to the greatest number of elements deposited in any bin. The *x*-range of the leftmost and rightmost bins extends to include the entire data range in the case when the user-specified range does not cover the data range; this often results in "boxes" at either or both edges of the distribution. If you want a plot in which this does not happen (that is, all bins have equal width), you can create a histogram-like display using the `bar` command.

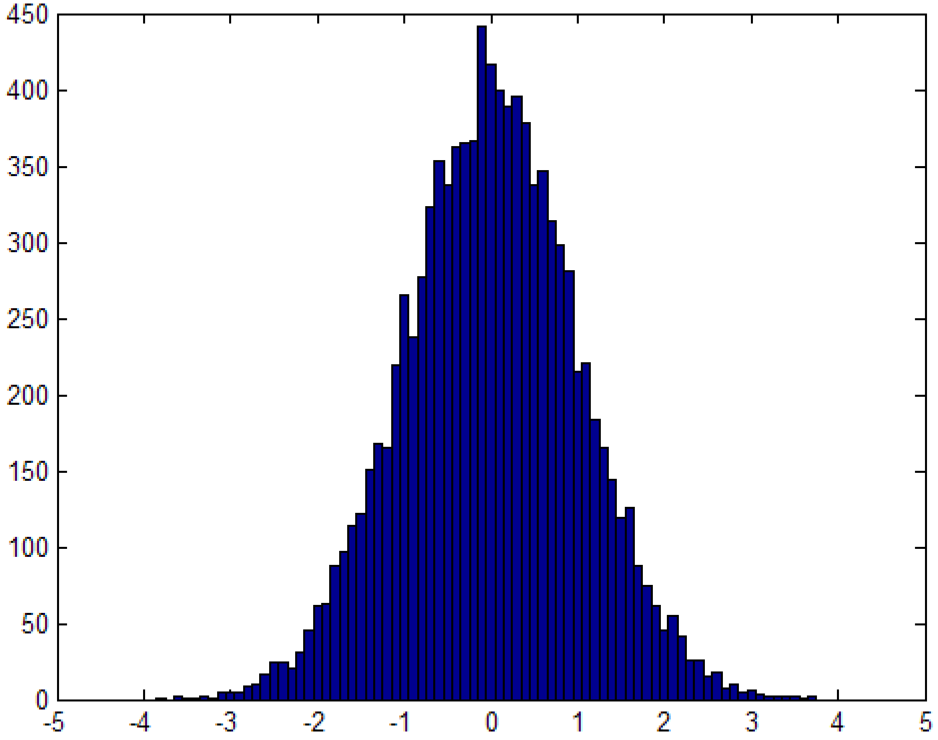
Histograms bins are created as patch objects and always plotted with a face color that maps to the first color in the current colormap (by default, blue) and with black edges. To change colors or other patch properties, use code similar to that given in the example.

The `hist` function does not accept data that contains `inf` values.

Example

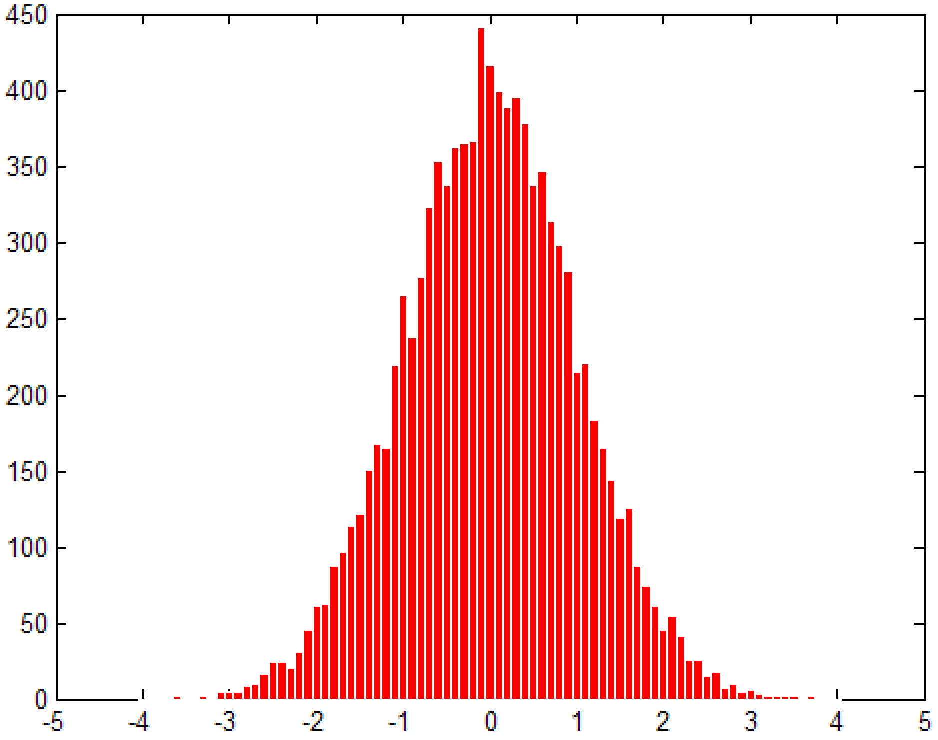
Generate a bell-curve histogram from Gaussian data.

```
x = -4:0.1:4;
y = randn(10000,1);
hist(y,x)
```



Change the color of the graph so that the bins are red and the edges of the bins are white.

```
h = findobj(gca,'Type','patch');
set(h,'FaceColor','r','EdgeColor','w')
```



See Also

`bar`, `ColorSpec`, `histc`, `mode`, `patch`, `rose`, `stairs`

"Specialized Plotting" on page 1-93 for related functions

for examples

Purpose

Histogram count

Syntax

```
n = histc(x,edges)
n = histc(x,edges,dim)
[n,bin] = histc(...)
```

Description

`n = histc(x,edges)` counts the number of values in vector `x` that fall between the elements in the `edges` vector (which must contain monotonically nondecreasing values). `n` is a `length(edges)` vector containing these counts. No elements of `x` can be complex.

`n(k)` counts the value `x(i)` if `edges(k) <= x(i) < edges(k+1)`. The last bin counts any values of `x` that match `edges(end)`. Values outside the values in `edges` are not counted. Use `-inf` and `inf` in `edges` to include all non-`NaN` values.

For matrices, `histc(x,edges)` returns a matrix of column histogram counts. For N-D arrays, `histc(x,edges)` operates along the first nonsingleton dimension.

`n = histc(x,edges,dim)` operates along the dimension `dim`.

`[n,bin] = histc(...)` also returns an index matrix `bin`. If `x` is a vector, `n(k) = sum(bin==k)`. `bin` is zero for out of range values. If `x` is an `M`-by-`N` matrix, then

```
for j=1:N,
n(k,j) = sum(bin(:,j)==k);
end
```

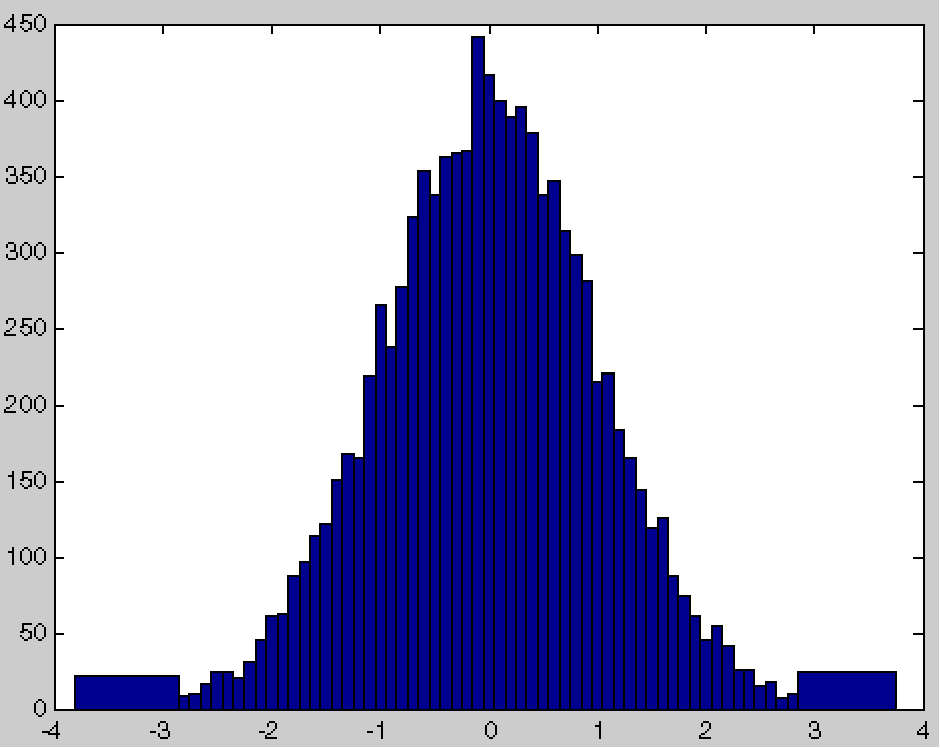
To plot the histogram, use the `bar` command.

Examples

Generate a cumulative histogram of a distribution.

Consider the following distribution:

```
x = -2.9:0.1:2.9;
y = randn(10000,1);
figure(1), hist(y,x)
```



Calculate number of elements in each bin

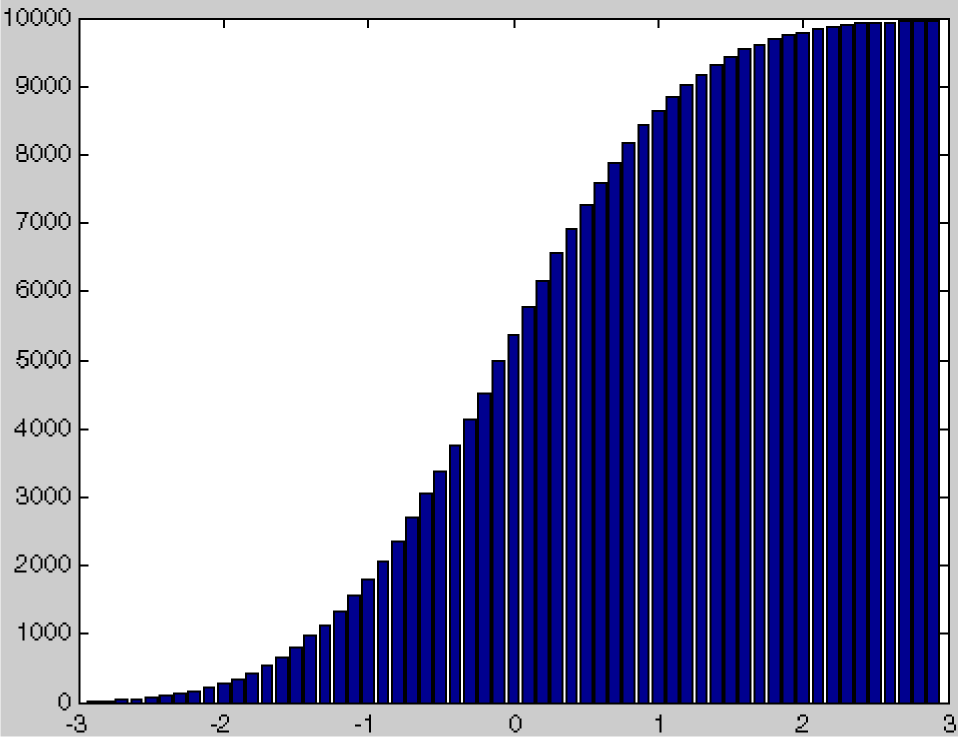
```
n_elements = histc(y,x);
```

Calculate the cumulative sum of these elements using `cumsum`

```
c_elements = cumsum(n_elements)
```

Plot the cumulative histogram

```
figure(2),bar(x,c_elements)
```



See Also `hist`, `mode`

"Specialized Plotting" on page 1-93 for related functions

Purpose

Retain current graph in figure

Syntax

```
hold on
hold off
hold all
hold
hold(axes_handle,...)
```

Description

The `hold` function determines whether new graphics objects are added to the graph or replace objects in the graph.

`hold on` retains the current plot and certain axes properties so that subsequent graphing commands add to the existing graph.

`hold off` resets axes properties to their defaults before drawing new plots. `hold off` is the default.

`hold all` holds the plot and the current line color and line style so that subsequent plotting commands do not reset the `ColorOrder` and `ColorOrder` property values to the beginning of the list. Plotting commands continue cycling through the predefined colors and linestyles from where the last plot stopped in the list.

`hold` toggles the hold state between adding to the graph and replacing the graph.

`hold(axes_handle,...)` applies the hold to the axes identified by the handle `axes_handle`.

Remarks

Test the hold state using the `ishold` function.

Although the hold state is `on`, some axes properties change to accommodate additional graphics objects. For example, the axes' limits increase when the data requires them to do so.

The `hold` function sets the `NextPlot` property of the current figure and the current axes. If several axes objects exist in a figure window, each axes has its own hold state. `hold` also creates an axes if one does not exist.

`hold on` sets the `NextPlot` property of the current figure and axes to `add`.

`hold off` sets the `NextPlot` property of the current axes to `replace`.

`hold` toggles the `NextPlot` property between the `add` and `replace` states.

See Also

`axis`, `cla`, `ishold`, `newplot`

The `NextPlot` property of axes and figure graphics objects

"Basic Plots and Graphs" on page 1-91 for related functions

Purpose

Send the cursor home

Syntax

```
home
```

Description

`home` moves the cursor to the upper-left corner of the window. When using the MATLAB desktop, `home` also scrolls the visible text in the window up and out of view. You can use the scroll bar to see what was previously on the screen.

Examples

Execute a MATLAB command that displays something in the Command Window and then run the `home` function. `home` moves the cursor to the upper-left corner of the screen and clears the screen.

```
magic(5)

ans =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

home
```

See Also

`clc`

horzcat

Purpose

Concatenate arrays horizontally

Syntax

`C = horzcat(A1, A2, ...)`

Description

`C = horzcat(A1, A2, ...)` horizontally concatenates matrices `A1`, `A2`, and so on. All matrices in the argument list must have the same number of rows.

`horzcat` concatenates N-dimensional arrays along the second dimension. The first and remaining dimensions must match.

MATLAB calls `C = horzcat(A1, A2,...)` for the syntax `C = [A1 A2 ...]` when any of `A1`, `A2`, etc., is an object.

Examples

Create a 3-by-5 matrix, `A`, and a 3-by-3 matrix, `B`. Then horizontally concatenate `A` and `B`.

```
A = magic(5);            % Create 3-by-5 matrix, A
A(4:5,:) = []

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22


B = magic(3)*100       % Create 3-by-3 matrix, B

B =

   800   100   600
   300   500   700
   400   900   200


C = horzcat(A, B)      % Horizontally concatenate A and B
```

```
C =

    17    24     1     8    15   800   100   600
    23     5     7    14    16   300   500   700
     4     6    13    20    22   400   900   200
```

See Also `vertcat`, `cat`, `strcat`, `strvcat`, `special character []`

horzcat (tscollection)

Purpose Horizontal concatenation for `tscollection` objects

Syntax

```
tsc = horzcat(tsc1,tsc2,...)
```

Description `tsc = horzcat(tsc1,tsc2,...)` performs horizontal concatenation for `tscollection` objects:

```
tsc = [tsc1 tsc2 ...]
```

This operation combines multiple `tscollection` objects, which must have the same time vectors, into one `tscollection` containing `timeseries` objects from all concatenated collections.

See Also `tscollection`, `vertcat (tscollection)`

Purpose Server host identification number

Syntax

```
id = hostid
```

hostid

Description

`id = hostid` usually returns a single element cell array containing the MATLAB server host identifier as a string. On UNIX[8] platforms, there can be more than one identifier. In that case, `hostid` returns a cell array with an identifier in each cell.

---

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

Purpose

Convert HSV colormap to RGB colormap

Syntax

```
M = hsv2rgb(H)
rgb_image = hsv2rgb(hsv_image)
```

Description

`M = hsv2rgb(H)` converts a hue-saturation-value (HSV) colormap to a red-green-blue (RGB) colormap. `H` is an m-by-3 matrix, where m is the number of colors in the colormap. The columns of `H` represent hue, saturation, and value, respectively. `M` is an m-by-3 matrix. Its columns are intensities of red, green, and blue, respectively.

`rgb_image = hsv2rgb(hsv_image)` converts the HSV image to the equivalent RGB image. HSV is an m-by-n-by-3 image array whose three planes contain the hue, saturation, and value components for the image. RGB is returned as an m-by-n-by-3 image array whose three planes contain the red, green, and blue components for the image.

Remarks

As `H(:,1)` varies from 0 to 1, the resulting color varies from red through yellow, green, cyan, blue, and magenta, and returns to red. When `H(:,2)` is 0, the colors are unsaturated (i.e., shades of gray). When `H(:,2)` is 1, the colors are fully saturated (i.e., they contain no white component). As `H(:,3)` varies from 0 to 1, the brightness increases.

The MATLAB `hsv` colormap uses `hsv2rgb([huesaturationvalue])` where `hue` is a linear ramp from 0 to 1, and `saturation` and `value` are all 1's.

See Also

`brighten`, `colormap`, `rgb2hsv`

"Color Operations" on page 1-103 for related functions

Purpose Square root of sum of squares

Syntax `c = hypot(a,b)`

Description `c = hypot(a,b)` returns the element-wise result of the following equation, computed to avoid underflow and overflow:

```
c = sqrt(abs(a).^2 + abs(b).^2)
```

Inputs `a` and `b` must follow these rules:

- Both `a` and `b` must be single- or double-precision, floating-point arrays.
- The sizes of the `a` and `b` arrays must either be equal, or one a scalar and the other nonscalar. In the latter case, `hypot` expands the scalar input to match the size of the nonscalar input.
- If `a` or `b` is an empty array (0-by-N or N-by-0), the other must be the same size or a scalar. The result `c` is an empty array having the same size as the empty input(s).

`hypot` returns the following in output `c`, depending upon the types of inputs:

- If the inputs to `hypot` are complex (`w+xi` and `y+zi`), then the statement `c = hypot(w+xi,y+zi)` returns the *positive real* result

  ```
  c = sqrt(abs(w).^2+abs(x).^2+abs(y).^2+abs(z).^2)
  ```

- If `a` or `b` is `–Inf`, `hypot` returns `Inf`.
- If neither `a` nor `b` is `Inf`, but one or both inputs is `NaN`, `hypot` returns `NaN`.
- If all inputs are finite, the result is finite. The one exception is when both inputs are very near the value of the MATLAB constant `realmax`. The reason for this is that the equation `c =`

`hypot(realmax,realmax)` is theoretically `sqrt(2)*realmax`, which overflows to `Inf`.

Examples

Example 1

To illustrate the difference between using the `hypot` function and coding the basic `hypot` equation in M-code, create an anonymous function that performs the same function as `hypot`, but without the consideration to underflow and overflow that `hypot` offers:

```
myhypot = @(a,b)sqrt(abs(a).^2+abs(b).^2);
```

Find the upper limit at which your coded function returns a useful value. You can see that this test function reaches its maximum at about `1e154`, returning an infinite result at that point:

```
myhypot(1e153,1e153)
ans =
    1.4142e+153

myhypot(1e154,1e154)
ans =
   Inf
```

Do the same using the `hypot` function, and observe that `hypot` operates on values up to about `1e308`, which is approximately equal to the value for `realmax` on your computer (the largest double-precision floating-point number you can represent on a particular computer):

```
hypot(1e308,1e308)
ans =
  1.4142e+308

hypot(1e309,1e309)
ans =
   Inf
```

Example 2

`hypot(a,a)` theoretically returns `sqrt(2)*abs(a)`, as shown in this example:

```
x = 1.271161e308;

y = x * sqrt(2)
y =
  1.7977e+308

y = hypot(x,x)
y =
  1.7977e+308
```

Algorithm

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

See Also

`sqrt`, `abs`, `norm`

Purpose

Imaginary unit

Syntax

```
i
a+bi
x+i*y
```

Description

As the basic imaginary unit `sqrt(-1)`, `i` is used to enter complex numbers. Since `i` is a function, it can be overridden and used as a variable. This permits you to use `i` as an index in `for` loops, etc.

If desired, use the character `i` without a multiplication sign as a suffix in forming a complex numerical constant.

You can also use the character `j` as the imaginary unit.

Examples

```
Z = 2+3i
Z = x+i*y
Z = r*exp(i*theta)
```

See Also

`conj`, `imag`, `j`, `real`

idealfilter (timeseries)

Purpose Apply ideal (noncausal) filter to `timeseries` object

Syntax

```
ts2 = idealfilter(ts1,Interval,FilterType)
ts2 = idealfilter(ts1,Interval,FilterType,Index)
```

Description

`ts2 = idealfilter(ts1,Interval,FilterType)` applies an ideal filter of `FilterType` `'pass'` or `'notch'` to one or more frequency intervals specified by `Interval` for the `timeseries` object `ts1`. You specify several frequency intervals as an `n`-by-2 array of start and end frequencies, where `n` represents the number of intervals.

`ts2 = idealfilter(ts1,Interval,FilterType,Index)` applies an ideal filter and uses the optional `Index` integer array to specify the columns or rows to filter. When `ts.IsTimeFirst` is set to `true`, `Index` specifies one or more data columns. When `ts.IsTimeFirst` is set to `false`, Index specifies one or more data rows.

Remarks

When to Use the Ideal Filter

You use the ideal *notch* filter when you want to remove variations in a specific frequency range. Alternatively, you use the ideal *pass* filter to allow only the variations in a specific frequency range.

These filters are ideal in the sense that they are not realizable; an ideal filter is noncausal and the ends of the filter amplitude are perfectly flat in the frequency domain.

Requirement for Uniform Samples in Time

If the time-series data is sampled nonuniformly, filtering resamples this data on a uniform time vector.

Interpolation of NaN Values

All `NaN`s in the time series are interpolated before filtering using the interpolation method you assigned to the `timeseries` object.

Examples You will apply an ideal notch filter to the data in `count.dat`.

1. Load the matrix `count` into the workspace.

```
load count.dat;
```

2 Create a `timeseries` object based on this matrix. The time vector ranges from 1 to 24 seconds in 1-second intervals.

```
count1=timeseries(count(:,1),1:24);
```

3 Enter the frequency interval in hertz.

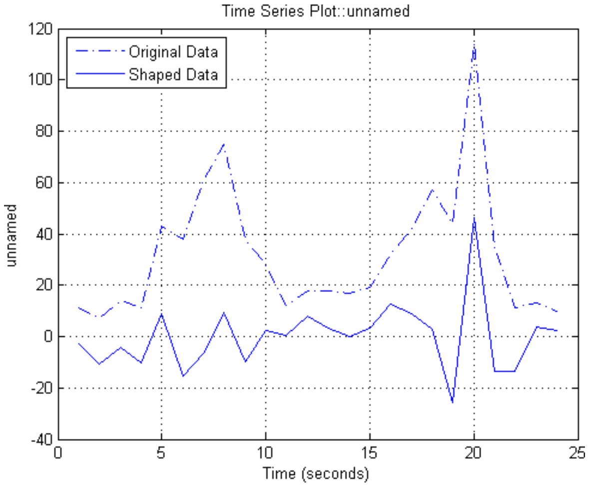
```
interval=[0.08 0.2];
```

4 Call the filter function:

```
idealfilter_count = idealfilter(count1,interval,'notch')
```

5 Compare the original data and the shaped data with an overlaid plot of the two curves.

```
plot(count1,'-.'), grid on, hold on
plot(filter_count,'-')
legend('Original Data','Shaped Data',2)
```



See Also `filter (timeseries)`, `timeseries`

Purpose

Integer division with rounding option

Syntax

```
C = idivide(A, B, opt)
C = idivide(A, B)
C = idivide(A, B, 'fix')
C = idivide(A, B, 'round')
C = idivide(A, B, 'floor')
C = idivide(A, B, 'ceil')
```

Description

`C = idivide(A, B, opt)` is the same as `A./B` for integer classes except that fractional quotients are rounded to integers using the optional rounding mode specified by `opt`. The default rounding mode is `'fix'`. Inputs `A` and `B` must be real and must have the same dimensions unless one is a scalar. At least one of the arguments `A` and `B` must belong to an integer class, and the other must belong to the same integer class or be a scalar double. The result `C` belongs to the integer class.

`C = idivide(A, B)` is the same as `A./B` except that fractional quotients are rounded toward zero to the nearest integers.

`C = idivide(A, B, 'fix')` is the same as the syntax shown immediately above.

`C = idivide(A, B, 'round')` is the same as `A./B` for integer classes. Fractional quotients are rounded to the nearest integers.

`C = idivide(A, B, 'floor')` is the same as `A./B` except that fractional quotients are rounded toward negative infinity to the nearest integers.

`C = idivide(A, B, 'ceil')` is the same as `A./B` except that the fractional quotients are rounded toward infinity to the nearest integers.

Examples

```
a = int32([-2 2]);
b = int32(3);

idivide(a,b)             % Returns [0 0]
idivide(a,b,'floor')     % Returns [-1 0]
idivide(a,b,'ceil')      % Returns [0 1]
```

```
idivide(a,b,'round')     % Returns [-1 1]
```

See Also `ldivide`, `rdivide`, `mldivide`, `mrdivide`

Purpose Execute statements if condition is true

Syntax `if expression, statements, end`

Description `if expression, statements, end` evaluates *`expression`* and, if the evaluation yields logical `1` (`true`) or a nonzero result, executes one or more MATLAB commands denoted here as *`statements`*.

`expression` is a MATLAB expression, usually consisting of variables or smaller expressions joined by relational operators (e.g., `count < limit`), or logical functions (e.g., `isreal(A)`). Simple expressions can be combined by logical operators (`&&`, `||`, `~`) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

```
(count < limit) && ((height - offset) >= 0)
```

Nested `if` statements must each be paired with a matching `end`.

The `if` function can be used alone or with the `else` and `elseif` functions. When using `elseif` and/or `else` within an `if` statement, the general form of the statement is

```
if expression1
    statements1
elseif expression2
    statements2
else
    statements3
end
```

See in the MATLAB Programming Fundamentals documentation for more information on controlling the flow of your program code.

Remarks

Nonscalar Expressions

If the evaluated `expression` yields a nonscalar value, then every element of this value must be true or nonzero for the entire expression to be considered true. For example, the statement `if (A < B)` is true only if each element of matrix `A` is less than its corresponding element in matrix `B`. See Example 2, below.

Partial Evaluation of the expression Argument

Within the context of an `if` or `while` expression, MATLAB does not necessarily evaluate all parts of a logical expression. In some cases it is possible, and often advantageous, to determine whether an expression is true or false through only partial evaluation.

For example, if `A` equals zero in statement 1 below, then the expression evaluates to `false`, regardless of the value of `B`. In this case, there is no need to evaluate `B` and MATLAB does not do so. In statement 2, if `A` is nonzero, then the expression is true, regardless of `B`. Again, MATLAB does not evaluate the latter part of the expression.

```
1)   if (A && B)                2)   if (A || B)
```

You can use this property to your advantage to cause MATLAB to evaluate a part of an expression only if a preceding part evaluates to the desired state. Here are some examples.

```
while (b ~= 0) && (a/b > 18.5)
if exist('myfun.m') && (myfun(x) >= y)
if iscell(A) && all(cellfun('isreal', A))
```

Empty Arrays

In most cases, using `if` on an empty array treats the array as `false`. There are some conditions however under which `if` evaluates as `true` on an empty array. Two examples of this, where `A` is equal to `[]`, are

```
if all(A), do_something, end
if 1|A, do_something, end
```

The latter expression is `true` because of short-circuiting, which causes MATLAB to ignore the right side operand of an OR statement whenever the left side evaluates to `true`.

Short-Circuiting Behavior

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

See for more information on this.

Examples

Example 1 - Simple if Statement

In this example, if both of the conditions are satisfied, then the student passes the course.

```
if ((attendance >= 0.90) && (grade_average >= 60))
   pass = 1;
end;
```

Example 2 - Nonscalar Expression

Given matrices `A` and `B`,

```
A =                     B =
     1     0                 1     1
     2     3                 3     4
```

Expression	Evaluates As	Because
`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.
`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

`else`, `elseif`, `end`, `for`, `while`, `switch`, `break`, `return`, `relational operators`, logical operators (`elementwise` and `short-circuit`),

Purpose Inverse discrete Fourier transform

Syntax

```
y = ifft(X)
y = ifft(X,n)
y = ifft(X,[],dim)
y = ifft(X,n,dim)
y = ifft(..., 'symmetric')
y = ifft(..., 'nonsymmetric')
```

Description

`y = ifft(X)` returns the inverse discrete Fourier transform (DFT) of vector `X`, computed with a fast Fourier transform (FFT) algorithm. If `X` is a matrix, `ifft` returns the inverse DFT of each column of the matrix.

`ifft` tests `X` to see whether vectors in `X` along the active dimension are *conjugate symmetric*. If so, the computation is faster and the output is real. An `N`-element vector `x` is conjugate symmetric if `x(i) = conj(x(mod(N-i+1,N)+1))` for each element of `x`.

If `X` is a multidimensional array, `ifft` operates on the first non-singleton dimension.

`y = ifft(X,n)` returns the `n`-point inverse DFT of vector `X`.

`y = ifft(X,[],dim)` and `y = ifft(X,n,dim)` return the inverse DFT of `X` across the dimension `dim`.

`y = ifft(..., 'symmetric')` causes `ifft` to treat `X` as conjugate symmetric along the active dimension. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft(..., 'nonsymmetric')` is the same as calling `ifft(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft(fft(X))` equals `X` to within roundoff error.

Algorithm

The algorithm for `ifft(X)` is the same as the algorithm for `fft(X)`, except for a sign change and a scale factor of `n = length(X)`. As for `fft`, the execution time for `ifft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifft` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

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

See Also

`fft`, `fft2`, `ifft2`, `ifftn`, `ifftshift`, `fftw`, `ifft2`, `ifftn`

`dftmtx` and `freqz`, in the Signal Processing Toolbox software.

Purpose

2-D inverse discrete Fourier transform

Syntax

```
Y = ifft2(X)
Y = ifft2(X,m,n)
y = ifft2(..., 'symmetric')
y = ifft2(..., 'nonsymmetric')
```

Description

`Y = ifft2(X)` returns the two-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifft2` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `M`-by-`N` matrix `X` is conjugate symmetric if `X(i,j) = conj(X(mod(M-i+1, M) + 1, mod(N-j+1, N) + 1))` for each element of `X`.

`Y = ifft2(X,m,n)` returns the `m`-by-`n` inverse fast Fourier transform of matrix `X`.

`y = ifft2(..., 'symmetric')` causes `ifft2` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft2(..., 'nonsymmetric')` is the same as calling `ifft2(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft2(fft2(X))` equals `X` to within roundoff error.

Algorithm

The algorithm for `ifft2(X)` is the same as the algorithm for `fft2(X)`, except for a sign change and scale factors of `[m,n] = size(X)`. The execution time for `ifft2` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifft2` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`ifft2` supports inputs of data types `double` and `single`. If you call `ifft2` with the syntax `y = ifft2(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`dftmtx` and `freqz` in the Signal Processing Toolbox, and:

`fft2`, `fftw`, `fftshift`, `ifft`, `ifftn`, `ifftshift`

Purpose

N-D inverse discrete Fourier transform

Syntax

```
Y = ifftn(X)
Y = ifftn(X,siz)
y = ifftn(..., 'symmetric')
y = ifftn(..., 'nonsymmetric')
```

Description

`Y = ifftn(X)` returns the n-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifftn` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `N1`-by-`N2`-by- ... `Nk` array `X` is conjugate symmetric if

```
X(i1,i2, ...,ik) = conj(X(mod(N1-i1+1,N1)+1, mod(N2-i2+1,N2)+1,
... mod(Nk-ik+1,Nk)+1))
```

for each element of `X`.

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

`y = ifftn(..., 'symmetric')` causes `ifftn` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifftn(..., 'nonsymmetric')` is the same as calling `ifftn(...)` without the argument `'nonsymmetric'`.

Remarks

For any `X`, `ifftn(fftn(X))` equals `X` within roundoff error.

Algorithm

`ifftn(X)` is equivalent to

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

This computes in-place the one-dimensional inverse DFT along each dimension of `X`.

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

Note You might be able to increase the speed of `ifftn` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

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

See Also

`fftn`, `fftw`, `ifft`, `ifft2`, `ifftshift`

Purpose

Inverse FFT shift

Syntax

```
ifftshift(X)
ifftshift(X,dim)
```

Description

`ifftshift(X)` swaps the left and right halves of the vector `X`. For matrices, `ifftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth. If `X` is a multidimensional array, `ifftshift(X)` swaps "half-spaces" of `X` along each dimension.

`ifftshift(X,dim)` applies the `ifftshift` operation along the dimension `dim`.

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

See Also

`fft`, `fft2`, `fftn`, `fftshift`

Purpose

Sparse incomplete LU factorization

Syntax

```
ilu(A,setup)
[L,U] = ilu(A,setup)
[L,U,P] = ilu(A,setup)
```

Description

`ilu` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`ilu(A,setup)` computes the incomplete LU factorization of `A`. `setup` is an input structure with up to five setup options. The fields must be named exactly as shown in the table below. You can include any number of these fields in the structure and define them in any order. Any additional fields are ignored.

Field Name	Description
`type`	Type of factorization. Values for `type` include: • `'nofill'`—Performs ILU factorization with `0` level of fill in, known as ILU(0). With `type` set to `'nofill'`, only the `milu` setup option is used; all other fields are ignored. • `'crout'`—Performs the Crout version of ILU factorization, known as ILUC. With `type` set to `'crout'`, only the `droptol` and `milu` setup options are used; all other fields are ignored. • `'ilutp'` (default)—Performs ILU factorization with threshold and pivoting. If `type` is not specified, the ILU factorization with pivoting ILUTP is performed. Pivoting is never performed with `type` set to `'nofill'` or `'crout'`.

Field Name	Description

`droptol`	Drop tolerance of the incomplete LU factorization. `droptol` is a non-negative scalar. The default value is `0`, which produces the complete LU factorization.
	The nonzero entries of `U` satisfy
	`abs(U(i,j)) >= droptol*norm((A:,j)),`
	with the exception of the diagonal entries, which are retained regardless of satisfying the criterion. The entries of `L` are tested against the local drop tolerance before being scaled by the pivot, so for nonzeros in `L`
	`abs(L(i,j)) >= droptol*norm(A(:,j))/U(j,j).`

`milu`	Modified incomplete LU factorization. Values for `milu` include: • `'row'`—Produces the row-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, `U`, preserving column sums. That is, `A*e = L*U*e`, where `e` is the vector of ones. • `'col'`—Produces the column-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, `U`, preserving column sums. That is, `e'*A = e'*L*U`. • `'off'` (default)—No modified incomplete LU factorization is produced.

Field Name	Description
`udiag`	If `udiag` is `1`, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is `0`.
`thresh`	Pivot threshold between `0` (forces diagonal pivoting) and `1`, the default, which always chooses the maximum magnitude entry in the column to be the pivot.

`ilu(A,setup)` returns `L+U-speye(size(A))`, where `L` is a unit lower triangular matrix and `U` is an upper triangular matrix.

`[L,U] = ilu(A,setup)` returns a unit lower triangular matrix in `L` and an upper triangular matrix in `U`.

`[L,U,P] = ilu(A,setup)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`.

Remarks

These incomplete factorizations may be useful as preconditioners for a system of linear equations being solved by iterative methods such as BICG (BiConjugate Gradients), GMRES (Generalized Minimum Residual Method).

Limitations

`ilu` works on sparse square matrices only.

Examples

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'crout';
setup.milu = 'row';
setup.droptol = 0.1;
[L,U] = ilu(A,setup);
e = ones(size(A,2),1);
norm(A*e-L*U*e)

ans =
```

```
   1.4251e-014
```

This shows that `A` and `L*U`, where `L` and `U` are given by the modified Crout `ILU`, have the same row-sum.

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'nofill';
nnz(A)
ans =

        7840

nnz(lu(A))
ans =

      126478

nnz(ilu(A,setup))
ans =

        7840
```

This shows that `A` has `7840` nonzeros, the complete LU factorization has `126478` nonzeros, and the incomplete LU factorization, with `0` level of fill-in, has `7840` nonzeros, the same amount as `A`.

See Also

`bicg`, `cholinc`,`gmres`,`luinc`

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

Purpose

Convert image to movie frame

Syntax

```
f = im2frame(X,map)
f = im2frame(X)
```

Description

`f = im2frame(X,map)` converts the indexed image `X` and associated colormap `map` into a movie frame `f`. If `X` is a truecolor (m-by-n-by-3) image, then `map` is optional and has no effect.

Typical usage:

```
M(1) = im2frame(X1,map);
M(2) = im2frame(X2,map);
 ...
M(n) = im2frame(Xn,map);
movie(M)
```

`f = im2frame(X)` converts the indexed image `X` into a movie frame `f` using the current colormap if `X` contains an indexed image.

See Also

`frame2im`, `movie`

"Bit-Mapped Images" on page 1-96 for related functions

Purpose

Convert image to Java image

Syntax

```
jimage = im2java(I)
jimage = im2java(X,MAP)
jimage = im2java(RGB)
```

Description

To work with a MATLAB image in the Java environment, you must convert the image from its MATLAB representation into an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(I)` converts the intensity image `I` to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(X,MAP)` converts the indexed image `X`, with colormap `MAP`, to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(RGB)` converts the RGB image `RGB` to an instance of the Java image class, `java.awt.Image`.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`.

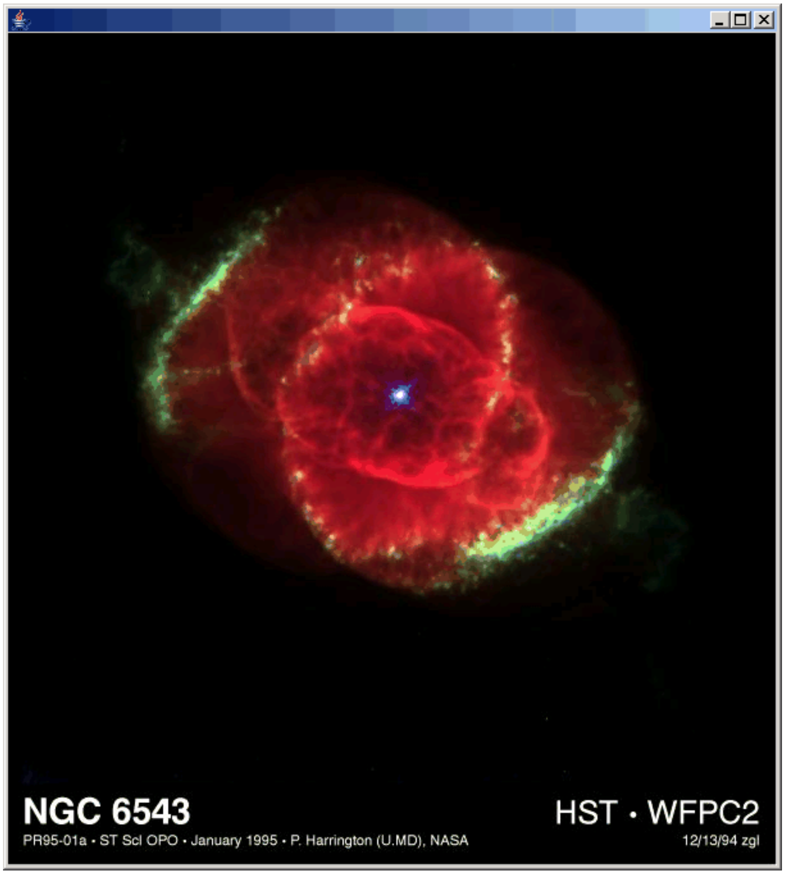
Note Java requires `uint8` data to create an instance of the Java image class, `java.awt.Image`. If the input image is of class `uint8`, `jimage` contains the same `uint8` data. If the input image is of class `double` or `uint16`, `im2java` makes an equivalent image of class `uint8`, rescaling or offsetting the data as necessary, and then converts this `uint8` representation to an instance of the Java image class, `java.awt.Image`.

Example

This example reads an image into the MATLAB workspace and then uses `im2java` to convert it into an instance of the Java image class.

```
I = imread('ngc6543a.jpg');
javaImage = im2java(I);
frame = javax.swing.JFrame;
icon = javax.swing.ImageIcon(javaImage);
label = javax.swing.JLabel(icon);
```

```
frame.getContentPane.add(label);
frame.pack
frame.show
```



See Also

“Bit-Mapped Images” on page 1-96 for related functions

Purpose Imaginary part of complex number

Syntax

```
Y = imag(Z)
```

Description `Y = imag(Z)` returns the imaginary part of the elements of array `Z`.

Examples

```
imag(2+3i)

ans =

     3
```

See Also `conj`, `i`, `j`, `real`

Purpose

Display image object

GUI Alternatives

To plot a selected matrix as an image use the Plot Selector

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

Syntax

```
image(C)
image(x,y,C)
image(x,y,C,'PropertyName',PropertyValue,...)
image('PropertyName',PropertyValue,...)
handle = image(...)
```

Description

`image` creates an image graphics object by interpreting each element in a matrix as an index into the figure's colormap or directly as RGB values, depending on the data specified.

The `image` function has two forms:

- A high-level function that calls `newplot` to determine where to draw the graphics objects and sets the following axes properties:
    - `XLim` and `YLim` to enclose the image
    - `Layer` to `top` to place the image in front of the tick marks and grid lines
    - `YDir` to `reverse`
    - `View` to `[0 90]`
- A low-level function that adds the image to the current axes without calling `newplot`. The low-level function argument list can contain only property name/property value pairs.

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

`image(C)` displays matrix `C` as an image. Each element of `C` specifies the color of a rectangular segment in the image.

`image(x,y,C)`, where `x` and `y` are two-element vectors, specifies the range of the *x*- and *y*-axis labels, but produces the same image as `image(C)`. This can be useful, for example, if you want the axis tick labels to correspond to real physical dimensions represented by the image. If `x(1) > x(2)` or `y(1) > y(2)`, the image is flipped left-right or up-down, respectively. It can also be useful when you want to place the image within a set of axes already created. In this case, use `hold on` with the current figure and enter *x* and *y* values corresponding to the corners of the desired image location. The image is stretched and oriented as applicable.

`image(x,y,C,'PropertyName',PropertyValue,...)` is a high-level function that also specifies property name/property value pairs. This syntax calls `newplot` before drawing the image.

`image('PropertyName',PropertyValue,...)` is the low-level syntax of the `image` function. It specifies only property name/property value pairs as input arguments.

`handle = image(...)` returns the handle of the image object it creates. You can obtain the handle with all forms of the `image` function.

Remarks

Image data can be either indexed or true color. An indexed image stores colors as an array of indices into the figure colormap. A true color image does not use a colormap; instead, the color values for each pixel are stored directly as RGB triplets. In MATLAB graphics, the `CData` property of a truecolor image object is a three-dimensional (`m`-by-`n`-by-3) array. This array consists of three `m`-by-`n` matrices (representing the red, green, and blue color planes) concatenated along the third dimension.

The `imread` function reads image data into MATLAB arrays from graphics files in various standard formats, such as TIFF. You can write MATLAB image data to graphics files using the `imwrite` function.

`imread` and `imwrite` both support a variety of graphics file formats and compression schemes.

When you read image data into the MATLAB workspace using `imread`, the data is usually stored as an array of 8-bit integers. However, `imread` also supports reading 16-bit-per-pixel data from TIFF and PNG files. These are more efficient storage methods than the double-precision (64-bit) floating-point numbers that MATLAB typically uses. However, it is necessary to interpret 8-bit and 16-bit image data differently from 64-bit data. This table summarizes these differences.

You cannot interactively pan or zoom outside the *x*-limits or *y*-limits of an image, unless the axes limits are already been set outside the bounds of the image, in which case there is no such restriction. If other objects (such as lineseries) occupy the axes and extend beyond the bounds of the image, you can pan or zoom to the bounds of the other objects, but no further.

Image Type	Double-Precision Data (double Array)	8-Bit Data (uint8 Array) 16-Bit Data (uint16 Array)
Indexed (colormap)	Image is stored as a two-dimensional (`m`-by-`n`) array of integers in the range [1, `length(colormap)`]; `colormap` is an `m`-by-3 array of floating-point values in the range [0, 1].	Image is stored as a two-dimensional (`m`-by-`n`) array of integers in the range [0, 255] (`uint8`) or [0, 65535] (`uint16`); `colormap` is an `m`-by-3 array of floating-point values in the range [0, 1].
True color (RGB)	Image is stored as a three-dimensional (`m`-by-`n`-by-3) array of floating-point values in the range [0, 1].	Image is stored as a three-dimensional (`m`-by-`n`-by-3) array of integers in the range [0, 255] (`uint8`) or [0, 65535] (`uint16`).

By default, `image` plots the *y*-axis from lowest to highest value, top to bottom. To reverse this, type `set(gca,'YDir','normal')`. This will reverse both the *y*-axis and the image.

Indexed Images

In an indexed image of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. In a `uint8` or `uint16` indexed image, there is an offset; the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on.

If you want to convert a `uint8` or `uint16` indexed image to `double`, you need to add 1 to the result. For example,

```
X64 = double(X8) + 1;
```

or

```
X64 = double(X16) + 1;
```

To convert from `double` to `uint8` or `uint16`, you need to first subtract 1, and then use `round` to ensure all the values are integers.

```
X8 = uint8(round(X64 - 1));
```

or

```
X16 = uint16(round(X64 - 1));
```

When you write an indexed image using `imwrite`, values are automatically converted if necessary.

Colormaps

MATLAB colormaps are always `m`-by-3 arrays of double-precision floating-point numbers in the range [0, 1]. In most graphics file formats, colormaps are stored as integers, but MATLAB colormaps cannot have integer values. `imread` and `imwrite` automatically convert colormap values when reading and writing files.

True Color Images

In a true color image of class `double`, the data values are floating-point numbers in the range [0, 1]. In a true color image of class `uint8`, the data values are integers in the range [0, 255], and for true color images of class `uint16` the data values are integers in the range [0, 65535].

If you want to convert a true color image from one data type to the other, you must rescale the data. For example, this statement converts a `uint8` true color image to `double`.

```
RGB64 = double(RGB8)/255;
```

or for `uint16` images,

```
RGB64 = double(RGB16)/65535;
```

This statement converts a `double` true color image to `uint8`:

```
RGB8 = uint8(round(RGB64*255));
```

or to obtain `uint16` images, type

```
RGB16 = uint16(round(RGB64*65535));
```

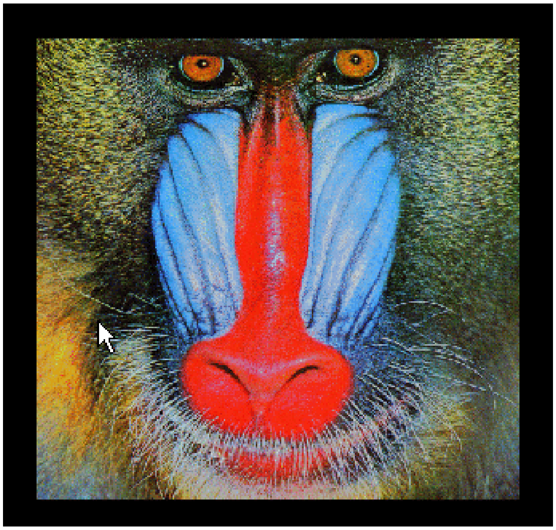
When you write a true color image using `imwrite`, values are automatically converted if necessary..

Example

Example 1

Load a mat-file containing a photograph of a colorful primate. Display the indexed image using its associated colormap.

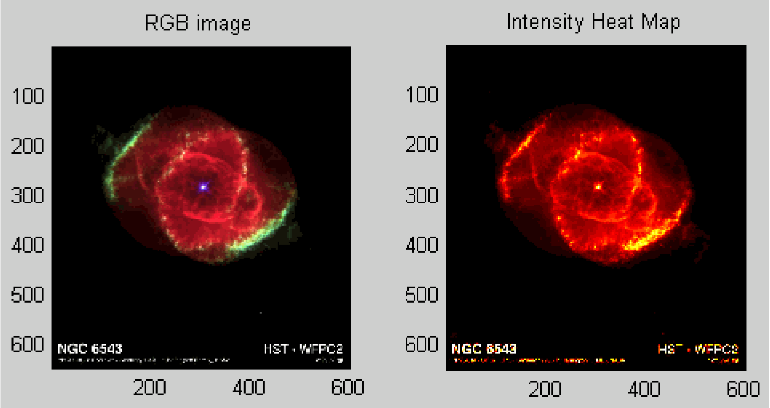
```
load mandrill
figure('color','k')
image(X)
colormap(map)
axis off          % Remove axis ticks and numbers
axis image        % Set aspect ratio to
obtain square pixels
```



Example 2

Load a JPEG image file of the Cat's Eye Nebula from the Hubble Space Telescope (image courtesy NASA). Display the original image using its RGB color values (left) as a subplot. Create a linked subplot (same size and scale) to display the transformed intensity image as a heat map (right).

```
figure
ax(1) = subplot(1,2,1);
rgb = imread('ngc6543a.jpg');
image(rgb); title('RGB image')
ax(2) = subplot(122);
im = mean(rgb,3);
image(im); title('Intensity Heat Map')
colormap(hot(256))
linkaxes(ax,'xy')
axis(ax,'image')
```



Setting Default Properties

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

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

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

See Also

`colormap`, `imagesc`, `imfinfo`, `imread`, `imwrite`, `newplot`, `pcolor`, `surface`

“Bit-Mapped Images” on page 1-96 for related functions

Image Properties for property descriptions

Purpose

Define image 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.

Image Properties

This section lists property names along with the types of values each property accepts.

`AlphaData`

m-by-n matrix of `double` or `uint8`

The transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The `AlphaData` can be of class double or uint8.

MATLAB software determines the transparency in one of three ways:

- Using the elements of `AlphaData` as transparency values (`AlphaDataMapping` set to `none`)
- Using the elements of `AlphaData` as indices into the current alphamap (`AlphaDataMapping` set to `direct`)
- Scaling the elements of `AlphaData` to range between the minimum and maximum values of the axes `ALim` property (`AlphaDataMapping` set to `scaled`, the default)

`AlphaDataMapping`

`{none} | direct| scaled`

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

- `none` — The transparency values of `AlphaData` are between 0 and 1 or are clamped to this range (the default).
- `scaled` — Transform the `AlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values.
- `direct` — Use the `AlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha value in the alphamap, and values greater than `length(alphamap)` to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest, lower integer. If `AlphaData` is an array of `uint8` integers, then the indexing begins at `0` (i.e., MATLAB maps a value of `0` to the first alpha value in the alphamap).

`Annotation`

`hg.Annotation` object Read Only

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

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

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

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

Setting the `IconDisplayStyle` property

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

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

Using the `IconDisplayStyle` property

See for more information and examples.

`BeingDeleted`

`on | {off}` Read Only

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

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

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

`ButtonDownFcn`

string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object.

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

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

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`

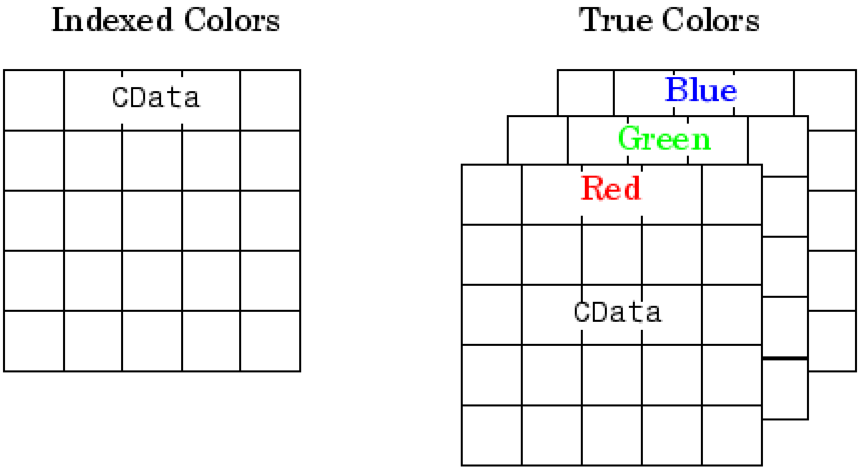
matrix or m-by-n-by-3 array

The image data. A matrix or 3-D array of values specifying the color of each rectangular area defining the image. `image(C)` assigns the values of `C` to `CData`. MATLAB determines the coloring of the image in one of three ways:

- Using the elements of `CData` as indices into the current colormap (the default) (`CDataMapping` set to `direct`)
- Scaling the elements of `CData` to range between the values `min(get(gca,'CLim'))` and `max(get(gca,'CLim'))` (`CDataMapping` set to `scaled`)
- Interpreting the elements of `CData` directly as RGB values (true color specification)

Note that the behavior of `NaN`s in image `CData` is not defined. See the image `AlphaData` property for information on using transparency with images.

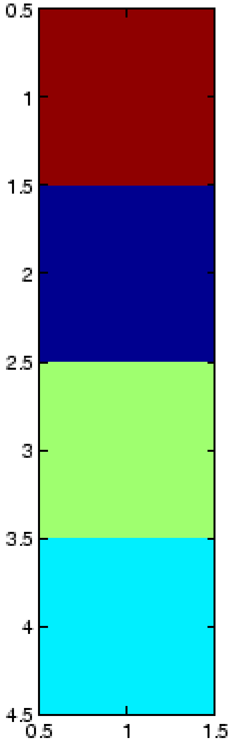
A true color specification for `CData` requires an m-by-n-by-3 array of RGB values. The first page contains the red component, the second page the green component, and the third page the blue component of each element in the image. RGB values range from 0 to 1. The following picture illustrates the relative dimensions of `CData` for the two color models.



If `CData` has only one row or column, the height or width respectively is always one data unit and is centered about the first `YData` or `XData` element respectively. For example, using a 4-by-1 matrix of random data,

```
C = rand(4,1);
image(C,'CDataMapping','scaled')
axis image
```

produces



`CDataMapping`
`scaled | {direct}`

Direct or scaled indexed colors. This property determines whether MATLAB interprets the values in `CData` as indices into the figure colormap (the default) or scales the values according to the values of the axes `CLim` property.

When `CDataMapping` is `direct`, the values of `CData` should be in the range 1 to `length(get(gcf,'Colormap'))`. If you use true color specification for `CData`, this property has no effect. If `CData` is of type `logical`, 0's will index the first color of the colormap and 1's will index the second color.

`Children`

handles

The empty matrix; image objects have no children.

`Clipping`

`on | off`

Clipping mode. By default, MATLAB clips images to the axes rectangle. If you set `Clipping` to `off`, the image can be displayed outside the axes rectangle. For example, if you create an image, set `hold` to `on`, freeze axis scaling (with `axis manual`), and then create a larger image, it extends beyond the axis limits.

`CreateFcn`

string or function handle

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates an image object. You must define this property as a default value for images or in a call to the `image` function to create a new image object. For example, the statement

```
set(0,'DefaultImageCreateFcn','axis image')
```

defines a default value on the root level that sets the aspect ratio and the axis limits so the image has square pixels. MATLAB executes this routine after setting all image properties. Setting this property on an existing image object has no effect.

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

See 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 image object. The `legend` function uses the string defined by the `DisplayName` property to label this image object in the legend.

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

- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See 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).

`Interruptible`

`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`Parent`

handle of parent axes, 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 graphics object. For image objects, `Type` is always '`image`'.

`UIContextMenu`

handle of a uicontextmenu object

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

`UserData`

array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`

`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`

`[1 size(CData,2)]` by default

Control placement of image along x-axis. A vector specifying the locations of the centers of the elements `CData(1,1)` and `CData(m,n)`, where `CData` has a size of `m`-by-`n`. Element `CData(1,1)` is centered over the coordinate defined by the first elements in `XData` and `YData`. Element `CData(m,n)` is centered over the coordinate defined by the last elements in `XData` and `YData`. The centers of the remaining elements of `CData` are evenly distributed between those two points.

The width of each `CData` element is determined by the expression

```
(XData(2)-XData(1))/(size(CData,2)-1)
```

You can also specify a single value for `XData`. In this case, `image` centers the first element at this coordinate and centers each following element one unit apart.

`YData`

`[1 size(CData,1)]` by default

Control placement of image along y-axis. A vector specifying the locations of the centers of the elements `CData(1,1)` and `CData(m,n)`, where `CData` has a size of `m`-by-`n`. Element `CData(1,1)` is centered over the coordinate defined by the first elements in `XData` and `YData`. Element `CData(m,n)` is centered over the coordinate defined by the last elements in `XData` and `YData`. The centers of the remaining elements of `CData` are evenly distributed between those two points.

The height of each `CData` element is determined by the expression

```
(YData(2)-YData(1))/(size(CData,1)-1)
```

You can also specify a single value for `YData`. In this case, `image` centers the first element at this coordinate and centers each following element one unit apart.

Purpose

Scale data and display image object

GUI Alternatives

To plot a selected matrix as an image use the Plot Selector 

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

Syntax

```
imagesc(C)
imagesc(x,y,C)
imagesc(...,clims)
imagesc('PropertyName',PropertyValue,...)
h = imagesc(...)
```

Description

The `imagesc` function scales image data to the full range of the current colormap and displays the image. (See "Examples" on page 2-1790 for an illustration.)

`imagesc(C)` displays `C` as an image. Each element of `C` corresponds to a rectangular area in the image. The values of the elements of `C` are indices into the current colormap that determine the color of each patch.

`imagesc(x,y,C)` displays `C` as an image and specifies the bounds of the x- and y-axis with vectors `x` and `y`. If `x(1) > x(2)` or `y(1) > y(2)`, the image is flipped left-right or up-down, respectively.

`imagesc(...,clims)` normalizes the values in `C` to the range specified by `clims` and displays `C` as an image. `clims` is a two-element vector that limits the range of data values in `C`. These values map to the full range of values in the current colormap.

`imagesc('PropertyName',PropertyValue,...)` is the low-level syntax of the `imagesc` function. It specifies only property name/property value pairs as input arguments.

`h = imagesc(...)` returns the handle for an image graphics object.

Remarks

`x` and `y` do not affect the elements in `C`; they only affect the annotation of the axes. If `length(x) > 2` or `length(y) > 2`, `imagesc` ignores all except the first and last elements of the respective vector.

`imagesc` creates an image with `CDataMapping` set to scaled, and sets the axes `CLim` property to the value passed in `clims`.

You cannot interactively pan or zoom outside the *x*-limits or *y*-limits of an image.

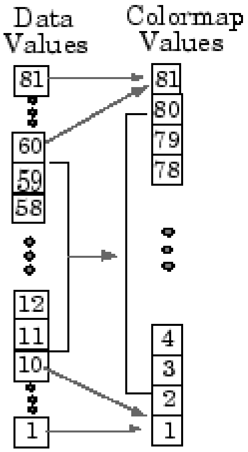
By default, `imagesc` plots the *y*-axis from lowest to highest value, top to bottom. To reverse this, type `set(gca,'YDir','normal')`. This will reverse both the *y*-axis and the image.

Examples

You can expand midrange color resolution by mapping low values to the first color and high values to the last color in the colormap by specifying color value limits (`clims`). If the size of the current colormap is 81-by-3, the statements

```
clims = [ 10 60 ]
imagesc(C,clims)
```

map the data values in C to the colormap as shown in this illustration and the code that follows:

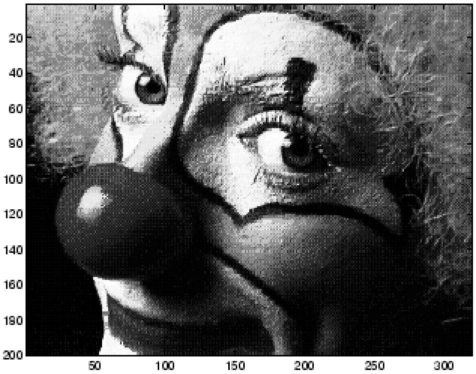


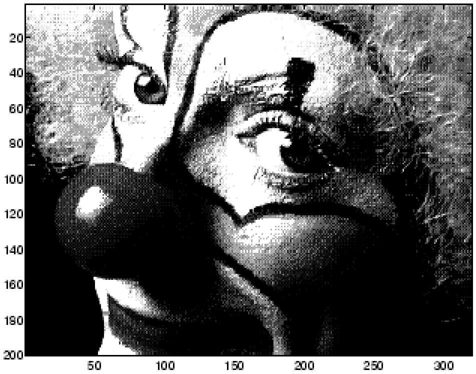
In this example, the left image maps to the `gray` colormap using the statements

```
load clown
imagesc(X)
colormap(gray)
```

The right image has values between `10` and `60` scaled to the full range of the `gray` colormap using the statements

```
load clown
clims = [10 60];
imagesc(X,clims)
colormap(gray)
```





See Also `image`, `imfinfo`, `imread`, `imwrite`, `colorbar`, `colormap`, `pcolor`, `surface`, `surf`

"Bit-Mapped Images" on page 1-96 for related functions

Purpose

Approximate indexed image using one with fewer colors

Syntax

```
[Y,newmap] = imapprox(X,map,n)
[Y,newmap] = imapprox(X,map,tol)
Y = imapprox(X,map,newmap)
Y = imapprox(...,dither_option)
```

Description

`[Y,newmap] = imapprox(X,map,n)` approximates the colors in the indexed image `X` and associated colormap `map` by using minimum variance quantization. `imapprox` returns the indexed image `Y` with colormap `newmap`, which has at most `n` colors.

`[Y,newmap] = imapprox(X,map,tol)` approximates the colors in `X` and `map` through uniform quantization. `newmap` contains at most `(floor(1/tol)+1)^3` colors. `tol` must be between 0 and 1.0.

`Y = imapprox(X,map,newmap)` approximates the colors in `map` by using colormap mapping to find the colors in `newmap` that best match the colors in `map`.

`Y = imapprox(...,dither_option)` enables or disables dithering. `dither_option` is a string that can have one of these values.

Value	Description
`{'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.

Class Support

The input image `X` can be of class `uint8`, `uint16`, or `double`. The output image `Y` is of class `uint8` if the length of `newmap` is less than or equal to 256. If the length of `newmap` is greater than 256, `Y` is of class `double`.

Algorithm

`imapprox` uses `rgb2ind` to create a new colormap that uses fewer colors.

Examples

Approximate the indexed image `trees.tif` by another indexed image containing only 16 colors.

```
[X, map] = imread('trees.tif');
[Y, newmap] = imapprox(X, map, 16);
figure, imshow(Y, newmap)
```

See Also

`cmunique`, `dither`, `rgb2ind`

Purpose

Information about graphics file

Syntax

```
info = imfinfo(filename,fmt)
info = imfinfo(filename)
info = imfinfo(URL,...)
```

Description

`info = imfinfo(filename,fmt)` returns a structure whose fields contain information about an image in a graphics file. `filename` is a string that specifies the name of the graphics file, and *`fmt`* is a string that specifies the format of the file. The file must be in the current folder or in a folder on the MATLAB path. If `imfinfo` cannot find a file named `filename`, it looks for a file named `filename.fmt`. The possible values for *`fmt`* are contained in the MATLAB file format registry. To view of list of these formats, run the `imformats` command.

If `filename` is a TIFF, HDF, ICO, GIF, or CUR file containing more than one image, `info` is a structure array with one element for each image in the file. For example, `info(3)` would contain information about the third image in the file.

`info = imfinfo(filename)` attempts to infer the format of the file from its contents.

`info = imfinfo(URL,...)` reads the image from the specified Internet `URL`. The URL must include the protocol type (e.g., `http://`)

Information Returned

The set of fields in `info` depends on the individual file and its format. However, the first nine fields are always the same. This table lists these common fields, in the order they appear in the structure, and describes their values. “Format-Specific Notes” on page 2-1796 contains information about some fields returned by certain formats.

Field	Value
`Filename`	A string containing the name of the file; if the file is not in the current folder, the string contains the full pathname of the file.

Field	**Value**
`FileModDate`	A string containing the date when the file was last modified
`FileSize`	An integer indicating the size of the file in bytes
`Format`	A string containing the file format, as specified by *`fmt`*; for formats with more than one possible extension (e.g. JPEG and TIFF files), `imfinfo` returns the first variant in the file format registry.
`FormatVersion`	A string or number describing the file format version
`Width`	An integer indicating the width of the image in pixels
`Height`	An integer indicating the height of the image in pixels
`BitDepth`	An integer indicating the number of bits per pixel
`ColorType`	A string indicating the type of image; this can include, but is not limited to, `'truecolor'` for a truecolor (RGB) image, `'grayscale'` for a grayscale intensity image, or `'indexed'` for an indexed image

Format-Specific Notes

- **JPEG and TIFF only** — If `filename` contains Exchangeable Image File Format (EXIF) tags, the `info` structure returned by `imfinfo` might also contain `'DigitalCamera'` or `'GPSInfo'` (global positioning system information) fields.
- **GIF only** — `imfinfo` returns the value of the `'DelayTime'` field in hundredths of seconds.

Example

```
info = imfinfo('canoe.tif')

info =
```

```
                   Filename: [1x76 char]
                FileModDate: '04-Dec-2000 13:57:55'
                   FileSize: 69708
                     Format: 'tif'
              FormatVersion: []
                      Width: 346
                     Height: 207
                   BitDepth: 8
                  ColorType: 'indexed'
            FormatSignature: [73 73 42 0]
                  ByteOrder: 'little-endian'
             NewSubFileType: 0
              BitsPerSample: 8
                Compression: 'PackBits'
  PhotometricInterpretation: 'RGB Palette'
               StripOffsets: [9x1 double]
            SamplesPerPixel: 1
               RowsPerStrip: 23
            StripByteCounts: [9x1 double]
                XResolution: 72
                YResolution: 72
             ResolutionUnit: 'Inch'
                   Colormap: [256x3 double]
        PlanarConfiguration: 'Chunky'
                  TileWidth: []
                 TileLength: []
                TileOffsets: []
             TileByteCounts: []
                Orientation: 1
                  FillOrder: 1
           GrayResponseUnit: 0.0100
             MaxSampleValue: 255
             MinSampleValue: 0
               Thresholding: 1
                     Offset: 67910
```

See Also

`imformats`, `imread`, `imwrite`

"Bit-Mapped Images" on page 1-96 for related functions

Purpose

Manage image file format registry

Syntax

```
imformats
formats = imformats
formats = imformats('fmt')
formats = imformats(format_struct)
formats = imformats('factory')
```

Description

`imformats` displays a table of information listing all the values in the MATLAB file format registry. This registry determines which file formats are supported by the `imfinfo`, `imread`, and `imwrite` functions.

`formats = imformats` returns a structure containing all the values in the MATLAB file format registry. The following tables lists the fields in the order they appear in the structure.

Field	Value
`ext`	A cell array of strings that specify filename extensions that are valid for this format
`isa`	A string specifying the name of the function that determines if a file is a certain format. This can also be a function handle.
`info`	A string specifying the name of the function that reads information about a file. This can also be a function handle.
`read`	A string specifying the name of the function that reads image data in a file. This can also be a function handle.
`write`	A string specifying the name of the function that writes MATLAB data to a file. This can also be a function handle.
`alpha`	Returns `1` if the format has an alpha channel, `0` otherwise
`description`	A text description of the file format

Note The values for the `isa`, `info`, `read`, and `write` fields must be functions on the MATLAB search path or function handles.

`formats = imformats('fmt')` searches the known formats in the MATLAB file format registry for the format associated with the filename extension `'fmt'`. If found, `imformats` returns a structure containing the characteristics and function names associated with the format. Otherwise, it returns an empty structure.

`formats = imformats(format_struct)` sets the MATLAB file format registry to the values in `format_struct`. The output structure, `formats`, contains the new registry settings.

Caution Using `imformats` to specify values in the MATLAB file format registry can result in the inability to load any image files. To return the file format registry to a working state, use `imformats` with the `'factory'` setting.

`formats = imformats('factory')` resets the MATLAB file format registry to the default format registry values. This removes any user-specified settings.

Changes to the format registry do not persist between MATLAB sessions. To have a format always available when you start MATLAB, add the appropriate `imformats` command to the MATLAB startup file, `startup.m`, located in `$MATLAB/toolbox/local` on UNIX systems, or `$MATLAB\toolbox\local` on Windows systems.

Example

```
formats = imformats;
formats(1)

ans =

           ext: {'bmp'}
```

```
         isa: @isbmp
        info: @imbmpinfo
        read: @readbmp
       write: @writebmp
       alpha: 0
 description: 'Windows Bitmap (BMP)'
```

See Also

`fileformats`, `imfinfo`, `imread`, `imwrite`, `path`

"Bit-Mapped Images" on page 1-96 for related functions

import

Purpose

Add package or class to current import list

Syntax

```
import package_name.*
import class_name
import cls_or_pkg_name1 cls_or_pkg_name2...
import
L = import
```

Description

`import package_name.*` adds specified `package_name` to the current import list. Note that `package_name` must be followed by `.*`.

`import class_name` adds a single class to the current import list. Note that `class_name` must be fully qualified (that is, it must include the package name).

`import cls_or_pkg_name1 cls_or_pkg_name2...` adds all named classes and packages to the current import list. Note that each class name must be fully qualified, and each package name must be followed by `.*`.

`import` with no input arguments displays the current import list, without adding to it.

`L = import` with no input arguments returns a cell array of strings containing the current import list, without adding to it.

The `import` function only affects the import list of the function within which it is used. When invoked at the command prompt, `import` uses the import list for the MATLAB command environment. If `import` is used in a script invoked from a function, it affects the import list of the function. If `import` is used in a script that is invoked from the command prompt, it affects the import list for the command environment.

The import list of a function is persistent across calls to that function and is only cleared when the function is cleared.

To clear the current import list, use the following command.

```
clear import
```

This command may only be invoked at the command prompt. Attempting to use `clear import` within a function results in an error.

Importing MATLAB Packages and Classes

You can `import` packages and classes into a MATLAB workspace (from the command line or in a function definition). For example:

```
import packagename.*
```

imports all classes and package functions so that you can reference those classes and functions by their simple names, without the package qualifier.

You can import just a single class from a package:

```
import packagename.ClassName
import Classname
```

You must still use the class name to call static methods:

```
ClassName.staticMethod()
```

For more information on how `import` works with MATLAB classes and packages, see .

Remarks

The `import` function allows your code to refer to an imported class by class name only, rather than with the fully qualified class name. `import` is particularly useful in streamlining calls to constructors, where most references to Java classes occur.

Examples

Add Java Class to Current Import List

```
import java.lang.String
s = String('hello');   % Create java.lang.String object
```

Add Multiple Java Packages to Current Import List

```
import java.util.* java.awt.*
f = Frame;             % Create java.awt.Frame object
```

```
methods Enumeration    % List java.util.Enumeration methods
```

See Also `clear`, `load`, `importdata`

Purpose

Load data from file

Syntax

```
importdata(filename)
A = importdata(filename)
A = importdata(filename, delimiter)
A = importdata(filename, delimiter, nheaderlines)
[A, delimiter] = importdata(...)
[A, delimiter, nheaderlines] = importdata(...)
[...] = importdata('-pastespecial', ...)
```

Description

`importdata(filename)` loads data from `filename` into the workspace.

`A = importdata(filename)` loads data into `A`.

`A = importdata(filename, delimiter)` interprets `delimiter` as the column separator in ASCII file `filename`.

`A = importdata(filename, delimiter, nheaderlines)` loads data from ASCII file `filename`, reading numeric data starting from line `nheaderlines+1`.

`[A, delimiter] = importdata(...)` returns the detected delimiter character for the input ASCII file.

`[A, delimiter, nheaderlines] = importdata(...)` returns the detected number of header lines in the input ASCII file.

`[...]  = importdata('-pastespecial', ...)` loads data from the system clipboard rather than from a file.

Inputs

`filename`

Name and extension of the file to import. If `importdata` recognizes the file extension, it calls the MATLAB helper function designed to import the associated file format (such as `load` for MAT-files or `xlsread` for spreadsheets). Otherwise, `importdata` interprets the file as a delimited ASCII file.

For a list of supported file formats, see the `file formats` table.

`delimiter`

Character in an ASCII file to interpret as a column separator, such as `','` or `';'`. Use `'\t'` for tab.

Default: interpreted from file

`nheaderlines`

Number of text header lines in the input ASCII file. `importdata` stores all the header text in the `textdata` field, and stores the last line of column header text in the `colheaders` field.

Default: interpreted from file

`'-pastespecial'`

Keyword to request that `importdata` load data from the system clipboard instead of a file.

Outputs

`A`

Data from the file. The class of *`A`* depends on the characteristics of the input file.

As described in the Inputs section, `importdata` calls a helper function to read the data. When the helper function returns more than one nonempty output, `importdata` combines the outputs into a `struct` array.

The following table lists the file formats associated with helper functions that can return more than one output, and the possible fields in *`A`*.

File Format	Possible Fields	Class
MAT-files	One field for each variable	Associated with each variable.

File Format	**Possible Fields**	**Class**

ASCII files and Spreadsheets	`data`
	`textdata`
	`colheaders`
	`rowheaders`
		For ASCII files, `data` contains a `double` array. Other fields contain `cell` arrays.
		For spreadsheets, each field contains a `struct`, with one field for each worksheet.

Images	`cdata`
	`colormap`
	`alpha`
		See `imread`.

Audio files	`data` `fs`	See `auread` or `wavread`.

The MATLAB helper functions for all other supported file formats return one output. For more information about the class of each output, see the functions listed in the `file formats` table.

For ASCII files and spreadsheets, `importdata` expects to find numeric data in a rectangular form (that is, like a matrix). Text headers can appear above or to the left of the numeric data, as follows:

- Column headers or file description text at the top of the file, above the numeric data.
- Row headers to the left of the numeric data.

To import ASCII files with nonnumeric characters anywhere else, including columns of character data or formatted dates or times, use `textscan` instead of `importdata`. For more information, see .

When importing spreadsheets with columns of nonnumeric data, `importdata` cannot always correctly interpret the column and row headers.

If the ASCII file or spreadsheet contains either column or row headers, but not both, `importdata` returns a `colheaders` or `rowheaders` field in the output structure, where:

- `colheaders` contains only the lowest line of column header text. `importdata` stores all text in the `textdata` field.
- `rowheaders` is created only when the file or worksheet contains a single column of row headers.

`delimiter`

The character that `importdata` detected as the column separator in the input ASCII file.

`nheaderlines`

The number of text header lines that `importdata` detected in the input ASCII file.

Examples

Import and display the image `ngc6543a.jpg`:

```
nebula_im = importdata('ngc6543a.jpg');
image(nebula_im);
```

`nebula_im` is class `uint8` because the helper function, `imread`, returns empty results for `colormap` and `alpha`.

Using a text editor, create a space-delimited ASCII file with column headers called `myfile.txt`:

```
Day1  Day2  Day3  Day4  Day5  Day6  Day7
95.01 76.21 61.54 40.57  5.79 20.28  1.53
23.11 45.65 79.19 93.55 35.29 19.87 74.68
60.68  1.85 92.18 91.69 81.32 60.38 44.51
```

```
48.60 82.14 73.82 41.03  0.99 27.22 93.18
89.13 44.47 17.63 89.36 13.89 19.88 46.60
```

Import the file, specifying the space delimiter and the single column header, and view columns 3 and 5:

```
M = importdata('myfile.txt', ' ', 1);

for k = [3, 5]
   disp(M.colheaders{1, k})
   disp(M.data(:, k))
   disp(' ')
end
```

Alternatives

The easiest way to import data is to use the Import Wizard, a graphical user interface. The Import Wizard imports the same file formats as `importdata`, but allows direct control over the variables to create. To start the Wizard, select **File > Import Data** or call `uiimport`.

See Also

`file formats` | `load` | `save` | `textscan` | `uiimport`

How To

-
-
-

imread

Purpose Read image from graphics file

Syntax

```
A = imread(filename, fmt)
[X, map] = imread(...)
[...] = imread(filename)
[...] = imread(URL,...)
[...] = imread(...,Param1,Val1,Param2,Val2...)
```

Description

`A = imread(filename, fmt)` reads a grayscale or color image from the file specified by the string `filename`. If the file is not in the current folder, or in a folder on the MATLAB path, specify the full pathname.

The text string *`fmt`* specifies the format of the file by its standard file extension. For example, specify `'gif'` for Graphics Interchange Format files. To see a list of supported formats, with their file extensions, use the `imformats` function. If `imread` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

The return value `A` is an array containing the image data. If the file contains a grayscale image, `A` is an M-by-N array. If the file contains a truecolor image, `A` is an M-by-N-by-3 array. For TIFF files containing color images that use the CMYK color space, `A` is an M-by-N-by-4 array. See TIFF in the Format-Specific Information section for more information.

The class of `A` depends on the bits-per-sample of the image data, rounded to the next byte boundary. For example, `imread` returns 24-bit color data as an array of `uint8` data because the sample size for each color component is 8 bits. See "Remarks" on page 2-1811 for a discussion of bitdepths, and see "Format-Specific Information" on page 2-1811 for more detail about supported bitdepths and sample sizes for a particular format.

`[X, map] = imread(...)` reads the indexed image in `filename` into `X` and its associated colormap into `map`. Colormap values in the image file are automatically rescaled into the range `[0,1]`.

`[...]  = imread(filename)` attempts to infer the format of the file from its content.

`[...] = imread(URL,...)` reads the image from an Internet URL. The `URL` must include the protocol type (e.g., `http://`).

`[...] = imread(...,Param1,Val1,Param2,Val2...)` specifies parameters that control various characteristics of the operations for specific formats. For more information, see "Format-Specific Information" on page 2-1811.

Remarks

Bitdepth is the number of bits used to represent each image pixel. Bitdepth is calculated by multiplying the bits-per-sample with the samples-per-pixel. Thus, a format that uses 8-bits for each color component (or sample) and three samples per pixel has a bitdepth of 24. Sometimes the sample size associated with a bitdepth can be ambiguous: does a 48-bit bitdepth represent six 8-bit samples, four 12-bit samples, or three 16-bit samples? The following format-specific sections provide sample size information to avoid this ambiguity.

Format-Specific Information

The following sections provide information about the support for specific formats, listed in alphabetical order by format name. These sections include information about format-specific syntaxes, if they exist.

"BMP — Windows Bitmap" on page 2-1812
"CUR — Cursor File" on page 2-1812
"GIF — Graphics Interchange Format" on page 2-1813
"JPEG — Joint Photographic Experts Group" on page 2-1814
"JPEG 2000 — Joint Photographic Experts Group 2000" on page 2-1815
"PBM — Portable Bitmap" on page 2-1816
"PNG — Portable Network Graphics" on page 2-1817
"PPM — Portable Pixmap" on page 2-1818
"RAS — Sun Raster" on page 2-1818

"HDF4 — Hierarchical Data Format" on page 2-1814	"PCX — Windows Paintbrush" on page 2-1816	"TIFF — Tagged Image File Format" on page 2-1818
"ICO — Icon File" on page 2-1814	"PGM — Portable Graymap" on page 2-1817	"XWD — X Window Dump" on page 2-1820

BMP — Windows Bitmap

Supported Bitdepths	No Compression	RLE Compression	Output Class	Notes
1-bit	x	–	logical	
4-bit	x	x	`uint8`	
8-bit	x	x	`uint8`	
16-bit	x	–	`uint8`	1 sample/pixel
24-bit	x	–	`uint8`	3 samples/pixel
32-bit	x	–	`uint8`	3 samples/pixel (1 byte padding)

CUR — Cursor File

Supported Bitdepths	No Compression	Compression	Output Class
1-bit	x	–	logical
4-bit	x	–	`uint8`
8-bit	x	–	`uint8`

Format-specific syntaxes:

`[...] = imread(..., idx)` reads in one image from a multi-image icon or cursor file. `idx` is an integer value that specifies the order that the image appears in the file. For example, if `idx` is `3`, `imread` reads the third image in the file. If you omit this argument, `imread` reads the first image in the file.

`[A, map, alpha] = imread(...)` returns the AND mask for the resource, which can be used to determine the transparency information. For cursor files, this mask may contain the only useful data.

Note By default, Microsoft Windows cursors are 32-by-32 pixels. MATLAB pointers must be 16-by-16. You will probably need to scale your image. If you have Image Processing Toolbox™, you can use the `imresize` function.

GIF — Graphics Interchange Format

Supported Bitdepths	No Compression	Compression	Output Class
1-bit	x	–	logical
2-bit to 8-bit	x	–	`uint8`

Format-specific syntaxes:

`[...]  = imread(..., idx)` reads in one or more frames from a multiframe (i.e., animated) GIF file. `idx` must be an integer scalar or vector of integer values. For example, if `idx` is `3`, `imread` reads the third image in the file. If `idx` is `1:5`, `imread` returns only the first five frames.

`[...]  = imread(..., 'frames', idx)` is the same as the syntax above except that `idx` can be `'all'`. In this case, all the frames are read and returned in the order that they appear in the file.

Note Because of the way that GIF files are structured, all the frames must be read when a particular frame is requested. Consequently, it is much faster to specify a vector of frames or `'all'` for `idx` than to call `imread` in a loop when reading multiple frames from the same GIF file.

HDF4 — Hierarchical Data Format

Supported Bitdepths	Raster Image with colormap	Raster image without colormap	Output Class	Notes
8-bit	x	x	`uint8`	
24-bit	–	x	`uint8`	3 samples/pixel

Format-specific syntaxes:

`[...]  = imread(..., ref)` reads in one image from a multi-image HDF4 file. `ref` is an integer value that specifies the reference number used to identify the image. For example, if `ref` is `12`, `imread` reads the image whose reference number is 12. (Note that in an HDF4 file the reference numbers do not necessarily correspond to the order of the images in the file. You can use `imfinfo` to match image order with reference number.) If you omit this argument, `imread` reads the first image in the file.

ICO — Icon File

See "CUR — Cursor File" on page 2-1812

JPEG — Joint Photographic Experts Group

`imread` can read any baseline JPEG image as well as JPEG images with some commonly used extensions. For information about support for JPEG 2000 files, see JPEG 2000.

Supported Bitdepths	Lossy Compression	Lossless Compression	Output Class	Notes
8-bit	x	x	`uint8`	Grayscale or RGB
12-bit	x	x	`uint16`	Grayscale

Supported Bitdepths	Lossy Compression	Lossless Compression	Output Class	Notes
16-bit	–	x	`uint16`	Grayscale
36-bit	x	x	`uint16`	RGB Three 12-bit samples/pixel

JPEG 2000 — Joint Photographic Experts Group 2000

For information about JPEG files, see JPEG.

Note Only 1- and 3-sample images are supported. Indexed JPEG 2000 images are not supported.

Supported Bitdepths (Bits-per-sample)	Lossy Compression	Lossless Compression	Output Class	Notes
1-bit	x	x	`logical`	Grayscale only
2- to 8-bit	x	x	`uint8`	Grayscale or RGB
9- to 16-bit	x	x	`uint16`	Grayscale or RGB

Format-specific syntaxes:

`[...]  = imread(..., 'Param1', value1, 'Param2', value2, ...)` uses parameter-value pairs to control the read operation, described in the following table.

Parameter	Value
`'ReductionLevel'`	A non-negative integer specifying the reduction in the resolution of the image. For a reduction level *L*, the image resolution is reduced by a factor of 2^*L*. Its default value is `0` implying no reduction. The reduction level is limited by the total number of decomposition levels as specified by the`'WaveletDecompositionLevels'` field in the structure returned by the `imfinfo` function.
`'PixelRegion'`	`{ROWS, COLS}` — The `imread` function returns the sub-image specified by the boundaries in `ROWS` and `COLS`. `ROWS` and `COLS` must both be two-element vectors that denote the 1-based indices `[START STOP]`. If `'ReductionLevel'` is greater than 0, then `ROWS` and `COLS` are coordinates in the reduced-sized image.

PBM — Portable Bitmap

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
1-bit	x	x	logical

PCX — Windows Paintbrush

Supported Bitdepths	Output Class	Notes
1-bit	`logical`	Grayscale only
8-bit	`uint8`	Grayscale or indexed
24-bit	`uint8`	RGB Three 8-bit samples/pixel

PGM — Portable Graymap

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
Up to 16-bit	x	–	`uint8`
Arbitrary	–	x	

PNG — Portable Network Graphics

Supported Bitdepths	Output Class	Notes
1-bit	`logical`	Grayscale
2-bit	`uint8`	Grayscale
4-bit	`uint8`	Grayscale
8-bit	`uint8`	Grayscale or Indexed
16-bit	`uint16`	Grayscale or Indexed
24-bit	`uint8`	RGB Three 8-bit samples/pixel.
48-bit	`uint16`	RGB Three 16-bit samples/pixel.

Format-specific syntaxes:

`[...]  = imread(...,'BackgroundColor',BG)` composites any transparent pixels in the input image against the color specified in `BG`. If `BG` is `'none'`, then no compositing is performed. If the input image is indexed, `BG` must be an integer in the range `[1,P]` where `P` is the colormap length. If the input image is grayscale, `BG` should be an integer in the range `[0,1]`. If the input image is RGB, `BG` should be a three-element vector whose values are in the range `[0,1]`. The string `'BackgroundColor'` may be abbreviated.

`[A, map, alpha] = imread(...)` returns the alpha channel if one is present; otherwise `alpha` is `[]`. Note that `map` may be empty if the file contains a grayscale or truecolor image.

If you specify the`alpha` output argument, `BG` defaults to `'none'`, if not specified. Otherwise, if the PNG file contains a background color chunk, that color is used as the default value for `BG`. If `alpha` is not used and the file does not contain a background color chunk, then the default value for `BG` is `1` for indexed images; `0` for grayscale images; and `[0 0 0]` for truecolor (RGB) images.

PPM — Portable Pixmap

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
Up to 16-bit	x	–	`uint8`
Arbitrary	–	x	

RAS — Sun Raster

The following table lists the supported bitdepths, compression, and output classes for RAS files.

Supported Bitdepths	Output Class	Notes
1-bit	`logical`	Bitmap
8-bit	`uint8`	Indexed
24-bit	`uint8`	RGB Three 8-bit samples/pixel
32-bit	`uint8`	RGB with Alpha Four 8-bit samples/pixel

TIFF — Tagged Image File Format

`imread` supports the following TIFF capabilities:

- Any number of samples-per-pixel
- CCITT group 3 and 4 FAX, Packbits, JPEG, LZW, Deflate, ThunderScan compression, and uncompressed images
- Logical, grayscale, indexed color, truecolor and hyperspectral images
- RGB, CMYK, CIELAB, ICCLAB color spaces. If the color image uses the CMYK color space, A is an M-by-N-by-4 array. To determine which color space is used, use `imfinfo` to get information about the graphics file and look at the value of the `PhotometricInterpretation` field. If a file contains CIELAB color data, `imread` converts it to ICCLAB before bringing it into the MATLAB workspace because 8- or 16-bit TIFF CIELAB-encoded values use a mixture of signed and unsigned data types that cannot be represented as a single MATLAB array.
- Data organized into tiles or scanlines

The following table lists the supported bit/sample and corresponding output classes for TIFF files.

Bits-per-Sample	Sample Format	Output Class
1	`integer`	`logical`
2 – 8	`integer`	`uint8`
9 – 16	`integer`	`uint16`
17 – 32	`integer`	`uint32`
32	`float`	`single`
33 – 64	`integer`	`uint64`
64	`float`	`double`

The following are format-specific syntaxes for TIFF files.

`[...]  = imread(..., idx)` reads in one image from a multi-image TIFF file. `idx` is an integer value that specifies the order in which the image appears in the file. For example, if `idx` is `3`, `imread` reads the third image in the file. If you omit this argument, `imread` reads the first image in the file.

`[...]  = imread(..., 'PixelRegion', {ROWS, COLS})` returns the subimage specified by the boundaries in `ROWS` and `COLS`. For tiled TIFF images, `imread` reads only the tiles that encompass the region specified by `ROWS` and `COLS`, improving memory efficiency and performance. `ROWS` and `COLS` must be either two or three element vectors. If two elements are provided, they denote the 1-based indices `[START STOP]`. If three elements are provided, the indices `[START INCREMENT STOP]` allow image downsampling.

For TIFF files, `imread` can read color data represented in the RGB, CIELAB, or ICCLAB color spaces.

XWD — X Window Dump

The following table lists the supported bitdepths, compression, and output classes for XWD files.

Supported Bitdepths	ZPixmaps	XYBitmaps	XYPixmaps	Output Class
1-bit	x	–	x	`logical`
8-bit	x	–	–	`uint8`

Class Support

For most image file formats, `imread` uses 8 or fewer bits per color plane to store image pixels. The following table lists the class of the returned array for the data types used by the file formats.

Data Type Used in File	Class of Array Returned by imread
1-bit per pixel	`logical`

Data Type Used in File	Class of Array Returned by imread
2- to 8-bits per color plane	`uint8`
9- to 16-bit per pixel	`uint16` (BMP, JPEG, PNG, and TIFF) For the 16-bit BMP packed format (5-6-5), MATLAB returns `uint8`

Note For indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself may be of class `uint8` or `uint16`.

Examples

This example reads the sixth image in a TIFF file.

```
[X,map] = imread('your_image.tif',6);
```

This example reads the fourth image in an HDF4 file.

```
info = imfinfo('your_hdf_file.hdf');
[X,map] = imread('your_hdf_file.hdf',info(4).Reference);
```

This example reads a 24-bit PNG image and sets any of its fully transparent (alpha channel) pixels to red.

```
bg = [1 0 0];
A = imread('your_image.png','BackgroundColor',bg);
```

This example returns the alpha channel (if any) of a PNG image.

```
[A,map,alpha] = imread('your_image.png');
```

This example reads an ICO image, applies a transparency mask, and then displays the image.

```
[a,b,c] = imread('your_icon.ico');
```

```
% Augment colormap for background color (white).
b2 = [b; 1 1 1];
% Create new image for display.
d = ones(size(a)) * (length(b2) - 1);
% Use the AND mask to mix the background and
% foreground data on the new image
d(c == 0) = a(c == 0);
% Display new image
image(uint8(d)), colormap(b2)
```

See Also

`double`, `fread`, `image`, `imfinfo`, `imformats`, `imwrite`, `uint8`, `uint16`

"Bit-Mapped Images" on page 1-96 for related functions

Purpose

Write image to graphics file

Syntax

```
imwrite(A,filename,fmt)
imwrite(X,map,filename,fmt)
imwrite(...,filename)
imwrite(...,Param1,Val1,Param2,Val2...)
```

Description

`imwrite(A,filename,fmt)` writes the image `A` to the file specified by `filename` in the format specified by `fmt`.

`A` can be an M-by-N (grayscale image) or M-by-N-by-3 (truecolor image) array, but it cannot be an empty array. For TIFF files, A can be an M-by-N-by-4 array containing color data that uses the CMYK color space. For GIF files, A can be an M-by-N-by-1-by-P array containing grayscale or indexed images — RGB images are not supported. For information about the class of the input array and the output image, see "Class Support" on page 2-1824.

`filename` is a string that specifies the name of the output file.

`fmt` can be any of the text strings listed in the table in "Supported Image Types" on page 2-1825. This list of supported formats is determined by the MATLAB image file format registry. See `imformats` for more information about this registry.

`imwrite(X,map,filename,fmt)` writes the indexed image in `X` and its associated colormap `map` to `filename` in the format specified by `fmt`. If `X` is of class `uint8` or `uint16`, `imwrite` writes the actual values in the array to the file. If `X` is of class `double`, `imwrite` offsets the values in the array before writing, using `uint8(X 1)`. `map` must be a valid MATLAB colormap. Note that most image file formats do not support colormaps with more than 256 entries.

When writing multiframe GIF images, `X` should be an 4-dimensional M-by-N-by-1-by-P array, where P is the number of frames to write.

`imwrite(...,filename)` writes the image to `filename`, inferring the format to use from the filename's extension. The extension must be one of the values for `fmt`, listed in "Supported Image Types" on page 2-1825.

`imwrite(...,Param1,Val1,Param2,Val2...)` specifies parameters that control various characteristics of the output file for HDF, JPEG, PBM, PGM, PNG, PPM, and TIFF files. For example, if you are writing a JPEG file, you can specify the quality of the output image. For the lists of parameters available for each format, see "Format-Specific Parameters" on page 2-1827.

Class Support

The input array `A` can be of class `logical`, `uint8`, `uint16`, or `double`. Indexed images (`X`) can be of class `uint8`, `uint16`, or `double`; the associated colormap, `map`, must be of class `double`. Input values must be full (non-sparse).

The class of the image written to the file depends on the format specified. For most formats, if the input array is of class `uint8`, `imwrite` outputs the data as 8-bit values. If the input array is of class `uint16` and the format supports 16-bit data (JPEG, PNG, and TIFF), `imwrite` outputs the data as 16-bit values. If the format does not support 16-bit values, `imwrite` issues an error. Several formats, such as JPEG and PNG, support a parameter that lets you specify the bit depth of the output data.

If the input array is of class `double`, and the image is a grayscale or RGB color image, `imwrite` assumes the dynamic range is [0,1] and automatically scales the data by 255 before writing it to the file as 8-bit values.

If the input array is of class `double`, and the image is an indexed image, `imwrite` converts the indices to zero-based indices by subtracting 1 from each element, and then writes the data as `uint8`.

If the input array is of class `logical`, `imwrite` assumes the data is a binary image and writes it to the file with a bit depth of 1, if the format allows it. BMP, PNG, or TIFF formats accept binary images as input arrays.

Supported Image Types

This table summarizes the types of images that `imwrite` can write. The MATLAB file format registry determines which file formats are supported. See `imformats` for more information about this registry. Note that, for certain formats, `imwrite` may take additional parameters, described in "Format-Specific Parameters" on page 2-1827.

Format	Full Name	Variants
`'bmp'`	Windows Bitmap (BMP)	1-bit, 8-bit, and 24-bit uncompressed images
`'gif'`	Graphics Interchange Format (GIF)	8-bit images
`'hdf'`	Hierarchical Data Format (HDF4)	8-bit raster image data sets, with or without associated colormap, 24-bit raster image data sets; uncompressed or with RLE or JPEG compression
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)	8-bit, 12-bit, and 16-bit Baseline JPEG images **Note** `imwrite` converts indexed images to RGB before writing data to JPEG files, because the JPEG format does not support indexed images.
`pbm`	Portable Bitmap (PBM)	Any 1-bit PBM image, ASCII (plain) or raw (binary) encoding
`'pcx'`	Windows Paintbrush (PCX)	8-bit images

Format	Full Name	Variants
`'pgm'`	Portable Graymap (PGM)	Any standard PGM image; ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per gray value
`'png'`	Portable Network Graphics (PNG)	1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit grayscale images with alpha channels; 1-bit, 2-bit, 4-bit, and 8-bit indexed images; 24-bit and 48-bit truecolor images; 24-bit and 48-bit truecolor images with alpha channels
`'pnm'`	Portable Anymap (PNM)	Any of the PPM/PGM/PBM formats, chosen automatically
`'ppm'`	Portable Pixmap (PPM)	Any standard PPM image. ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per color component
`'ras'`	Sun Raster (RAS)	Any RAS image, including 1-bit bitmap, 8-bit indexed, 24-bit truecolor and 32-bit truecolor with alpha
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF)	Baseline TIFF images, including 1-bit, 8-bit, 16-bit, and 24-bit uncompressed images, images with packbits compression, images with LZW compression, and images with Deflate compression; 1-bit images with CCITT 1D, Group 3, and Group 4 compression; CIELAB, ICCLAB, and CMYK images
`'xwd'`	X Windows Dump (XWD)	8-bit ZPixmaps

Format-Specific Parameters

GIF-Specific Parameters

Parameter	Values
`'BackgroundColor'`	A scalar integer that specifies which index in the colormap should be treated as the transparent color for the image and is used for certain disposal methods in animated GIFs. If `X` is `uint8` or `logical`, indexing starts at `0`. If `X` is `double`, indexing starts at `1`.
`'Comment'`	A string or cell array of strings containing a comment to be added to the image. For a cell array of strings, a carriage return is added after each row.
`'DelayTime'`	A scalar value between 0 and 655 inclusive, that specifies the delay in seconds before displaying the next image.
`'DisposalMethod'`	One of the following strings, which sets the disposal method of an animated GIF: `'leaveInPlace'`, `'restoreBG'`, `'restorePrevious'`, or `'doNotSpecify'`.
`'Location'`	A two-element vector specifying the offset of the top left corner of the screen relative to the top left corner of the image. The first element is the offset from the top, and the second element is the offset from the left.
`'LoopCount'`	A finite integer between 0 and 65535 or the value `Inf` (the default) which specifies the number of times to repeat the animation. By default, the animation loops continuously. If you specify 0, the animation will be played once. If you specify the value 1, the animation will be played twice, etc. To enable animation within Microsoft® PowerPoint®, specify a value for the `'LoopCount'` parameter within the range [1 65535]. Some Microsoft applications interpret the value 0 to mean do not loop at all.

Parameter	Values
`'ScreenSize'`	A two-element vector specifying the height and width of the frame. When used with `'Location'`, `ScreenSize` provides a way to write frames to the image that are smaller than the whole frame. `'DisposalMethod'` determines the fill value used for pixels outside the frame.
`'TransparentColor'`	A scalar integer. This value specifies which index in the colormap should be treated as the transparent color for the image. If X (the image) is `uint8` or `logical`, indexing starts at `0`. If X is `double`, indexing starts at `1`.
`'WriteMode'`	`'overwrite'` (the default) or `'append'`. In append mode, `imwrite` adds a single frame to the existing file.

HDF4-Specific Parameters

Parameter	Values
`'Compression'`	`'none'` (the default) `'jpeg'`(valid only for grayscale and RGB images) `'rle'` (valid only for grayscale and indexed images)
`'Quality'`	A number between 0 and 100; this parameter applies only if `'Compression'` is `'jpeg'`. Higher numbers mean higher *quality* (less image degradation due to compression), but the resulting file size is larger. The default value is 75.
`'WriteMode'`	`'overwrite'` (the default) `'append'`

JPEG-Specific Parameters

Parameter	Values	Default
`'Bitdepth'`	A scalar value indicating desired bitdepth; for grayscale images this can be 8, 12, or 16; for color images this can be 8 or 12.	8 (grayscale) and 8 bit per plane for color images
`'Comment'`	A column vector cell array of strings or a character matrix. `imwrite` writes each row of input as a comment in the JPEG file.	Empty
`'Mode'`	Specifies the type of compression used:`'lossy'` or `'lossless'`	`'lossy'`
`'Quality'`	A number between 0 and 100; higher numbers mean higher quality (less image degradation due to compression), but the resulting file size is larger.	`75`

PBM-, PGM-, and PPM-Specific Parameters

This table describes the available parameters for PBM, PGM, and PPM files.

Parameter	Values	Default
`'Encoding'`	One of these strings: `'ASCII'` for plain encoding `'rawbits'` for binary encoding	`'rawbits'`
`'MaxValue'`	A scalar indicating the maximum gray or color value. Available only for PGM and PPM files. For PBM files, this value is always `1`.	Default is `65535` if image array is `'uint16'`; `255` otherwise.

PNG-Specific Parameters

The following table lists the available parameters for PNG files, in alphabetical order. In addition to these PNG parameters, you can use any parameter name that satisfies the PNG specification for keywords; that is, uses only printable characters, contains 80 or fewer characters, and no contains no leading or trailing spaces. The value corresponding to these user-specified parameters must be a string that contains no control characters other than linefeed.

Parameter	Values
`'Alpha'`	A matrix specifying the transparency of each pixel individually. The row and column dimensions must be the same as the data array; they can be `uint8`, `uint16`, or `double`, in which case the values should be in the range `[0,1]`.
`'Author'`	A string
`'Background'`	The value specifies background color to be used when compositing transparent pixels. For indexed images: an integer in the range `[1,P]`, where `P` is the colormap length. For grayscale images: a scalar in the range `[0,1]`. For truecolor images: a three-element vector in the range `[0,1]`.
`'bitdepth'`	A scalar value indicating desired bit depth. For grayscale images this can be 1, 2, 4, 8, or 16. For grayscale images with an alpha channel this can be 8 or 16. For indexed images this can be 1, 2, 4, or 8. For truecolor images with or without an alpha channel this can be 8 or 16. By default, `imwrite` uses 8 bits per pixel, if image is double or `uint8`; 16 bits per pixel if image is `uint16`; 1 bit per pixel if image is `logical`.

Parameter	Values
`'Chromaticities'`	An eight-element vector `[wx wy rx ry gx gy bx by]` that specifies the reference white point and the primary chromaticities
`'Comment'`	A string
`'Copyright'`	A string
`'CreationTime'`	A string
`'Description'`	A string
`'Disclaimer'`	A string
`'Gamma'`	A nonnegative scalar indicating the file gamma
`'ImageModTime'`	A MATLAB serial date number (see the `datenum` function) or a string convertible to a date vector via the `datevec` function. Values should be in Coordinated Universal Time (UTC).
`'InterlaceType'`	Either `'none'` (the default) or `'adam7'`
`'ResolutionUnit'`	Either `'unknown'` or `'meter'`
`'SignificantBits'`	A scalar or vector indicating how many bits in the data array should be regarded as significant; values must be in the range [1,`BitDepth`]. For indexed images: a three-element vector. For grayscale images: a scalar. For grayscale images with an alpha channel: a two-element vector. For truecolor images: a three-element vector. For truecolor images with an alpha channel: a four-element vector.
`'Software'`	A string
`'Source'`	A string

Parameter	Values
`'Transparency'`	This value is used to indicate transparency information only when no alpha channel is used. Set to the value that indicates which pixels should be considered transparent. (If the image uses a colormap, this value represents an index number to the colormap.) For indexed images: a `Q`-element vector in the range [0,1], where `Q` is no larger than the colormap length and each value indicates the transparency associated with the corresponding colormap entry. In most cases, `Q = 1`. For grayscale images: a scalar in the range [0,1]. The value indicates the grayscale color to be considered transparent. For truecolor images: a three-element vector in the range [0,1]. The value indicates the truecolor color to be considered transparent. **Note** You cannot specify `'Transparency'` and `'Alpha'` at the same time.
`'Warning'`	A string
`'XResolution'`	A scalar indicating the number of pixels/unit in the horizontal direction
`'YResolution'`	A scalar indicating the number of pixels/unit in the vertical direction

RAS-Specific Parameters

This table describes the available parameters for RAS files.

Parameter	Values	Default
`'Alpha'`	A matrix specifying the transparency of each pixel individually; the row and column dimensions must be the same as the data array; can be `uint8`, `uint16`, or `double`. Can only be used with truecolor images.	Empty matrix (`[]`)
`'Type'`	One of these strings: `'standard'` (uncompressed, b-g-r color order with truecolor images) `'rgb'` (like `'standard'`, but uses r-g-b color order for truecolor images) `'rle'` (run-length encoding of 1-bit and 8-bit images)	`'standard'`

TIFF-Specific Parameters

This table describes the available parameters for TIFF files.

Parameter	Values
`'ColorSpace'`	Specifies the color space used to represent the color data. `'rgb'` `'cielab'` `'icclab'` [1] (default is `'rgb'`). Note: To use the CMYK color space in a TIFF file, do not use the `'ColorSpace'` parameter. It is sufficient to specify an M-by-N-by-4 input array.

Parameter	Values

`'Compression'`	`'none'`
	`'packbits'` (default for non-binary images)
	`'lzw'`
	`'deflate'`
	`'jpeg'`
	`'ccitt'` (binary images only; default)
	`'fax3'` (binary images only)
	`'fax4'` (binary images only)
	Note: `'jpeg'` is a lossy compression scheme; other compression modes are lossless. Also, if you specify `'jpeg'` compression, you must specify the `'RowsPerStrip'` parameter and the value must be a multiple of 8.

`'Description'`	Any string; fills in the `ImageDescription` field returned by `imfinfo`. By default, the field is empty.
`'Resolution'`	A two-element vector containing the XResolution and YResolution, or a scalar indicating both resolutions. The default value is `72`.
`'RowsPerStrip'`	A scalar value specifying the number of rows to include in each strip. The default will be such that each strip is about 8K bytes. Note: You must specify the `'RowsPerStrip'` parameter if you specify `'jpeg'` compression. The value must be a multiple of 8.
`'WriteMode'`	`'overwrite'` (default) `'append'`

[1] `imwrite` can write color image data that uses the *L*a*b** color space to TIFF files. The 1976 CIE *L*a*b** specification defines numeric values that represent luminance (*L**) and chrominance (*a** and *b**) information. To store *L*a*b** color data in a TIFF file, the values must be encoded to fit into either 8-bit or 16-bit storage. `imwrite` can store *L*a*b** color data in a TIFF file using 8-bit and 16-bit encodings defined by the TIFF specification, called the CIELAB encodings, or using 8-bit and 16-bit encodings defined by the International Color Consortium, called ICCLAB encodings.

The output class and encoding used by `imwrite` depends on the class of the input array and the value of the TIFF-specific `ColorSpace` parameter. The following table explains these options. (The 8-bit and 16-bit CIELAB encodings cannot be input arrays because they use a mixture of signed and unsigned values and cannot be represented as a single MATLAB array.)

Input Class and Encoding	ColorSpace Parameter Value	Output Class and Encoding
8-bit ICCLAB Represents values as integers in the range [0 255]. L^* values are multiplied by `255/100`. `128` is added to both the a^* and b^* values.	’`icclab`’	8-bit ICCLAB
	’`cielab`’	8-bit CIELAB

Input Class and Encoding	**ColorSpace Parameter Value**	**Output Class and Encoding**
16-bit ICCLAB Represents the values as integers in the range [0, 65280]. *L** values are multiplied by `65280/100`. `32768` is added to both the *a** and *b** values, which are represented as integers in the range [0,65535].	`'icclab'`	16-bit ICCLAB
	`'cielab'`	16-bit CIELAB
Double-precision 1976 CIE *L*a*b** values *L** is in the dynamic range [0, 100]. *a** and *b** can take any value. Setting *a** and *b** to 0 (zero) produces a neutral color (gray).	`'icclab'`	8-bit ICCLAB
	`'cielab'`	8-bit CIELAB

Example

This example appends an indexed image `X` and its colormap `map` to an existing uncompressed multipage HDF4 file.

```
imwrite(X,map,'your_hdf_file.hdf','Compression','none',...
'WriteMode','append')
```

See Also

`fwrite`, `getframe`, `imfinfo`, `imformats`, `imread`

"Bit-Mapped Images" on page 1-96 for related functions

Purpose

Incenters of specified simplices

Syntax

```
IC = incenters(TR,SI)
[IC RIC] = incenters(TR, SI)
```

Description

`IC = incenters(TR,SI)` returns the coordinates of the incenter of each specified simplex `SI`.

`[IC RIC] = incenters(TR, SI)` returns the incenters and the corresponding radius of the inscribed circle/sphere.

Inputs

`TR`	Triangulation representation.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the incenter information for the entire triangulation is returned, where the incenter associated with simplex `i` is the `i`'th row of `IC`.

Outputs

`IC`	`m`-by-`n` matrix, where `m = length(SI)`, the number of specified simplices, and `n` is the dimension of the space where the triangulation resides. Each row `IC(i,:)` represents the coordinates of the incenter of simplex `SI(i)`.
`RIC`	Vector of length `length(SI)`, the number of specified simplices.

Definitions

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

Examples

Example 1

Load a 3-D triangulation:

```
load tetmesh
```

Use `TriRep` to compute the incenters of the first five tetrahedra.

```
 trep = TriRep(tet, X)
 ic = incenters(trep, [1:5]')
```

Example 2

Query a 2-D triangulation created with `DelaunayTri`.

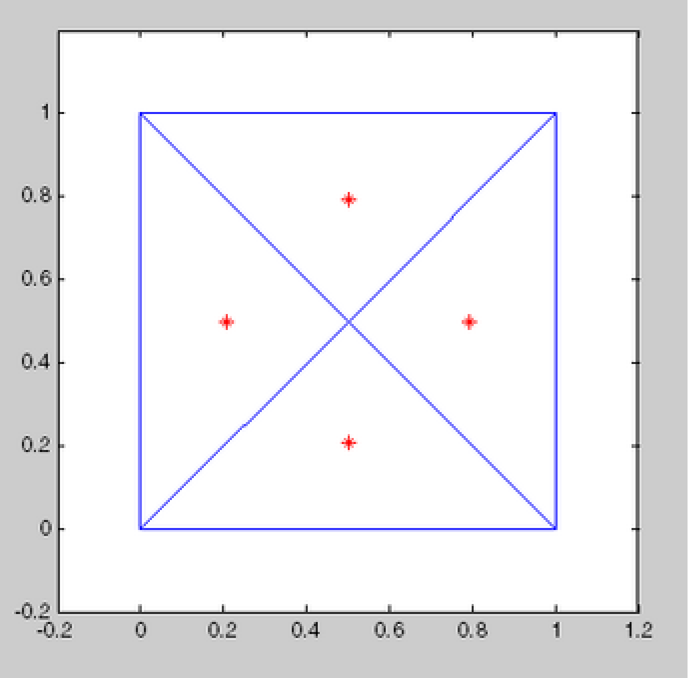
```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
dt = DelaunayTri(x,y);
```

Compute incenters of the triangles:

```
ic = incenters(dt);
```

Plot the triangles and incenters:

```
triplot(dt);
axis equal;
axis([-0.2 1.2 -0.2 1.2]);
hold on;
plot(ic(:,1),ic(:,2),'*r');
hold off;
```



See Also

`DelaunayTri`
`circumcenters`

Purpose

Status of triangles in 2-D constrained Delaunay triangulation

Syntax

`IN = inOutStatus(DT)`

Description

`IN = inOutStatus(DT)` returns the in/out status of the triangles in a 2-D constrained Delaunay triangulation of a geometric domain. Given a Delaunay triangulation that has a set of constrained edges that define a bounded geometric domain. The `i`'th triangle in the triangulation is classified as inside the domain if `IN(i) = 1` and outside otherwise.

Note `inOutStatus` is only relevant for 2-D constrained Delaunay triangulations where the imposed edge constraints bound a closed geometric domain.

Inputs

`DT`	Delaunay triangulation.

Outputs

`IN`	Logical array of length equal to the number of triangles in the triangulation. The constrained edges in the triangulation define the boundaries of a valid geometric domain.

Example

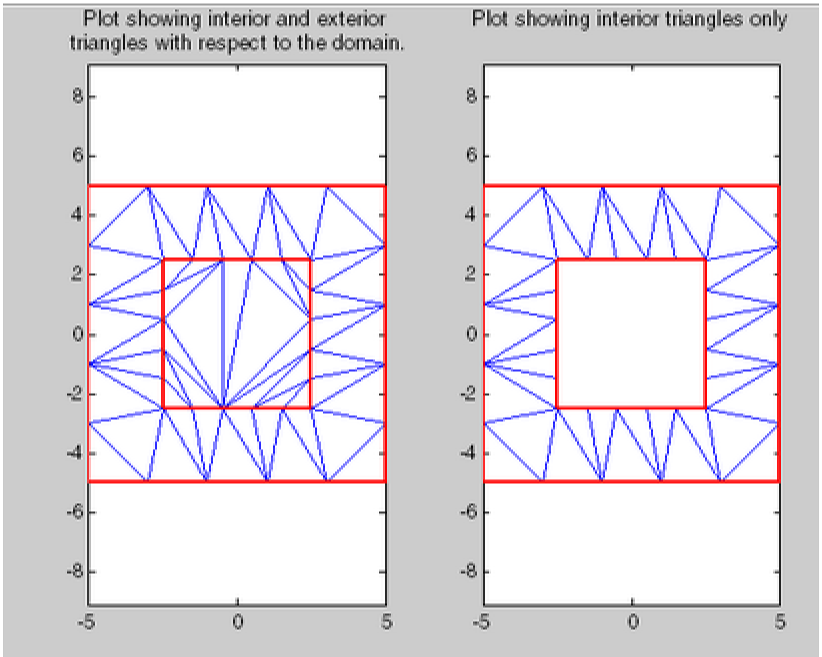
Create a geometric domain that consists of a square with a square hole:

```
outerprofile = [-5 -5; -3 -5; -1 -5; 1 -5; 3 -5; ...
 5 -5; 5 -3; 5 -1; 5 1; 5 3;...
 5 5; 3 5; 1 5; -1 5; -3 5; ...
 -5 5; -5 3; -5 1; -5 -1; -5 -3; ];
innerprofile = outerprofile.*0.5;
profile = [outerprofile; innerprofile];
outercons = [(1:19)' (2:20)'; 20 1;];
innercons = [(21:39)' (22:40)'; 40 21];
```

```
edgeconstraints = [outercons; innercons];
```

Create a constrained Delaunay triangulation of the domain:

```
dt = DelaunayTri(profile, edgeconstraints)
subplot(1,2,1);
triplot(dt);
hold on;
plot(dt.X(outercons',1), dt.X(outercons',2), ...
     '-r', 'LineWidth', 2);
plot(dt.X(innercons',1), dt.X(innercons',2), ...
     '-r', 'LineWidth', 2);
axis equal;
title(sprintf('Plot showing interior and exterior\n ...
     triangles with respect to the domain.'));
hold off;
subplot(1,2,2);
inside = inOutStatus(dt);
triplot(dt(inside, :), dt.X(:,1), dt.X(:,2));
hold on;
plot(dt.X(outercons',1), dt.X(outercons',2), ...
     '-r', 'LineWidth', 2);
plot(dt.X(innercons',1), dt.X(innercons',2), ...
     '-r', 'LineWidth', 2);
axis equal;
title(sprintf('Plot showing interior triangles only\n'));
hold off;
```


Plot showing interior and exterior triangles with respect to the domain.
Plot showing interior triangles only
8
6
4
2
0
-2
-4
-6
-8
-5
0
5

ind2rgb

Purpose Convert indexed image to RGB image

Syntax `RGB = ind2rgb(X,map)`

Description `RGB = ind2rgb(X,map)` converts the matrix `X` and corresponding colormap `map` to RGB (truecolor) format.

Class Support `X` can be of class `uint8`, `uint16`, or `double`. RGB is an m-by-n-by-3 array of class `double`.

See Also `image`

"Bit-Mapped Images" on page 1-96 for related functions

Purpose

Subscripts from linear index

Syntax

```
[I,J] = ind2sub(siz,IND)
[I1,I2,I3,...,In] = ind2sub(siz,IND)
```

Description

The `ind2sub` command determines the equivalent subscript values corresponding to a single index into an array.

`[I,J] = ind2sub(siz,IND)` returns the matrices `I` and `J` containing the equivalent row and column subscripts corresponding to each linear index in the matrix `IND` for a matrix of size `siz`. `siz` is a vector with `ndim(A)` elements (in this case, 2), where `siz(1)` is the number of rows and `siz(2)` is the number of columns.

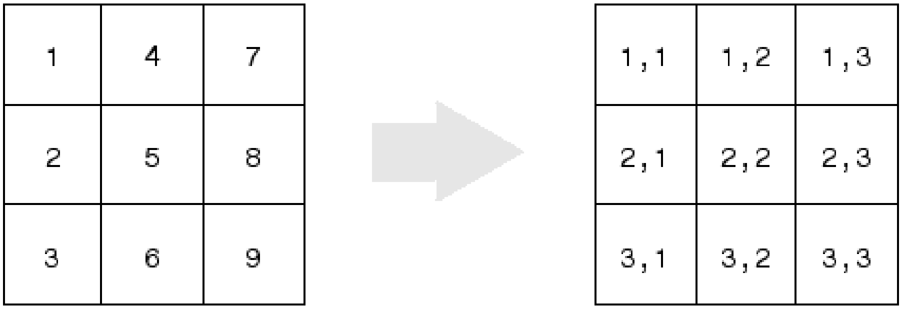
Note For matrices, `[I,J] = ind2sub(size(A),find(A>5))` returns the same values as `[I,J] = find(A>5)`.

`[I1,I2,I3,...,In] = ind2sub(siz,IND)` returns `n` subscript arrays `I1`,`I2`,`...`,`In` containing the equivalent multidimensional array subscripts equivalent to `IND` for an array of size `siz`. `siz` is an `n`-element vector that specifies the size of each array dimension.

Examples

Example 1 — Two-Dimensional Matrices

The mapping from linear indexes to subscript equivalents for a 3-by-3 matrix is



This code determines the row and column subscripts in a 3-by-3 matrix, of elements with linear indices 3, 4, 5, 6.

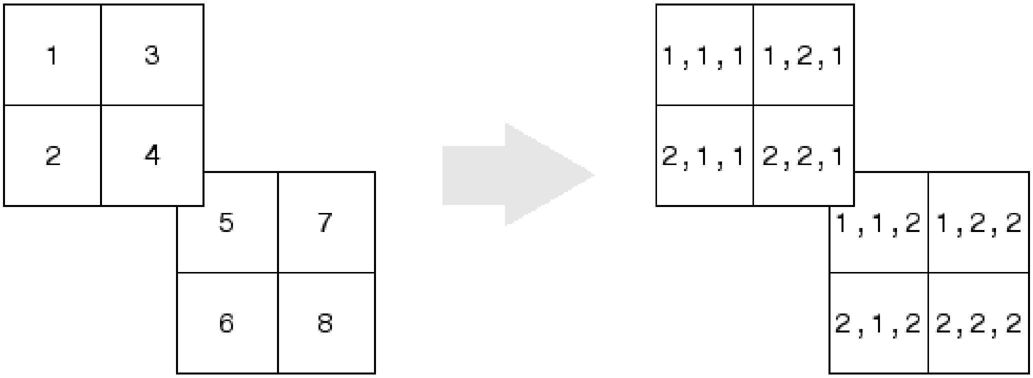
```
IND = [3 4 5 6]
s = [3,3];
[I,J] = ind2sub(s,IND)

I =
     3     1     2     3

J =
     1     2     2     2
```

Example 2 — Three-Dimensional Matrices

The mapping from linear indexes to subscript equivalents for a 2-by-2-by-2 array is



This code determines the subscript equivalents in a 2-by-2-by-2 array, of elements whose linear indices 3, 4, 5, 6 are specified in the `IND` matrix.

```
IND = [3 4;5 6];
s = [2,2,2];
[I,J,K] = ind2sub(s,IND)

I =
     1     2
     1     2

J =
     2     2
     1     1

K =
     1     1
     2     2
```

Example 3 — Effects of Returning Fewer Outputs

When calling `ind2sub` for an N-dimensional matrix, you would typically supply N output arguments in the call: one for each dimension of the matrix. This example shows what happens when you return three, two, and one output when calling `ind2sub` on a 3-dimensional matrix.

The matrix is 2-by-2-by-2 and the linear indices are 1 through 8:

```
dims = [2 2 2];
indices = [1 2 3 4 5 6 7 8];
```

The 3-output call to `ind2sub` returns the expected subscripts for the 2-by-2-by-2 matrix:

```
[rowsub colsub pagsub] = ind2sub(dims, indices)
rowsub =
     1     2     1     2     1     2     1     2
colsub =
     1     1     2     2     1     1     2     2
pagsub =
     1     1     1     1     2     2     2     2
```

If you specify only two outputs (row and column), `ind2sub` still returns a subscript for each specified index, but drops the third dimension from the matrix, returning subscripts for a 2-dimensional, 2-by-4 matrix instead:

```
[rowsub colsub] = ind2sub(dims, indices)
rowsub =
     1     2     1     2     1     2     1     2
colsub =
     1     1     2     2     3     3     4     4
```

If you specify one output (row), `ind2sub` drops both the second and third dimensions from the matrix, and returns subscripts for a 1-dimensional, 1-by-8 matrix instead:

```
[rowsub] = ind2sub(dims, indices)
rowsub =
     1     2     3     4     5     6     7     8
```

See Also

`find`, `size`, `sub2ind`

Purpose

Infinity

Syntax

```
Inf
Inf('double')
Inf('single')
Inf(n)
Inf(m,n)
Inf(m,n,p,...)
Inf(...,classname)
```

Description

`Inf` returns the IEEE arithmetic representation for positive infinity. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values.

`Inf('double')` is the same as `Inf` with no inputs.

`Inf('single')` is the single precision representation of `Inf`.

`Inf(n)` is an `n`-by-`n` matrix of `Inf`s.

`Inf(m,n)` or `inf([m,n])` is an `m`-by-`n` matrix of `Inf`s.

`Inf(m,n,p,...)` or `Inf([m,n,p,...])` is an `m`-by-`n`-by-`p`-by-... array of `Inf`s.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`Inf(...,classname)` is an array of `Inf`s of class specified by `classname`. `classname` must be either `'single'` or `'double'`.

Examples

`1/0`, `1.e1000`, `2^2000`, and `exp(1000)` all produce `Inf`.

`log(0)` produces `-Inf`.

`Inf-Inf` and `Inf/Inf` both produce `NaN` (Not-a-Number).

See Also `isinf`, `NaN`

Purpose

Specify inferior class relationship

Syntax

`inferiorto('class1','class2',...)`

Description

`inferiorto('class1','class2',...)` establishes that the class invoking this function in its constructor has lower precedence than the classes in the argument list. MATLAB uses this precedence to determines which method or function MATLAB calls in any given situation.

Use this function only from a constructor that calls the `class` function to create objects (classes defined before MATLAB 7.6).

Examples

Specify class precedence.

Suppose `a` is an object of class `class_a`, `b` is an object of class `class_b`, and `c` is an object of class `class_c`. Suppose the constructor method of `class_c` contains the statement:

```
inferiorto('class_a')
```

This function call establishes `class_a` as taking precedence over `class_c` for function dispatching. Therefore, either of the following two statements:

```
e = fun(a,c);
e = fun(c,a);
```

Invoke `class_a/fun`.

If you call a function with two objects having an unspecified relationship, the two objects have equal precedence. In this case, MATLAB calls the method of the left-most object. So `fun(b, c)` calls `class_b/fun`, while `fun(c, b)` calls `class_c/fun`.

See Also

`superiorto`

info

Purpose Information about contacting The MathWorks

Syntax `info`

Description `info` displays in the Command Window, information about contacting The MathWorks.

See Also `help`, `version`

Purpose

Construct inline object

Syntax

```
inline(expr)
inline(expr,arg1,arg2,...)
inline(expr,n)
```

Description

`inline(expr)` constructs an inline function object from the MATLAB expression contained in the string `expr`. The input argument to the inline function is automatically determined by searching `expr` for an isolated lower case alphabetic character, other than `i` or `j`, that is not part of a word formed from several alphabetic characters. If no such character exists, `x` is used. If the character is not unique, the one closest to `x` is used. If two characters are found, the one later in the alphabet is chosen.

`inline(expr,arg1,arg2,...)` constructs an inline function whose input arguments are specified by the strings `arg1`, `arg2`,.... Multicharacter symbol names may be used.

`inline(expr,n)` where `n` is a scalar, constructs an inline function whose input arguments are `x`, `P1`, `P2`, ...

Remarks

Three commands related to `inline` allow you to examine an inline function object and determine how it was created.

`char(fun)` converts the inline function into a character array. This is identical to `formula(fun)`.

`argnames(fun)` returns the names of the input arguments of the inline object `fun` as a cell array of strings.

`formula(fun)` returns the formula for the inline object `fun`.

A fourth command `vectorize(fun)` inserts a `.` before any `^`, `*` or `/`' in the formula for `fun`. The result is a vectorized version of the inline function.

Examples

Example 1

This example creates a simple inline function to square a number.

```
g = inline('t^2')
g =

     Inline function:
     g(t) = t^2
```

You can convert the result to a string using the `char` function.

```
char(g)

ans =

t^2
```

Example 2

This example creates an inline function to represent the formula $f = 3\sin(2x^2)$. The resulting inline function can be evaluated with the `argnames` and `formula` functions.

```
f = inline('3*sin(2*x.^2)')

f =
     Inline function:
     f(x) = 3*sin(2*x.^2)

argnames(f)

ans =
    'x'

formula(f)
ans =

3*sin(2*x.^2)
```

Example 3

This call to `inline` defines the function `f` to be dependent on two variables, `alpha` and `x`:

```
f = inline('sin(alpha*x)')

f =
     Inline function:
     f(alpha,x) = sin(alpha*x)
```

If `inline` does not return the desired function variables or if the function variables are in the wrong order, you can specify the desired variables explicitly with the `inline` argument list.

```
g = inline('sin(alpha*x)','x','alpha')

g =

     Inline function:
     g(x,alpha) = sin(alpha*x)
```

inmem

Purpose

Names of M-files, MEX-files, Sun Java classes in memory

Syntax

```
M = inmem
[M, X] = inmem
[M, X, J] = inmem
[...] = inmem('-completenames')
```

Description

`M = inmem` returns a cell array of strings containing the names of the M-files that are currently loaded.

`[M, X] = inmem` returns an additional cell array `X` containing the names of the MEX-files that are currently loaded.

`[M, X, J] = inmem` also returns a cell array `J` containing the names of the Java classes that are currently loaded.

`[...]  = inmem('-completenames')` returns not only the names of the currently loaded M- and MEX-files, but the path and filename extension for each as well. No additional information is returned for loaded Java classes.

Examples

Example 1

This example lists the M-files that are required to run `erf`.

```
clear all;        % Clear the workspace
erf(0.5);

M = inmem
M =
   'erf'
```

Example 2

Generate a plot, and then find the M- and MEX-files that had been loaded to perform this operation:

```
clear all
surf(peaks)
```

```
[m x] = inmem('-completenames');

m{1:5}
ans =
   F:\matlab\toolbox\matlab\general\usejava.m
ans =
   F:\matlab\toolbox\matlab\graph3d\private\surfchk.m
ans =
   F:\matlab\toolbox\matlab\graphics\gcf.m
ans =
   F:\matlab\toolbox\matlab\datatypes\@opaque\char.m
ans =
   F:\matlab\toolbox\matlab\graphics\findall.m

x
x =
   Empty cell array: 0-by-1
```

See Also

`clear`

inpolygon

Purpose Points inside polygonal region

Syntax

```
IN = inpolygon(X,Y,xv,yv)
[IN ON] = inpolygon(X,Y,xv,yv)
```

Description `IN = inpolygon(X,Y,xv,yv)` returns a matrix `IN` the same size as `X` and `Y`. Each element of `IN` is assigned the value 1 or 0 depending on whether the point `(X(p,q),Y(p,q))` is inside the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

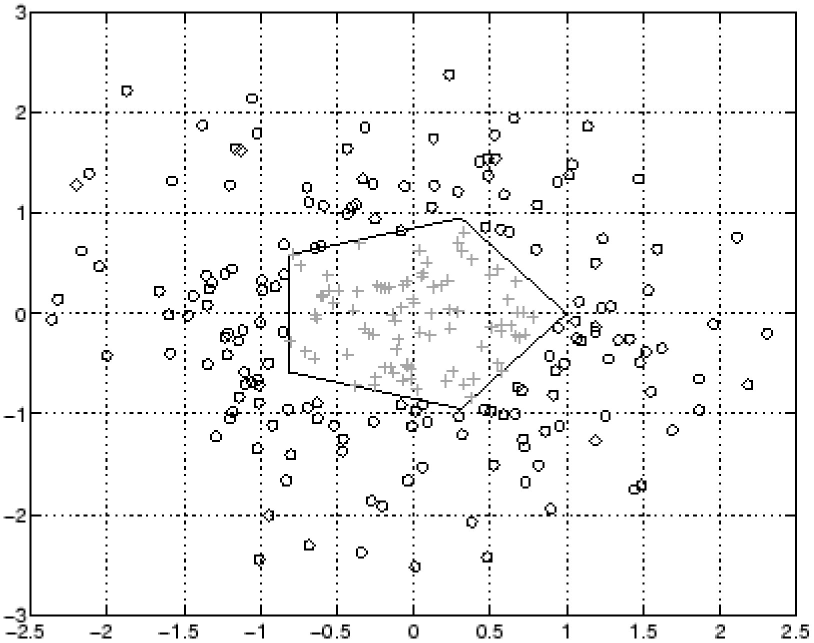
`IN(p,q) = 1`	If `(X(p,q),Y(p,q))` is inside the polygonal region or on the polygon boundary
`IN(p,q) = 0`	If `(X(p,q),Y(p,q))` is outside the polygonal region

`[IN ON] = inpolygon(X,Y,xv,yv)` returns a second matrix `ON` the same size as `X` and `Y`. Each element of `ON` is assigned the value 1 or 0 depending on whether the point `(X(p,q),Y(p,q))` is on the boundary of the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

`ON(p,q) = 1`	If `(X(p,q),Y(p,q))` is on the polygon boundary
`ON(p,q) = 0`	If `(X(p,q),Y(p,q))` is inside or outside the polygon boundary

Examples

```
L = linspace(0,2.*pi,6); xv = cos(L)';yv = sin(L)';
xv = [xv ; xv(1)]; yv = [yv ; yv(1)];
x = randn(250,1); y = randn(250,1);
in = inpolygon(x,y,xv,yv);
plot(xv,yv,x(in),y(in),'r+',x(~in),y(~in),'bo')
```


3
2
1
0
−1
−2
−3
−2.5
−2
−1.5
−1
−0.5
0
0.5
1
1.5
2
2.5

input

Purpose

Request user input

Syntax

```
user_entry = input('prompt')
user_entry = input('prompt', 's')
```

Description

The response to the `input` prompt can be any MATLAB expression, which is evaluated using the variables in the current workspace.

`user_entry = input('`*`prompt`*`')` displays *`prompt`* as a prompt on the screen, waits for input from the keyboard, and returns the value entered in `user_entry`.

`user_entry = input('`*`prompt`*`', 's')` returns the entered string as a text variable rather than as a variable name or numerical value.

Remarks

If you press the **Return** key without entering anything, `input` returns an empty matrix.

The text string for the prompt can contain one or more `'\n'` characters. The `'\n'` means to skip to the next line. This allows the prompt string to span several lines. To display just a backslash, use `'\\'`.

If you enter an invalid expression at the prompt, MATLAB displays the relevant error message and then prompts you again to enter input.

Examples

Press **Return** to select a default value by detecting an empty matrix:

```
reply = input('Do you want more? Y/N [Y]: ', 's');
if isempty(reply)
    reply = 'Y';
end
```

See Also

`keyboard`, `menu`, `ginput`, `uicontrol`

Purpose

Create and open input dialog box

Syntax

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,dlg_title)
answer = inputdlg(prompt,dlg_title,num_lines)
answer = inputdlg(prompt,dlg_title,num_lines,defAns)
answer = inputdlg(prompt,dlg_title,num_lines,defAns,options)
```

Description

`answer = inputdlg(prompt)` creates a modal dialog box and returns user input for multiple prompts in the cell array. `prompt` is a cell array containing prompt strings.

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

`answer = inputdlg(prompt,dlg_title)` `dlg_title` specifies a title for the dialog box.

`answer = inputdlg(prompt,dlg_title,num_lines)` `num_lines` specifies the number of lines for each user-entered value. `num_lines` can be a scalar, column vector, or matrix.

- If `num_lines` is a scalar, it applies to all prompts.
- If `num_lines` is a column vector, each element specifies the number of lines of input for a prompt.
- If `num_lines` is a matrix, it should be size m-by-2, where m is the number of prompts on the dialog box. Each row refers to a prompt. The first column specifies the number of lines of input for a prompt. The second column specifies the width of the field in characters.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns)` `defAns` specifies the default value to display for each prompt. `defAns` must contain the same number of elements as `prompt` and all elements must be strings.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns,options)` If `options` is the string `'on'`, the dialog is made resizable in the horizontal direction. If options is a structure, the fields shown in the following table are recognized:

Field	Description
`Resize`	Can be `'on'` or `'off'` (default). If `'on'`, the window is resizable horizontally.
`WindowStyle`	Can be either `'normal'` or `'modal'` (default).
`Interpreter`	Can be either `'none'` (default) or `'tex'`. If the value is `'tex'`, the prompt strings are rendered using LaTeX.

If the user clicks the **Cancel** button to close an `inputdlg` box, the dialog returns an empty cell array:

```
answer =
     {}
```

Remarks

`inputdlg` uses the `uiwait` function to suspend execution until the user responds.

The returned variable `answer` is a cell array containing strings, one string per text entry field, starting from the top of the dialog box.

To convert a member of the cell array to a number, use `str2num`. To do this, you can add the following code to the end of any of the examples below:

```
[val status] = str2num(answer{1});  % Use curly bracket for subscript
if ~status
    % Handle empty value returned for unsuccessful conversion
    % ...
end
```

```
% val is a scalar or matrix converted from the first input
```

Users can enter scalar or vector values into `inputdlg` fields; `str2num` converts space- and comma-delimited strings into row vectors, and semicolon-delimited strings into column vectors. For example, if `answer{1}` contains `'1 2 3;4 -5 6+7i'`, the conversion produces:

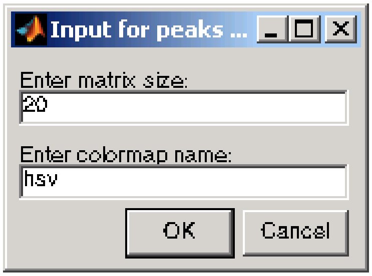
```
val = str2num(answer{1})
val =
   1.0000      2.0000      3.0000
   4.0000     -5.0000      6.0000 + 7.0000i
```

Example

Example 1

Create a dialog box to input an integer and colormap name. Allow one line for each value.

```
prompt = {'Enter matrix size:','Enter colormap name:'};
dlg_title = 'Input for peaks function';
num_lines = 1;
def = {'20','hsv'};
answer = inputdlg(prompt,dlg_title,num_lines,def);
```



Enter matrix size: 20

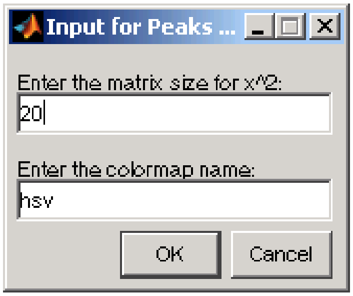
Enter colormap name: hsv

Example 2

Create a dialog box using the default options. Then use the options to make it resizable and not modal, and to interpret the text using LaTeX.

```
prompt={'Enter the matrix size for x^2:',...
        'Enter the colormap name:'};
```

```
name='Input for Peaks function';
numlines=1;
defaultanswer={'20','hsv'};
answer=inputdlg(prompt,name,numlines,defaultanswer);
```



Enter the matrix size for x^2:

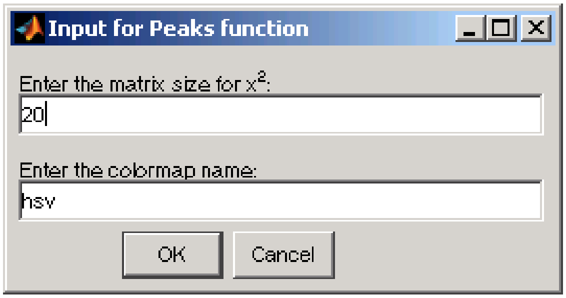
20

Enter the colormap name:

hsv

```
options.Resize='on';
options.WindowStyle='normal';
options.Interpreter='tex';

answer=inputdlg(prompt,name,numlines,defaultanswer,options);
```



Enter the matrix size for x^2:

20

Enter the colormap name:

hsv

See Also

`dialog`, `errordlg`, `helpdlg`, `listdlg`, `msgbox`, `questdlg`, `warndlg`

`figure`, `str2num`, `uiwait`, `uiresume`

for related functions

Purpose

Variable name of function input

Syntax

`inputname(argnum)`

Description

This command can be used only inside the body of a function.

`inputname(argnum)` returns the workspace variable name corresponding to the argument number *`argnum`*. If the input argument has no name (for example, if it is an expression instead of a variable), the `inputname` command returns the empty string (`''`).

Examples

Suppose the function `myfun.m` is defined as

```
function c = myfun(a,b)
fprintf('First calling variable is "%s"\n.', inputname(1))
```

Then

```
x = 5;  y = 3;  myfun(x,y)
```

produces

```
First calling variable is "x".
```

But

```
myfun(pi+1, pi-1)
```

produces

```
First calling variable is "".
```

See Also

`nargin`, `nargout`, `nargchk`

Purpose

Construct input parser object

Syntax

```
p = inputParser
```

Description

`p = inputParser` constructs an empty `inputParser` object. Use this utility object to parse and validate input arguments to the functions that you develop. The input parser object follows handle semantics; that is, methods called on it affect the original object, not a copy of it.

The MATLAB software configures `inputParser` objects to recognize an input schema. Use any of the following methods to create the schema for parsing a particular function.

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

Methods

Method	Description
`addOptional`	Add an optional argument to the schema
`addParamValue`	Add a parameter-value pair argument to the schema
`addRequired`	Add a required argument to the schema
`createCopy`	Create a copy of the `inputParser` object
`parse`	Parse and validate the named inputs

Properties

Property	Description
`CaseSensitive`	Enable or disable case-sensitive matching of argument names
`FunctionName`	Function name to be included in error messages
`KeepUnmatched`	Enable or disable errors on unmatched arguments

Property	Description
`Parameters`	Names of arguments defined in `inputParser` schema
`Results`	Names and values of arguments passed in function call that are in the schema for this function
`StructExpand`	Enable or disable passing arguments in a structure
`Unmatched`	Names and values of arguments passed in function call that are not in the schema for this function
`UsingDefaults`	Names of arguments not passed in function call that are given default values

Property Descriptions

Properties of the `inputParser` class are described below.

CaseSensitive

Purpose — Enable or disable case sensitive matching of argument names

`p.CaseSensitive = TF` enables or disables case-sensitivity when matching entries in the argument list with argument names in the schema. Set `CaseSensitive` to logical 1 (`true`) to enable case-sensitive matching, or to logical 0 (`false`) to disable it. By default, case-sensitive matching is disabled.

FunctionName

Purpose — Function name to be included in error messages

`p.FunctionName = name` stores a function `name` that is to be included in error messages that might be thrown in the process of validating input arguments to the function. The `name` input is a string containing the name of the function for which you are parsing inputs with `inputParser`.

KeepUnmatched

Purpose — Enable or disable errors on unmatched arguments

`p.KeepUnmatched = TF` controls whether MATLAB throws an error when the function being called is passed an argument that has not been defined in the `inputParser` schema for this file. When this property is set to logical 1 (`true`), MATLAB does not throw an error, but instead stores the names and values of unmatched arguments in the `Unmatched` property of object p. When `KeepUnmatched` is set to logical 0 (false), MATLAB does throw an error whenever this condition is encountered and the `Unmatched` property is not affected.

Parameters

Purpose — Names of arguments defined in `inputParser` schema

`c = p.Parameters` is a cell array of strings containing the names of those arguments currently defined in the schema for the object. Each row of the `Parameters` cell array is a string containing the full name of a known argument.

Results

Purpose — Names and values of arguments passed in function call that are in the schema for this function

`arglist = p.Results` is a structure containing the results of the most recent parse of the input argument list. Each argument passed to the function is represented by a field in the `Results` structure, and the value of that argument is represented by the value of that field.

StructExpand

Purpose — Enable or disable passing arguments in a structure

`p.StructExpand = TF`, when set to logical 1 (`true`), tells MATLAB to accept a structure as an input in place of individual parameter-value arguments. If `StructExpand` is set to logical 0 (`false`), a structure is treated as a regular, single input.

Unmatched

Purpose — Names and values of arguments passed in function call that are not in the schema for this function

`c = p.Unmatched` is a structure array containing the names and values of all arguments passed in a call to the function that are not included in the schema for the function. `Unmatched` only contains this list of the `KeepUnmatched` property is set to true. If `KeepUnmatched` is set to `false`, MATLAB throws an error when unmatched arguments are passed in the function call. The `Unmatched` structure has the same format as the `Results` property of the `inputParser` class.

UsingDefaults

Purpose — Names of arguments not passed in function call that are given default values

`defaults = p.UsingDefaults` is a cell array of strings containing the names of those arguments that were not passed in the call to this function and consequently are set to their default values.

Examples

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

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

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

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

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

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

Execution of the `parse` method validates each argument and also builds a structure from the input arguments. The name of the structure is `Results`, which is accessible as a property of the object. To get the value of any input argument, type

```
p.Results.argname
```

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

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

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

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

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

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

The maximum height is 300.

List of all arguments:
              format: 'ppt'
```

```
 maxHeight: 300
  maxWidth: 500
 outputDir: 'C:/matlab/test'
    script: 'ipscript.m'
```

See Also

`addRequired(inputParser)`, `addOptional(inputParser)`, `addParamValue(inputParser)`, `parse(inputParser)`, `createCopy(inputParser)`, `validateattributes`, `validatestring`, `varargin`, `nargchk`, `nargin`

Purpose

Open Property Inspector

Syntax

```
inspect
inspect(h)
inspect([h1,h2,...])
```

Description

`inspect` creates a separate Property Inspector window to enable the display and modification of the properties of any object you select in the figure window or Layout Editor. If no object is selected, the Property Inspector is blank.

`inspect(h)` creates a Property Inspector window for the object whose handle is `h`.

`inspect([h1,h2,...])` displays properties that objects `h1` and `h2` have in common, or a blank window if there are no such properties; any number of objects can be inspected and edited in this way (for example, handles returned by the `bar` command).

The Property Inspector has the following behaviors:

- Only one Property Inspector window is active at any given time; when you inspect a new object, its properties replace those of the object last inspected.
- When the Property Inspector is open and plot edit mode is on, clicking any object in the figure window displays the properties of that object (or set of objects) in the Property Inspector.
- When you select and inspect two or more objects of different types, the Property Inspector only shows the properties that all objects have in common.
- To change the value of any property, click on the property name shown at the left side of the window, and then enter the new value in the field at the right.

The Property Inspector provides two different views:

- List view — properties are ordered alphabetically (default); this is the only view available for annotation objects.
- Group view — properties are grouped under classified headings (Handle Graphics objects only)

To view alphabetically, click the “AZ” Icon in the Property Inspector toolbar. To see properties in groups, click

the “++” icon. When properties are grouped, the “-” and “+” icons are enabled; click to expand all categories and click to collapse all categories. You can also expand and collapse individual categories by clicking on the “+” next to the category name. Some properties expand and collapse

---

Notes To see a complete description of any property, right-click on its name or value and select **What’s This**; a help window opens that displays the reference page entry for it.

The Property Inspector displays most, but not all, properties of Handle Graphics objects. For example, the `parent` and `children` of HG objects are not shown.
`inspect h` displays a Property Inspector window that enables modification of the string `'h'`, not the object whose handle is `h`.
If you modify properties at the MATLAB command line, you must refresh the Property Inspector window to see the change reflected there. Refresh the Property Inspector by reinvoking `inspect` on the object.

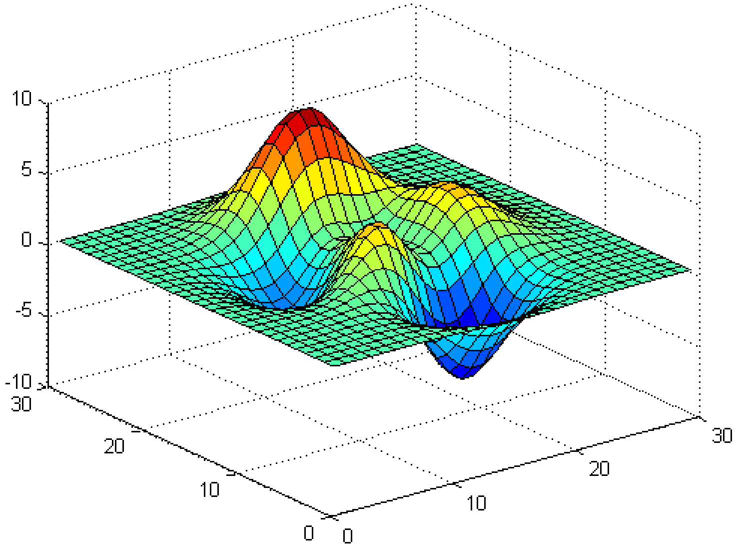
Examples

Example 1

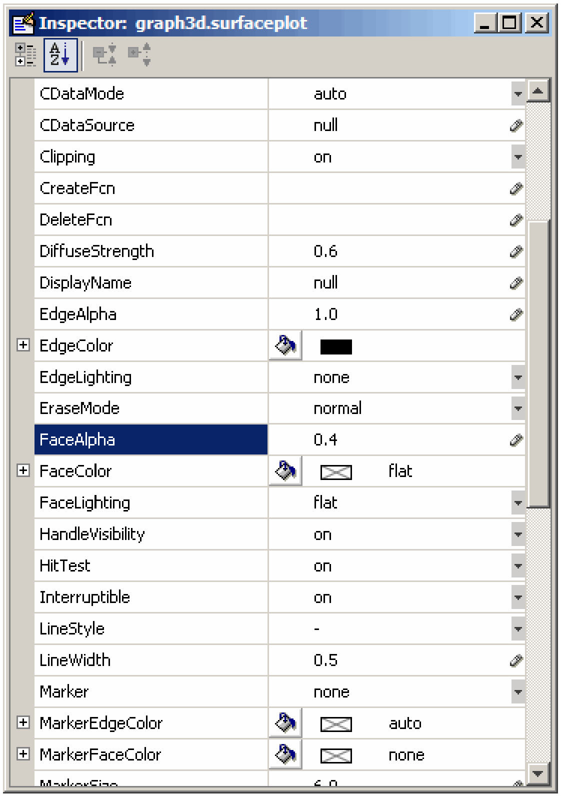
Create a surface mesh plot and view its properties with the Property Inspector:

```
Z = peaks(30);
h = surf(Z)
```

```
inspect(h)
```

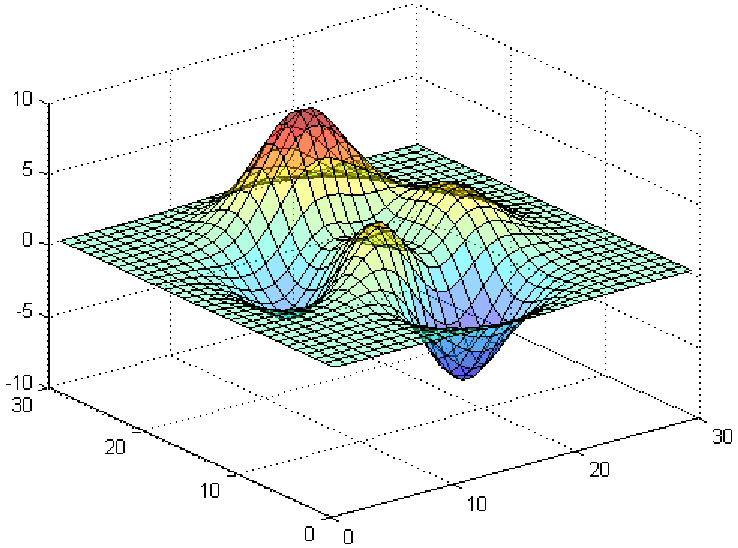


Use the Property Inspector to change the `FaceAlpha` property from `1.0` to `0.4` (equivalent to the command `set(h,'FaceAlpha',0.4)`). `FaceAlpha` controls the transparency of patch faces.



CDataMode	auto
CDataSource	null
Clipping	on
CreateFcn	
DeleteFcn	
DiffuseStrength	0.6
DisplayName	null
EdgeAlpha	1.0
EdgeColor	■
EdgeLighting	none
EraseMode	normal
FaceAlpha	0.4
FaceColor	☒ flat
FaceLighting	flat
HandleVisibility	on
HitTest	on
Interruptible	on
LineStyle	-
LineWidth	0.5
Marker	none
MarkerEdgeColor	☒ auto
MarkerFaceColor	☒ none

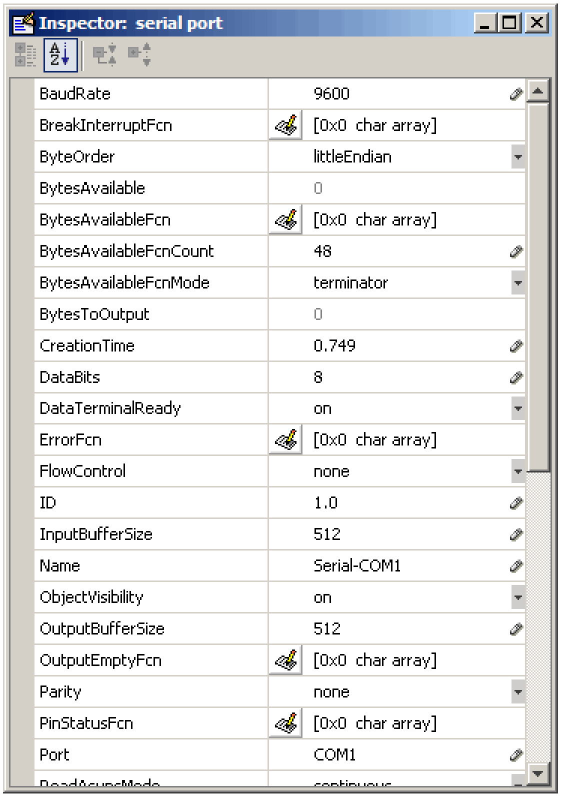
When you press **Enter** or click a different field, the `FaceAlpha` property of the surface object is updated:



Example 2

Create a serial port object for `COM1` on a Windows platform and use the Property Inspector to peruse its properties:

```
s = serial('COM1');
inspect(s)
```



BaudRate	9600
BreakInterruptFcn	[0x0 char array]
ByteOrder	littleEndian
BytesAvailable	0
BytesAvailableFcn	[0x0 char array]
BytesAvailableFcnCount	48
BytesAvailableFcnMode	terminator
BytesToOutput	0
CreationTime	0.749
DataBits	8
DataTerminalReady	on
ErrorFcn	[0x0 char array]
FlowControl	none
ID	1.0
InputBufferSize	512
Name	Serial-COM1
ObjectVisibility	on
OutputBufferSize	512
OutputEmptyFcn	[0x0 char array]
Parity	none
PinStatusFcn	[0x0 char array]
Port	COM1
ReadAsyncMode	continuous

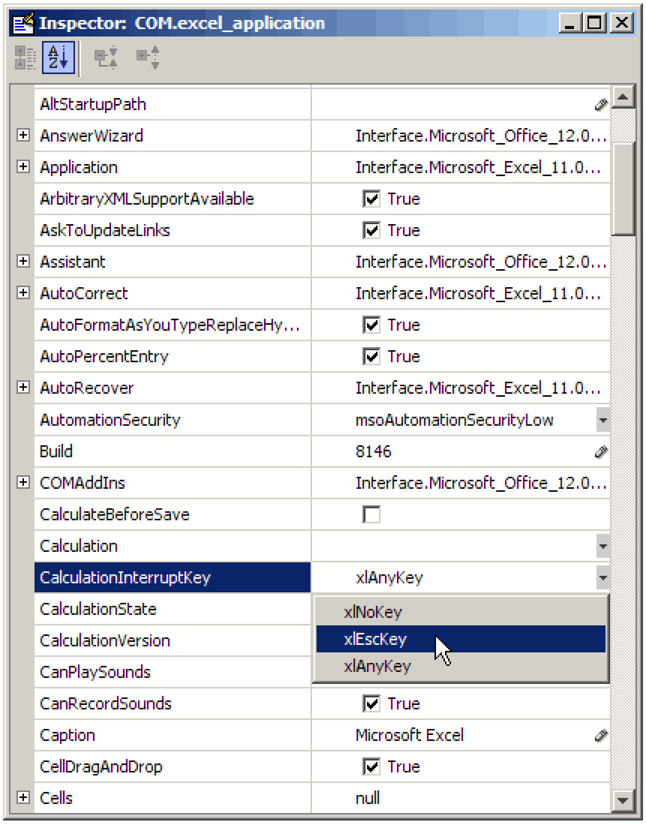
Because COM objects do not define property groupings, only the alphabetical list view of their properties is available.

Example 3

Create a COM Excel server and open a Property Inspector window with `inspect`:

```
h = actxserver('excel.application');
inspect(h)
```

Scroll down until you see the `CalculationInterruptKey` property, which by default is `xlAnyKey`. Click on the down-arrow in the right margin of the property inspector and select `xlEscKey` from the drop-down menu, as shown below:



	AltStartupPath	
⊞	AnswerWizard	Interface.Microsoft_Office_12.0...
⊞	Application	Interface.Microsoft_Excel_11.0...
	ArbitraryXMLSupportAvailable	☑ True
	AskToUpdateLinks	☑ True
⊞	Assistant	Interface.Microsoft_Office_12.0...
⊞	AutoCorrect	Interface.Microsoft_Excel_11.0...
	AutoFormatAsYouTypeReplaceHy...	☑ True
	AutoPercentEntry	☑ True
⊞	AutoRecover	Interface.Microsoft_Excel_11.0...
	AutomationSecurity	msoAutomationSecurityLow
	Build	8146
⊞	COMAddIns	Interface.Microsoft_Office_12.0...
	CalculateBeforeSave	☐
	Calculation	
	CalculationInterruptKey	xlAnyKey
	CalculationState	
	CalculationVersion	
	CanPlaySounds	
	CanRecordSounds	☑ True
	Caption	Microsoft Excel
	CellDragAndDrop	☑ True
⊞	Cells	null

Check this field in the MATLAB command window using `get` to confirm that it has changed:

```
get(h,'CalculationInterruptKey')

ans =
xlEscKey
```

See Also `get`, `set`, `isprop`, `guide`, `addproperty`, `deleteproperty`

instrcallback

Purpose Event information when event occurs

Syntax `instrcallback(obj,event)`

Description `instrcallback(obj,event)` displays a message that contains the event type, `event`, the time the event occurred, and the name of the serial port object, `obj`, that caused the event to occur.

For error events, the error message is also displayed. For pin status events, the pin that changed value and its value are also displayed.

Remarks You should use `instrcallback` as a template from which you create callback functions that suit your specific application needs.

Example The following example creates the serial port objects `s`, on a Windows platform. It configures `s` to execute `instrcallback` when an output-empty event occurs. The event occurs after the `*IDN?` command is written to the instrument.

```
s = serial('COM1');
set(s,'OutputEmptyFcn',@instrcallback)
fopen(s)
fprintf(s,'*IDN?','async')
```

The resulting display from `instrcallback` is shown below.

```
OutputEmpty event occurred at 08:37:49 for the object:
Serial-COM1.
```

Read the identification information from the input buffer and end the serial port session.

```
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

Purpose

Read serial port objects from memory to MATLAB workspace

Syntax

```
out = instrfind
out = instrfind('PropertyName',PropertyValue,...)
out = instrfind(S)
out = instrfind(obj,'PropertyName',PropertyValue,...)
```

Description

`out = instrfind` returns all valid serial port objects as an array to `out`.

`out = instrfind('`*`PropertyName`*`',PropertyValue,...)` returns an array of serial port objects whose property names and property values match those specified.

`out = instrfind(S)` returns an array of serial port objects whose property names and property values match those defined in the structure `S`. The field names of `S` are the property names, while the field values are the associated property values.

`out = instrfind(obj,'`*`PropertyName`*`',PropertyValue,...)` restricts the search for matching property name/property value pairs to the serial port objects listed in `obj`.

Remarks

Refer to for a list of serial port object properties that you can use with `instrfind`.

You must specify property values using the same format as the `get` function returns. For example, if `get` returns the `Name` property value as `MyObject`, `instrfind` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of string values. For example, `instrfind` will find an object with a `Parity` property value of `Even` or `even`.

You can use property name/property value string pairs, structures, and cell array pairs in the same call to `instrfind`.

Example

Suppose you create the following two serial port objects on a Windows platform.

```
s1 = serial('COM1');
```

```
s2 = serial('COM2');
set(s2,'BaudRate',4800)
fopen([s1 s2])
```

You can use `instrfind` to return serial port objects based on property values.

```
out1 = instrfind('Port','COM1');
out2 = instrfind({'Port','BaudRate'},{'COM2',4800});
```

You can also use `instrfind` to return cleared serial port objects to the MATLAB workspace.

```
clear s1 s2
newobjs = instrfind

   Instrument Object Array
   Index:   Type:         Status:     Name:
   1        serial        open        Serial-COM1
   2        serial        open        Serial-COM2
```

To close both `s1` and `s2`

```
fclose(newobjs)
```

See Also

Functions

`clear`, `get`

Purpose

Find visible and hidden serial port objects

Syntax

```
out = instrfindall
out = instrfindall('P1',V1,...)
out = instrfindall(s)
out = instrfindall(objs,'P1',V1,...)
```

Description

`out = instrfindall` finds all serial port objects, regardless of the value of the objects' `ObjectVisibility` property. The object or objects are returned to `out`.

`out = instrfindall('P1',V1,...)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified as arguments.

`out = instrfindall(s)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified in the structure `s`, where the field names correspond to property names and the field values correspond to the current value of the respective property.

`out = instrfindall(objs,'P1',V1,...)` restricts the search for objects with matching property name/value pairs to the serial port objects listed in `objs`.

Note that you can use string property name/property value pairs, structures, and cell array property name/property value pairs in the same call to `instrfindall`.

Remarks

`instrfindall` differs from `instrfind` in that it finds objects whose `ObjectVisibility` property is set to `off`.

Property values are case sensitive. You must specify property values using the same format as that returned by the `get` function. For example, if `get` returns the `Name` property value as `'MyObject'`, `instrfindall` will not find an object with a `Name` property value of `'myobject'`. However, this is not the case for properties that have a finite set of string values. For example, `instrfindall` will find an object with a `Parity` property value of `'Even'` or `'even'`.

Examples

Suppose you create the following serial port objects on a Windows platform:

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s2,'ObjectVisibility','off')
```

Because object `s2` has its `ObjectVisibility` set to `'off'`, it is not visible to commands like `instrfind`:

```
instrfind

   Serial Port Object : Serial-COM1
```

However, `instrfindall` finds all objects regardless of the value of `ObjectVisibility`:

```
instrfindall

 Instrument Object Array
 Index:   Type:          Status:     Name:
 1        serial         closed      Serial-COM1
 2        serial         closed      Serial-COM2
```

The following statements use `instrfindall` to return objects with specific property settings, which are passed as cell arrays:

```
props = {'PrimaryAddress','SecondaryAddress};
vals = {2,0};
obj = instrfindall(props,vals);
```

You can use `instrfindall` as an argument when you want to apply the command to all objects, visible and invisible. For example, the following statement makes all objects visible:

```
set(instrfindall,'ObjectVisibility','on')
```

See Also

Functions

`get`, `instrfind`

Properties

`ObjectVisibility`

Purpose Convert integer to string

Syntax `str = int2str(N)`

Description `str = int2str(N)` converts an integer to a string with integer format. The input `N` can be a single integer or a vector or matrix of integers. Noninteger inputs are rounded before conversion.

Examples `int2str(2+3)` is the string `'5'`.

One way to label a plot is

```
title(['case number ' int2str(n)])
```

For matrix or vector inputs, `int2str` returns a string matrix:

```
int2str(eye(3))

ans =

1  0  0
0  1  0
0  0  1
```

See Also `fprintf`, `num2str`, `sprintf`

Purpose Convert to signed integer

Syntax

```
I = int8(X)
I = int16(X)
I = int32(X)
I = int64(X)
```

Description `I = int*(X)` converts the elements of array `X` into signed integers. `X` can be any numeric object (such as a `double`). The results of an `int*` operation are shown in the next table.

Operation	Output Range	Output Type	Bytes per Element	Output Class
`int8`	-128 to 127	Signed 8-bit integer	1	`int8`
`int16`	-32,768 to 32,767	Signed 16-bit integer	2	`int16`
`int32`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	4	`int32`
`int64`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	8	`int64`

`double` and `single` values are rounded to the nearest `int*` value on conversion. A value of `X` that is above or below the range for an integer class is mapped to one of the endpoints of the range. For example,

```
int16(40000)
ans =
   32767
```

If `X` is already a signed integer of the same class, then `int*` has no effect.

You can define or overload your own methods for `int*` (as you can for any object) by placing the appropriately named method in an `@int*` directory within a directory on your path. Type `help datatypes` for the names of the methods you can overload.

Remarks

Most operations that manipulate arrays without changing their elements are defined for integer values. Examples are `reshape`, `size`, the logical and relational operators, subscripted assignment, and subscripted reference.

Some arithmetic operations are defined for integer arrays on interaction with other integer arrays of the same class (e.g., where both operands are `int16`). Examples of these operations are `+`, `-`, `.*`, `./`, `.\` and `.^`. If at least one operand is scalar, then `*`, `/`, `\`, and `^` are also defined. Integer arrays may also interact with scalar `double` variables, including constants, and the result of the operation is an integer array of the same class. Integer arrays saturate on overflow in arithmetic.

Note Only the lower order integer data types support math operations. Math operations are not supported for `int64` and `uint64`.

A particularly efficient way to initialize a large array is by specifying the data type (i.e., class name) for the array in the `zeros`, `ones`, or `eye` function. For example, to create a 100-by-100 `int64` array initialized to zero, type

```
I = zeros(100, 100, 'int64');
```

An easy way to find the range for any MATLAB integer type is to use the `intmin` and `intmax` functions as shown here for `int32`:

```
intmin('int32')                      intmax('int32')
ans =                                ans =
   -2147483648                          2147483647
```

See Also `double`, `single`, `uint8`, `uint16`, `uint32`, `uint64`, `intmax`, `intmin`

interfaces

Purpose List custom interfaces exposed by COM server object

Syntax

```
customlist = h.interfaces
customlist = interfaces(h)
```

Description

`customlist = h.interfaces` returns cell array of strings `customlist` listing all custom interfaces implemented by the component in a specific COM server object. The server is designated by input argument `h`, the handle returned by the `actxcontrol` or `actxserver` function when creating that server.

`customlist = interfaces(h)` is an alternate syntax.

The `interfaces` function only lists the custom interfaces exposed by the object; it does not return interfaces. Use the `invoke` function to return a handle to a specific custom interface.

COM functions are available on Microsoft Windows systems only.

See Also `actxcontrol` | `actxserver` | `invoke` | `get (COM)`

How To

-

Purpose

1-D data interpolation (table lookup)

Syntax

```
yi = interp1(x,Y,xi)
yi = interp1(Y,xi)
yi = interp1(x,Y,xi,method)
yi = interp1(x,Y,xi,method,'extrap')
yi = interp1(x,Y,xi,method,extrapval)
pp = interp1(x,Y,method,'pp')
```

Description

`yi = interp1(x,Y,xi)` interpolates to find `yi`, the values of the underlying function `Y` at the points in the vector or array `xi`. `x` must be a vector. Y can be a scalar, a vector, or an array of any dimension, subject to the following conditions:

- If `Y` is a vector, it must have the same length as `x`. A scalar value for `Y` is expanded to have the same length as `x`. `xi` can be a scalar, a vector, or a multidimensional array, and `yi` has the same size as `xi`.
- If `Y` is an array that is not a vector, the size of `Y` must have the form `[n,d1,d2,...,dk]`, where `n` is the length of `x`. The interpolation is performed for each `d1`-by-`d2`-by-...-`dk` value in `Y`. The sizes of `xi` and `yi` are related as follows:
    - If `xi` is a scalar or vector, `size(yi)` equals `[length(xi), d1, d2, ..., dk]`.
    - If `xi` is an array of size `[m1,m2,...,mj]`, `yi` has size `[m1,m2,...,mj,d1,d2,...,dk]`.

`yi = interp1(Y,xi)` assumes that `x = 1:N`, where `N` is the length of `Y` for vector `Y`, or `size(Y,1)` for matrix `Y`.

`yi = interp1(x,Y,xi,method)` interpolates using alternative methods:

`’nearest’`	Nearest neighbor interpolation
`’linear’`	Linear interpolation (default)

`’spline’`	Cubic spline interpolation
`’pchip’`	Piecewise cubic Hermite interpolation
`’cubic’`	(Same as `’pchip'`)
`’v5cubic’`	Cubic interpolation used in MATLAB 5. This method does not extrapolate. Also, if `x` is not equally spaced, `'spline'` is used/

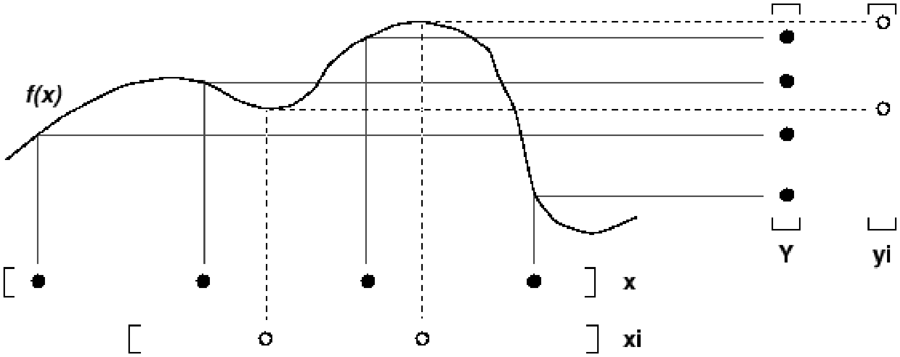
For the `'nearest'`, `'linear'`, and `'v5cubic'` methods, `interp1(x,Y,xi,method)` returns `NaN` for any element of `xi` that is outside the interval spanned by `x`. For all other methods, `interp1` performs extrapolation for out of range values.

`yi = interp1(x,Y,xi,method,'extrap')` uses the specified method to perform extrapolation for out of range values.

`yi = interp1(x,Y,xi,method,extrapval)` returns the scalar `extrapval` for out of range values. `NaN` and `0` are often used for `extrapval`.

`pp = interp1(x,Y,method,'pp')` uses the specified method to generate the piecewise polynomial form (ppform) of `Y`. You can use any of the methods in the preceding table, except for `'v5cubic'`. `pp` can then be evaluated via `ppval`. `ppval(pp,xi)` is the same as `interp1(x,Y,xi,method,'extrap')`.

The `interp1` command interpolates between data points. It finds values at intermediate points, of a one-dimensional function $f(x)$ that underlies the data. This function is shown below, along with the relationship between vectors `x`, `Y`, `xi`, and `yi`.



Interpolation is the same operation as *table lookup*. Described in table lookup terms, the *table* is `[x,Y]` and `interp1` *looks up* the elements of `xi` in `x`, and, based upon their locations, returns values `yi` interpolated within the elements of `Y`.

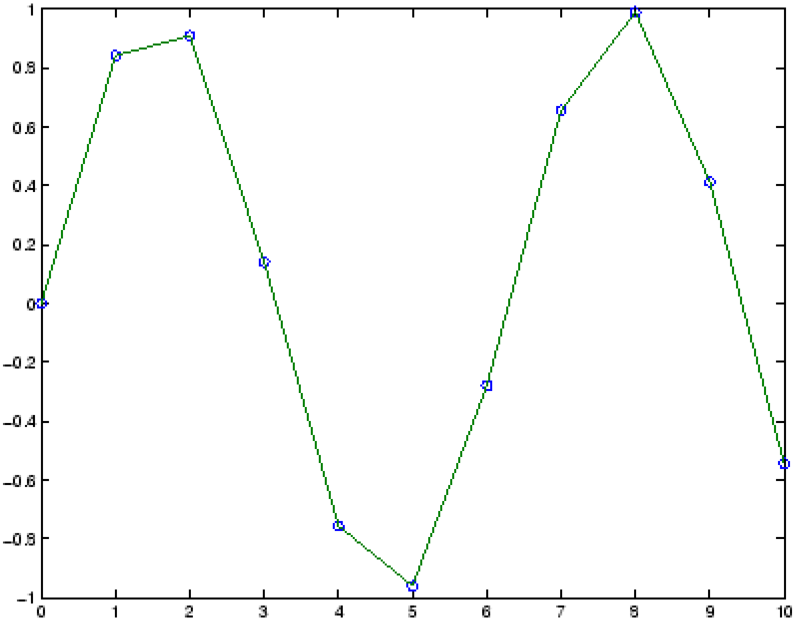
Note `interp1q` is quicker than `interp1` on non-uniformly spaced data because it does no input checking. For `interp1q` to work properly, `x` must be a monotonically increasing column vector and `Y` must be a column vector or matrix with `length(X)` rows. Type `help interp1q` at the command line for more information.

Examples

Example 1

Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = 0:10;
y = sin(x);
xi = 0:.25:10;
yi = interp1(x,y,xi);
plot(x,y,'o',xi,yi)
```



Example 2

The following multidimensional example creates 2-by-2 matrices of interpolated function values, one matrix for each of the three functions x^2, x^3, and x^4.

```
x = [1:10]'; y = [ x.^2, x.^3, x.^4 ];
xi = [1.5, 1.75; 7.5, 7.75];
yi = interp1(x,y,xi);
```

The result `yi` has size 2-by-2-by-3.

```
size(yi)

ans =

     2     2     3
```

Example 3

Here are two vectors representing the census years from 1900 to 1990 and the corresponding United States population in millions of people.

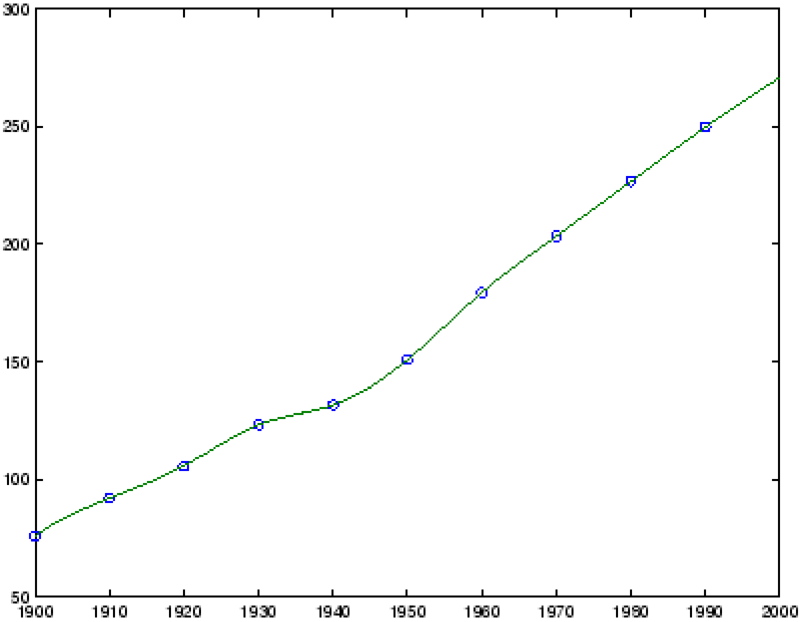
```
t = 1900:10:1990;
p = [75.995  91.972  105.711  123.203  131.669...
     150.697  179.323  203.212  226.505  249.633];
```

The expression `interp1(t,p,1975)` interpolates within the census data to estimate the population in 1975. The result is

```
ans =
    214.8585
```

Now interpolate within the data at every year from 1900 to 2000, and plot the result.

```
x = 1900:1:2000;
y = interp1(t,p,x,'spline');
plot(t,p,'o',x,y)
```



Sometimes it is more convenient to think of interpolation in table lookup terms, where the data are stored in a single table. If a portion of the census data is stored in a single 5-by-2 table,

```
tab =
    1950    150.697
    1960    179.323
    1970    203.212
    1980    226.505
    1990    249.633
```

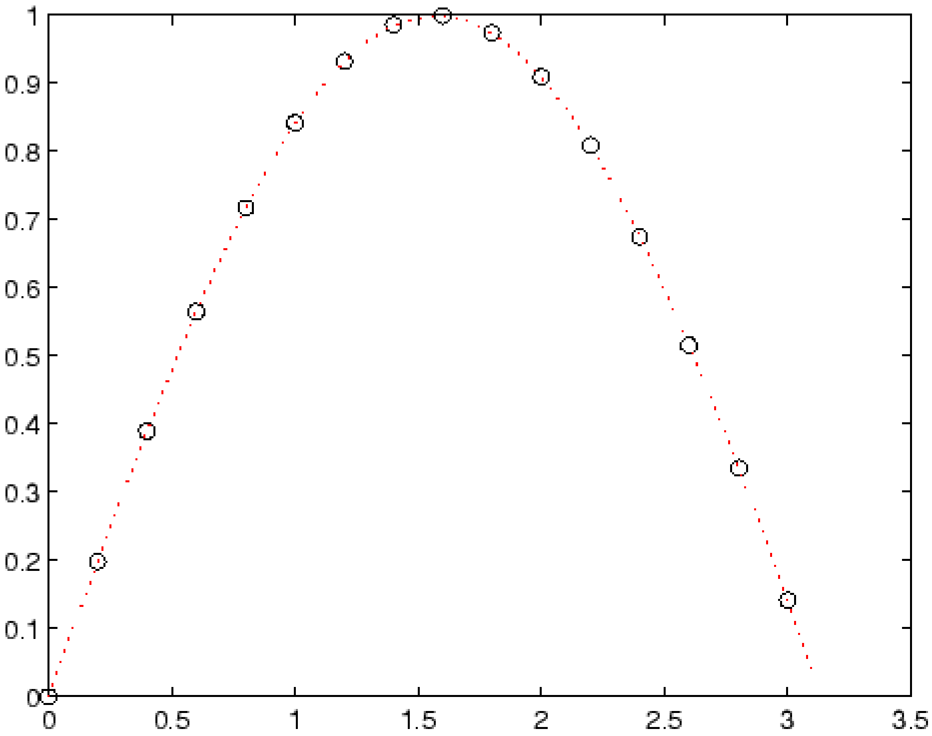
then the population in 1975, obtained by table lookup within the matrix `tab`, is

```
p = interp1(tab(:,1),tab(:,2),1975)
p =
    214.8585
```

Example 4

The following example uses the `'cubic'` method to generate the piecewise polynomial form (ppform) of `Y`, and then evaluates the result using `ppval`.

```
x = 0:.2:pi; y = sin(x);
pp = interp1(x,y,'cubic','pp');
xi = 0:.1:pi;
yi = ppval(pp,xi);
plot(x,y,'ko'), hold on, plot(xi,yi,'r:'), hold off
```



Algorithm

The `interp1` command is a MATLAB M-file. The ’`nearest'` and ’`linear'` methods have straightforward implementations.

For the ’`spline'` method, `interp1` calls a function `spline` that uses the functions `ppval`, `mkpp`, and `unmkpp`. These routines form a small suite of functions for working with piecewise polynomials. `spline` uses them to perform the cubic spline interpolation. For access to more advanced features, see the `spline` reference page, the M-file help for these functions, and the Spline Toolbox™ .

For the `'pchip'` and `'cubic'` methods, `interp1` calls a function `pchip` that performs piecewise cubic interpolation within the vectors `x` and `y`. This method preserves monotonicity and the shape of the data. See the `pchip` reference page for more information.

Interpolating Complex Data

For Real x and Complex Y. For `interp1(x,Y,...)` where `x` is real and `Y` is complex, you can use any `interp1` method except for `'pchip'`. The shape-preserving aspect of the `'pchip'` algorithm involves the signs of the slopes between the data points. Because there is no notion of sign with complex data, it is impossible to talk about whether a function is increasing or decreasing. Consequently, the `'pchip'` algorithm does not generalize to complex data.

The `'spline'` method is often a good choice because piecewise cubic splines are derived purely from smoothness conditions. The second derivative of the interpolant must be continuous across the interpolating points. This does not involve any notion of sign or shape and so generalizes to complex data.

For Complex x. For `interp1(x,Y,...)` where `x` is complex and `Y` is either real or complex, use the two-dimensional interpolation routine `interp2(REAL(x), IMAG(x),Y,...)` instead.

See Also

`interp1q`, `interpft`, `interp2`, `interp3`, `interpn`, `pchip`, `spline`

References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

Purpose

Quick 1-D linear interpolation

Syntax

`yi = interp1q(x,Y,xi)`

Description

`yi = interp1q(x,Y,xi)` returns the value of the 1-D function `Y` at the points of column vector `xi` using linear interpolation. The vector `x` specifies the coordinates of the underlying interval. The length of output `yi` is equal to the length of `xi`.

`interp1q` is quicker than `interp1` on non-uniformly spaced data because it does no input checking.

For `interp1q` to work properly,

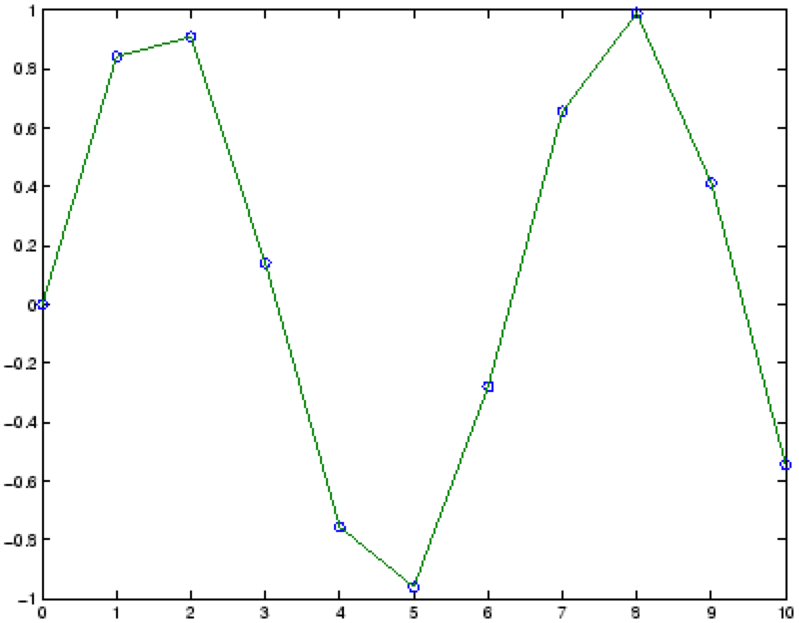
- `x` must be a monotonically increasing column vector.
- `Y` must be a column vector or matrix with `length(x)` rows.
- `xi` must be a column vector

`interp1q` returns NaN for any values of `xi` that lie outside the coordinates in `x`. If `Y` is a matrix, then the interpolation is performed for each column of `Y`, in which case `yi` is `length(xi)-by-size(Y,2)`.

Example

Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = (0:10)';
y = sin(x);
xi = (0:.25:10)';
yi = interp1q(x,y,xi);
plot(x,y,'o',xi,yi)
```



See Also `interp1`, `interp2`, `interp3`, `interpn`

Purpose

2-D data interpolation (table lookup)

Syntax

```
ZI = interp2(X,Y,Z,XI,YI)
ZI = interp2(Z,XI,YI)
ZI = interp2(Z,ntimes)
ZI = interp2(X,Y,Z,XI,YI,method)
ZI = interp2(...,method, extrapval)
```

Description

`ZI = interp2(X,Y,Z,XI,YI)` returns matrix `ZI` containing elements corresponding to the elements of `XI` and `YI` and determined by interpolation within the two-dimensional function specified by matrices `X`, `Y`, and `Z`. `X` and `Y` must be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. Matrices `X` and `Y` specify the points at which the data `Z` is given. Out of range values are returned as `NaN`s.

`XI` and `YI` can be matrices, in which case `interp2` returns the values of `Z` corresponding to the points `(XI(i,j),YI(i,j))`. Alternatively, you can pass in the row and column vectors `xi` and `yi`, respectively. In this case, `interp2` interprets these vectors as if you issued the command `meshgrid(xi,yi)`.

`ZI = interp2(Z,XI,YI)` assumes that `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`.

`ZI = interp2(Z,ntimes)` expands `Z` by interleaving interpolates between every element, working recursively for `ntimes`. `interp2(Z)` is the same as `interp2(Z,1)`.

`ZI = interp2(X,Y,Z,XI,YI,method)` specifies an alternative interpolation method:

`'nearest'`	Nearest neighbor interpolation
`'linear'`	Linear interpolation (default)

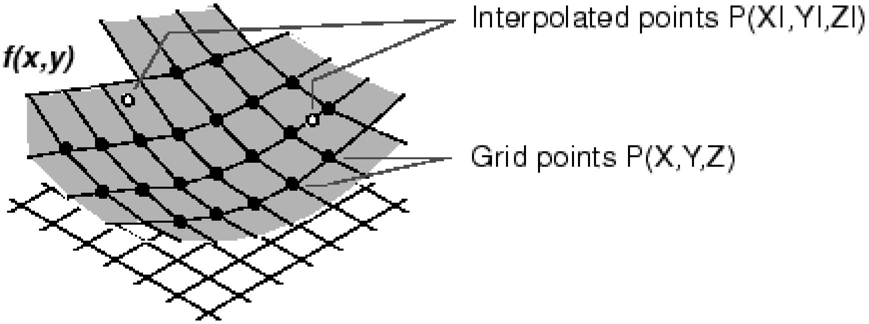
`'spline'`	Cubic spline interpolation
`'cubic'`	Cubic interpolation, as long as data is uniformly-spaced. Otherwise, this method is the same as `'spline'`.

All interpolation methods require that `X` and `Y` be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. If you provide two monotonic vectors, `interp2` changes them to a plaid internally. Variable spacing is handled by mapping the given values in `X`, `Y`, `XI`, and `YI` to an equally spaced domain before interpolating. For faster interpolation when `X` and `Y` are equally spaced and monotonic, use the methods `'*linear'`, `'*cubic'`, `'*spline'`, or `'*nearest'`.

`ZI = interp2(...,method, extrapval)` specifies a method and a scalar value for `ZI` outside of the domain created by `X` and `Y`. Thus, `ZI` equals `extrapval` for any value of `YI` or `XI` that is not spanned by `Y` or `X` respectively. A method must be specified to use `extrapval`. The default method is `'linear'`.

Remarks

The `interp2` command interpolates between data points. It finds values of a two-dimensional function $f(x, y)$ underlying the data at intermediate points.



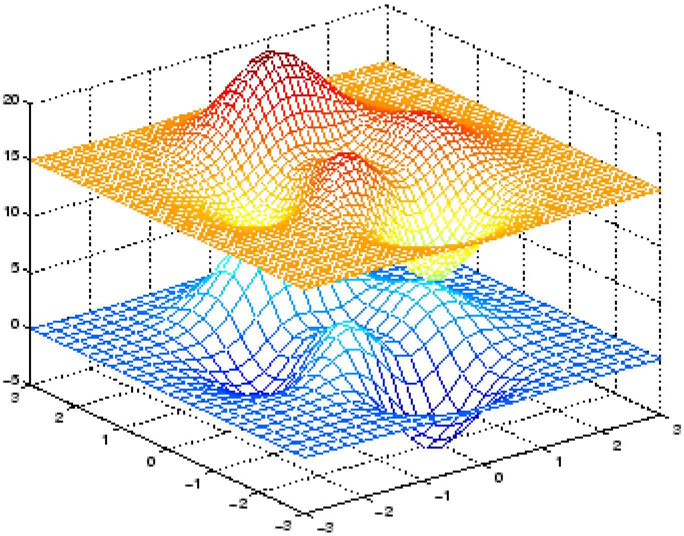
Interpolation is the same operation as table lookup. Described in table lookup terms, the table is `tab = [NaN,Y;X,Z]` and `interp2` looks up the elements of `XI` in `X`, `YI` in `Y`, and, based upon their location, returns values `ZI` interpolated within the elements of `Z`.

Examples

Example 1

Interpolate the `peaks` function over a finer grid.

```
[X,Y] = meshgrid(-3:.25:3);
Z = peaks(X,Y);
[XI,YI] = meshgrid(-3:.125:3);
ZI = interp2(X,Y,Z,XI,YI);
mesh(X,Y,Z), hold, mesh(XI,YI,ZI+15)
hold off
axis([-3 3 -3 3 -5 20])
```



Example 2

Given this set of employee data,

```
years = 1950:10:1990;
service = 10:10:30;
```

```
wage = [150.697 199.592 187.625
179.323 195.072 250.287
203.212 179.092 322.767
226.505 153.706 426.730
249.633 120.281 598.243];
```

it is possible to interpolate to find the wage earned in 1975 by an employee with 15 years' service:

```
w = interp2(service,years,wage,15,1975)
w =
  190.6287
```

See Also

`griddata`, `interp1`, `interp1q`, `interp3`, `interpn`, `meshgrid`

Purpose

3-D data interpolation (table lookup)

Syntax

```
VI = interp3(X,Y,Z,V,XI,YI,ZI)
VI = interp3(V,XI,YI,ZI)
VI = interp3(V,ntimes)
VI = interp3(...,method)
VI = interp3(...,method,extrapval)
```

Description

`VI = interp3(X,Y,Z,V,XI,YI,ZI)` interpolates to find `VI`, the values of the underlying three-dimensional function `V` at the points in arrays `XI`, `YI` and `ZI`. `XI`,`YI`, `ZI` must be arrays of the same size, or vectors. Vector arguments that are not the same size, and have mixed orientations (i.e. with both row and column vectors) are passed through `meshgrid` to create the `Y1`, `Y2`, `Y3` arrays. Arrays `X`, `Y`, and `Z` specify the points at which the data `V` is given. Out of range values are returned as `NaN`.

`VI = interp3(V,XI,YI,ZI)` assumes `X=1:N`, `Y=1:M`, `Z=1:P` where `[M,N,P]=size(V)`.

`VI = interp3(V,ntimes)` expands `V` by interleaving interpolates between every element, working recursively for `ntimes` iterations. The command `interp3(V)` is the same as `interp3(V,1)`.

`VI = interp3(...,method)` specifies alternative methods:

’`nearest`’	Nearest neighbor interpolation
’`linear`’	Linear interpolation (default)
’`spline`’	Cubic spline interpolation
’`cubic`’	Cubic interpolation, as long as data is uniformly-spaced. Otherwise, this method is the same as `'spline'`.

`VI = interp3(...,method,extrapval)` specifies a method and a value for `VI` outside of the domain created by `X`, `Y` and `Z`. Thus, `VI` equals `extrapval` for any value of `XI`, `YI` or `ZI` that is not spanned by `X`, `Y`, and `Z`, respectively. You must specify a method to use `extrapval`. The default method is `'linear'`.

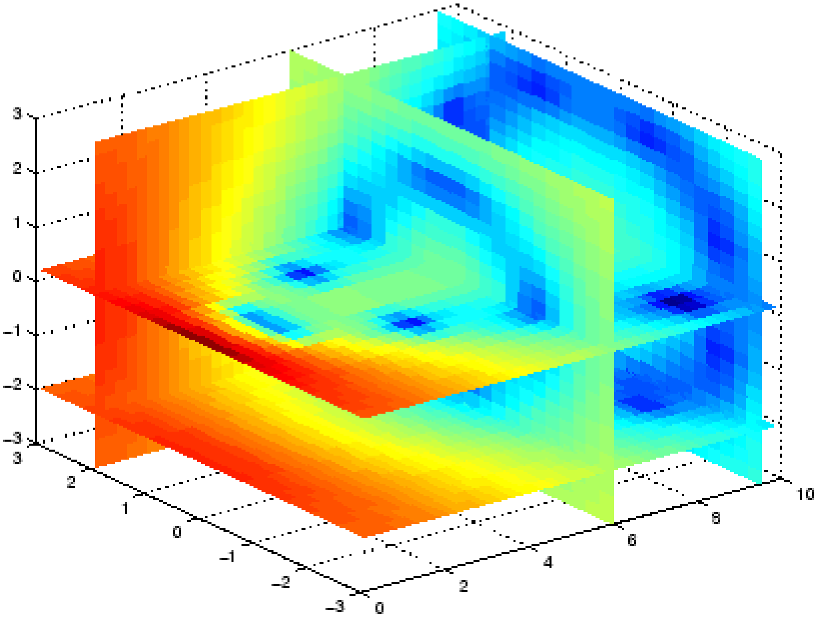
Discussion

All the interpolation methods require that `X`,`Y` and `Z` be monotonic and have the same format (“plaid”) as if they were created using `meshgrid`. `X`, `Y`, and `Z` can be non-uniformly spaced. For faster interpolation when `X`, `Y`, and `Z` are equally spaced and monotonic, use the methods ’`*linear`’, ’`*cubic`’, or ’`*nearest`’.

Examples

To generate a coarse approximation of `flow` and interpolate over a finer mesh:

```
[x,y,z,v] = flow(10);
[xi,yi,zi] = meshgrid(.1:.25:10, -3:.25:3, -3:.25:3);
vi = interp3(x,y,z,v,xi,yi,zi); % vi is 25-by-40-by-25
slice(xi,yi,zi,vi,[6 9.5],2,[-2 .2]), shading flat
```



See Also

`interp1`, `interp1q`, `interp2`, `interpn`, `meshgrid`

Purpose

1-D interpolation using FFT method

Syntax

```
y = interpft(x,n)
y = interpft(x,n,dim)
```

Description

`y = interpft(x,n)` returns the vector `y` that contains the value of the periodic function `x` resampled to `n` equally spaced points.

If `length(x) = m`, and `x` has sample interval `dx`, then the new sample interval for `y` is `dy = dx*m/n`. Note that `n` cannot be smaller than `m`.

If `X` is a matrix, `interpft` operates on the columns of `X`, returning a matrix `Y` with the same number of columns as `X`, but with `n` rows.

`y = interpft(x,n,dim)` operates along the specified dimension.

Algorithm

The `interpft` command uses the FFT method. The original vector `x` is transformed to the Fourier domain using `fft` and then transformed back with more points.

Examples

Interpolate a triangle-like signal using an interpolation factor of 5. First, set up signal to be interpolated:

```
y = [0 .5 1 1.5 2 1.5 1 .5 0 -.5 -1 -1.5 -2 -1.5 -1 -.5 0];
N = length(y);
```

Perform the interpolation:

```
L = 5;
M = N*L;
x = 0:L:L*N-1;
xi = 0:M-1;
yi = interpft(y,M);
plot(x,y,'o',xi,yi,'*')
legend('Original data','Interpolated data')
```

See Also

`interp1`

interpn

Purpose

N-D data interpolation (table lookup)

Syntax

```
VI = interpn(X1,X2,X3,...,V,Y1,Y2,Y3,...)
VI = interpn(V,Y1,Y2,Y3,...)
VI = interpn(V,ntimes)
VI = interpn(...,method)
VI = interpn(...,method,extrapval)
```

Description

`VI = interpn(X1,X2,X3,...,V,Y1,Y2,Y3,...)` interpolates to find `VI`, the values of the underlying multidimensional function `V` at the points in the arrays `Y1`, `Y2`, `Y3`, etc. For an n-dimensional array `V`, `interpn` is called with `2*N+1` arguments. Arrays `X1`, `X2`, `X3`, etc. specify the points at which the data `V` is given. Out of range values are returned as `NaN`s. `Y1`, `Y2`, `Y3`, etc. must be arrays of the same size, or vectors. Vector arguments that are not the same size, and have mixed orientations (i.e. with both row and column vectors) are passed through `ndgrid` to create the `Y1`, `Y2`, `Y3`, etc. arrays. `interpn` works for all n-dimensional arrays with 2 or more dimensions.

`VI = interpn(V,Y1,Y2,Y3,...)` interpolates as above, assuming `X1 = 1:size(V,1)`, `X2 = 1:size(V,2)`, `X3 = 1:size(V,3)`, etc.

`VI = interpn(V,ntimes)` expands `V` by interleaving interpolates between each element, working recursively for `ntimes` iterations. `interpn(V)` is the same as `interpn(V,1)`.

`VI = interpn(...,method)` specifies alternative methods:

'nearest'	Nearest neighbor interpolation
`'linear'`	Linear interpolation (default)
`'spline'`	Cubic spline interpolation
`'cubic'`	Cubic interpolation, as long as data is uniformly-spaced. Otherwise, this method is the same as `'spline'`.

`VI = interpn(...,method,extrapval)` specifies a method and a value for `VI` outside of the domain created by `X1`, `X2`, `...`. Thus, `VI` equals `extrapval` for any value of `Y1`, `Y2`,... that is not spanned by `X1`, `X2`,... respectively. You must specify a method to use `extrapval`. The default method is `'linear'`.

`interpn` requires that `X1`, `X2`, `X3`, ... be monotonic and plaid (as if they were created using `ndgrid`). `X1`, `X2`, `X3`, and so on can be non-uniformly spaced.

Discussion

All the interpolation methods require that `X1`,`X2`, `X3` ... be monotonic and have the same format ("plaid") as if they were created using `ndgrid`. `X1`,`X2`,`X3`,... and `Y1`, `Y2`, `Y3`, etc. can be non-uniformly spaced. For faster interpolation when `X1`, `X2`, `X3`, etc. are equally spaced and monotonic, use the methods ’`*linear`’, ’`*cubic`’, or ’`*nearest`’.

Examples

Start by defining an anonymous function to compute $f = te^{-x^2-y^2-z^2}$:

```
f = @(x,y,z,t) t.*exp(-x.^2 - y.^2 - z.^2);
```

Build the lookup table by evaluating the function `f` on a grid constructed by `ndgrid`:

```
[x,y,z,t] = ndgrid(-1:0.2:1,-1:0.2:1,-1:0.2:1,0:2:10);
v = f(x,y,z,t);
```

Now construct a finer grid:

```
[xi,yi,zi,ti] = ndgrid(-1:0.05:1,-1:0.08:1,-1:0.05:1, ...
                       0:0.5:10);
```

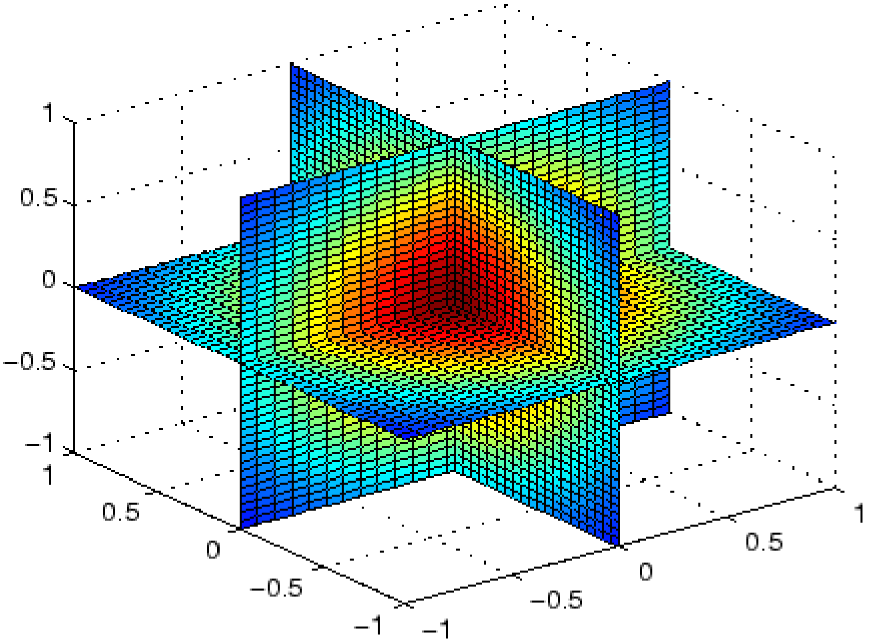
Compute the spline interpolation at `xi`, `yi`, `zi`, and `ti`:

```
vi = interpn(x,y,z,t,v,xi,yi,zi,ti,'spline');
```

Plot the interpolated function, and then create a movie from the plot:

```
nframes = size(ti, 4);
for j = 1:nframes
   slice(yi(:,:,:,j), xi(:,:,:,j), zi(:,:,:,j), ...
```

```
              vi(:,:,:,j),0,0,0);
    caxis([0 10]);
    M(j) = getframe;
end
movie(M);
```



See Also `interp1`, `interp2`, `interp3`, `ndgrid`

Purpose Interpolate stream-line vertices from flow speed

Syntax

```
interpstreamspeed(X,Y,Z,U,V,W,vertices)
interpstreamspeed(U,V,W,vertices)
interpstreamspeed(X,Y,Z,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(X,Y,U,V,vertices)
interpstreamspeed(U,V,vertices)
interpstreamspeed(X,Y,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(...,sf)
vertsout = interpstreamspeed(...)
```

Description

`interpstreamspeed(X,Y,Z,U,V,W,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`).

`interpstreamspeed(U,V,W,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(U)`.

`interpstreamspeed(X,Y,Z,speed,vertices)` uses the 3-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p]=size(speed)`.

`interpstreamspeed(X,Y,U,V,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`. The arrays `X`, `Y` define the coordinates for `U`, `V` and must be monotonic and 2-D plaid (as if produced by `meshgrid`).

`interpstreamspeed(U,V,vertices)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[M N]=size(U)`.

`interpstreamspeed(X,Y,speed,vertices)` uses the 2-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[M,N]= size(speed)`.

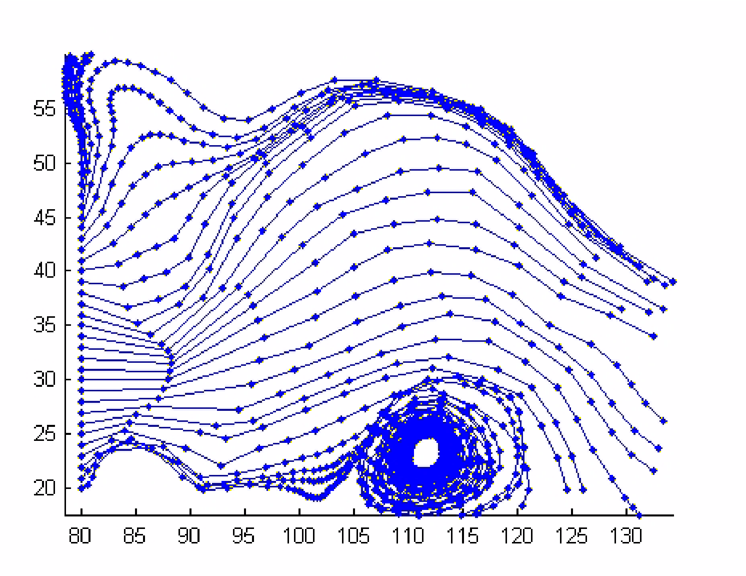
`interpstreamspeed(...,sf)` uses `sf` to scale the magnitude of the vector data and therefore controls the number of interpolated vertices. For example, if `sf` is 3, then `interpstreamspeed` creates only one-third of the vertices.

`vertsout = interpstreamspeed(...)` returns a cell array of vertex arrays.

Examples

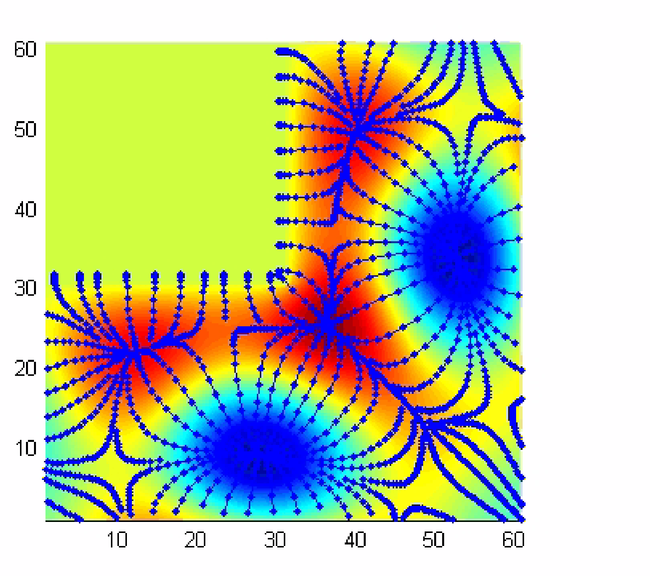
This example draws streamlines using the vertices returned by `interpstreamspeed`. Dot markers indicate the location of each vertex. This example enables you to visualize the relative speeds of the flow data. Streamlines having widely spaced vertices indicate faster flow; those with closely spaced vertices indicate slower flow.

```
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.2);
sl = streamline(iverts);
set(sl,'Marker','.')
axis tight; view(2); daspect([1 1 1])
```



This example plots streamlines whose vertex spacing indicates the value of the gradient along the streamline.

```
z = membrane(6,30);
[u v] = gradient(z);
[verts averts] = streamslice(u,v);
iverts = interpstreamspeed(u,v,verts,15);
sl = streamline(iverts);
set(sl,'Marker','.')
hold on; pcolor(z); shading interp
axis tight; view(2); daspect([1 1 1])
```



See Also

`stream2`, `stream3`, `streamline`, `streamslice`, `streamparticles`

"Volume Visualization" on page 1-106 for related functions

Purpose

Find set intersection of two vectors

Syntax

```
c = intersect(A, B)
c = intersect(A, B, 'rows')
[c, ia, ib] = intersect(a, b)
```

Description

`c = intersect(A, B)` returns the values common to both `A` and `B`. In set theoretic terms, this is `A`[[INTERSECT]] B. Inputs `A` and `B` can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted in ascending order.

`c = intersect(A, B, 'rows')` when `A` and `B` are matrices with the same number of columns returns the rows common to both `A` and `B`. `MATLAB ignores the rows flag for all cell arrays.

`[c, ia, ib] = intersect(a, b)` also returns column index vectors `ia` and `ib` such that `c = a(ia)` and `c = b(ib)` (or `c = a(ia,:)` and `c = b(ib,:)`).

Remarks

Because NaN is considered to be not equal to itself, it is never included in the result `c`.

Examples

```
A = [1 2 3 6]; B = [1 2 3 4 6 10 20];
[c, ia, ib] = intersect(A, B);
disp([c; ia; ib])
      1     2     3     6
      1     2     3     4
      1     2     3     5
```

See Also

`ismember`, `issorted`, `setdiff`, `setxor`, `union`, `unique`

intmax

Purpose

Largest value of specified integer type

Syntax

```
v = intmax
v = intmax('classname')
```

Description

`v = intmax` is the largest positive value that can be represented in the MATLAB software with a 32-bit integer. Any value larger than the value returned by `intmax` saturates to the `intmax` value when cast to a 32-bit integer.

`v = intmax('classname')` is the largest positive value in the integer class `classname`. Valid values for the string `classname` are

`'int8'`	`'int16'`	`'int32'`	`'int64'`
`'uint8'`	`'uint16'`	`'uint32'`	`'uint64'`

`intmax('int32')` is the same as `intmax` with no arguments.

Examples

Find the maximum value for a 64-bit signed integer:

```
v = intmax('int64')
v =
  9223372036854775807
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
  2147483647
```

Compare the result with the default value returned by `intmax`:

```
isequal(x, intmax)
ans =
     1
```

See Also

`intmin`, `realmax`, `realmin`, `int8`, `uint8`, `isa`, `class`

Purpose

Smallest value of specified integer type

Syntax

```
v = intmin
v = intmin('classname')
```

Description

`v = intmin` is the smallest value that can be represented in the MATLAB software with a 32-bit integer. Any value smaller than the value returned by `intmin` saturates to the `intmin` value when cast to a 32-bit integer.

`v = intmin('classname')` is the smallest positive value in the integer class `classname`. Valid values for the string `classname` are

`'int8'`	`'int16'`	`'int32'`	`'int64'`
`'uint8'`	`'uint16'`	`'uint32'`	`'uint64'`

`intmin('int32')` is the same as `intmin` with no arguments.

Examples

Find the minimum value for a 64-bit signed integer:

```
v = intmin('int64')
v =
 -9223372036854775808
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
  2147483647
```

Compare the result with the default value returned by `intmin`:

```
isequal(x, intmin)
ans =
     1
```

See Also

`intmax`, `realmin`, `realmax`, `int8`, `uint8`, `isa`, `class`

intwarning

Purpose

Control state of integer warnings

Syntax

```
intwarning('action')
s = intwarning('action')
intwarning(s)
sOld = intwarning(sNew)
```

Description

The MATLAB software offers four types of warnings on invalid operations that involve integers. The `intwarning` function enables, disables, or returns information on these warnings:

- `MATLAB:intConvertNaN` — Warning on an attempt to convert NaN (Not a Number) to an integer. The result of the operation is zero.
- `MATLAB:intConvertNonIntVal` — Warning on an attempt to convert a non-integer value to an integer. The result is that the input value is rounded to the nearest integer for that class.
- `MATLAB:intConvertOverflow` — Warning on overflow when attempting to convert from a numeric class to an integer class. The result is the maximum value for the target class.
- `MATLAB:intMathOverflow` — Warning on overflow when attempting an integer arithmetic operation. The result is the maximum value for the class of the input value. MATLAB also issues this warning when NaN is computed (e.g., `int8(0)/0`).

`intwarning('`*`action`*`')` sets or displays the state of integer warnings in MATLAB according to the string, *`action`*. There are three possible actions, as shown here. The default state is `'off'`.

Action	**Description**
`off`	Disable the display of integer warnings
`on`	Enable the display of integer warnings
`query`	Display the state of all integer warnings

`s = intwarning('`*`action`*`')` sets the state of integer warnings in MATLAB according to the string *`action`*, and then returns the previous state in a 4-by-1 structure array, `s`. The return structure array has two fields: `identifier` and `state`.

`intwarning(s)` sets the state of integer warnings in MATLAB according to the `identifier` and `state` fields in structure array `s`.

`sOld = intwarning(sNew)` sets the state of integer warnings in MATLAB according to `sNew`, and then returns the previous state in `sOld`.

Remarks

Caution Enabling the `MATLAB:intMathOverflow` warning slows down integer arithmetic. It is recommended that you enable this particular warning only when you need to diagnose unusual behavior in your code, and disable it during normal program operation. The other integer warnings listed here do not affect program performance.

Examples

General Usage

Examples of the four types of integer warnings are shown here:

- **MATLAB:intConvertNaN**

Attempt to convert NaN (Not a Number) to an unsigned integer:

```
uint8(NaN);
Warning: NaN converted to uint8(0).
```

- **MATLAB:intConvertNonIntVal**

Attempt to convert a floating point number to an unsigned integer:

```
uint8(2.7);
Warning: Conversion rounded non-integer floating point
   value to nearest uint8 value.
```

- **MATLAB:intConvertOverflow**

Attempt to convert a large unsigned integer to a signed integer, where the operation overflows:

```
int8(uint8(200));
Warning: Out of range value converted to intmin('int8')
   or intmax('int8').
```

- **MATLAB:intMathOverflow**

Attempt an integer arithmetic operation that overflows:

```
intmax('uint8') + 5;
Warning: Out of range value or NaN computed in
integer arithmetic.
```

Example 1

Check the initial state of integer warnings:

```
intwarning('query')
The state of warning 'MATLAB:intConvertNaN' is 'off'.
The state of warning 'MATLAB:intConvertNonIntVal' is 'off'.
The state of warning 'MATLAB:intConvertOverflow' is 'off'.
The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

Convert a floating point value to an 8-bit unsigned integer. MATLAB does the conversion, but that requires rounding the resulting value. Because all integer warnings have been disabled, no warning is displayed:

```
uint8(2.7)
ans =
    3
```

Store this state in structure array `iwState`:

```
iwState = intwarning('query');
```

Change the state of the `ConvertNonIntVal` warning to `'on'` by first setting the state to `'on'` in the `iwState` structure array, and then loading `iwState` back into the internal integer warning settings for your MATLAB session:

```
maxintwarn = 4;

for k = 1:maxintwarn
   if strcmp(iwState(k).identifier, ...
            'MATLAB:intConvertNonIntVal')
      iwState(k).state = 'on';
      intwarning(iwState);
   end
end
```

Verify that the state of `ConvertNonIntVal` has changed:

```
intwarning('query')
The state of warning 'MATLAB:intConvertNaN' is 'off'.
The state of warning 'MATLAB:intConvertNonIntVal' is 'on'.
The state of warning 'MATLAB:intConvertOverflow' is 'off'.
The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

Now repeat the conversion from floating point to integer. This time MATLAB displays the warning:

```
uint8(2.7)
Warning: Conversion rounded non-integer floating point
   value to nearest uint8 value.
ans =
    3
```

See Also

`warning`, `lastwarn`

inv

Purpose

Matrix inverse

Syntax

```
Y = inv(X)
```

Description

`Y = inv(X)` returns the inverse of the square matrix `X`. A warning message is printed if `X` is badly scaled or nearly singular.

In practice, it is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises when solving the system of linear equations $Ax = b$. One way to solve this is with `x = inv(A)*b`. A better way, from both an execution time and numerical accuracy standpoint, is to use the matrix division operator `x = A\b`. This produces the solution using Gaussian elimination, without forming the inverse. See `\` and `/` for further information.

Examples

Here is an example demonstrating the difference between solving a linear system by inverting the matrix with `inv(A)*b` and solving it directly with `A\b`. A random matrix `A` of order 500 is constructed so that its condition number, `cond(A)`, is `1.e10`, and its norm, `norm(A)`, is `1`. The exact solution `x` is a random vector of length 500 and the right-hand side is `b = A*x`. Thus the system of linear equations is badly conditioned, but consistent.

On a 300 MHz, laptop computer the statements

```
n = 500;
Q = orth(randn(n,n));
d = logspace(0,-10,n);
A = Q*diag(d)*Q';
x = randn(n,1);
b = A*x;
tic, y = inv(A)*b; toc
err = norm(y-x)
res = norm(A*y-b)
```

produce

```
elapsed_time =
```

```
    1.4320
err =
    7.3260e-006
res =
    4.7511e-007
```

while the statements

```
tic, z = A\b, toc
err = norm(z-x)
res = norm(A*z-b)
```

produce

```
elapsed_time =
    0.6410
err =
    7.1209e-006
res =
    4.4509e-015
```

It takes almost two and one half times as long to compute the solution with `y = inv(A)*b` as with `z = A\b`. Both produce computed solutions with about the same error, `1.e-6`, reflecting the condition number of the matrix. But the size of the residuals, obtained by plugging the computed solution back into the original equations, differs by several orders of magnitude. The direct solution produces residuals on the order of the machine accuracy, even though the system is badly conditioned.

The behavior of this example is typical. Using `A\b` instead of `inv(A)*b` is two to three times as fast and produces residuals on the order of machine accuracy, relative to the magnitude of the data.

Algorithm

Inputs of Type Double

For inputs of type `double`, `inv` uses the following LAPACK routines to compute the matrix inverse:

Matrix	Routine
Real	`DLANGE`, `DGETRF`, `DGECON`, `DGETRI`
Complex	`ZLANGE`, `ZGETRF`, `ZGECON`, `ZGETRI`

Inputs of Type Single

For inputs of type `single`, `inv` uses the following LAPACK routines to compute the matrix inverse:

Matrix	Routine
Real	`SLANGE`, `SGETRF`, `SGECON`, `SGETRI`
Complex	`CLANGE`, `CGETRF`, `CGECON`, `CGETRI`

See Also

`det`, `lu`, `rref`

The arithmetic operators `\`, `/`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Inverse of Hilbert matrix

Syntax

```
H = invhilb(n)
```

Description

`H = invhilb(n)` generates the exact inverse of the exact Hilbert matrix for `n` less than about 15. For larger `n`, `invhilb(n)` generates an approximation to the inverse Hilbert matrix.

Limitations

The exact inverse of the exact Hilbert matrix is a matrix whose elements are large integers. These integers may be represented as floating-point numbers without roundoff error as long as the order of the matrix, `n`, is less than 15.

Comparing `invhilb(n)` with `inv(hilb(n))` involves the effects of two or three sets of roundoff errors:

- The errors caused by representing `hilb(n)`
- The errors in the matrix inversion process
- The errors, if any, in representing `invhilb(n)`

It turns out that the first of these, which involves representing fractions like 1/3 and 1/5 in floating-point, is the most significant.

Examples

`invhilb(4)` is

```
      16     -120      240     -140
    -120     1200    -2700     1680
     240    -2700     6480    -4200
    -140     1680    -4200     2800
```

See Also

`hilb`

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

invoke

Purpose

Invoke method on COM object or interface, or display methods

Syntax

```
S = h.invoke
S = h.invoke('methodname')
S = h.invoke('methodname', arg1, arg2, ...)
S = h.invoke('custominterfacename')
S = invoke(h, ...)
```

Description

`S = h.invoke` returns structure array `S` containing a list of all methods supported by the object or interface, `h`, along with the prototypes for these methods. If `S` is empty, either there are no properties or methods in the object, or the MATLAB software cannot read the object's type library. Refer to the COM vendor's documentation.

`S = h.invoke('methodname')` invokes the method specified in the string `methodname`, and returns an output value, if any, in `S`. The data type of the return value depends on the invoked method, which is determined by the control or server.

`S = h.invoke('methodname', arg1, arg2, ...)` invokes the method specified in the string `methodname` with input arguments `arg1`, `arg2`, etc.

`S = h.invoke('custominterfacename')` returns an `Interface` object `S`, which is a handle to a custom interface implemented by the COM component. The `h` argument is a handle to the COM object. The `custominterfacename` argument is a string returned by the `interfaces` function.

`S = invoke(h, ...)` is an alternate syntax. For Automation objects, if the vendor provides documentation for specific properties or methods, use the `S = invoke(h, ...)` syntax to call them.

If the method returns a COM interface, then `invoke` returns a new MATLAB COM object that represents the interface returned. For a description of how MATLAB converts COM types, see Handling COM Data in MATLAB in the External Interfaces documentation.

COM functions are available on Microsoft Windows systems only.

Examples

Invoke the `Redraw` method in the `mwsamp` control:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.1', [0 0 200 200], f);
h.Radius = 100;
h.invoke('Redraw');
```

Call the method directly:

```
h.Redraw;
```

Display all `mwsamp` methods:

```
h.invoke
```

MATLAB displays (in part):

```
ans =
   AboutBox = void AboutBox(handle)
   Beep = void Beep(handle)
   FireClickEvent = void FireClickEvent(handle)
             .
             .
```

Getting a Custom Interface Example

Once you have created a COM server, you can query the server component to see if any custom interfaces are implemented. Use the `interfaces` function to return a list of all available custom interfaces:

```
h = actxserver('mytestenv.calculator');
customlist = h.interfaces
```

MATLAB displays:

```
customlist =
   ICalc1
```

```
    ICalc2
    ICalc3
```

To get a handle to the custom interface you want, use the `invoke` function, specifying the handle returned by `actxcontrol` or `actxserver` and also the name of the custom interface:

```
c1 = h.invoke('ICalc1')
```

MATLAB displays:

```
c1 =
    Interface.Calc_1.0_Type_Library.ICalc_Interface
```

You can now use this handle with most of the COM client functions to access the properties and methods of the object through the selected custom interface.

See Also

`methods` | `ismethod` | `interfaces`

How To

-
-

Purpose

Inverse permute dimensions of N-D array

Syntax

`A = ipermute(B,order)`

Description

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

Remarks

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

Examples

Consider the 2-by-2-by-3 array `a`:

```
a = cat(3,eye(2),2*eye(2),3*eye(2))

a(:,:,1) =                a(:,:,2) =
     1     0                   2     0
     0     1                   0     2

a(:,:,3) =
     3     0
     0     3
```

Permuting and inverse permuting `a` in the same fashion restores the array to its original form:

```
B = permute(a,[3 2 1]);
C = ipermute(B,[3 2 1]);
isequal(a,C)
ans=

    1
```

See Also

`permute`

Purpose

Interquartile range of `timeseries` data

Syntax

```
ts_iqr = iqr(ts)
iqr(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_iqr = iqr(ts)` returns the interquartile range of `ts.Data`. When `ts.Data` is a vector, ts_iqr is the difference between the 75th and the 25th percentiles of the `ts.Data` values. When `ts.Data` is a matrix, ts_iqr is a row vector containing the interquartile range of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `iqr` always operates along the first nonsingleton dimension of `ts.Data`.

`iqr(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`.
  When you specify `'time'`, larger time values correspond to larger weights.

Examples

Create a time series with a missing value, represented by `NaN`.

```
ts = timeseries([3.0 NaN 5 6.1 8], 1:5);
```

Calculate the interquartile range of `ts.Data` after removing the missing value from the calculation.

```
iqr(ts,'MissingData','remove')
```

```
ans =

    3.0500
```

See Also `timeseries`

Purpose Detect state

Description These functions detect the state of MATLAB entities:

`isa`	Detect object of given MATLAB class or Java class
`isappdata`	Determine if object has specific application-defined data
`iscell`	Determine if input is cell array
`iscellstr`	Determine if input is cell array of strings
`ischar`	Determine if input is character array
`iscom`	Determine if input is Component Object Model (COM) object
`isdir`	Determine if input is directory
`isempty`	Determine if input is empty array
`isequal`	Determine if arrays are numerically equal
`isequalwithequalnans`	Determine if arrays are numerically equal, treating `NaN`s as equal
`isevent`	Determine if input is object event
`isfield`	Determine if input is MATLAB structure array field
`isfinite`	Detect finite elements of array
`isfloat`	Determine if input is floating-point array
`isglobal`	Determine if input is global variable
`ishandle`	Detect valid graphics object handles
`ishold`	Determine if graphics hold state is on
`isinf`	Detect infinite elements of array
`isinteger`	Determine if input is integer array
`isinterface`	Determine if input is Component Object Model (COM) interface

`isjava`	Determine if input is Java object
`iskeyword`	Determine if input is MATLAB keyword
`isletter`	Detect elements that are alphabetic letters
`islogical`	Determine if input is logical array
`ismac`	Determine if running MATLAB for Macintosh OS X platform
`ismember`	Detect members of specific set
`ismethod`	Determine if input is object method
`isnan`	Detect elements of array that are not a number (NaN)
`isnumeric`	Determine if input is numeric array
`isobject`	Determine if input is MATLAB object
`ispc`	Determine if running MATLAB for PC (Windows) platform
`isprime`	Detect prime elements of array
`isprop`	Determine if input is object property
`isreal`	Determine if all array elements are real numbers
`isscalar`	Determine if input is scalar
`issorted`	Determine if set elements are in sorted order
`isspace`	Detect space characters in array
`issparse`	Determine if input is sparse array
`isstrprop`	Determine if string is of specified category
`isstruct`	Determine if input is MATLAB structure array
`isstudent`	Determine if Student Version of MATLAB

`isunix`	Determine if running MATLAB for UNIX[9] platform.
`isvarname`	Determine if input is valid variable name
`isvector`	Determine if input is vector

See Also

`isa`

---

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

Purpose Determine whether input is object of given class

Syntax

```
K = isa(obj, 'class_name')
```

Description `K = isa(obj, 'class_name')` returns logical `1` (`true`) if `obj` is of class (or a subclass of) *`class_name`*, and logical `0` (`false`) otherwise.

The argument `obj` is a MATLAB object or an object of the Java programming language. The argument *`class_name`* is the name of a MATLAB (predefined or user-defined) or a Java class. Predefined MATLAB classes include

`logical`	Logical array of `true` and `false` values
`char`	Characters array
`numeric`	Integer or floating-point array
`integer`	Signed or unsigned integer array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64-bit unsigned integer array
`float`	Single- or double-precision floating-point array
`single`	Single-precision floating-point array
`double`	Double-precision floating-point array
`cell`	Cell array
`struct`	Structure array

`function_handle`	Function handle
'*`class_name`*'	MATLAB class or Java class

To check for a sparse array, use `issparse`. To check for a complex array, use `~isreal`.

Examples

```
isa(rand(3,4),'double')
ans =
    1
```

The following example creates an instance of the user-defined MATLAB class named `polynom`. The `isa` function identifies the object as being of the `polynom` class.

```
polynom_obj = polynom([1 0 -2 -5]);
isa(polynom_obj, 'polynom')
ans =
     1
```

See Also

`class`, `is*`

Purpose True if application-defined data exists

Syntax `isappdata(h,name)`

Description `isappdata(h,name)` returns `1` if application-defined data with the specified `name` exists on the object specified by handle `h`, and returns `0` otherwise.

Remarks Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property.

See Also `getappdata`, `rmappdata`, `setappdata`

Purpose

Determine whether input is cell array

Syntax

```
tf = iscell(A)
```

Description

`tf = iscell(A)` returns logical `1` (`true`) if `A` is a cell array and logical `0` (`false`) otherwise.

Examples

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];
A{1,2} = 'Anne Smith';
A{2,1} = 3+7i;
A{2,2} = -pi:pi/10:pi;

iscell(A)

ans =

     1
```

See Also

`cell`, `iscellstr`, `isstruct`, `isnumeric`, `islogical`, `isobject`, `isa`, `is*`

Purpose

Determine whether input is cell array of strings

Syntax

`tf = iscellstr(A)`

Description

`tf = iscellstr(A)` returns logical `1` (`true`) if `A` is a cell array of strings (or an empty cell array), and logical `0` (`false`) otherwise. A cell array of strings is a cell array where every element is a character array.

Examples

```
A{1,1} = 'Thomas Lee';
A{1,2} = 'Marketing';
A{2,1} = 'Allison Jones';
A{2,2} = 'Development';

iscellstr(A)

ans =

      1
```

See Also

`cellstr`, `iscell`, `isstrprop`, `strings`, `char`, `isstruct`, `isa`, `is*`

ischar

Purpose

Determine whether item is character array

Syntax

```
tf = ischar(A)
```

Description

`tf = ischar(A)` returns logical `1` (`true`) if `A` is a character array and logical `0` (`false`) otherwise.

Examples

Given the following cell array,

```
C{1,1} = magic(3);           % double array
C{1,2} = 'John Doe';         % char array
C{1,3} = 2 + 4i              % complex double

C =

    [3x3 double]    'John Doe'    [2.0000+ 4.0000i]
```

`ischar` shows that only `C{1,2}` is a character array.

```
for k = 1:3
x(k) = ischar(C{1,k});
end

x

x =

     0     1     0
```

See Also

`char`, `strings`, `isletter`, `isspace`, `isstrprop`, `iscellstr`, `isnumeric`, `isa`, `is*`

Purpose

Determine whether input is COM or ActiveX object

Syntax

```
tf = h.iscom
tf = iscom(h)
```

Description

`tf = h.iscom` returns logical `1` (`true`) if handle `h` is a COM or Microsoft ActiveX object. Otherwise, returns logical `0` (`false`).

`tf = iscom(h)` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Test an instance of a Microsoft Excel application:

```
h = actxserver('Excel.Application');
h.iscom
```

MATLAB displays `true`, indicating object `h` is a COM object.

Test an Excel interface:

```
h = actxserver('Excel.Application');
%Create a workbooks object
w = h.get('workbooks');
w.iscom
```

MATLAB displays `false`, indicating object `w` is not a COM object.

How To

-

isdir

Purpose Determine whether input is folder

Syntax `tf = isdir('A')`

Description `tf = isdir('A')` returns logical `1` (`true`) if `A` is a folder. Otherwise, it returns logical `0` (`false`).

Examples Run:

```
tf=isdir('mymfiles/results')
```

MATLAB returns

```
tf =
 1
```

indicating that `mymfiles/results` is a folder.

See Also `dir`, `is*`

TriRep.isEdge

Purpose

Test if vertices are joined by edge

Syntax

```
TF = isEdge(TR, V1, V2)
TF = isEdge(TR, EDGE)
```

Description

`TF = isEdge(TR, V1, V2)` returns an array of 1/0 (true/false) flags, where each entry `TF(i)` is true if `V1(i)`, `V2(i)` is an edge in the triangulation. `V1`, `V2` are column vectors representing the indices of the vertices in the mesh, that is, indices into the vertex coordinate arrays.

`TF = isEdge(TR, EDGE)` specifies the edge start and end indices in matrix format.

Inputs

`TR`	Triangulation representation.
`V1, V2`	Column vectors of mesh vertices.
`EDGE`	Matrix of size `n`-by-2 where `n` is the number of query edges.

Outputs

`TF`	Array of 1/0 (true/false) flags, where each entry `TF(i)` is true if `V1(i)`, `V2(i)` is an edge in the triangulation.

Examples

Example 1

Load a 2-D triangulation and use `TriRep` to query the presence of an edge between pairs of points.

```
load trimesh2d
trep = TriRep(tri, x,y);
```

Test if vertices 3 and 117 are connected by an edge

```
isEdge(trep, 3, 117)
```

Test if vertices 3 and 164 are connected by an edge

```
isEdge(trep, 3, 164)
```

Example 2

Direct query of a 3-D Delaunay triangulation created using `DelaunayTri`.

```
X = rand(10,3)
dt = DelaunayTri(X)
```

Test if vertices 2 and 7 are connected by an edge

```
isEdge(dt, 2, 7);
```

See Also

`DelaunayTri`

Purpose Determine whether array is empty

Syntax `TF = isempty(A)`

Description `TF = isempty(A)` returns logical 1 (`true`) if `A` is an empty array and logical 0 (`false`) otherwise. An empty array has at least one dimension of size zero, for example, 0-by-0 or 0-by-5.

Examples

```
B = rand(2,2,2);
B(:,:,:) = [];

isempty(B)

ans = 1
```

See Also `is*`

isempty (timeseries)

Purpose

Determine whether `timeseries` object is empty

Syntax

```
isempty(ts)
```

Description

`isempty(ts)` returns a logical value for `timeseries` object `ts`, as follows:

- `1` — When `ts` contains no data samples or `ts.Data` is empty.
- `0` — When `ts` contains data samples

See Also

`length (timeseries)`, `size (timeseries)`, `timeseries`, `tsprops`

Purpose Determine whether `tscollection` object is empty

Syntax `isempty(tsc)`

Description `isempty(tsc)` returns a logical value for `tscollection` object `tsc`, as follows:

- `1` — When `tsc` contains neither `timeseries` members nor a time vector
- `0` — When `tsc` contains either `timeseries` members or a time vector

See Also `length (tscollection)`, `size (tscollection)`, `timeseries`, `tscollection`

isequal

Purpose

Test arrays for equality

Syntax

```
tf = isequal(A, B, ...)
```

Description

`tf = isequal(A, B, ...)` returns logical `1` (`true`) if the input arrays have the same contents, and logical `0` (`false`) otherwise. Nonempty arrays must be of the same data type and size.

Remarks

When comparing structures, the order in which the fields of the structures were created is not important. As long as the structures contain the same fields, with corresponding fields set to equal values, `isequal` considers the structures to be equal. See Example 2, below.

When comparing numeric values, `isequal` does not consider the data type used to store the values in determining whether they are equal. See Example 3, below. This is also true when comparing numeric values with certain nonnumeric values, such as logical `true` and 1, or the character `A` and its numeric equivalent, 65.

`NaN`s (Not a Number), by definition, are not equal. Therefore, arrays that contain `NaN` elements are not equal, and `isequal` returns zero when comparing such arrays. See Example 4, below. Use the `isequalwithequalnans` function when you want to test for equality with `NaN`s treated as equal.

`isequal` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequal` returns logical `1`.

Examples

Example 1

Given

```
A =                   B =                   C =
     1     0               1     0               1     0
     0     1               0     1               0     0
```

`isequal(A,B,C)` returns `0`, and `isequal(A,B)` returns `1`.

Example 2

When comparing structures with `isequal`, the order in which the fields of the structures were created is not important:

```
A.f1 = 25;    A.f2 = 50
A =
    f1: 25
    f2: 50

B.f2 = 50;    B.f1 = 25
B =
    f2: 50
    f1: 25

isequal(A, B)
ans =
     1
```

Example 3

When comparing numeric values, the data types used to store the values are not important:

```
A = [25 50];   B = [int8(25) int8(50)];

isequal(A, B)
ans =
     1
```

Example 4

Arrays that contain `NaN` (Not a Number) elements cannot be equal, since `NaN`s, by definition, are not equal:

```
A = [32 8 -29 NaN 0 5.7];
B = A;

isequal(A, B)
ans =
```

```
0
```

See Also `isequalwithequalnans`, `strcmp`, `isa`, `is*`, `relational operators`

Purpose

Compare `MException` objects for equality

Syntax

```
TF = isequal(eObj1, eObj2)
```

Description

`TF = isequal(eObj1, eObj2)` tests `MException` objects `eObj1` and `eObj2` for equality, returning logical 1 (`true`) if the two objects are identical, otherwise returning logical 0 (`false`).

See Also

`try`, `catch`, `error`, `assert`, `MException`, `eq(MException)`, `ne(MException)`, `getReport(MException)`, `disp(MException)`, `throw(MException)`, `rethrow(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `last(MException)`,

isequalwithequalnans

Purpose

Test arrays for equality, treating NaNs as equal

Syntax

`tf = isequalwithequalnans(A, B, ...)`

Description

`tf = isequalwithequalnans(A, B, ...)` returns logical `1` (`true`) if the input arrays are the same type and size and hold the same contents, and logical `0` (`false`) otherwise. `NaN` (Not a Number) values are considered to be equal to each other. Numeric data types and structure field order do not have to match.

Remarks

`isequalwithequalnans` is the same as `isequal`, except `isequalwithequalnans` considers `NaN` (Not a Number) values to be equal, and `isequal` does not.

`isequalwithequalnans` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequalwithequalnans` returns logical `1`.

Examples

Arrays containing `NaN`s are handled differently by `isequal` and `isequalwithequalnans`. `isequal` does not consider `NaN`s to be equal, while `isequalwithequalnans` does.

```
A = [32 8 -29 NaN 0 5.7];
B = A;
isequal(A, B)
ans =
     0

isequalwithequalnans(A, B)
ans =
     1
```

The position of `NaN` elements in the array does matter. If they are not in the same position in the arrays being compared, then `isequalwithequalnans` returns zero.

```
A = [2 4 6 NaN 8];   B = [2 4 NaN 6 8];
```

```
isequalwithequalnans(A, B)
ans =
     0
```

See Also `isequal`, `strcmp`, `isa`, `is*`, `relational operators`

isevent

Purpose Determine whether input is COM object event

Syntax

```
tf = h.isevent('eventname')
tf = isevent(h, 'eventname')
```

Description `tf = h.isevent('eventname')` returns logical `1` (`true`) if `event_name` is an event recognized by COM object `h`. Otherwise, returns logical `0` (`false`). The `event_name` argument is not case sensitive.

`tf = isevent(h, 'eventname')` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples Test events in a MATLAB sample control object:

1. Create an instance of the `mwsamp` control and test `DblClick`:

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

   MATLAB displays `true`, indicating `DblClick` is an event.

2. Try the same test on`Redraw`:

   ```
   h.isevent('Redraw')
   ```

   MATLAB displays `false`, indicating `Redraw` is not an event; it is a method.

Test events in a Microsoft Excel workbook object:

1. Create a `Workbook` object `wb`:

   ```
   myApp = actxserver('Excel.Application');
   wbs = myApp.Workbooks;
   wb = wbs.Add;
   ```

2 Test `Activate`:

```
wb.isevent('Activate')
```

MATLAB displays `true`, indicating `Activate` is an event.

3 Test `Save`:

```
wb.isevent('Save')
```

MATLAB displays `false`, indicating `Save` is not an event; it is a method.

See Also

`events (COM)` | `eventlisteners` | `registerevent`

How To

-
-

isfield

Purpose

Determine whether input is structure array field

Syntax

```
tf = isfield(S, 'fieldname')
tf = isfield(S, C)
```

Description

`tf = isfield(S, 'fieldname')` examines structure `S` to see if it includes the field specified by the quoted string `'fieldname'`. Output `tf` is set to logical `1` (`true`) if `S` contains the field, or logical `0` (`false`) if not. If `S` is not a structure array, `isfield` returns `false`.

`tf = isfield(S, C)` examines structure `S` for multiple fieldnames as specified in cell array of strings `C`, and returns an array of logical values to indicate which of these fields are part of the structure. Elements of output array `tf` are set to a logical 1 (`true`) if the corresponding element of `C` holds a fieldname that belongs to structure `S`. Otherwise, logical 0 (`false`) is returned in that element. In other words, if structure `S` contains the field specified in `C{m,n}`, `isfield` returns a logical 1 (`true`) in `tf(m,n)`.

Note `isfield` returns `false` if the `field` or `fieldnames` input is empty.

Examples

Example 1 — Single Fieldname Syntax

Given the following MATLAB structure,

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

`isfield` identifies `billing` as a field of that structure.

```
isfield(patient,'billing')
ans =
     1
```

Example 2 — Multiple Fieldname Syntax

Check structure `S` for any of four possible fieldnames. Only the first is found, so the first element of the return value is set to `true`:

```
S = struct('one', 1, 'two', 2);

fields = isfield(S, {'two', 'pi', 'One', 3.14})
fields =
     1     0     0     0
```

See Also

`fieldnames`, `setfield`, `getfield`, `orderfields`, `rmfield`, `struct`, `isstruct`, `iscell`, `isa`, `is*`, dynamic field names

isfinite

Purpose

Array elements that are finite

Syntax

`TF = isfinite(A)`

Description

`TF = isfinite(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of the array `A` are finite and logical 0 (`false`) where they are infinite or `NaN`. For a complex number `z`, `isfinite(z)` returns `1` if both the real and imaginary parts of `z` are finite, and `0` if either the real or the imaginary part is infinite or `NaN`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2];
isfinite(1./a)
ans =
     1     1     0     1     1
isfinite(0./a)
ans =
     1     1     0     1     1
```

See Also

`isinf`, `isnan`, `is*`

Purpose Determine whether input is floating-point array

Syntax `isfloat(A)`

Description `isfloat(A)` returns a logical 1 (`true`) if A is a floating-point array and a logical 0 (`false`) otherwise. The only floating-point data types in the MATLAB programming language are `single` and `double`.

See Also `isa`, `isinteger`, `double`, `single`, `isnumeric`

isglobal

Purpose

Determine whether input is global variable

Note Support for the `isglobal` function will be removed in a future release of the MATLAB software. See Remarks below.

Syntax

```
tf = isglobal(A)
```

Description

`tf = isglobal(A)` returns logical `1` (`true`) if `A` has been declared to be a global variable in the context from which `isglobal` is called, and logical `0` (`false`) otherwise.

Remarks

`isglobal` is most commonly used in conjunction with conditional global declaration. An alternate approach is to use a pair of variables, one local and one declared global.

Instead of using

```
if condition
   global x
end

x = some_value

if isglobal(x)
   do_something
end
```

You can use

```
global gx
if condition
   gx = some_value
else
   x = some_value
end
```

```
if condition
   do_something
end
```

If no other workaround is possible, you can replace the command

```
isglobal(variable)
```

with

```
~isempty(whos('global','variable'))
```

See Also `global`, `isvarname`, `isa`, `is*`

ishandle

Purpose Determine whether input is valid Handle Graphics handle

Syntax `ishandle(H)`

Description `ishandle(H)` returns an array whose elements are `1` where the elements of `H` are valid graphics or Sun Java object handles, and `0` where they are not.

You should use the `isa` function to determine the class and validity of MATLAB objects.

See Also `findobj`, `gca`, `gcf`, `gco`, `isa` `set`

for more information.

ishghandle

Purpose

True for Handle Graphics object handles

Syntax

`ishghandle(h)`

Description

`ishghandle(h)` returns an array that contains 1's where the elements of `h` are handles to existing graphic objects and 0's where they are not. Differs from `ishandle` in that Simulink objects handles return false.

Examples

Create a plot and find the valid handles:

```
x = [1:10];
y = [1:10];
p=plot(x,y);
ishghandle([x y p])
% This returns a 1-by-21 array of values with ones at the first,
% eleventh, and last values, if the figure handle is 1.
```

See Also

`isa` | `ishandle` | `findobj` | `gca` | `gcf` | `set`

How To

-

ishold

Purpose Current hold state

Syntax `ishold`

Description `ishold` returns `1` if `hold` is `on`, and `0` if it is `off`. When `hold` is `on`, the current plot and most axis properties are held so that subsequent graphing commands add to the existing graph.

A state of `hold on` implies that both figure and axes `NextPlot` properties are set to `add`.

See Also `hold`, `newplot`

for related information

"Axes Operations" on page 1-101 for related functions

Purpose

Array elements that are infinite

Syntax

`TF = isinf(A)`

Description

`TF = isinf(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of `A` are `+Inf` or `-Inf` and logical 0 (`false`) where they are not. For a complex number `z`, `isinf(z)` returns `1` if either the real or imaginary part of `z` is infinite, and `0` if both the real and imaginary parts are finite or `NaN`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2]

isinf(1./a)
Warning: Divide by zero.

ans =
     0     0     1     0     0

isinf(0./a)
Warning: Divide by zero.

ans =
     0     0     0     0     0
```

See Also

`isfinite`, `isnan`, `is*`

isinteger

Purpose

Determine whether input is integer array

Syntax

Description

`isinteger(A)` returns a logical 1 (`true`) if the array `A` has integer data type and a logical 0 (`false`) otherwise. The integer data types in the MATLAB language are

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `int64`
- `uint64`

See Also

`isa`, `isnumeric`, `isfloat`

Purpose

Determine whether input is COM interface

Syntax

```
tf = h.isinterface
tf = isinterface(h)
```

Description

`tf = h.isinterface` returns logical `1` (`true`) if handle `h` is a COM interface. Otherwise, returns logical `0` (`false`).

`tf = isinterface(h)` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Test an instance of a Microsoft Excel application:

```
h = actxserver('Excel.Application');
h.isinterface
```

MATLAB displays `false`, indicating object `h` is not an interface.

Test a `workbooks` object:

```
w = h.get('workbooks');
w.isinterface
```

MATLAB displays `true`, indicating object `w` is an interface.

See Also

`iscom` | `interfaces`

How To

-

isjava

Purpose

Determine whether input is Sun Java object

Syntax

```
tf = isjava(A)
```

Description

`tf = isjava(A)` returns logical `1` (`true`) if `A` is a Java object, and logical `0` (`false`) otherwise.

Examples

Create an instance of the Java Date class and `isjava` indicates that it is a Java object.

```
myDate = java.util.Date;
isjava(myDate)
```

The MATLAB software displays:

```
ans =

     1
```

Note that `isobject`, which tests for MATLAB objects, returns logical `0` (`false`). Type:

```
isobject(myDate)
```

MATLAB displays:

```
ans =

     0
```

See Also

`isobject`, `javaArray`, `javaMethod`, `javaObject`, `isa`, `is*`

Purpose

Check if `containers.Map` contains key

Syntax

```
tf = isKey(M, keys)
```

Description

`tf = isKey(M, keys)` looks for the specified keys in the `Map` instance `M`, and returns logical 1 (`true`) for those elements that it finds, and logical 0 (`false`) for those it does not. `keys` is a scalar key or cell array of keys. If `keys` is nonscalar, then return value `tf` is a nonscalar logical array that has the same dimensions and size as `keys`.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

Examples

Construct a `Map` object where the keys are states in the United States and the value associated with each key is that state's capital city:

```
US_Capitals = containers.Map( ...
{'Arizona', 'Nebraska', 'Nevada', 'New York', ...
 'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...
{'Phoenix', 'Lincoln', 'Carson City', 'Albany', ...
 'Atlanta', 'Juneau', 'Montpelier', 'Salem'})
```

Check three states to see if they are in the map:

```
isKey(US_Capitals, {'Georgia', 'Alaska', 'Wyoming'})
ans =
     1     1     0
```

Check two states and a capital to see if they are all keys in the map:

```
isKey(US_Capitals, {'Georgia'; 'Montpelier'; 'Alaska'})
ans =
     1     0     1
```

Identify the capital city of a specific state, but only attempt this if you know that this state is in the map:

```
S = 'Nebraska';
```

```
if isKey(US_Capitals, S)
sprintf('   The capital of %s is %s', S, US_Capitals(S))
else error('The state of %s is not in the map', S)
end
ans =
   The capital of Nebraska is Lincoln

S =  Montana';
if isKey(US_Capitals, S)
sprintf('   The capital of %s is %s', S, US_Capitals(S))
else error('The state of %s is not in the map', S)
end

??? The state of Montana is not in the map
```

See Also

`containers.Map`, `keys(Map)`, `values(Map)`, `size(Map)`, `length(Map)`, `remove(Map)`, `handle`

Purpose Determine whether input is MATLAB keyword

Syntax

```
tf = iskeyword('str')
iskeyword str
iskeyword
```

Description `tf = iskeyword('str')` returns logical `1` (`true`) if the string `str` is a keyword in the MATLAB language and logical `0` (`false`) otherwise.

`iskeyword str` uses the MATLAB command format.

`iskeyword` returns a list of all MATLAB keywords.

Examples To test if the word `while` is a MATLAB keyword,

```
iskeyword while
ans =
    1
```

To obtain a list of all MATLAB keywords,

```
iskeyword
    'break'
    'case'
    'catch'
    'classdef'
    'continue'
    'else'
    'elseif'
    'end'
    'for'
    'function'
    'global'
    'if'
    'otherwise'
    'parfor'
    'persistent'
    'return'
```

```
'spmd'
'switch'
'try'
'while'
```

See Also `isvarname`, `genvarname`, `is*`

Purpose Array elements that are alphabetic letters

Syntax `tf = isletter('str')`

Description `tf = isletter('str')` returns an array the same size as `str` containing logical `1` (`true`) where the elements of `str` are letters of the alphabet and logical `0` (`false`) where they are not.

Examples Find the letters in character array `s`.

```
s = 'A1,B2,C3';

isletter(s)
ans =
      1     0     0     1     0     0     1     0
```

See Also `ischar`, `isspace`, `isstrprop`, `iscellstr`, `isnumeric`, `char`, `strings`, `isa`, `is*`

islogical

Purpose

Determine whether input is logical array

Syntax

```
tf = islogical(A)
```

Description

`tf = islogical(A)` returns logical `1` (`true`) if `A` is a logical array and logical `0` (`false`) otherwise.

Examples

Given the following cell array,

```
C{1,1} = pi;                  % double
C{1,2} = 1;                   % double
C{1,3} = ispc;                % logical
C{1,4} = magic(3)             % double array

C =
    [3.1416]    [1]    [1]    [3x3 double]
```

`islogical` shows that only `C{1,3}` is a logical array.

```
for k = 1:4
x(k) = islogical(C{1,k});
end

x
x =
     0     0     1     0
```

See Also

`logical`, `isnumeric`, `ischar`, `isreal`, , logical operators (`elementwise` and `short-circuit`), `isa`, `is*`

Purpose Determine if version is for Mac OS X platform

Syntax `tf = ismac`

Description `tf = ismac` returns logical `1` (`true`) if the version of MATLAB software is for the Apple Mac OS X platform, and returns logical `0` (`false`) otherwise.

See Also `isunix`, `ispc`, `isstudent`, `is*`

ismember

Purpose Array elements that are members of set

Syntax

```
tf = ismember(A, S)
tf = ismember(A, S, 'rows')
[tf, loc] = ismember(A, S, ...)
```

Description

`tf = ismember(A, S)` returns an array the same size as `A`, containing logical `1` (`true`) where the elements of `A` are in the set `S`, and logical `0` (`false`) elsewhere. In set theory terms, `k` is 1 where `A` ϵ `S`. Inputs `A` and `S` can be numeric or character arrays or cell arrays of strings.

`tf = ismember(A, S, 'rows')`, when `A` and `S` are matrices with the same number of columns, returns a vector containing `1` where the rows of `A` are also rows of `S` and `0` otherwise. You cannot use this syntax if `A` or `S` is a cell array of strings.

`[tf, loc] = ismember(A, S, ...)` returns an array `loc` containing the highest index in `S` for each element in `A` that is a member of `S`. For those elements of `A` that do not occur in `S`, `ismember` returns `0`.

Remarks Because NaN is considered to be not equal to anything, it is never a member of any set.

Examples

```
set = [0 2 4 6 8 10 12 14 16 18 20];
a = (1:5)'
a =
     1
     2
     3
     4
     5

ismember(a, set)
ans =
     0
     1
     0
```

```
     1
     0
set = [5 2 4 2 8 10 12 2 16 18 20 3];
[tf, index] = ismember(a, set);

index
index =
     0
     8
    12
     3
     1
```

See Also `issorted`, `intersect`, `setdiff`, `setxor`, `union`, `unique`, `is*`

ismethod

Purpose

Determine whether input is COM object method

Syntax

```
tf = h.ismethod('methodname')
tf = ismethod(h, 'methodname')
```

Description

`tf = h.ismethod('methodname')` returns logical `1` (`true`) if the specified `methodname` is a method you can call on COM object `h`. Otherwise, returns logical `0` (`false`).

`tf = ismethod(h, 'methodname')` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Test members of an instance of a Microsoft Excel application:

```
h = actxserver ('Excel.Application');
ismethod(h, 'SaveWorkspace')
```

MATLAB displays `true`, `SaveWorkspace` is a method.

Try the same test on `UsableWidth`:

```
ismethod(h, 'UsableWidth')
```

MATLAB displays `false`, `UsableWidth` is not a method; it is a property.

See Also

`methods` | `methodsview` | `isprop` | `isevent` | `isobject` | `class`

How To

-

Purpose

Array elements that are NaN

Syntax

`TF = isnan(A)`

Description

`TF = isnan(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of `A` are `NaN`s and logical 0 (`false`) where they are not. For a complex number `z`, `isnan(z)` returns `1` if either the real or imaginary part of `z` is `NaN`, and `0` if both the real and imaginary parts are finite or `Inf`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2]

isnan(1./a)
Warning: Divide by zero.

ans =
     0     0     0     0     0

isnan(0./a)
Warning: Divide by zero.

ans =
     0     0     1     0     0
```

See Also

`isfinite`, `isinf`, `is*`

isnumeric

Purpose

Determine whether input is numeric array

Syntax

```
tf = isnumeric(A)
```

Description

`tf = isnumeric(A)` returns logical `1` (`true`) if `A` is a numeric array and logical `0` (`false`) otherwise. For example, sparse arrays and double-precision arrays are numeric, while strings, cell arrays, and structure arrays and logicals are not.

Examples

Given the following cell array,

```
C{1,1} = pi;                 % double
C{1,2} = 'John Doe';         % char array
C{1,3} = 2 + 4i;             % complex double
C{1,4} = ispc;               % logical
C{1,5} = magic(3)            % double array

C =
   [3.1416] 'John Doe' [2.0000+ 4.0000i] [1][3x3 double]
```

`isnumeric` shows that all but `C{1,2}` and `C{1,4}` are numeric arrays.

```
for k = 1:5
x(k) = isnumeric(C{1,k});
end

x
x =
     1     0     1     0     1
```

See Also

`isstrprop`, `isnan`, `isreal`, `isprime`, `isfinite`, `isinf`, `isa`, `is*`

Purpose Is input MATLAB object

Syntax `tf = isobject(A)`

Description `tf = isobject(A)` returns `true` if `A` is an object of a MATLAB class. Otherwise, it returns `false`. Handle Graphics objects return `false`. Use `ishghandle` to test for Handle Graphics objects.

Examples Define the following MATLAB class:

```
classdef button < handle
   properties
      UiHandle
   end
   methods
      function obj = button(pos)
         obj.UiHandle = uicontrol('Position',pos,...
            'Style','pushbutton');
      end
   end
end
```

Determine which objects are instances of MATLAB classes. For example:

```
h = button([20 20 60 60]);
isobject(h)
ans =
     1
isobject(h.UiHandle)
ans =
     0
```

Use `isjava` to test for Sun Java objects in MATLAB, where it returns `false` for MATLAB objects:

```
isjava(h)
ans =
     0
```

See Also `class` | `isa` | `is*`

Tutorials

-

Purpose

Compute isosurface end-cap geometry

Syntax

```
fvc = isocaps(X,Y,Z,V,isovalue)
fvc = isocaps(V,isovalue)
fvc = isocaps(...,'enclose')
fvc = isocaps(...,'whichplane')
[f,v,c] = isocaps(...)
isocaps(...)
```

Description

`fvc = isocaps(X,Y,Z,V,isovalue)` computes isosurface end-cap geometry for the volume data `V` at isosurface value `isovalue`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`.

The struct `fvc` contains the face, vertex, and color data for the end-caps and can be passed directly to the `patch` command.

`fvc = isocaps(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isocaps(...,'enclose')` specifies whether the end-caps enclose data values above or below the value specified in `isovalue`. The string *`enclose`* can be either `above` (default) or `below`.

`fvc = isocaps(...,'whichplane')` specifies on which planes to draw the end-caps. Possible values for *`whichplane`* are `all` (default), `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, or `zmax`.

`[f,v,c] = isocaps(...)` returns the face, vertex, and color data for the end-caps in three arrays instead of the struct `fvc`.

`isocaps(...)` without output arguments draws a patch with the computed faces, vertices, and colors.

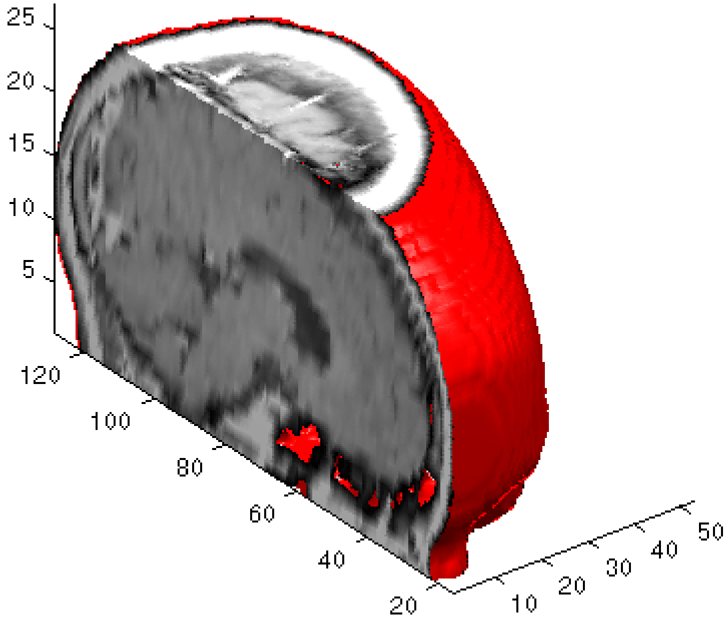
Examples

This example uses a data set that is a collection of MRI slices of a human skull. It illustrates the use of `isocaps` to draw the end-caps on this cutaway volume.

The red `isosurface` shows the outline of the volume (skull) and the end-caps show what is inside of the volume.

The `patch` created from the end-cap data (`p2`) uses interpolated face coloring, which means the gray `colormap` and the light sources determine how it is colored. The isosurface patch (`p1`) used a flat red face color, which is affected by the lights, but does not use the colormap.

```
load mri
D = squeeze(D);
D(:,1:60,:) = [];
p1 = patch(isosurface(D, 5),'FaceColor','red',...
 'EdgeColor','none');
p2 = patch(isocaps(D, 5),'FaceColor','interp',...
 'EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight left; camlight; lighting gouraud
isonormals(D,p1)
```



See Also

`isosurface`, `isonormals`, `smooth3`, `subvolume`, `reducevolume`, `reducepatch`

for more illustrations of isocaps

“Volume Visualization” on page 1-106 for related functions

isocolors

Purpose Calculate isosurface and patch colors

Syntax

```
nc = isocolors(X,Y,Z,C,vertices)
nc = isocolors(X,Y,Z,R,G,B,vertices)
nc = isocolors(C,vertices)
nc = isocolors(R,G,B,vertices)
nc = isocolors(...,PatchHandle)
isocolors(...,PatchHandle)
```

Description

`nc = isocolors(X,Y,Z,C,vertices)` computes the colors of isosurface (patch object) vertices (`vertices`) using color values `C`. Arrays `X`, `Y`, `Z` define the coordinates for the color data in `C` and must be monotonic vectors or 3-D plaid arrays (as if produced by `meshgrid`). The colors are returned in `nc`. `C` must be 3-D (index colors).

`nc = isocolors(X,Y,Z,R,G,B,vertices)` uses `R`, `G`, `B` as the red, green, and blue color arrays (true color).

`nc = isocolors(C,vertices)`, and `nc = isocolors(R,G,B,vertices)` assume `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(C)`.

`nc = isocolors(...,PatchHandle)` uses the vertices from the patch identified by `PatchHandle`.

`isocolors(...,PatchHandle)` sets the `FaceVertexCData` property of the patch specified by `PatchHandle` to the computed colors.

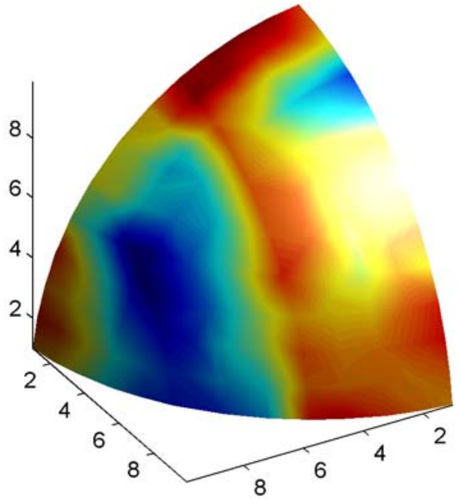
Examples

Indexed Color Data

This example displays an isosurface and colors it with random data using indexed color. (See for information on how patch objects interpret color data.)

```
[x y z] = meshgrid(1:20,1:20,1:20);
```

```
data = sqrt(x.^2 + y.^2 + z.^2);
cdata = smooth3(rand(size(data)),'box',7);
p = patch(isosurface(x,y,z,data,10));
isonormals(x,y,z,data,p);
isocolors(x,y,z,cdata,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);axis tight
camlight; lighting phong;
```

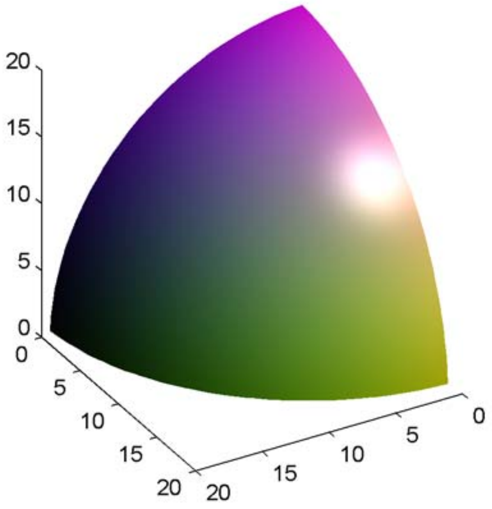


True Color Data

This example displays an isosurface and colors it with true color (RGB) data.

```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(x,y,z,data,20));
isonormals(x,y,z,data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
```

```
isocolors(x,y,z,r/20,g/20,b/20,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);
camlight; lighting phong;
```

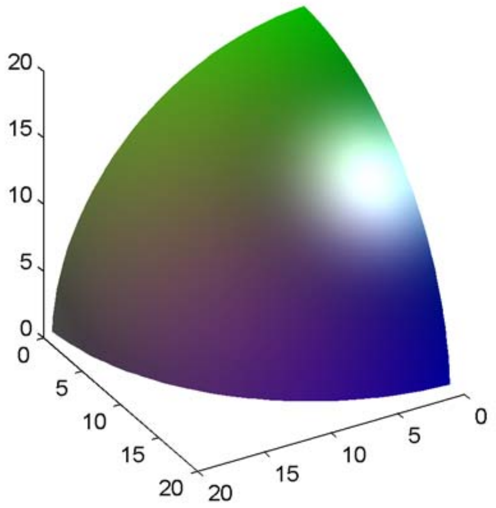


Modified True Color Data

This example uses `isocolors` to calculate the true color data using the isosurface's (patch object's) vertices, but then returns the color data in a variable (`c`) in order to modify the values. It then explicitly sets the isosurface's `FaceVertexCData` to the new data (`1-c`).

```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(data,20));
isonormals(data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
c = isocolors(r/20,g/20,b/20,p);
set(p,'FaceVertexCData',1-c)
```

```
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);
camlight; lighting phong;
```



See Also `isosurface`, `isocaps`, `smooth3`, `subvolume`, `reducevolume`, `reducepatch`, `isonormals`

"Volume Visualization" on page 1-106 for related functions

isonormals

Purpose

Compute normals of isosurface vertices

Syntax

```
n = isonormals(X,Y,Z,V,vertices)
n = isonormals(V,vertices)
n = isonormals(V,p) and n = isonormals(X,Y,Z,V,p)
n = isonormals(...,'negate')
isonormals(V,p) and isonormals(X,Y,Z,V,p)
```

Description

`n = isonormals(X,Y,Z,V,vertices)` computes the normals of the isosurface vertices from the vertex list, `vertices`, using the gradient of the data `V`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The computed normals are returned in `n`.

`n = isonormals(V,vertices)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`n = isonormals(V,p)` and `n = isonormals(X,Y,Z,V,p)` compute normals from the vertices of the patch identified by the handle `p`.

`n = isonormals(...,'negate')` negates (reverses the direction of) the normals.

`isonormals(V,p)` and `isonormals(X,Y,Z,V,p)` set the `VertexNormals` property of the patch identified by the handle `p` to the computed normals rather than returning the values.

Examples

This example compares the effect of different surface normals on the visual appearance of lit isosurfaces. In one case, the triangles used to draw the isosurface define the normals. In the other, the `isonormals` function uses the volume data to calculate the vertex normals based on the gradient of the data points. The latter approach generally produces a smoother-appearing isosurface.

Define a 3-D array of volume data (`cat`, `interp3`):

```
data = cat(3, [0 .2 0; 0 .3 0; 0 0 0], ...
              [.1 .2 0; 0 1 0; .2 .7 0],...
              [0 .4 .2; .2 .4 0;.1 .1 0]);
```

```
data = interp3(data,3,'cubic');
```

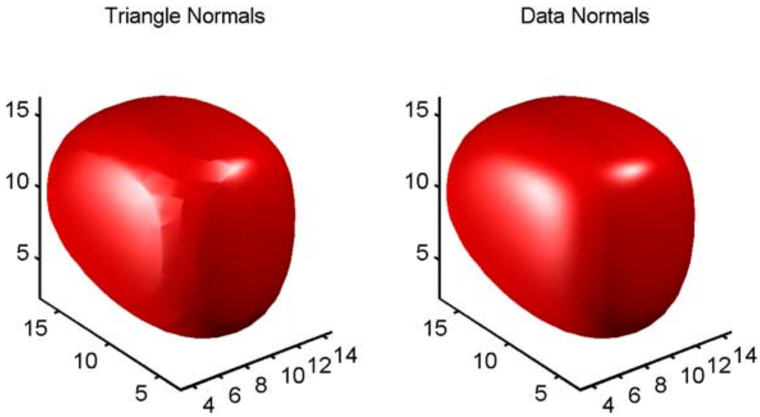
Draw an isosurface from the volume data and add lights. This isosurface uses triangle normals (`patch`, `isosurface`, `view`, `daspect`, `axis`, `camlight`, `lighting`, `title`):

```
subplot(1,2,1)
p1 = patch(isosurface(data,.5),...
'FaceColor','red','EdgeColor','none');
view(3); daspect([1,1,1]); axis tight
camlight; camlight(-80,-10); lighting phong;
title('Triangle Normals')
```

Draw the same lit isosurface using normals calculated from the volume data:

```
subplot(1,2,2)
p2 = patch(isosurface(data,.5),...
    'FaceColor','red','EdgeColor','none');
isonormals(data,p2)
view(3); daspect([1 1 1]); axis tight
camlight;  camlight(-80,-10); lighting phong;
title('Data Normals')
```

These isosurfaces illustrate the difference between triangle and data normals:



See Also

`interp3`, `isosurface`, `isocaps`, `smooth3`, `subvolume`, `reducevolume`, `reducepatch`

"Volume Visualization" on page 1-106 for related functions

Purpose

Extract isosurface data from volume data

Syntax

```
fv = isosurface(X,Y,Z,V,isovalue)
fv = isosurface(V,isovalue)
fvc = isosurface(...,colors)
fv = isosurface(...,'noshare')
fv = isosurface(...,'verbose')
[f,v] = isosurface(...)
[f,v,c] = isosurface(...)
isosurface(...)
```

Description

`fv = isosurface(X,Y,Z,V,isovalue)` computes isosurface data from the volume data `V` at the isosurface value specified in `isovalue`. That is, the isosurface connects points that have the specified value much the way contour lines connect points of equal elevation.

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The structure `fv` contains the faces and vertices of the isosurface, which you can pass directly to the `patch` command.

`fv = isosurface(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isosurface(...,colors)` interpolates the array `colors` onto the scalar field and returns the interpolated values in the `facevertexcdata` field of the `fvc` structure. The size of the `colors` array must be the same as `V`. The `colors` argument enables you to control the color mapping of the isosurface with data different from that used to calculate the isosurface (e.g., temperature data superimposed on a wind current isosurface).

`fv = isosurface(...,'noshare')` does not create shared vertices. This is faster, but produces a larger set of vertices.

`fv = isosurface(...,'verbose')` prints progress messages to the command window as the computation progresses.

`[f,v] = isosurface(...)` or `[f,v,c] = isosurface(...)` returns the faces and vertices (and `faceVertexcCData`) in separate arrays instead of a struct.

`isosurface(...)` with no output arguments, creates a patch in the current axes with the computed faces and vertices. If no current axes exists, a new axes is created with a 3-D view and appropriate lighting.

Special Case Behavior — isosurface Called with No Output Arguments

If there is no current axes and you call `isosurface` with without assigning output arguments, MATLAB creates a new axes, sets it to a 3-D view, and adds lighting to the isosurface graph.

Remarks

You can pass the `fv` structure created by `isosurface` directly to the `patch` command, but you cannot pass the individual faces and vertices arrays (`f`, `v`) to `patch` without specifying property names. For example,

```
patch(isosurface(X,Y,Z,V,isovalue))
```

or

```
[f,v] = isosurface(X,Y,Z,V,isovalue);
patch('Faces',f,'Vertices',v)
```

Examples

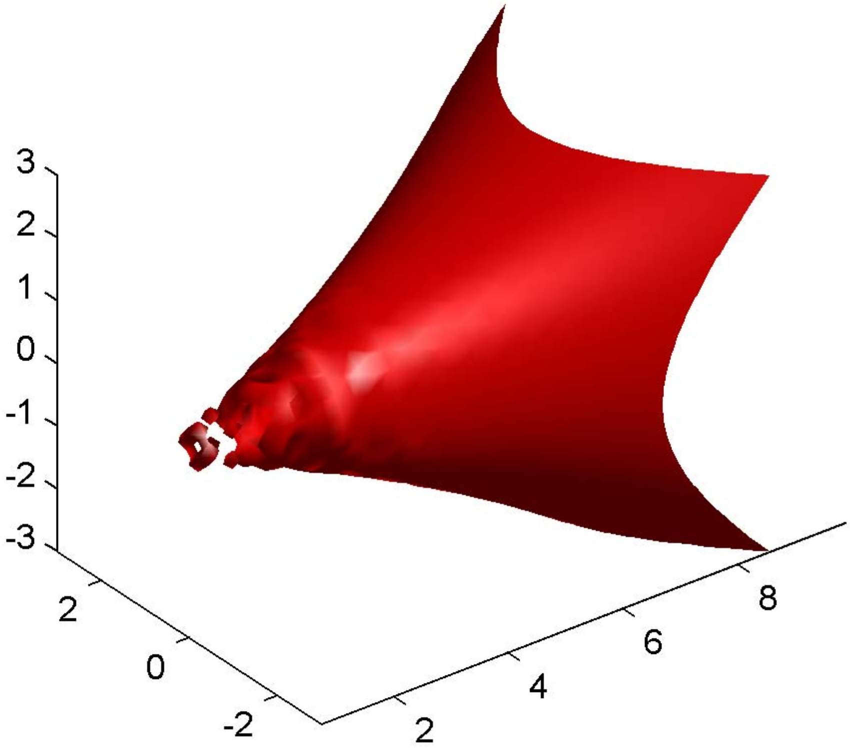
Example 1

This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type `help flow` for more information). The isosurface is drawn at the data value of -3. The statements that follow the `patch` command prepare the isosurface for lighting by

- Recalculating the isosurface normals based on the volume data (`isonormals`)
- Setting the face and edge color (`set`, `FaceColor`, `EdgeColor`)
- Specifying the view (`daspect`, `view`)

- Adding lights (`camlight`, `lighting`)

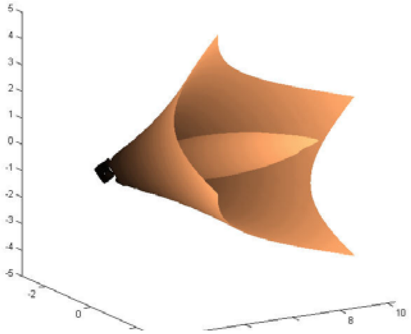
```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p)
set(p,'FaceColor','red','EdgeColor','none');
daspect([1 1 1])
view(3); axis tight
camlight
lighting gouraud
```



Example 2

Visualize the same flow data as above, but color-code the surface to indicate magnitude along the X-axis. Use a sixth argument to `isosurface`, which provides a means to overlay another data set by coloring the resulting isosurface. The `colors` variable is a vector containing a scalar value for each vertex in the isosurface, to be portrayed with the current color map. In this case, it is one of the variables that define the surface, but it could be entirely independent. You can apply a different color scheme by changing the current figure color map.

```
[x,y,z,v] = flow;
[faces,verts,colors] = isosurface(x,y,z,v,-3,x);
patch('Vertices', verts, 'Faces', faces, ...
    'FaceVertexCData', colors, ...
    'FaceColor','interp', ...
    'edgecolor', 'interp');
view(30,-15);
axis vis3d;
colormap copper
```



See Also

`isonormals`, `shrinkfaces`, `smooth3`, `subvolume`

for more examples

“Volume Visualization” on page 1-106 for related functions

ispc

Purpose Determine if version is for Windows (PC) platform

Syntax `tf = ispc`

Description `tf = ispc` returns logical `1` (`true`) if the version of MATLAB software is for the Microsoft Windows platform, and returns logical `0` (`false`) otherwise.

See Also `isunix`, `ismac`, `isstudent`, `is*`

Purpose

Test for existence of preference

Syntax

```
ispref('group','pref')
ispref('group')
ispref('group',{'pref1','pref2',...'prefn'})
```

Description

`ispref('group','pref')` returns `1` if the preference specified by `group` and `pref` exists, and `0` otherwise.

`ispref('group')` returns `1` if the GROUP exists, and `0` otherwise.

`ispref('group',{'pref1','pref2',...'prefn'})` returns a logical array the same length as the cell array of preference names, containing `1` where each preference exists, and `0` elsewhere.

Examples

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

ans =
     1.0
```

See Also

`addpref`, `getpref`, `rmpref`, `setpref`, `uigetpref`, `uisetpref`

Purpose

Array elements that are prime numbers

Syntax

`TF = isprime(A)`

Description

`TF = isprime(A)` returns an array the same size as `A` containing logical 1 (`true`) for the elements of `A` which are prime, and logical 0 (`false`) otherwise. `A` must contain only positive integers.

Examples

```
c = [2 3 0 6 10]

c =
     2     3     0     6    10

isprime(c)

ans =
     1     1     0     0     0
```

See Also

`is*`

Purpose

Determine whether input is COM object property

Syntax

```
tf = h.isprop('propertyname')
tf = isprop(h, 'propertyname')
```

Description

`tf = h.isprop('propertyname')` returns logical `1` (`true`) if the specified `name` is a property of COM object `h`. Otherwise, returns logical `0` (`false`).

`tf = isprop(h, 'propertyname')` is an alternate syntax.

Examples

Test a property of an instance of a Microsoft Excel application:

```
h = actxserver ('Excel.Application');
isprop(h, 'UsableWidth')
```

MATLAB displays `true`, `UsableWidth` is a property.

Try the same test on `SaveWorkspace`:

```
isprop(h, 'SaveWorkspace')
```

MATLAB displays `false`. `SaveWorkspace` is not a property; it is a method.

See Also

`inspect` | `ismethod` | `isevent` |

How To

-

isreal

Purpose

Check if input is real array

Syntax

```
TF = isreal(A)
```

Description

`TF = isreal(A)` returns logical `1` (`true`) if `A` does not have an imaginary part. It returns logical `0` (`false`) otherwise. If `A` has a stored imaginary part of value `0`, `isreal(A)` returns logical `0` (`false`).

Note For `logical` and `char` data classes, `isreal` always returns `true`. For numeric data types, if `A` does not have an imaginary part `isreal` returns `true`; if `A` does have an imaginary part `isreal` returns `false`. For `cell`, `struct`, `function_handle`, and `object` data types, `isreal` always returns `false`.

`~isreal(x)` returns `true` for arrays that have at least one element with an imaginary component. The value of that component can be `0`.

Remarks

If `A` is real, `complex(A)` returns a complex number whose imaginary component is `0`, and `isreal(complex(A))` returns `false`. In contrast, the addition `A + 0i` returns the real value `A`, and `isreal(A + 0i)` returns `true`.

If `B` is real and `A = complex(B)`, then `A` is a complex matrix and `isreal(A)` returns `false`, while `A(m:n)` returns a real matrix and `isreal(A(m:n))` returns `true`.

Because MATLAB software supports complex arithmetic, certain of its functions can introduce significant imaginary components during the course of calculations that appear to be limited to real numbers. Thus, you should use `isreal` with discretion.

Examples

Example 1

If a computation results in a zero-value imaginary component, `isreal` returns `true`.

```
x=3+4i;
y=5-4i;
isreal(x+y)

ans =

     1
```

Example 2

These examples use `isreal` to detect the presence or absence of imaginary numbers in an array. Let

```
x = magic(3);
y = complex(x);
```

`isreal(x)` returns `true` because no element of `x` has an imaginary component.

```
isreal(x)
ans =
     1
```

`isreal(y)` returns `false`, because every element of `x` has an imaginary component, even though the value of the imaginary components is `0`.

```
isreal(y)
ans =
     0
```

This expression detects strictly real arrays, i.e., elements with `0`-valued imaginary components are treated as real.

```
~any(imag(y(:)))
ans =
     1
```

Example 3

Given the following cell array,

```
C{1} = pi;                   % double
C{2} = 'John Doe';           % char array
C{3} = 2 + 4i;               % complex double
C{4} = ispc;                 % logical
C{5} = magic(3);             % double array
C{6} = complex(5,0)          % complex double

C =
  [3.1416]  'John Doe'  [2.0000+ 4.0000i]  [1]  [3x3 double]  [5]
```

`isreal` shows that all but `C{1,3}` and `C{1,6}` are real arrays.

```
for k = 1:6
x(k) = isreal(C{k});
end

x
x =
     1     1     0     1     1    0
```

See Also `complex`, `isnumeric`, `isnan`, `isprime`, `isfinite`, `isinf`, `isa`, `is*`

Purpose Determine whether input is scalar

Syntax `TF = isscalar(A)`

Description `TF = isscalar(A)` returns logical `1` (`true`) if `A` is a 1-by-1 matrix, and logical `0` (`false`) otherwise.

The `A` argument can be a structure or cell array. It also be a MATLAB object, as described in , as long as that object overloads the `size` function.

Examples Test matrix `A` and one element of the matrix:

```
A = rand(5);

isscalar(A)
ans =
     0

isscalar(A(3,2))
ans =
     1
```

See Also `isvector`, `isempty`, `isnumeric`, `islogical`, `ischar`, `isa`, `is*`

Purpose Determine whether set elements are in sorted order

Syntax

```
TF = issorted(A)
TF = issorted(A, 'rows')
```

Description

`TF = issorted(A)` returns logical `1` (`true`) if the elements of `A` are in sorted order, and logical `0` (`false`) otherwise. Input `A` can be a vector or an N-by-1 or 1-by-N cell array of strings. `A` is considered to be sorted if `A` and the output of `sort(A)` are equal.

`TF = issorted(A, 'rows')` returns logical `1` (`true`) if the rows of two-dimensional matrix `A` are in sorted order, and logical `0` (`false`) otherwise. Matrix `A` is considered to be sorted if `A` and the output of `sortrows(A)` are equal.

Note Only the `issorted(A)` syntax supports `A` as a cell array of strings.

Remarks

For character arrays, `issorted` uses ASCII, rather than alphabetical, order.

You cannot use `issorted` on arrays of greater than two dimensions.

Examples

Example 1 — Using issorted on a vector

```
A = [5 12 33 39 78 90 95 107 128 131];

issorted(A)
ans =
     1
```

Example 2 — Using issorted on a matrix

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
```

```
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

issorted(A, 'rows')
ans =
     0

B = sortrows(A)
B =
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
    17    24     1     8    15
    23     5     7    14    16

issorted(B)
ans =
     1
```

Example 3 — Using issorted on a cell array

```
x = {'one'; 'two'; 'three'; 'four'; 'five'};
issorted(x)
ans =
     0

y = sort(x)
y =
    'five'
    'four'
    'one'
    'three'
    'two'

issorted(y)
```

See Also `sort`, `sortrows`, `ismember`, `unique`, `intersect`, `union`, `setdiff`, `setxor`, `is*`

Purpose

Array elements that are space characters

Syntax

`tf = isspace('str')`

Description

`tf = isspace('str')` returns an array the same size as `'str'` containing logical `1` (`true`) where the elements of `str` are ASCII white spaces and logical `0` (`false`) where they are not. White spaces in ASCII are space, newline, carriage return, tab, vertical tab, or formfeed characters.

Examples

```
isspace('  Find spa ces ')
  Columns 1 through 13
     1    1    0    0    0    0    1    0    0    0    1    0    0
  Columns 14 through 15
     0    1
```

See Also

`isletter`, `isstrprop`, `ischar`, `strings`, `isa`, `is*`

issparse

Purpose Determine whether input is sparse

Syntax `TF = issparse(S)`

Description `TF = issparse(S)` returns logical 1 (`true`) if the storage class of `S` is sparse and logical 0 (`false`) otherwise.

See Also `is*`, `sparse`, `full`

Purpose

Determine whether input is character array

Note Use the `ischar` function in place of `isstr`. The `isstr` function will be removed in a future version of MATLAB.

See Also

`ischar`, `isa`, `is*`

isstrprop

Purpose Determine whether string is of specified category

Syntax

```
tf = isstrprop('str', 'category')
```

Description `tf = isstrprop('str', 'category')` returns a logical array the same size as `str` containing logical `1` (`true`) where the elements of `str` belong to the specified *`category`*, and logical `0` (`false`) where they do not.

The `str` input can be a character array, cell array, or any MATLAB numeric type. If `str` is a cell array, then the return value is a cell array of the same shape as `str`.

The *`category`* input can be any of the strings shown in the left column below:

Category	Description
`alpha`	`True` for those elements of `str` that are alphabetic
`alphanum`	`True` for those elements of `str` that are alphanumeric
`cntrl`	`True` for those elements of `str` that are control characters (for example, `char(0:20)`)
`digit`	`True` for those elements of `str` that are numeric digits

`graphic`	True for those elements of str that are graphic characters. These are all values that represent any characters except for the following:
	`unassigned, space, line separator,`
	`paragraph separator, control characters,`
	`Unicode format control characters,`
	`private user-defined characters,`
	`Unicode surrogate characters,`
	`Unicode other characters`

`lower`	`True` for those elements of `str` that are lowercase letters
`print`	`True` for those elements of `str` that are graphic characters, plus `char(32)`

Category	Description
`punct`	`True` for those elements of `str` that are punctuation characters
`wspace`	`True` for those elements of `str` that are white-space characters. This range includes the ANSI® C definition of white space, `{' ','\t','\n','\r','\v','\f'}`.
`upper`	`True` for those elements of `str` that are uppercase letters
`xdigit`	`True` for those elements of `str` that are valid hexadecimal digits

Remarks

Numbers of type `double` are converted to `int32` according to MATLAB rules of double-to-integer conversion. Numbers of type `int64` and `uint64` bigger than `int32(inf)` saturate to `int32(inf)`.

MATLAB classifies the elements of the `str` input according to the Unicode definition of the specified category. If the numeric value of an element in the input array falls within the range that defines a Unicode character category, then this element is classified as being of that category. The set of Unicode character codes includes the set of ASCII character codes, but also covers a large number of languages beyond the scope of the ASCII set. The classification of characters is dependent on the global location of the platform on which MATLAB is installed.

Examples

Test for alphabetic characters in a string:

```
A = isstrprop('abc123def', 'alpha')
A =
   1 1 1 0 0 0 1 1 1
```

Test for numeric digits in a string:

```
A = isstrprop('abc123def', 'digit')
A =
   0 0 0 1 1 1 0 0 0
```

Test for hexadecimal digits in a string:

```
A = isstrprop('abcd1234efgh', 'xdigit')
A =
   1 1 1 1 1 1 1 1 1 1 0 0
```

Test for numeric digits in a character array:

```
A = isstrprop(char([97 98 99 49 50 51 101 102 103]), ...
              'digit')
A =
   0 0 0 1 1 1 0 0 0
```

Test for alphabetic characters in a two-dimensional cell array:

```
A = isstrprop({'abc123def';'456ghi789'}, 'alpha')
A =
   [1x9 logical]
   [1x9 logical]

A{:,:}
ans =
   1 1 1 0 0 0 1 1 1
   0 0 0 1 1 1 0 0 0
```

Test for white-space characters in a string:

```
A = isstrprop(sprintf('a bc\n'), 'wspace')
A =
   0 1 0 0 1
```

See Also

`strings`, `ischar`, `isletter`, `isspace`, `iscellstr`, `isnumeric`, `isa`, `is*`

Purpose

Determine whether input is structure array

Syntax

`tf = isstruct(A)`

Description

`tf = isstruct(A)` returns logical `1` (`true`) if `A` is a MATLAB structure and logical `0` (`false`) otherwise.

Examples

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];

isstruct(patient)

ans =

     1
```

See Also

`struct`, `isfield`, `iscell`, `ischar`, `isobject`, `isnumeric`, `islogical`, `isa`, `is*`, dynamic field names

isstudent

Purpose Determine if version is Student Version

Syntax `tf = isstudent`

Description `tf = isstudent` returns logical `1` (`true`) if the version of MATLAB software is the Student Version, and returns logical `0` (`false`) for commercial versions.

See Also `ver`, `version`, `license`, `ispc`, `isunix`, `is*`

Tiff.isTiled

Purpose

Determine if tiled image

Syntax

```
bool = tiffobj.isTiled()
```

Description

`bool = tiffobj.isTiled()` returns `true` if the image has a tiled organization and `false` if the image has a stripped organization.

Examples

Open a `Tiff` object and check if the image in the TIFF file has a tiled or stipped organization. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r');
tf = t.isTiled();
```

References

This method corresponds to the `TIFFIsTiled` 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`.

Tutorials

-
-

isunix

Purpose

Determine if version is for UNIX platform

Syntax

```
tf = isunix
```

Description

`tf = isunix` returns logical `1` (`true`) if the version of MATLAB software is for the UNIX[10] platform, and returns logical `0` (`false`) otherwise.

See Also

`ispc`, `ismac`, `isstudent`, `is*`

---

[10] UNIX is a registered trademark of The Open Group in the United States and other countries

Purpose Is object valid handle class object

Syntax `Hl = isvalid(Hobj)`

Description `Hl = isvalid(Hobj)` returns a logical array (or scalar if `Hobj` is scalar) in which each element is `true` if the corresponding element in `Hobj` is a valid handle. This method is `Sealed`, so you cannot override it in a handle subclass.

Note This method does not work with Handle Graphics objects. To determine the validity of a Handle Graphics object handle, use the `ishandle` function.

See Also `delete (handle)`, `handle`

isvalid (serial)

Purpose

Determine whether serial port objects are valid

Syntax

```
out = isvalid(obj)
```

Description

`out = isvalid(obj)` returns the logical array `out`, which contains a `0` where the elements of the serial port object, `obj` are invalid serial port objects and a `1` where the elements of `obj` are valid serial port objects.

Remarks

`obj` becomes invalid after it is removed from memory with the `delete` function. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the `clear` command.

Example

Suppose you create the following two serial port objects.

```
s1 = serial('COM1');
s2 = serial('COM1');
```

`s2` becomes invalid after it is deleted.

```
delete(s2)
```

`isvalid` verifies that `s1` is valid and `s2` is invalid.

```
sarray = [s1 s2];
isvalid(sarray)
ans =
     1     0
```

See Also

Functions

`clear`, `delete`

Purpose Determine whether timer object is valid

Syntax

```
out = isvalid(obj)
```

Description `out = isvalid(obj)` returns a logical array, `out`, that contains a `0` where the elements of `obj` are invalid timer objects and a `1` where the elements of `obj` are valid timer objects.

An invalid timer object is an object that has been deleted and cannot be reused. Use the `clear` command to remove an invalid timer object from the workspace.

Examples Create a valid timer object.

```
t = timer;
out = isvalid(t)
out =

    1
```

Delete the timer object, making it invalid.

```
delete(t)
out1 = isvalid(t)
out1 =

    0
```

See Also `timer`, `delete(timer)`

isvarname

Purpose

Determine whether input is valid variable name

Syntax

```
tf = isvarname('str')
isvarname str
```

Description

`tf = isvarname('str')` returns logical `1` (`true`) if the string `str` is a valid MATLAB variable name and logical `0` (`false`) otherwise. A valid variable name is a character string of letters, digits, and underscores, totaling not more than `namelengthmax` characters and beginning with a letter.

MATLAB keywords are not valid variable names. Type the command `iskeyword` with no input arguments to see a list of MATLAB keywords.

`isvarname str` uses the MATLAB command format.

Examples

This variable name is valid:

```
isvarname foo
ans =
   1
```

This one is not because it starts with a number:

```
isvarname 8th_column
ans =
   0
```

If you are building strings from various pieces, place the construction in parentheses.

```
d = date;

isvarname(['Monday_', d(1:2)])
ans =
   1
```

See Also

`genvarname`, `isglobal`, `iskeyword`, `namelengthmax`, `is*`

Purpose

Determine whether input is vector

Syntax

`TF = isvector(A)`

Description

`TF = isvector(A)` returns logical `1` (`true`) if `A` is a 1-by-`N` or `N`-by-1 vector where `N >= 0`, and logical `0` (`false`) otherwise.

The `A` argument can also be a MATLAB object, as described in , as long as that object overloads the `size` function.

Examples

Test matrix `A` and its row and column vectors:

```
A = rand(5);

isvector(A)
ans =
     0

isvector(A(3, :))
ans =
     1

isvector(A(:, 2))
ans =
     1
```

See Also

`isscalar`, `isempty`, `isnumeric`, `islogical`, `ischar`, `isa`, `is*`

Purpose

Imaginary unit

Syntax

```
j
x+yj
x+j*y
```

Description

Use the character `j` in place of the character `i`, if desired, as the imaginary unit.

As the basic imaginary unit `sqrt(-1)`, `j` is used to enter complex numbers. Since `j` is a function, it can be overridden and used as a variable. This permits you to use `j` as an index in `for` loops, etc.

It is possible to use the character `j` without a multiplication sign as a suffix in forming a numerical constant.

Examples

```
Z = 2+3j
Z = x+j*y
Z = r*exp(j*theta)
```

See Also

`conj`, `i`, `imag`, `real`

Purpose

Add entries to dynamic Sun Java class path

Syntax

```
javaaddpath('dpath')
javaaddpath('dpath', '-end')
```

Description

`javaaddpath('dpath')` adds one or more directories or JAR files to the beginning of the current dynamic Java class path. `dpath` is a string or cell array of strings containing the directory or JAR file. (See the Remarks section for a description of static and dynamic Java paths.)

`javaaddpath('dpath', '-end')` adds one or more directories or files to the end of the current dynamic Java path.

Remarks

The Java path consists of two segments: a static path (read only at startup) and a dynamic path. The MATLAB software always searches the static path (defined in `classpath.txt`) before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path. Use `javaclasspath` to see the current static and dynamic Java paths.

MATLAB calls the `clear java` command whenever you change the dynamic path.

Path Type	**Description**
Static	Loaded at the start of each MATLAB session from the file `classpath.txt`. The static Java path offers better Java class loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file `classpath.txt` and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the `javaclasspath` function. You can define the dynamic path (using `javaclasspath`), modify the path (using `javaaddpath` and `javarmpath`), and refresh the Java class definitions for all classes on the dynamic path (using `clear java`) without restarting MATLAB.

Examples

Create function to set initial dynamic Java class path:

```
function setdynpath
javaclasspath({
   'C:\Work\Java\ClassFiles', ...
   'C:\Work\JavaTest\curvefit.jar', ...
   'C:\Work\JavaTest\timer.jar', ...
   'C:\Work\JavaTest\patch.jar'});
% end of file
```

Call this function to set up your dynamic class path. Then, use the `javaclasspath` function with no arguments to display all current static and dynamic paths:

```
setdynpath;

javaclasspath

         STATIC JAVA PATH

   D:\Sys0\Java\util.jar
   D:\Sys0\Java\widgets.jar
   D:\Sys0\Java\beans.jar
             .
             .

         DYNAMIC JAVA PATH

   C:\Work\Java\ClassFiles
   C:\Work\JavaTest\curvefit.jar
   C:\Work\JavaTest\timer.jar
   C:\Work\JavaTest\patch.jar
```

At some later time, add the following two entries to the dynamic path. (Calling `javaaddpath` clears all variables from the workspace). One entry specifies a directory and the other a Java Archive (JAR) file.

When you add a directory to the path, MATLAB includes all files in that directory as part of the path:

```
javaaddpath({
   'C:\Work\Java\Curvefit\Test', ...
   'C:\Work\Java\mywidgets.jar'});
```

Use `javaclasspath` with just an output argument to return the dynamic path alone:

```
p = javaclasspath
p =
   'C:\Work\Java\ClassFiles'
   'C:\Work\JavaTest\curvefit.jar'
   'C:\Work\JavaTest\timer.jar'
   'C:\Work\JavaTest\patch.jar'
   'C:\Work\Java\Curvefit\Test'
   'C:\Work\Java\mywidgets.jar'
```

Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

If you modify one or more classes that are defined on the dynamic path, you need to clear the former definition for those classes from MATLAB memory. You can clear all dynamic Java class definitions from memory using:

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

Use `javarmpath` to remove a file or directory from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

Other Examples

Add a JAR file from an internet URL to your dynamic Java path:

```
javaaddpath http://www.example.com/my.jar
```

Add the current directory with the following statement:

```
javaaddpath(pwd)
```

See Also

`javaclasspath`, `javarmpath`, `clear`

See for more information.

Purpose

Construct Sun Java array

Syntax

`javaArray('package_name.class_name',x1,...,xn)`

Description

`javaArray('package_name.class_name',x1,...,xn)` constructs an empty Java array capable of storing objects of Java class, `'class_name'`. The dimensions of the array are `x1` by `...` by `xn`. You must include the package name when specifying the class.

The array that you create with `javaArray` is equivalent to the array that you would create with the Java code

```
A = new class_name[x1]...[xn];
```

Examples

The following example constructs and populates a 4-by-5 array of `java.lang.Double` objects.

```
dblArray = javaArray ('java.lang.Double', 4, 5);
for m = 1:4
   for n = 1:5
   dblArray(m,n) = java.lang.Double((m*10) + n);
   end
end

dblArray

dblArray =
java.lang.Double[][]:
     [11]    [12]    [13]    [14]    [15]
     [21]    [22]    [23]    [24]    [25]
     [31]    [32]    [33]    [34]    [35]
     [41]    [42]    [43]    [44]    [45]
```

See Also

`javaObject`, `javaMethod`, `class`, `methodsview`, `isjava`

Purpose

Generate error message based on Sun Java feature support

Syntax

```
javachk(feature)
javachk(feature, component)
```

Description

`javachk(feature)` returns a generic error message if the specified Java `feature` is not available in the current MATLAB session. If it is available, `javachk` returns an empty matrix. Possible `feature` arguments are shown in the following table.

Feature	Description
`'awt'`	Abstract Window Toolkit components[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

`javachk(feature, component)` works the same as the above syntax, except that the specified `component` is also named in the error message. (See the example below.)

Examples

The following M-file displays an error with the message "`CreateFrame is not supported on this platform.`" when run in a MATLAB session in which the AWT's GUI components are not available. The second argument to `javachk` specifies the name of the M-file, which is then included in the error message generated by MATLAB.

```
javamsg = javachk('awt', mfilename);
if isempty(javamsg)
   myFrame = java.awt.Frame;
   myFrame.setVisible(1);
else
   error(javamsg);
end
```

See Also

`usejava`

javaclasspath

Purpose

Get and set Sun Java class path

Syntax

```
javaclasspath
javaclasspath('-dynamic')
javaclasspath('-static')
dpath = javaclasspath
spath = javaclasspath('-static')
jpath = javaclasspath('-all')
javaclasspath(dpath)
javaclasspath(dpath1, dpath2)
javaclasspath(statusmsg)
```

Description

`javaclasspath` displays the static and dynamic segments of the Java path. (See the Remarks section, below, for a description of static and dynamic Java paths.)

`javaclasspath('-dynamic')` displays the dynamic Java path.

`javaclasspath('-static')` displays the static Java path.

`dpath = javaclasspath` returns the dynamic segment of the Java path in cell array, `dpath`. If no dynamic paths are defined, `javaclasspath` returns an empty cell array.

`spath = javaclasspath('-static')` returns the static segment of the Java path in cell array, `spath`. No path information is displayed unless you specify an output variable. If no static paths are defined, `javaclasspath` returns an empty cell array.

`jpath = javaclasspath('-all')` returns the entire Java path in cell array, `jpath`. The returned cell array contains first the static segment of the path, and then the dynamic segment. No path information is displayed unless you specify an output variable. If no dynamic paths are defined, `javaclasspath` returns an empty cell array.

`javaclasspath(dpath)` changes the dynamic Java path to `dpath`, where `dpath` can be a string or cell array of strings representing path entries. Relative paths are converted to absolute paths. Uses the `clear java` command to refresh the classes defined on the dynamic Java path.

`javaclasspath(dpath1, dpath2)` changes the dynamic Java path to the concatenation of the two paths `dpath1` and `dpath2`, where `dpath1` and `dpath2` can be a string or cell array of strings representing path entries. Relative paths are converted to absolute paths. Uses the `clear java` command to refresh the classes defined on the dynamic Java path.

`javaclasspath(`*`statusmsg`*`)` enables or disables the display of status messages from the `javaclasspath`, `javaaddpath`, and `javarmpath` functions. Values for the *`statusmsg`* argument are shown in the following table:

statusmsg	Description
`'-v1'`	Display status messages while loading the Java path from the file system.
`'-v0'`	Do not display status messages. This is the default.

Remarks

The Java path consists of two segments: a static path (read only at startup) and a dynamic path. The MATLAB software always searches the static path (defined in `classpath.txt`) before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path. Use `javaclasspath` to see the current static and dynamic Java paths.

MATLAB calls the `clear java` command whenever you change the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the file `classpath.txt`. The static Java path offers better Java class loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file `classpath.txt` and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the `javaclasspath` function. You can define the dynamic path (using `javaclasspath`), modify the path (using `javaaddpath` and `javarmpath`), and refresh the Java class definitions for all classes on the dynamic path (using `clear java`) without restarting MATLAB.

Examples

Create a function to set your initial dynamic Java class path:

```
function setdynpath
javaclasspath({
   'C:\Work\Java\ClassFiles', ...
   'C:\Work\JavaTest\curvefit.jar', ...
   'C:\Work\JavaTest\timer.jar', ...
   'C:\Work\JavaTest\patch.jar'});
%             end of file
```

Call this function to set up your dynamic class path. Then, use the `javaclasspath` function with no arguments to display all current static and dynamic paths:

```
setdynpath;

javaclasspath

         STATIC JAVA PATH

   D:\Sys0\Java\util.jar
   D:\Sys0\Java\widgets.jar
   D:\Sys0\Java\beans.jar
```

```
                .
                .

            DYNAMIC JAVA PATH

        C:\Work\Java\ClassFiles
        C:\Work\JavaTest\curvefit.jar
        C:\Work\JavaTest\timer.jar
        C:\Work\JavaTest\patch.jar
```

At some later time, add the following two entries to the dynamic path. One entry specifies a directory and the other a Java Archive (JAR) file. When you add a directory to the path, The MATLAB software includes all files in that directory as part of the path:

```
javaaddpath({
   'C:\Work\Java\Curvefit\Test', ...
   'C:\Work\Java\mywidgets.jar'});
```

Use `javaclasspath` with just an output argument to return the dynamic path alone:

```
p = javaclasspath
p =
   'C:\Work\Java\ClassFiles'
   'C:\Work\JavaTest\curvefit.jar'
   'C:\Work\JavaTest\timer.jar'
   'C:\Work\JavaTest\patch.jar'
   'C:\Work\Java\Curvefit\Test'
   'C:\Work\Java\mywidgets.jar'
```

Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

If, at some time, you modify one or more classes that are defined on the dynamic path, you will need to clear the former definition for those classes from MATLAB memory. You can clear all dynamic Java class definitions from memory using:

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

Use `javarmpath` to remove a file or directory from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

See Also

`javaaddpath`, `javarmpath`, `clear`

Purpose

Call Sun Java method

Syntax

```
javaMethod('MethodName','ClassName',x1,...,xn)
javaMethod('MethodName',J,x1,...,xn)
```

Description

`javaMethod('`*`MethodName`*`','`*`ClassName`*`',x1,...,xn)` calls the static method *`MethodName`* in the class *`ClassName`*, with the argument list `x1,...,xn`.

`javaMethod('`*`MethodName`*`',J,x1,...,xn)` calls the nonstatic method *`MethodName`* on the object `J`, with the argument list `x1,...,xn`.

Remarks

Use the `javaMethod` function to:

- Use methods having names longer than 31 characters.
- Specify the method you want to invoke at run-time, for example, as input from an application user.

The `javaMethod` function enables you to use methods having names longer than 31 characters. This is the only way you can invoke such a method in the MATLAB software. For example:

```
javaMethod('DataDefinitionAndDataManipulationTransactions', T);
```

With `javaMethod`, you can also specify the method to be invoked at run-time. In this situation, your code calls `javaMethod` with a string variable in place of the method name argument. When you use `javaMethod` to invoke a static method, you can also use a string variable in place of the class name argument.

Note Typically, you do not need to use `javaMethod`. Use the default MATLAB syntax for invoking a Java methods instead. Use `javaMethod` for the cases described above.

Examples

To invoke the static Java method `isNaN` on class `java.lang.Double` type:

```
javaMethod('isNaN','java.lang.Double',2.2)
```

The following example invokes the nonstatic method `setMonth`, where `myDate` is a `java.util.Date` object.

```
myDate = java.util.Date;
javaMethod('setMonth', myDate, 3);
```

See Also

`javaArray`, `javaObject`, `import`, `methods`, `isjava`, `javaMethodEDT`

javaMethodEDT

Purpose Call Sun Java method from Event Dispatch Thread (EDT)

Syntax `javaMethodEDT('MethodName',JavaObject,x1,...,xn)`

Description `javaMethodEDT('MethodName',JavaObject,x1,...,xn)` calls the named method on the specified Java object from the Event Dispatch Thread (EDT), with the argument list `x1,...,xn`.

See Also `javaObjectEDT`, `javaMethod`

javaObject

Purpose

Construct Sun Java object

Syntax

`javaObject('ClassName',x1,...,xn)`

Description

`javaObject('ClassName',x1,...,xn)` calls the Java constructor for class *`ClassName`* with the argument list that matches `x1,...,xn`, to return a new object.

If there is no constructor that matches the class name and argument list passed to `javaObject`, an error occurs.

Remarks

Use the `javaObject` function to:

- Use classes having names with more than 31 consecutive characters.
- Specify the class for an object at run-time, for example, as input from an application user.

The default MATLAB constructor syntax requires that no segment of the input class name be longer than 31 characters. (A *name segment*, is any portion of the class name before, between, or after a period. For example, there are three segments in class, `java.lang.String`.) Any class name segment that exceeds 31 characters is truncated by MATLAB. In the rare case where you need to use a class name of this length, you must use `javaObject` to instantiate the class.

The `javaObject` function also allows you to specify the Java class for the object being constructed at run-time. In this situation, you call `javaObject` with a string variable in place of the class name argument.

```
class  = 'java.lang.String';
text = 'hello';
strObj = javaObject(class, text);
```

In the usual case, when the class to instantiate is known at development time, it is more convenient to use the MATLAB constructor syntax. For example, to create a `java.lang.String` object, type:

```
strObj = java.lang.String('hello');
```

Note Typically, you do not need to use `javaObject`. Use the default MATLAB syntax for instantiating a Java class instead. Use `javaObject` for the cases described above.

Examples

The following example constructs and returns a Java object of class `java.lang.String`:

```
strObj = javaObject('java.lang.String','hello')
```

See Also

`javaArray`, `javaMethod`, `import`, `methods`, `fieldnames`, `isjava`, `javaObjectEDT`

javaObjectEDT

Purpose Construct Sun Java object on Event Dispatch Thread (EDT)

Syntax `javaObjectEDT('`*`ClassName`*`',x1,...,xn)`

Description `javaObjectEDT('`*`ClassName`*`',x1,...,xn)` instantiates a new Java object and returns a handle to it. Constructor parameters `x1,...,xn` may be passed in following the class name. If no parameters are specified, the no-argument constructor is called.

See Also `javaMethodEDT`, `javaObject`

Purpose

Remove entries from dynamic Sun Java class path

Syntax

```
javarmpath('dpath')
javarmpath dpath1 dpath2 ... dpathN
javarmpath(v1, v2, ..., vN)
```

Description

`javarmpath('dpath')` removes a directory or file from the current dynamic Java path. `dpath` is a string containing the directory or file specification. (See the Remarks section, below, for a description of static and dynamic Java paths.)

`javarmpath dpath1 dpath2 ...  dpathN` removes those directories and files specified by `dpath1`, `dpath2`, `...`, `dpathN` from the dynamic Java path. Each input argument is a string containing a directory or file specification.

`javarmpath(v1, v2, ..., vN)` removes those directories and files specified by `v1`, `v2`, `...`, `vN` from the dynamic Java path. Each input argument is a variable to which a directory or file specification is assigned.

Remarks

The Java path consists of two segments: a static path (read only at startup) and a dynamic path. The MATLAB software always searches the static path (defined in `classpath.txt`) before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path. Use `javaclasspath` to see the current static and dynamic Java paths.

MATLAB calls the `clear java` command whenever you change the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the file `classpath.txt`. The static Java path offers better Java class loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file `classpath.txt` and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the `javaclasspath` function. You can define the dynamic path (using `javaclasspath`), modify the path (using `javaaddpath` and `javarmpath`), and refresh the Java class definitions for all classes on the dynamic path (using `clear java`) without restarting MATLAB.

Examples

Create a function to set your initial dynamic Java class path:

```
function setdynpath
javaclasspath({
   'C:\Work\Java\ClassFiles', ...
   'C:\Work\JavaTest\curvefit.jar', ...
   'C:\Work\JavaTest\timer.jar', ...
   'C:\Work\JavaTest\patch.jar'});
% end of file
```

Call this function to set up your dynamic class path. Then, use the `javaclasspath` function with no arguments to display all current static and dynamic paths:

```
setdynpath;

javaclasspath

         STATIC JAVA PATH

   D:\Sys0\Java\util.jar
   D:\Sys0\Java\widgets.jar
   D:\Sys0\Java\beans.jar
```

```
                    .
                    .

                DYNAMIC JAVA PATH

            C:\Work\Java\ClassFiles
            C:\Work\JavaTest\curvefit.jar
            C:\Work\JavaTest\timer.jar
            C:\Work\JavaTest\patch.jar
```

At some later time, add the following two entries to the dynamic path. One entry specifies a directory and the other a Java Archive (JAR) file. When you add a directory to the path, MATLAB includes all files in that directory as part of the path:

```
javaaddpath({
   'C:\Work\Java\Curvefit\Test', ...
   'C:\Work\Java\mywidgets.jar'});
```

Use `javaclasspath` with just an output argument to return the dynamic path alone:

```
p = javaclasspath
p =
   'C:\Work\Java\ClassFiles'
   'C:\Work\JavaTest\curvefit.jar'
   'C:\Work\JavaTest\timer.jar'
   'C:\Work\JavaTest\patch.jar'
   'C:\Work\Java\Curvefit\Test'
   'C:\Work\Java\mywidgets.jar'
```

Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

If, at some time, you modify one or more classes that are defined on the dynamic path, you will need to clear the former definition for those classes from MATLAB memory. You can clear all dynamic Java class definitions from memory using:

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

Use `javarmpath` to remove a file or directory from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

See Also

`javaclasspath`, `javaaddpath`, `clear`

Purpose

Input from keyboard

Syntax

```
keyboard
```

Description

`keyboard` , when placed in an M-file, stops execution of the file and gives control to the keyboard. The special status is indicated by a `K` appearing before the prompt. You can examine or change variables; all MATLAB commands are valid. This keyboard mode is useful for debugging your M-files..

To terminate the keyboard mode, type the command

```
return
```

then press the **Return** key.

See Also

`dbstop`, `input`, `quit`, `pause`, `return`

keys (Map)

Purpose

Return all keys of `containers.Map` object

Syntax

```
k = keys(M)
```

Description

`k = keys(M)` returns cell array `k` that contains all of the keys stored in `Map` object `M`.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

Examples

Construct a `Map` object that relates states in the United States to their capital cities:

```
US_Capitals = containers.Map( ...
{'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...
{'Atlanta', 'Juneau', 'Montpelier', 'Salem'})
```

Use the keys and values methods to list all keys and values in the map:

```
keys(US_Capitals)
ans =
    'Arizona'    'Nebraska'    'New York'    'Oregon'

values(US_Capitals)
ans =
    'Phoenix'    'Lincoln'    'Albany'    'Salem'
```

Use the map to look up a capital when given a specific state:

```
sprintf('   The capital of %s is %s', ...
'Alaska', US_Capitals('Alaska'))
ans =
   The capital of Alaska is Juneau
```

See Also

`containers.Map`, `values(Map)`, `size(Map)`, `length(Map)isKey(Map)`, `remove(Map)`, `handle`

Purpose

Kronecker tensor product

Syntax

```
K = kron(X,Y)
```

Description

`K = kron(X,Y)` returns the Kronecker tensor product of `X` and `Y`. The result is a large array formed by taking all possible products between the elements of `X` and those of `Y`. If `X` is `m`-by-`n` and `Y` is `p`-by-`q`, then `kron(X,Y)` is `m*p`-by-`n*q`.

Examples

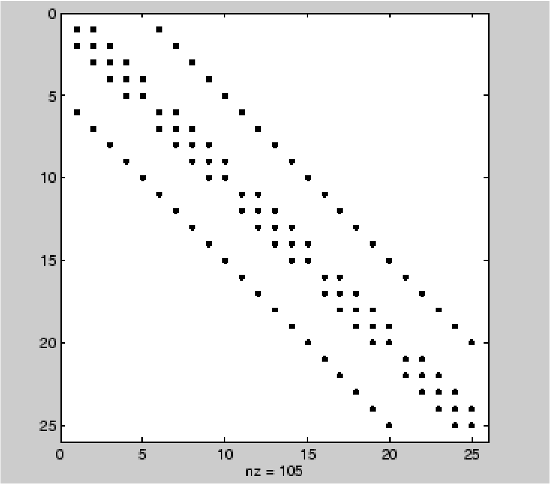
If `X` is 2-by-3, then `kron(X,Y)` is

```
[ X(1,1)*Y X(1,2)*Y X(1,3)*Y
  X(2,1)*Y X(2,2)*Y X(2,3)*Y ]
```

The matrix representation of the discrete Laplacian operator on a two-dimensional, `n`-by-`n` grid is a `n^2`-by-`n^2` sparse matrix. There are at most five nonzero elements in each row or column. The matrix can be generated as the Kronecker product of one-dimensional difference operators with these statements:

```
I = speye(n,n);
E = sparse(2:n,1:n-1,1,n,n);
D = E+E'-2*I;
A = kron(D,I)+kron(I,D);
```

Plotting this with the `spy` function for `n = 5` yields:



See Also `hankel`, `toeplitz`

Purpose

Last uncaught exception

Syntax

```
errRecord = MException.last
MException.last('reset')
```

Description

`errRecord = MException.last` displays the contents of the `MException` object representing your most recent uncaught error. This is a static method of the `MException` class; it is not a method of an `MException` class object. Use this method from the MATLAB command line only, and not within an M-file.

`MException.last('reset')` sets the `identifier` and `message` properties of the most recent exception to the empty string, the `stack` property to a 0-by-1 structure, and `cause` property to an empty cell array.

`last` is not set in a `try-catch` statement.

Examples

This example displays the last error that was caught during this MATLAB session:

```
A = 25;
A(2)
??? Index exceeds matrix dimensions.

MException.last
ans =

 MException object with properties:

    identifier: 'MATLAB:badsubscript'
       message: 'Index exceeds matrix dimensions.'
         stack: [0x1 struct]
         cause: {}
```

See Also

`try`, `catch`, `error`, `assert`, `MException`, `throw(MException)`, `rethrow(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `getReport(MException)`,

Tiff.lastDirectory

Purpose

Determine if current IFD is last in file

Syntax

```
bool = tiffobj.lastDirectory()
```

Description

`bool = tiffobj.lastDirectory()` returns `true` if the current image file directory (IFD) is the last IFD in the TIFF file; otherwise, `false`.

Examples

Open a `Tiff` object and determine if the current directory is the last directory in the file. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path. If the file contains only one image, the current IFD will be the last:

```
t = Tiff('myfile.tif', 'r');
tf = t.lastDirectory();
```

References

This method corresponds to the `TIFFLastDirectory` 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

-
-

Purpose

Last error message

Note `lasterr` will be removed in a future version. You can obtain information about any error that has been generated by catching an `MException`. See in the Programming Fundamentals documentation.

Syntax

```
msgstr = lasterr
[msgstr, msgid] = lasterr
lasterr('new_msgstr')
lasterr('new_msgstr', 'new_msgid')
[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')
```

Description

`msgstr = lasterr` returns the last error message generated by the MATLAB software.

`[msgstr, msgid] = lasterr` returns the last error in `msgstr` and its message identifier in `msgid`. If the error was not defined with an identifier, `lasterr` returns an empty string for `msgid`. See in the MATLAB Programming Fundamentals documentation for more information on the `msgid` argument and how to use it.

`lasterr('new_msgstr')` sets the last error message to a new string, `new_msgstr`, so that subsequent invocations of `lasterr` return the new error message string. You can also set the last error to an empty string with `lasterr('')`.

`lasterr('new_msgstr', 'new_msgid')` sets the last error message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lasterr` return the new error message and message identifier.

`[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')` returns the last error message and its identifier, also changing these values so that subsequent invocations of `lasterr` return the message and identifier strings specified by `new_msgstr` and `new_msgid` respectively.

Examples

Example 1

Here is a function that examines the `lasterr` string and displays its own message based on the error that last occurred. This example deals with two cases, each of which is an error that can result from a matrix multiply:

```
function matrix_multiply(A, B)
try
   A * B
catch
   errmsg = lasterr;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
   else
      if(strfind(errmsg, 'not defined for variables of class'))
         disp('** Both arguments must be double matrices')
      end
   end
end
```

If you call this function with matrices that are incompatible for matrix multiplication (e.g., the column dimension of `A` is not equal to the row dimension of `B`), MATLAB catches the error and uses `lasterr` to determine its source:

```
A = [1  2  3; 6  7  2; 0  -1  5];
B = [9  5  6; 0  4  9];

matrix_multiply(A, B)
** Wrong dimensions for matrix multiply
```

Example 2

Specify a message identifier and error message string with `error`:

```
error('MyToolbox:angleTooLarge', ...
        'The angle specified must be less than 90 degrees.');
```

In your error handling code, use `lasterr` to determine the message identifier and error message string for the failing operation:

```
[errmsg, msgid] = lasterr
errmsg =
   The angle specified must be less than 90 degrees.
msgid =
   MyToolbox:angleTooLarge
```

See Also

`error`, `lasterror`, `rethrow`, `warning`, `lastwarn`

lasterror

Purpose

Last error message and related information

Note `lasterror` will be removed in a future version. You can obtain information about any error that has been generated by catching an `MException`. See in the Programming Fundamentals documentation.

Syntax

```
s = lasterror
s = lasterror(err)
s = lasterror('reset')
```

Description

`s = lasterror` returns a structure `s` containing information about the most recent error issued by the MATLAB software. The return structure contains the following fields:

Fieldname	Description
`message`	Character array containing the text of the error message.
`identifier`	Character array containing the message identifier of the error message. If the last error issued by MATLAB had no message identifier, then the `identifier` field is an empty character array.
`stack`	Structure providing information on the location of the error. The structure has fields `file`, `name`, and `line`, and is the same as the structure returned by the `dbstack` function. If `lasterror` returns no stack information, `stack` is a 0-by-1 structure having the same three fields.

Note The `lasterror` return structure might contain additional fields in future versions of MATLAB.

The fields of the structure returned in `stack` are

Fieldname	Description
`file`	Name of the file in which the function generating the error appears. This field is the empty string if there is no file.
`name`	Name of the function in which the error occurred. If this is the primary function of the M-file, and the function name differs from the M-file name, `name` is set to the M-file name.
`line`	M-file line number where the error occurred.

See in the MATLAB Programming Fundamentals documentation for more information on the syntax and usage of message identifiers.

`s = lasterror(err)` sets the last error information to the error message and identifier specified in the structure `err`. Subsequent invocations of `lasterror` return this new error information. The optional return structure `s` contains information on the previous error.

`s = lasterror('reset')` sets the last error information to the default state. In this state, the `message` and `identifier` fields of the return structure are empty strings, and the `stack` field is a 0-by-1 structure.

Remarks

The MathWorks is gradually transitioning MATLAB error handling to an object-oriented scheme that is based on the `MException` class. Although support for `lasterror` is expected to continue, using the static `last` method of `MException` is preferable.

Warning

`lasterror` and `MException.last` are not guaranteed to always return identical results. For example, `MException.last` updates its error status only on uncaught errors, where `lasterror` can update its error status on any error, whether it is caught or not.

Examples

Example 1

Save the following MATLAB code in an M-file called `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector elements.
% Nonvector input results in an error.
check_inputs(x)
y = sum(x)/length(x);      % The actual computation

function check_inputs(x)
[m,n] = size(x);
if (~((m == 1) || (n == 1)) || (m == 1 && n == 1))
    error('AVG:NotAVector', 'Input must be a vector.')
end
```

Now run the function. Because this function requires vector input, passing a scalar value to it forces an error. The error occurs in subroutine `check_inputs`:

```
average(200)
??? Error using ==> average>check_inputs
Input must be a vector.

Error in ==> average at 5
check_inputs(x)
```

Get the three fields from `lasterror`:

```
err = lasterror
err =
       message: [1x61 char]
    identifier: 'AVG:NotAVector'
         stack: [2x1 struct]
```

Display the text of the error message:

```
msg = err.message
```

```
msg =
    Error using ==> average>check_inputs
    Input must be a vector.
```

Display the fields containing the `stack` information. `err.stack` is a 2-by-1 structure because it provides information on the failing subroutine `check_inputs` and also the outer, primary function `average`:

```
st1 = err.stack(1,1)
st1 =
    file: 'd:\matlab_test\average.m'
    name: 'check_inputs'
    line: 11

st2 = err.stack(2,1)
st2 =
    file: 'd:\matlab_test\average.m'
    name: 'average'
    line: 5
```

Note As a rule, the name of your primary function should be the same as the name of the M-file containing that function. If these names differ, MATLAB uses the M-file name in the `name` field of the `stack` structure.

Example 2

`lasterror` is often used in conjunction with the `rethrow` function in `try-catch` statements. For example,

```
try
   do_something
catch
   do_cleanup
   rethrow(lasterror)
end
```

See Also `last(MException)`, `MException`, `try`, `catch`, `error`, `assert`, `rethrow`, `lastwarn`, `dbstack`

Purpose

Last warning message

Syntax

```
msgstr = lastwarn
[msgstr, msgid] = lastwarn
lastwarn('new_msgstr')
lastwarn('new_msgstr', 'new_msgid')
[msgstr, msgid] = lastwarn('new_msgstr', 'new_msgid')
```

Description

`msgstr = lastwarn` returns the last warning message generated by the MATLAB software.

`[msgstr, msgid] = lastwarn` returns the last warning in `msgstr` and its message identifier in `msgid`. If the warning was not defined with an identifier, `lastwarn` returns an empty string for `msgid`. See and in the MATLAB Programming Fundamentals documentation for more information on the `msgid` argument and how to use it.

`lastwarn('new_msgstr')` sets the last warning message to a new string, `new_msgstr`, so that subsequent invocations of `lastwarn` return the new warning message string. You can also set the last warning to an empty string with `lastwarn('')`.

`lastwarn('new_msgstr', 'new_msgid')` sets the last warning message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lastwarn` return the new warning message and message identifier.

`[msgstr, msgid] = lastwarn('new_msgstr', 'new_msgid')` returns the last warning message and its identifier, also changing these values so that subsequent invocations of `lastwarn` return the message and identifier strings specified by `new_msgstr` and `new_msgid`, respectively.

Remarks

`lastwarn` does not return warnings that are reported during the parsing of MATLAB commands. (Warning messages that include the failing file name and line number are parse-time warnings.)

Examples

Specify a message identifier and warning message string with `warning`:

```
warning('MATLAB:divideByZero', 'Divide by zero');
```

Use `lastwarn` to determine the message identifier and error message string for the operation:

```
[warnmsg, msgid] = lastwarn
warnmsg =
   Divide by zero
msgid =
   MATLAB:divideByZero
```

See Also

`warning`, `error`, `lasterr`, `lasterror`

Purpose

Least common multiple

Syntax

```
L = lcm(A,B)
```

Description

`L = lcm(A,B)` returns the least common multiple of corresponding elements of arrays `A` and `B`. Inputs `A` and `B` must contain positive integer elements and must be the same size (or either can be scalar).

Examples

```
lcm(8,40)

ans =

    40
lcm(pascal(3),magic(3))

ans =
     8     1     6
     3    10    21
     4     9     6
```

See Also

`gcd`

Purpose

Block LDL' factorization for Hermitian indefinite matrices

Syntax

```
L = ldl(A)
[L,D] = ldl(A)
[L,D,P] = ldl(A)
[L,D,p] = ldl(A,'vector')
[U,D,P] = ldl(A,'upper')
[U,D,p] = ldl(A,'upper','vector')
[L,D,P,S] = ldl(A)
[L,D,P,S] = LDL(A,THRESH)
[U,D,p,S] = LDL(A,THRESH,'upper','vector')
```

Description

`L = ldl(A)` returns only the "psychologically lower triangular matrix" `L` as in the two-output form. The permutation information is lost, as is the block diagonal factor `D`. By default, `ldl` references only the diagonal and lower triangle of `A`, and assumes that the upper triangle is the complex conjugate transpose of the lower triangle. Therefore `[L,D,P] = ldl(TRIL(A))` and `[L,D,P] = ldl(A)`both return the exact same factors. Note, this syntax is not valid for sparse `A`.

`[L,D] = ldl(A)` stores a block diagonal matrix `D` and a "psychologically lower triangular matrix" (i.e a product of unit lower triangular and permutation matrices) in `L` such that `A = L*D*L'`. The block diagonal matrix `D` has 1-by-1 and 2-by-2 blocks on its diagonal. Note, this syntax is not valid for sparse `A`.

`[L,D,P] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, and permutation matrix `P` such that `P'*A*P = L*D*L'`. This is equivalent to `[L,D,P] = ldl(A,'matrix')`.

`[L,D,p] = ldl(A,'vector')` returns the permutation information as a vector, `p`, instead of a matrix. The `p` output is a row vector such that `A(p,p) = L*D*L'`.

`[U,D,P] = ldl(A,'upper')` references only the diagonal and upper triangle of `A` and assumes that the lower triangle is the complex conjugate transpose of the upper triangle. This syntax returns a unit upper triangular matrix `U` such that `P'*A*P = U'*D*U` (assuming that

`A` is Hermitian, and not just upper triangular). Similarly, `[L,D,P] = ldl(A,'lower')` gives the default behavior.

`[U,D,p] = ldl(A,'upper','vector')` returns the permutation information as a vector, `p`, as does `[L,D,p] = ldl(A,'lower','vector')`. `A` must be a full matrix.

`[L,D,P,S] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, permutation matrix `P`, and scaling matrix `S` such that `P'*S*A*S*P = L*D*L'`. This syntax is only available for real sparse matrices, and only the lower triangle of `A` is referenced. `ldl` uses MA57 for sparse real symmetric `A`.

`[L,D,P,S] = LDL(A,THRESH)` uses `THRESH` as the pivot tolerance in MA57. `THRESH` must be a double scalar lying in the interval `[0, 0.5]`. The default value for `THRESH` is `0.01`. Using smaller values of `THRESH` may give faster factorization times and fewer entries, but may also result in a less stable factorization. This syntax is available only for real sparse matrices.

`[U,D,p,S] = LDL(A,THRESH,'upper','vector')` sets the pivot tolerance and returns upper triangular `U` and permutation vector `p` as described above.

Examples

These examples illustrate the use of the various forms of the `ldl` function, including the one-, two-, and three-output form, and the use of the `vector` and `upper` options. The topics covered are:

- “Example 1 — Two-Output Form of ldl” on page 2-2066
- “Example 2 — Three Output Form of ldl” on page 2-2066
- “Example 3 — The Structure of D” on page 2-2067
- “Example 4 — Using the ’vector’ Option” on page 2-2067
- “Example 5 — Using the ’upper’ Option” on page 2-2068
- “Example 6 — linsolve and the Hermitian indefinite solver” on page 2-2068

Before running any of these examples, you will need to generate the following positive definite and indefinite Hermitian matrices:

```
A = full(delsq(numgrid('L', 10)));
B = gallery('uniformdata',10,0);
M = [eye(10) B; B' zeros(10)];
```

The structure of `M` here is very common in optimization and fluid-flow problems, and `M` is in fact indefinite. Note that the positive definite matrix `A` must be full, as `ldl` does not accept sparse arguments.

Example 1 — Two-Output Form of ldl

The two-output form of `ldl` returns `L` and `D` such that `A-(L*D*L')` is small, `L` is "psychologically unit lower triangular" (i.e., a permuted unit lower triangular matrix), and `D` is a block 2-by-2 diagonal. Note also that, because `A` is positive definite, the diagonal of `D` is all positive:

```
[LA,DA] = ldl(A);
fprintf(1, ...
'The factorization error ||A - LA*DA*LA''|| is %g\n', ...
norm(A - LA*DA*LA'));
neginds = find(diag(DA) < 0)
```

Given `a b`, solve `Ax=b` using `LA`, `DA`:

```
bA = sum(A,2);
x = LA'\(DA\(LA\bA));
fprintf(...
'The absolute error norm ||x - ones(size(bA))|| is %g\n', ...
norm(x - ones(size(bA))));
```

Example 2 — Three Output Form of ldl

The three output form returns the permutation matrix as well, so that `L` is in fact unit lower triangular:

```
[Lm, Dm, Pm] = ldl(M);
fprintf(1, ...
'The error norm ||Pm''*M*Pm - Lm*Dm*Lm''|| is %g\n', ...
```

```
norm(Pm'*M*Pm - Lm*Dm*Lm'));
fprintf(1, ...
'The difference between Lm and tril(Lm) is %g\n', ...
norm(Lm - tril(Lm)));
```

Given `b`, solve `Mx=b` using `Lm`, `Dm`, and `Pm`:

```
bM = sum(M,2);
x = Pm*(Lm'\(Dm\(Lm\(Pm'*bM))));
fprintf(...
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

Example 3 — The Structure of D

`D` is a block diagonal matrix with 1-by-1 blocks and 2-by-2 blocks. That makes it a special case of a tridiagonal matrix. When the input matrix is positive definite, `D` is almost always diagonal (depending on how definite the matrix is). When the matrix is indefinite however, `D` may be diagonal or it may express the block structure. For example, with `A` as above, `DA` is diagonal. But if you shift `A` just a bit, you end up with an indefinite matrix, and then you can compute a `D` that has the block structure.

```
figure; spy(DA); title('Structure of D from ldl(A)');
[Las, Das] = ldl(A - 4*eye(size(A)));
figure; spy(Das);
title('Structure of D from ldl(A - 4*eye(size(A)))');
```

Example 4 — Using the 'vector' Option

Like the `lu` function, `ldl` accepts an argument that determines whether the function returns a permutation vector or permutation matrix. `ldl` returns the latter by default. When you select `'vector'`, the function executes faster and uses less memory. For this reason, specifying the `'vector'` option is recommended. Another thing to note is that indexing is typically faster than multiplying for this kind of operation:

```
[Lm, Dm, pm] = ldl(M, 'vector');
```

```
fprintf(1, 'The error norm ||M(pm,pm) - Lm*Dm*Lm''|| is %g\n', ...
  norm(M(pm,pm) - Lm*Dm*Lm'));

% Solve a system with this kind of factorization.
clear x;
x(pm,:) = Lm'\(Dm\(Lm\(bM(pm,:))));
fprintf('The absolute error norm ||x - ones(size(b))|| is %g\n', ...
  norm(x - ones(size(bM))));
```

Example 5 — Using the 'upper' Option

Like the `chol` function, `ldl` accepts an argument that determines which triangle of the input matrix is referenced, and also whether `ldl` returns a lower (`L`) or upper (`L'`) triangular factor. For dense matrices, there are no real savings with using the upper triangular version instead of the lower triangular version:

```
Ml = tril(M);
[Lml, Dml, Pml] = ldl(Ml, 'lower'); % 'lower' is default behavior.
fprintf(1, ...
'The difference between Lml and Lm is %g\n', norm(Lml - Lm));
[Umu, Dmu, pmu] = ldl(triu(M), 'upper', 'vector');
fprintf(1, ...
'The difference between Umu and Lm'' is %g\n', norm(Umu - Lm'));

% Solve a system using this factorization.
clear x;
x(pm,:) = Umu\(Dmu\(Umu'\(bM(pmu,:))));
fprintf(...
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

When specifying both the `'upper'` and `'vector'` options, `'upper'` must precede `'vector'` in the argument list.

Example 6 — linsolve and the Hermitian indefinite solver

When using the `linsolve` function, you may experience better performance by exploiting the knowledge that a system has a symmetric matrix. The matrices used in the examples above are a bit small to see this so, for this example, generate a larger matrix. The matrix here is symmetric positive definite, and below we will see that with each bit of knowledge about the matrix, there is a corresponding speedup. That is, the symmetric solver is faster than the general solver while the symmetric positive definite solver is faster than the symmetric solver:

```
Abig = full(delsq(numgrid('L', 30)));
bbig = sum(Abig, 2);
LSopts.POSDEF = false;
LSopts.SYM = false;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.SYM = true;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.POSDEF = true;
tic; linsolve(Abig, bbig, LSopts); toc;
```

Algorithm

`ldl` uses the MA57 routines in the Harwell Subroutine Library (HSL) for real sparse matrices.

References

[1] Ashcraft, C., R.G. Grimes, and J.G. Lewis. “Accurate Symmetric Indefinite Linear Equations Solvers.” *SIAM J. Matrix Anal. Appl.* Vol. 20. Number 2, 1998, pp. 513–561.

[2] Duff, I. S. "MA57 — A new code for the solution of sparse symmetric definite and indefinite systems." Technical Report RAL-TR-2002-024, Rutherford Appleton Laboratory, 2002.

See Also

`chol`, `lu`, `qr`

ldivide, rdivide

Purpose

Left or right array division

Syntax

```
ldivide(A,B)
A.\B
rdivide(A,B)
A./B
```

Description

`ldivide(A,B)` and the equivalent `A.\B` divides each entry of `B` by the corresponding entry of `A`. `A` and `B` must be arrays of the same size. A scalar value for either `A` or `B` is expanded to an array of the same size as the other.

`rdivide(A,B)` and the equivalent `A./B` divides each entry of `A` by the corresponding entry of `B`. `A` and `B` must be arrays of the same size. A scalar value for either `A` or `B` is expanded to an array of the same size as the other.

Example

```
A = [1 2 3;4 5 6];
B = ones(2, 3);
A.\B

ans =

    1.0000    0.5000    0.3333
    0.2500    0.2000    0.1667
```

See Also

Arithmetic Operators, `mldivide`, `mrdivide`

Purpose

Test for less than or equal to

Syntax

```
A <= B
le(A, B)
```

Description

`A <= B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is less than or equal to `B`, or set to logical 0 (`false`) where `A` is greater than `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLAB software returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`le(A, B)` is called for the syntax `A < =B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are less than or equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A <= B
ans =
     0     1     1     0     0     0
     1     0     1     0     0     0
     0     1     1     0     1     0
     1     0     0     1     0     1
```

```
     0     1     0     0     1     1
     1     0     0     0     1     0
```

See Also `lt`, `eq`, `ge`, `gt`, `ne`, Relational Operators

Purpose

Graph legend for lines and patches

Syntax

```
legend
legend('string1','string2',...)
legend(h,'string1','string2',...)
legend(M)
legend(h,M)
legend(M,'parameter_name','parameter_value',...)
legend(h,M,'parameter_name','parameter_value',...)
legend(axes_handle,...)
legend('off'), legend(axes_handle,'off')
legend('toggle'), legend(axes_handle,'toggle')
legend('hide'), legend(axes_handle,'hide')
legend('show'), legend(axes_handle,'show')
legend('boxoff'), legend(axes_handle,'boxoff')
legend('boxon'), legend(axes_handle,'boxon')
legend_handle = legend(...)
legend(...,'Location',location)
legend(...,'Orientation','orientation')
[legend_h,object_h,plot_h,text_strings] = legend(...)
legend(li_object,string1,string2,string3)
legend(li_objects,M)
legend('v6',M,...)
legend('v6',AX)
```

Description

`legend` places a legend on various types of graphs (line plots, bar graphs, pie charts, etc.). For each line plotted, the legend shows a sample of the line type, marker symbol, and color beside the text label you specify. When plotting filled areas (patch or surface objects), the legend contains a sample of the face color next to the text label.

The font size and font name for the legend strings match the axes `FontSize` and `FontName` properties.

`legend('string1','string2',...)` displays a legend in the current axes using the specified strings to label each set of data.

`legend(h,'string1','string2',...)` displays a legend on the plot containing the objects identified by the handles in the vector `h` and uses the specified strings to label the corresponding graphics object (line, barseries, etc.).

`legend(M)` adds a legend containing the rows of the matrix or cell array of strings `M` as labels. For matrices, this is the same as `legend(M(1,:),M(2,:),...)`.

`legend(h,M)` associates each row of the matrix or cell array of strings `M` with the corresponding graphics object (patch or line) in the vector of handles `h`.

`legend(M,'parameter_name','parameter_value',...)` and `legend(h,M,'parameter_name','parameter_value',...)` allow parameter/value pairs to be set when creating a legend (you can also assign them with `set` or with the Property Editor or Property Inspector). `M` must be a cell array of names. Legends inherit the properties of axes, although not all of them are relevant to legend objects.

`legend(axes_handle,...)` displays the legend for the axes specified by `axes_handle`.

`legend('off')`, `legend(axes_handle,'off')` removes the legend in the current axes or the axes specified by `axes_handle`.

`legend('toggle')`, `legend(axes_handle,'toggle')` toggles the legend on or off. If no legend exists for the current axes, one is created using default strings.

The *default string* for an object is the value of the object's `DisplayName` property, if you have defined a value for `DisplayName` (which you can do using the Property Editor or calling `set`). Otherwise, `legend` constructs a string of the form `data1`, `data2`, etc. Setting display names is useful when you are experimenting with legends and might forget how objects in a lineseries, for example, are ordered.

When you specify legend strings in a `legend` command, their respective `DisplayName`s are set to these strings. If you delete a legend and then create a new legend without specifying labels for it, the values of `DisplayName` are (re)used as label names. Naturally, the associated plot objects must have a `DisplayName` property for this to happen: all *`_series`* and *`_group`* plot objects have a `DisplayName` property; Handle Graphics primitives, such as `line` and `patch`, do not.

Legends for graphs that contain groups of objects such as lineseries, barseries, contourgroups, etc. created by high-level plotting commands such as `plot`, `bar`, `contour`, etc., by default display a single legend entry for the entire group, regardless of how many member objects it contains. However, you can customize such legends to show individual entries for all or selected member objects and assign a unique `DisplayName` to any of them. You control how groups appear in the legend by setting values for their `Annotation` and `DisplayName` properties with M-code. For information and examples about customizing legends in this manner, see in the MATLAB Graphics documentation.

You can specify `EdgeColor` and `TextColor` as RGB triplets or as `ColorSpecs`. You cannot set these colors to `'none'`. To hide the box surrounding a legend, set the `Box` property to `'off'`. To allow the background to show through the legend box, set the legend's `Color` property to `'none'`, for example,

```
set(legend_handle, 'Box', 'off')
set(legend_handle, 'Color', 'none')
```

This is similar to the effect of the command `legend boxoff`, except that `boxoff` also hides the legend's border.

You can use a legend's handle to set text properties for all the strings in a legend at once with a cell array of strings, rather than looping through each of them. See the last line of the example below, which demonstrates setting a legend's `Interpreter` property. In that example, you could reset the `String` property of the legend as follows:

```
set(h,'String',{'cos(x)','sin(x)'})
```

See the documentation for Text Properties for additional details.

`legend('hide'), legend(axes_handle,'hide')` makes the legend in the current axes or the axes specified by `axes_handle` invisible.

`legend('show')`, `legend(axes_handle,'show')` makes the legend in the current axes or the axes specified by `axes_handle` visible. A legend is created if one did not exist previously. Legends created automatically are limited to depict only the first 20 lines in the plot; if you need more legend entries, you can manually create a legend for them all with `legend('string1','string2',...)` syntax.

`legend('boxoff')`, `legend(axes_handle,'boxoff')` removes the box from the legend in the current axes or the axes specified by `axes_handle`, and makes its background transparent.

`legend('boxon')`, `legend(axes_handle,'boxon')` adds a box with an opaque background to the legend in the current axes or the axes specified by `axes_handle`.

You can also type the above six commands using the syntax

```
legend keyword
```

If the keyword is not recognized, it is used as legend text, creating a legend or replacing the current legend.

`legend_handle = legend(...)` returns the handle to the legend on the current axes, or `[]` if no legend exists.

`legend(...,'Location',`*`location`*`)` uses *`location`* to determine where to place the legend. *`location`* can be either a 1-by-4 position vector ([left bottom width height]) or one of the following strings.

Specifier	Location in Axes
`North`	Inside plot box near top
`South`	Inside bottom
`East`	Inside right
`West`	Inside left
`NorthEast`	Inside top right (default for 2-D plots)
`NorthWest`	Inside top left

Specifier	Location in Axes
`SouthEast`	Inside bottom right
`SouthWest`	Inside bottom left
`NorthOutside`	Outside plot box near top
`SouthOutside`	Outside bottom
`EastOutside`	Outside right
`WestOutside`	Outside left
`NorthEastOutside`	Outside top right (default for 3-D plots)
`NorthWestOutside`	Outside top left
`SouthEastOutside`	Outside bottom right
`SouthWestOutside`	Outside bottom left
`Best`	Least conflict with data in plot
`BestOutside`	Least unused space outside plot

If the legend text does not fit in the 1-by-4 position vector, the position vector is resized around the midpoint to fit the legend text given its font and size, making the legend taller or wider. The *`location`* string can be all lowercase and can be abbreviated by sentinel letter (e.g., N, NE, NEO, etc.). Using one of the `...Outside` values for *`location`* ensures that the legend does not overlap the plot, whereas overlaps can occur when you specify any of the other cardinal values. The *`location`* property applies to colorbars and legends, but not to axes.

Obsolete Location Values

The first column of the following table shows the now-obsolete specifiers for legend locations that were in use prior to Version 7, along with a description of the locations and their current equivalent syntaxes:

Obsolete Specifier	Location in Axes	Current Specifier
`-1`	Outside axes on right side	`NorthEastOutside`

Obsolete Specifier	Location in Axes	Current Specifier
0	Inside axes	`Best`
1	Upper right corner of axes	`NorthEast`
2	Upper left corner of axes	`NorthWest`
3	Lower left corner of axes	`SouthWest`
4	Lower right corner of axes	`SouthEast`

`legend(...,'Orientation','`*`orientation`*`')` creates a legend with the legend items arranged in the specified orientation. *`orientation`* can be `vertical` (the default) or `horizontal`.

`[legend_h,object_h,plot_h,text_strings] = legend(...)` returns

- `legend_h` — Handle of the legend axes
- `object_h` — Handles of the line, patch, and text graphics objects used in the legend
- `plot_h` — Handles of the lines and other objects used in the plot
- `text_strings` — Cell array of the text strings used in the legend

These handles enable you to modify the properties of the respective objects.

`legend(li_object,string1,string2,string3)` creates a legend for legendinfo objects `li_objects` with strings `string1`, etc.

`legend(li_objects,M)` creates a legend of legendinfo objects `li_objects`, where `M` is a string matrix or cell array of strings corresponding to the legendinfo objects.

Backward Compatibility

`legend('v6',M,...)`, for a cell array of strings `M`, creates a legend compatible with MATLAB 6.5 from the strings in `M` and any additional inputs.

`legend('v6',AX)`, for an axes handle `AX`, updates any Version 6 legends and returns the legend handle.

The following calls to `legend` are passed to the Version 6 legend mechanism to maintain backward compatibility:

```
legend('DeleteLegend')
legend('EditLegend',h)
legend('ShowLegendPlot',h)
legend('ResizeLegend')
legend('RestoreSize',hLegend)
legend('RecordSize',hPlot)
```

Note The `v6` option lets users of MATLAB Version 7.x create FIG-files that previous versions can open. It is obsolete and will be removed in a future MATLAB version.

Relationship to Axes

`legend` associates strings with the objects in the axes in the same order that they are listed in the axes `Children` property. By default, the legend annotates the current axes.

You can only display one legend per axes. `legend` positions the legend based on a variety of factors, such as what objects the legend obscures.

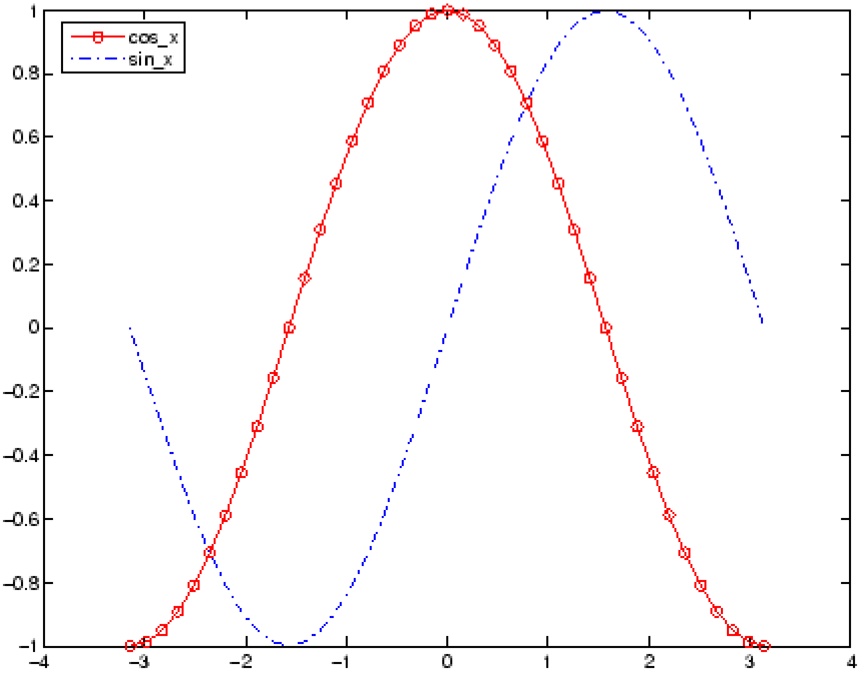
The properties that legends do not share with axes are

- `Location`
- `Orientation`
- `EdgeColor`
- `TextColor`
- `Interpreter`
- `String`

Examples

Add a legend to a graph showing a sine and cosine function:

```
x = -pi:pi/20:pi;
plot(x,cos(x),'-ro',x,sin(x),'-.b')
h = legend('cos_x','sin_x',2);
set(h,'Interpreter','none')
```



In this example, the `plot` command specifies a solid, red line (`'-r'`) for the cosine function and a dash-dot, blue line (`'-.b'`) for the sine function.

Alternatives

Add a legend to a selected axes on a graph with the **Insert Legend** tool on the figure toolbar, or use **Insert —> Legend** from the figure menu. Use the Property Editor to modify the position, font, and other properties of a legend. For details, see Using Plot Edit Mode in the MATLAB Graphics documentation.

Moving the Legend

Move the legend by pressing the left mouse button while the cursor is over the legend and dragging the legend to a new location. Double-clicking a label allows you to edit the label.

See Also

`LineSpec` | `plot`

How To

-

legendre

Purpose Associated Legendre functions

Syntax

```
P = legendre(n,X)
S = legendre(n,X,'sch')
N = legendre(n,X,'norm')
```

Definitions

Associated Legendre Functions

The Legendre functions are defined by

$$P_n^m(x) = (-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}P_n(x)$$

where

$$P_n(x)$$

is the Legendre polynomial of degree n.

$$P_n(x) = \frac{1}{2^n n!}\left[\frac{d^n}{dx^n}(x^2-1)^n\right]$$

Schmidt Seminormalized Associated Legendre Functions

The Schmidt seminormalized associated Legendre functions are related to the nonnormalized associated Legendre functions $P_n^m(x)$ by

$P_n(x)$ for $m = 0$

$S_n^m(x) = (-1)^m\sqrt{\frac{2(n-m)!}{(n+m)!}}P_n^m(x)$ for $m > 0$.

Fully Normalized Associated Legendre Functions

The fully normalized associated Legendre functions are normalized such that

$$\int_{-1}^{1} (N_n^m(x))^2 dx = 1$$

and are related to the unnormalized associated Legendre functions $P_n^m(x)$ by

$$N_n^m(x) = (-1)^m \sqrt{\frac{\left(n+\frac{1}{2}\right)(n-m)!}{(n+m)!}} P_n^m(x)$$

Description

`P = legendre(n,X)` computes the associated Legendre functions $P_n^m(x)$ of degree `n` and order `m = 0,1,...,n`, evaluated for each element of `X`. Argument `n` must be a scalar integer, and `X` must contain real values in the domain $-1 \le x \le 1$.

If `X` is a vector, then `P` is an `(n+1)`-by-`q` matrix, where `q = length(X)`. Each element `P(m+1,i)` corresponds to the associated Legendre function of degree `n` and order `m` evaluated at `X(i)`.

In general, the returned array `P` has one more dimension than `X`, and each element `P(m+1,i,j,k,...)` contains the associated Legendre function of degree `n` and order `m` evaluated at `X(i,j,k,...)`. Note that the first row of `P` is the Legendre polynomial evaluated at `X`, i.e., the case where `m` = 0.

`S = legendre(n,X,'sch')` computes the Schmidt seminormalized associated Legendre functions $S_n^m(x)$.

`N = legendre(n,X,'norm')` computes the fully normalized associated Legendre functions $N_n^m(x)$.

Examples

Example 1

The statement `legendre(2,0:0.1:0.2)` returns the matrix

	x = 0	**x = 0.1**	**x = 0.2**
`m = 0`	`-0.5000`	`-0.4850`	`-0.4400`
`m = 1`	`0`	`-0.2985`	`-0.5879`
`m = 2`	`3.0000`	`2.9700`	`2.8800`

Example 2

Given,

```
X = rand(2,4,5);
n = 2;
P = legendre(n,X)
```

then

```
size(P)
ans =
     3     2     4     5
```

and

```
P(:,1,2,3)
ans =
   -0.2475
   -1.1225
    2.4950
```

is the same as

```
legendre(n,X(1,2,3))
ans =
   -0.2475
   -1.1225
    2.4950
```

Algorithm

`legendre` uses a three-term backward recursion relationship in `m`. This recursion is on a version of the Schmidt seminormalized associated

Legendre functions $Q_n^m(x)$, which are complex spherical harmonics. These functions are related to the standard Abramowitz and Stegun [1] functions $P_n^m(x)$ by

$$P_n^m(x) = \sqrt{\frac{(n+m)!}{(n-m)!}}\,Q_n^m(x)$$

They are related to the Schmidt form given previously by

$$S_n^m(x) = Q_n^0(x) \text{ for } m = 0$$

$$S_n^m(x) = (-1)^m\sqrt{2}\,Q_n^m(x) \text{ for } m > 0$$

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Ch.8.

[2] Jacobs, J. A., *Geomagnetism*, Academic Press, 1987, Ch.4.

length

Purpose

Length of vector or largest array dimension

Syntax

`numberOfElements = length(array)`

Description

`numberOfElements = length(array)` finds the number of elements along the largest dimension of an array. `array` is an array of any MATLAB data type and any valid dimensions. `numberOfElements` is a whole number of the MATLAB `double` class.

For nonempty arrays, `numberOfElements` is equivalent to `max(size(array))`. For empty arrays, `numberOfElements` is zero.

Examples

Create a 1-by-8 array `X` and use `length` to find the number of elements in the second (largest) dimension:

```
X = [5, 3.4, 72, 28/4, 3.61, 17 94 89];

length(X)
ans =
     8
```

Create a 4-dimensional array `Y` in which the third dimension is the largest. Use `length` to find the number of elements in that dimension:

```
Y = rand(2, 5, 17, 13);

length(Y)
ans =
    10
```

Create a struct array `S` with character and numeric fields of different lengths. Use the `structfun` function to apply `length` to each field of `S`:

```
S = struct('f1', 'Name:', 'f2', 'Charlie', ...
           'f3', 'DOB:', 'f4', 1917)
```

```
S =
    f1: 'Name:'
    f2: 'Charlie'
    f3: 'DOB:'
    f4: 1917

structfun(@(field)length(field), S)
ans =
     5
     7
     4
     1
```

See Also

`numel` | `size` | `ndims`

length (Map)

Purpose

Length of `containers.Map` object

Syntax

```
L = length(M)
```

Description

`L = length(M)` returns the number of pairs in the map `M`. The number returned by this method is equivalent to `size(M,1)`.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

Examples

Create a `Map` object containing the names of several US states and the capital city of each:

```
US_Capitals = containers.Map( ...
{'Arizona', 'Nebraska', 'Nevada', 'New York', ...
 'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...
{'Phoenix', 'Lincoln', 'Carson City', 'Albany', ...
 'Atlanta', 'Juneau', 'Montpelier', 'Salem'});
```

Find out how many keys are in the map:

```
length(US_Capitals)
ans =
     8
```

This should be equal to the `Count` property for the object:

```
length(US_Capitals) == US_Capitals.Count
ans =
     1
```

See Also

`containers.Map`, `keys(Map)`, `values(Map)`, `size(Map)`, `isKey(Map)`, `remove(Map)`, `handle`

length (serial)

Purpose Length of serial port object array

Syntax `length(obj)`

Description `length(obj)` returns the length of the serial port object, `obj`. It is equivalent to the command `max(size(obj))`.

See Also

Functions

`size`

length (timeseries)

Purpose Length of time vector

Syntax `length(ts)`

Description `length(ts)` returns an integer that represents the length of the time vector for the `timeseries` object `ts`. It returns `0` if `ts` is empty.

See Also `isempty (timeseries)`, `size (timeseries)`

Purpose Length of time vector

Syntax `length(tsc)`

Description `length(tsc)` returns an integer that represents the length of the time vector for the `tscollection` object `tsc`.

See Also `isempty (tscollection)`, `size (tscollection)`, `tscollection`

libfunctions

Purpose

Return information on functions in shared library

Syntax

```
m = libfunctions('libname')
m = libfunctions('libname', '-full')
libfunctions libname -full
```

Description

`m = libfunctions('libname')` returns the names of all functions defined in the external shared library, `libname`, that has been loaded into the MATLAB software with the `loadlibrary` function. The return value, `m`, is a cell array of strings.

`m = libfunctions('libname', '-full')` returns a full description of the functions in the library, including function signatures. This includes duplicate function names with different signatures. The return value, `m`, is a cell array of strings.

`libfunctions libname -full` is the command format for this function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

Examples

To list the functions in the MATLAB `libmx` library, see .

See Also

`loadlibrary`, `libfunctionsview`, `calllib`, `unloadlibrary`

Purpose View functions in shared library

Syntax

```
libfunctionsview('libname')
libfunctionsview libname
```

Description `libfunctionsview('libname')` displays the names of the functions in the external shared library, `libname`, that has been loaded into the MATLAB software with the `loadlibrary` function.

`libfunctionsview libname` is the command format for this function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

MATLAB creates a new window in response to the `libfunctionsview` command. This window displays all of the functions defined in the specified library. For each of these functions, the following information is supplied:

- Type returned by the function
- Name of the function
- Arguments passed to the function

An additional column entitled "Inherited From" is displayed at the far right of the window. The information in this column is not useful for external libraries.

Examples To open a window showing functions in the `libmx` library, see .

See Also `loadlibrary`, `libfunctions`, `calllib`, `unloadlibrary`

libisloaded

Purpose

Determine if shared library is loaded

Syntax

```
libisloaded('libname')
libisloaded libname
```

Description

`libisloaded('libname')` returns logical `1` (`true`) if the shared library `libname` is loaded and logical `0` (`false`) otherwise.

`libisloaded libname` is the command format for this function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

Examples

Example 1

Load the `shrlibsample` library and check to see if the load was successful before calling one of its functions:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h

if libisloaded('shrlibsample')
   x = calllib('shrlibsample', 'addDoubleRef', 1.78, 5.42, 13.3)
end
```

Since the library is successfully loaded, the call to `addDoubleRef` works as expected and returns

```
x =
   20.5000

unloadlibrary shrlibsample
```

Example 2

Load the same library, this time giving it an alias. If you use `libisloaded` with the library name, `shrlibsample`, it now returns `false`. Since you loaded the library using an alias, all further references to the library must also use that alias:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h alias lib

libisloaded shrlibsample
ans =
     0

libisloaded lib
ans =
     1

unloadlibrary lib
```

See Also

`loadlibrary`, `unloadlibrary`

libpointer

Purpose

Create pointer object for use with shared libraries

Syntax

```
p = libpointer
p = libpointer('type')
p = libpointer('type',value)
```

Description

`p = libpointer` returns an empty (void) pointer.

`p = libpointer('type')` returns an empty pointer that contains a reference to the specified `type`. This `type` can be any MATLAB numeric type, or a structure or enumerated type defined in an external library that has been loaded into MATLAB with the `loadlibrary` function. For valid `types`, see the table under in the MATLAB External Interfaces documentation.

Note Using this syntax, `p` is a `NULL` pointer. You, therefore, must ensure that any library function to which you pass `p` must be able to accept a `NULL` pointer as an argument.

`p = libpointer('type',value)` returns a pointer to the specified data `type` and initialized to the `value` supplied.

Remarks

MATLAB automatically converts data passed to and from external library functions to the data type expected by the external function. The `libpointer` function enables you to convert your argument data manually. This is an advanced feature available to experienced C programmers. For more information about using pointer objects, see in the MATLAB External Interfaces documentation. Additional examples for using `libpointer` can be found in in the same documentation.

Examples

This example passes an `int16` pointer to a function that multiplies each value in a matrix by its index. The function `multiplyShort` is defined in the MATLAB sample shared library, `shrlibsample`.

Here is the C function:

```
void multiplyShort(short *x, int size)
{
    int i;
    for (i = 0; i < size; i++)
    *x++ *= i;
}
```

Load the `shrlibsample` library. Create the matrix, `v`, and also a pointer to it, `pv`:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h

v = [4 6 8; 7 5 3];

pv = libpointer('int16Ptr', v);
get(pv, 'Value')
ans =
     4     6     8
     7     5     3
```

Now call the C function in the library, passing the pointer to `v`. If you were to pass a *copy* of `v`, the results would be lost once the function terminates. Passing a pointer to `v` enables you to get back the results:

```
calllib('shrlibsample', 'multiplyShort', pv, 6);
get(pv, 'Value')
ans =
     0    12    32
     7    15    15

unloadlibrary shrlibsample
```

See Also `loadlibrary`, `libstruct`

libstruct

Purpose Create structure pointer for use with shared libraries

Syntax

```
s = libstruct('structtype')
s = libstruct('structtype',mlstruct)
```

Description `s = libstruct('structtype')` returns a `libstruct` object `s` that is a MATLAB object designed to resemble a C structure of type specified by `structtype`. The structure type, `structtype`, is defined in an external library that must be loaded into MATLAB using the `loadlibrary` function.

Note Using this syntax, `s` is a `NULL` pointer. You, therefore, must ensure that any library function to which you pass `s` must be able to accept a `NULL` pointer as an argument.

`s = libstruct('structtype',mlstruct)` returns a `libstruct` object `s` with its fields initialized from MATLAB structure, `mlstruct`.

The `libstruct` function creates a C-style structure that you can pass to functions in an external library. You handle this structure in MATLAB as you would a true MATLAB structure.

What Types Are Available

To determine which MATLAB types to use when passing arguments to library functions, see the output of `libfunctionsview` or `libfunctions -full`. These functions list all of the functions found in a particular library along with a specification of the types required for each argument.

Examples This example performs a simple addition of the fields of a structure. The function `addStructFields` is defined in the MATLAB sample shared library, `shrlibsample`.

Here is the C function:

```
double addStructFields(struct c_struct st)
```

```
{
    double t = st.p1 + st.p2 + st.p3;
    return t;
}
```

Start by loading the `shrlibsample` library and creating the structure, `sm`:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h

sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
```

Construct a `libstruct` object `sc` that uses the `c_struct` template:

```
sc = libstruct('c_struct', sm);

get(sc)
    p1: 476
    p2: -299
    p3: 1000
```

Now call the function, passing the `libstruct` object, `sc`:

```
calllib('shrlibsample', 'addStructFields', sc)
ans =
    1177
```

You must clear the `libstruct` object before unloading the library:

```
clear sc
unloadlibrary shrlibsample
```

Note In most cases, you can pass a MATLAB structure and MATLAB automatically converts the argument to a C structure. See in the MATLAB External Interfaces documentation for more information.

See Also `loadlibrary`, `libpointer`

Purpose

Return license number or perform licensing task

Syntax

```
license
license('inuse')
S = license('inuse')
S = license('inuse', feature)
license('test',feature)
license('test',feature,toggle)
result = license('checkout',feature)
```

Description

`license` returns the license number for this MATLAB product. The return value is always a string but is not guaranteed to be a number. The following table lists text strings that `license` can return.

String	Description
`'demo'`	MATLAB is a demonstration version
`'student'`	MATLAB is the student version
`'unknown'`	License number cannot be determined

`license('inuse')` returns a list of licenses checked out in the current MATLAB session. In the list, products are listed alphabetically by their license feature names, i.e., the text string used to identify products in the `INCREMENT` lines in a License File (`license.dat`). Note that the feature names returned in the list contain only lower-case characters.

`S = license('inuse')` returns an array of structures, where each structure represents a checked-out license. The structures contains two fields: feature and user. The feature field contains the license feature name. The user field contains the username of the person who has the license checked out.

`S = license('inuse', feature)` checks if the product specified by the text string `feature` is checked out in the current MATLAB session. If the product is checked out, the `license` function returns the product name and the username of the person who has it checked out in the structure `S`. If the product is not currently checked out, the fields in the structure are empty.

The `feature` string must be a license feature name, spelled exactly as it appears in the `INCREMENT` lines in a License File. For example, the string `'Identification_Toolbox'` is the feature name for the System Identification Toolbox™. The `feature` string is not case-sensitive and must not exceed 27 characters.

`license('test',feature)` tests if a license exists for the product specified by the text string `feature`. The `license` command returns `1` if the license exists and `0` if the license does not exist. The `feature` string identifies a product, as described in the previous syntax.

Note Testing for a license only confirms that the license exists. It does not confirm that the license can be checked out. For example, license will return 1 if a license exists, even if the license has expired or if a system administrator has excluded you from using the product in an options file. The existence of a license does not indicate that the product is installed.

`license('test',feature,toggle)` enables or disables testing of the product specified by the text string `feature`, depending on the value of `toggle`. The parameter `toggle` can have either of two values:

`'enable'`	The syntax `license('test',feature)` returns `1` if the product license exists and 0 if the product license does not exist.
`'disable'`	The syntax `license('test',feature)` always returns `0` (product license does not exist) for the specified product.

Note Disabling a test for a particular product can impact other tests for the existence of the license, not just tests performed using the `license` command.

`result = license('checkout',feature)` checks out a license for the product identified by the text string `feature`. The `license` command returns `1` if it could check out a license for the product and `0` if it could not check out a license for the product.

Examples

Get the license number for this MATLAB.

```
license
```

Get a list of licenses currently being used. Note that the products appear in alphabetical order by their license feature name in the list returned.

```
license('inuse')

image_toolbox
map_toolbox
matlab
```

Get a list of licenses in use with information about who is using the license.

```
S = license('inuse');

S(1)

ans =

    feature: 'image_toolbox'
       user: 'juser'
```

Determine if the license for MATLAB is currently in use.

```
S = license('inuse','MATLAB')

S =

    feature: 'matlab'
       user: 'jsmith'
```

Determine if a license exists for the Mapping Toolbox™.

```
license('test','map_toolbox')

ans =

    1
```

Check out a license for the Control System Toolbox.

```
license('checkout','control_toolbox')

ans =

    1
```

Determine if the license for the Control System Toolbox is checked out.

```
license('inuse')

control_toolbox
image_toolbox
map_toolbox
matlab
```

See Also

`isstudent`

Purpose

Create light object

Syntax

```
light('PropertyName',propertyvalue,...)
handle = light(...)
```

Description

`light` creates a light object in the current axes. Lights affect only patch and surface objects.

`light('PropertyName',propertyvalue,...)` creates a light object using the specified values for the named properties. The MATLAB software parents the light to the current axes unless you specify another axes with the `Parent` property.

`handle = light(...)` returns the handle of the light object created.

Remarks

You cannot see a light object *per se*, but you can see the effects of the light source on patch and surface objects. You can also specify an axes-wide ambient light color that illuminates these objects. However, ambient light is visible only when at least one light object is present and visible in the axes.

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

See also the patch and surface `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, `SpecularColorReflectance`, and `VertexNormals` properties. Also see the `lighting` and `material` commands.

Examples

Light the `peaks` surface plot with a light source located at infinity and oriented along the direction defined by the vector `[1 0 0]`, that is, along the *x*-axis.

```
h = surf(peaks);
set(h,'FaceLighting','phong','FaceColor','interp',...
      'AmbientStrength',0.5)
light('Position',[1 0 0],'Style','infinite');
```

Setting Default Properties

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

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

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

See Also

`lighting`, `material`, `patch`, `surface`

for more information about lighting

"Lighting" on page 1-106 for related functions

Light Properties for property descriptions

Purpose

Light 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 .

See for general information about this type of object.

Light Property Descriptions

This section lists property names along with the type of values each accepts.

`BeingDeleted`

`on | {off}` Read Only

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

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

`BusyAction`

`cancel | {queue}`

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

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

`ButtonDownFcn`
function handle

This property is not used on lights.

`Children`
handles

The empty matrix; light objects have no children.

`Clipping`
`on | off`

`Clipping` has no effect on light objects.

`Color`
`ColorSpec`

Color of light. This property defines the color of the light emanating from the light object. Define it as a three-element RGB vector or one of the MATLAB predefined names. See the `ColorSpec` reference page for more information.

`CreateFcn`

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

Callback function executed during object creation. A callback function that executes when MATLAB creates a light object. You must define this property as a default value for lights or in a call to the `light` function to create a new light object. For example, the following statement:

```
set(0,'DefaultLightCreateFcn',@light_create)
```

defines a default value for the line `CreateFcn` property on the root level that sets the current figure colormap to `gray` and uses a reddish light color whenever you create a light object.

```
function light_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
 set(src,'Color',[.9 .2 .2])
 set(gcbf,'Colormap',gray)
end
```

MATLAB executes this function after setting all light properties. Setting this property on an existing light object has no effect. The function must define at least two input arguments (handle of light object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

`DeleteFcn`

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

Delete light callback function. A callback function that executes when you delete the light object (e.g., when you issue a `delete` command or clear the axes `cla` or figure `clf`). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

`HandleVisibility`

`{on} | callback | off`

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

Handles are always visible when `HandleVisibility` is `on`.

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

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

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

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

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

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

`HitTest`

`{on} | off`

This property is not used by light objects.

`Interruptible`

`{on} | off`

Callback routine interruption mode. Light object callback routines defined for the `DeleteFcn` property are not affected by the `Interruptible` property.

`Parent`

handle of parent axes

Parent of light object. This property contains the handle of the light object's parent. The parent of a light object is the `axes` object that contains it.

Note that light objects cannot be parented to hggroup or hgtransform objects.

See for more information on parenting graphics objects.

`Position`

`[x,y,z]` in axes data units

Location of light object. This property specifies a vector defining the location of the light object. The vector is defined from the origin to the specified x-, y-, and z-coordinates. The placement of the light depends on the setting of the `Style` property:

- If the `Style` property is set to `local`, `Position` specifies the actual location of the light (which is then a point source that radiates from the location in all directions).

- If the `Style` property is set to `infinite`, `Position` specifies the direction from which the light shines in parallel rays.

`Selected`

`on | off`

This property is not used by light objects.

`SelectionHighlight`

`{on} | off`

This property is not used by light objects.

`Style`

`{infinite} | local`

Parallel or divergent light source. This property determines whether MATLAB places the light object at infinity, in which case the light rays are parallel, or at the location specified by the `Position` property, in which case the light rays diverge in all directions. See the `Position` property.

`Tag`

string

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

`Type`

string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For light objects, `Type` is always `'light'`.

`UIContextMenu`

handle of a uicontextmenu object

This property is not used by light objects.

`UserData`

matrix

User-specified data. This property can be any data you want to associate with the light object. The light does not use this property, but you can access it using `set` and `get`.

`Visible`

`{on} | off`

Light visibility. While light objects themselves are not visible, you can see the light on patch and surface objects. When you set `Visible` to `off`, the light emanating from the source is not visible. There must be at least one light object in the axes whose `Visible` property is `on` for any lighting features to be enabled (including the axes `AmbientLightColor` and patch and surface `AmbientStrength`).

Purpose Create or position `light` object in spherical coordinates

Syntax

```
lightangle(az,el)
light_handle = lightangle(az,el)
lightangle(light_handle,az,el)
[az,el] = lightangle(light_handle)
```

Description `lightangle(az,el)` creates a light at the position specified by azimuth and elevation. `az` is the azimuthal (horizontal) rotation and `el` is the vertical elevation (both in degrees). The interpretation of azimuth and elevation is the same as that of the `view` command.

`light_handle = lightangle(az,el)` creates a light and returns the handle of the light in `light_handle`.

`lightangle(light_handle,az,el)` sets the position of the light specified by `light_handle`.

`[az,el] = lightangle(light_handle)` returns the azimuth and elevation of the light specified by `light_handle`.

Remarks By default, when a light is created, its style is `infinite`. If the light handle passed in to `lightangle` refers to a local light, the distance between the light and the camera target is preserved as the position is changed.

Examples

```
surf(peaks)
axis vis3d
h = light;
for az = -50:10:50
 lightangle(h,az,30)
 drawnow
end
```

See Also `light`, `camlight`, `view`

for more information about lighting

"Lighting" on page 1-106 for related functions

lighting

Purpose

Specify lighting algorithm

Syntax

```
lighting flat
lighting gouraud
lighting phong
lighting none
```

Description

`lighting` selects the algorithm used to calculate the effects of `light` objects on all `surface` and `patch` objects in the current axes. In order for the `lighting` command to have any effects, however, you must create a lighting object by using the `light` function.

`lighting flat` produces uniform lighting across each of the faces of the object. Select this method to view faceted objects.

`lighting gouraud` calculates the vertex normals and interpolates linearly across the faces. Select this method to view curved surfaces.

`lighting phong` interpolates the vertex normals across each face and calculates the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but it takes longer to render.

`lighting none` turns off lighting.

Remarks

The `surf`, `mesh`, `pcolor`, `fill`, `fill3`, `surface`, and `patch` functions create graphics objects that are affected by light sources. The `lighting` command sets the `FaceLighting` and `EdgeLighting` properties of surfaces and patches appropriately for the graphics object.

See Also

`fill`, `fill3`, `light`, `material`, `mesh`, `patch`, `pcolor`, `shading`, `surface`

for more information about lighting

“Lighting” on page 1-106 for related functions

Purpose Convert linear audio signal to mu-law

Syntax `mu = lin2mu(y)`

Description `mu = lin2mu(y)` converts linear audio signal amplitudes in the range `-1 ≤ Y ≤ 1` to mu-law encoded “flints” in the range `0 ≤ u ≤ 255`.

See Also `auwrite`, `mu2lin`

line

Purpose Create line object

Syntax

```
line
line(X,Y)
line(X,Y,Z)
line(X,Y,Z,'PropertyName',propertyvalue,...)
line('XData',x,'YData',y,'ZData',z,...)
h = line(...)
```

Description `line` creates a line object in the current axes with default values `x = [0 1]` and `y = [0 1]`. You can specify the color, width, line style, and marker type, as well as other characteristics.

The `line` function has two forms:

- Automatic color and line style cycling. When you specify matrix coordinate data using the informal syntax (i.e., the first three arguments are interpreted as the coordinates),

  ```
  line(X,Y,Z)
  ```

  MATLAB cycles through the axes `ColorOrder` and `LineStyleOrder` property values the way the `plot` function does. However, unlike `plot`, `line` does not call the `newplot` function.
- Purely low-level behavior. When you call `line` with only property name/property value pairs,

  ```
  line('XData',x,'YData',y,'ZData',z)
  ```

  MATLAB draws a line object in the current axes using the default line color (see the `colordef` function for information on color defaults). Note that you cannot specify matrix coordinate data with the low-level form of the `line` function.

`line(X,Y)` adds the line defined in vectors `X` and `Y` to the current axes. If `X` and `Y` are matrices of the same size, `line` draws one line per column.

`line(X,Y,Z)` creates lines in three-dimensional coordinates.

`line(X,Y,Z,'`*`PropertyName`*`',propertyvalue,...)` creates a line using the values for the property name/property value pairs specified and default values for all other properties.

See the `LineStyle` and `Marker` properties for a list of supported values.

`line('XData',x,'YData',y,'ZData',z,...)` creates a line in the current axes using the property values defined as arguments. This is the low-level form of the `line` function, which does not accept matrix coordinate data as the other informal forms described above.

`h = line(...)` returns a column vector of handles corresponding to each line object the function creates.

Remarks

In its informal form, the `line` function interprets the first three arguments (two for 2-D) as the `X`, `Y`, and `Z` coordinate data, allowing you to omit the property names. You must specify all other properties as name/value pairs. For example,

```
line(X,Y,Z,'Color','r','LineWidth',4)
```

The low-level form of the `line` function can have arguments that are only property name/property value pairs. For example,

```
line('XData',x,'YData',y,'ZData',z,'Color','r','LineWidth',4)
```

Line properties control various aspects of the line object and are described in the "Line Properties" section. You can also set and query property values after creating the line using `set` and `get`.

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

Unlike high-level functions such as `plot`, `line` does not respect the settings of the figure and axes `NextPlot` properties. It simply adds line objects to the current axes. However, axes properties that are under automatic control, such as the axis limits, can change to accommodate the line within the current axes.

Connecting the dots

The coordinate data is interpreted as vectors of corresponding x, y, and z values:

```
X = [x(1) x(2) x(3)...x(n)]
Y = [y(1) x(2) y(3)...y(n)]
Z = [z(1) z(2) x(3)...z(n)]
```

where a point is determined by the corresponding vector elements:

```
p1(x(i),y(i),z(i))
```

For example, to draw a line from the point located at `x = .3` and `y = .4` and `z = 1` to the point located at `x = .7` and `y = .9` and `z = 1`, use the following data:

```
axis([0 1 0 1])
line([.3 .7],[.4 .9],[1 1],'Marker','.','LineStyle','-')
```

Examples

This example uses the `line` function to add a shadow to plotted data. First, plot some data and save the line's handle:

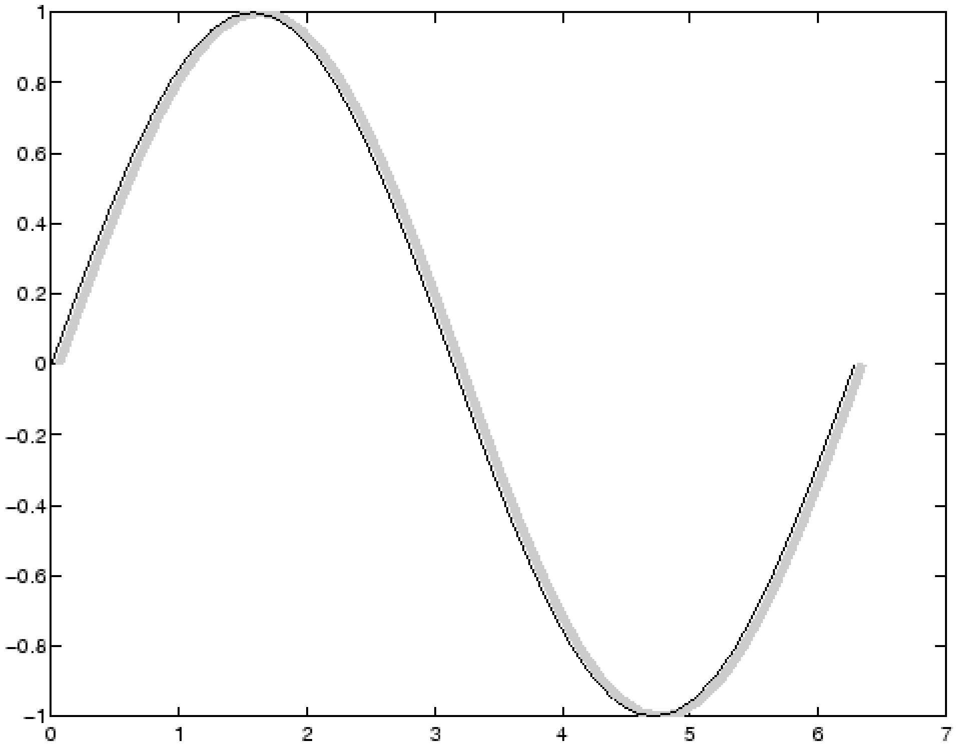
```
t = 0:pi/20:2*pi;
hline1 = plot(t,sin(t),'k');
```

Next, add a shadow by offsetting the *x*-coordinates. Make the shadow line light gray and wider than the default `LineWidth`:

```
hline2 = line(t+.06,sin(t),'LineWidth',4,'Color',[.8 .8 .8]);
```

Finally, pull the first line to the front:

```
set(gca,'Children',[hline1 hline2])
```



Drawing Lines Interactively

You can use the `ginput` function to select points from a figure. For example:

```
axis([0 1 0 1])
for n = 1:5
 [x(n),y(n)] = ginput(1);
end
line(x,y)
```

The `for` loop enables you to select five points and build the `x` and `y` arrays. Because `line` requires arrays of corresponding x and y coordinates, you can just pass these arrays to the `line` function.

Drawing with mouse motion

You can use the axes `CurrentPoint` property and the figure `WindowButtonDownFcn` and `WindowButtonMotionFcn` properties to select a point with a mouse click and draw a line to another point by dragging the mouse, like a simple drawing program. The following example illustrates a few useful techniques for doing this type of interactive drawing.

Click to view in editor — This example enables you to click and drag the cursor to draw lines.

Click to run example — Click the left mouse button in the axes and move the cursor, left-click to define the line end point, right-click to end drawing mode.

Input Argument Dimensions — Informal Form

This statement reuses the one-column matrix specified for `ZData` to produce two lines, each having four points.

```
line(rand(4,2),rand(4,2),rand(4,1))
```

If all the data has the same number of columns and one row each, MATLAB transposes the matrices to produce data for plotting. For example,

```
line(rand(1,4),rand(1,4),rand(1,4))
```

is changed to

```
line(rand(4,1),rand(4,1),rand(4,1))
```

This also applies to the case when just one or two matrices have one row. For example, the statement

```
line(rand(2,4),rand(2,4),rand(1,4))
```

is equivalent to

```
line(rand(4,2),rand(4,2),rand(4,1))
```

Setting Default Properties

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

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

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

See Also

`annotationaxes`, `newplot`, `plot`, `plot3`

"Object Creation" on page 1-99 for related functions

Line Properties for property descriptions

Purpose

Line properties

Modifying Properties

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

- The 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 Core Graphics Objects for general information about this type of object.

Line Property Descriptions

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

`Annotation`

`hg.Annotation` object Read Only

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

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

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

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

Setting the `IconDisplayStyle` property

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

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

Using the `IconDisplayStyle` property

See for more information and examples.

`BeingDeleted`

`on | {off}` Read Only

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

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

`BusyAction`

`cancel | {queue}`

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

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

`ButtonDownFcn`

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

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

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

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of line associated with the button down event and an event structure, which is empty for this property)

The following example shows how to access the callback object's handle as well as the handle of the figure that contains the object from the callback function.

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   sel_typ = get(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 line object and that the `button_down` function is on your MATLAB path. The following statement assigns the function above to the `ButtonDownFcn`:

```
set(h,'ButtonDownFcn',@button_down)
```

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

`Children`

vector of handles

The empty matrix; line objects have no children.

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips lines to the axes plot box by default. If you set `Clipping` to `off`, lines are displayed outside the axes plot box. This can occur if you create a line, set `hold` to `on`, freeze axis scaling (set `axis` to `manual`), and then create a longer line.

`Color`

`ColorSpec`

Line color. A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

`CreateFcn`

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

Callback function executed during object creation. A callback function that executes when MATLAB creates a line object. You must define this property as a default value for lines or in a call to the `line` function to create a new line object. For example, the statement

```
set(0,'DefaultLineCreateFcn',@line_create)
```

defines a default value for the line `CreateFcn` property on the root level that sets the axes `LineStyleOrder` whenever you create a line object. The callback function must be on your MATLAB path when you execute the above statement.

```
function line_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
 axh = get(src,'Parent');
 set(axh,'LineStyleOrder','-.|--')
end
```

MATLAB executes this function after setting all line properties. Setting this property on an existing line object has no effect. The function must define at least two input arguments (handle of line object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

`DeleteFcn`

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

Delete line callback function. A callback function that executes when you delete the line object (e.g., when you issue a `delete` command or clear the axes `cla` or figure `clf`). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of line object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

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

`DisplayName`

string (default is empty string)

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

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

See for more examples.

The following code shows how to use the `DisplayName` property from the command line or in an M-file.

```
t = 0:.1:2*pi;
a(:,1)=sin(t); a(:,2)=cos(t);
```

```
h = plot(a);
set(h,{'DisplayName'},{'Sine','Cosine'}')
legend show
```

`EraseMode`

`{normal} | none | xor | background`

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

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

Printing with Nonnormal Erase Modes

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

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

`HitTest`

`{on} | off`

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

`HandleVisibility`

`{on} | callback | off`

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

Handles are always visible when `HandleVisibility` is `on`.

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

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

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

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

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

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

`Interruptible`

`{on} | off`

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

`LineStyle`

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

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

Symbol	Line Style
`'  '`	Solid line (default)
`'--'`	Dashed line
`':'`	Dotted line
`' .'`	Dash-dot line
`'none'`	No line

You can use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`

scalar

The width of the line object. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`

character (see table)

Marker symbol. The `Marker` property specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the table.

Marker Specifier	**Description**
`'+'`	Plus sign
`'o'`	Circle
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No marker (default)

`MarkerEdgeColor`

`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the line's `Color` property.

`MarkerFaceColor`

`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or the figure color, if the axes `Color` property is set to `none` (which is the factory default for axes).

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker, in points. The default value for `MarkerSize` is six points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of line object. This property contains the handle of the line object's parent. The parent of a line object is the axes that contains it. You can reparent line objects to other axes, hggroup, or hgtransform objects.

See for more information on parenting graphics objects.

`Selected`
`on | off`

Is object selected? When this property is `on`. MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is on, MATLAB indicates the selected state by drawing handles at each vertex. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`

string

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

`Type`

string (read only)

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

`UIContextMenu`

handle of a uicontextmenu object

Associate a context menu with the line. Assign this property the handle of a uicontextmenu object created in the same figure as the line. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the line.

`UserData`

matrix

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

`Visible`

`{on} | off`

Line visibility. By default, all lines are visible. When set to `off`, the line is not visible, but still exists, and you can `get` and `set` its properties.

`XData`

vector of coordinates

X-coordinates. A vector of *x*-coordinates defining the line. `YData` and `ZData` must be the same length and have the same number of rows. (See "Examples" on page 2-2120.)

`YData`

vector of coordinates

Y-coordinates. A vector of *y*-coordinates defining the line. `XData` and `ZData` must be the same length and have the same number of rows.

`ZData`

vector of coordinates

Z-coordinates. A vector of *z*-coordinates defining the line. `XData` and `YData` must have the same number of rows.

Purpose

Define lineseries properties

Modifying Properties

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

See for more information on lineseries objects.

Note that you cannot define default properties for lineseries objects.

Lineseries Property Descriptions

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

`Annotation`

`hg.Annotation` object Read Only

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

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

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

IconDisplayStyle Value	Purpose
`on`	Include the lineseries object in a legend as one entry, but not its children objects
`off`	Do not include the lineseries or its children in a legend (default)
`children`	Include only the children of the lineseries as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

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

Using the IconDisplayStyle Property

See for more information and examples.

`BeingDeleted`

`on | {off}` Read Only

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

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

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object.

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

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

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`
vector of handles

The empty matrix; line objects have no children.

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`

ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

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

`CreateFcn`

string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

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

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

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 lineseries object. The `legend` function uses the string defined by the `DisplayName` property to label this lineseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this lineseries object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object

based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.

- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See 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).

`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)

Class of graphics object. For lineseries objects, `Type` is always the string `line`.

`UIContextMenu`

handle of a uicontextmenu object

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

`UserData`

array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`

`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`

vector or matrix

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

You can use `XData` to define meaningful coordinates for an underlying surface whose topography is being mapped. See for more information.

`XDataMode`

{auto} | manual

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

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`

string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

`YData`

vector or matrix of coordinates

Y-coordinates. A vector of *y*-coordinates defining the values along the *y*-axis for the graph. `XData` and `ZData` must be the same length and have the same number of rows.

`YDataSource`

string (MATLAB variable)

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

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

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

See the `refreshdata` reference page for more information.

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

`ZData`

vector of coordinates

Z-coordinates. A vector defining the *z*-coordinates for the graph. `XData` and `YData` must be the same length and have the same number of rows.

`ZDataSource`

string (MATLAB variable)

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.

LineSpec (Line Specification)

Purpose Line specification string syntax

GUI Alternative To modify the style, width, and color of lines on a graph, use the Property Editor, one of the plotting tools. For details, see The Property Editor in the MATLAB Graphics documentation.

Description This page describes how to specify the properties of lines used for plotting. MATLAB graphics give you control over these visual characteristics:

- Line style
- Line width
- Color
- Marker type
- Marker size
- Marker face and edge coloring (for filled markers)

You indicate the line styles, marker types, and colors you want to display using *string specifiers*, detailed in the following tables:

Line Style Specifiers

Specifier	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker Specifiers

Specifier	Marker Type
`+`	Plus sign

Specifier	Marker Type
`o`	Circle
`*`	Asterisk
`.`	Point (see note below)
`x`	Cross
`'square'` or `s`	Square
`'diamond'` or `d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`'pentagram'` or `p`	Five-pointed star (pentagram)
`'hexagram'` or `h`	Six-pointed star (hexagram)

Note The point (.) marker type does not change size when the specified value is less than 5.

Color Specifiers

Specifier	Color
`r`	Red
`g`	Green
`b`	Blue
`c`	Cyan
`m`	Magenta
`y`	Yellow

Specifier	Color
`k`	Black
`w`	White

All high-level plotting functions (except for the `ez...` family of function-plotting functions) accept a `LineSpec` argument that defines three components used to specify lines:

- Line style
- Marker symbol
- Color

For example:

```
plot(x,y,'-.or')
```

plots `y` versus `x` using a dash-dot line (`-.`), places circular markers (`o`) at the data points, and colors both line and marker red (`r`). Specify the components (in any order) as a quoted string after the data arguments. Note that linespecs are single strings, not property-value pairs.

Plotting Data Points with No Line

If you specify a marker, but not a line style, only the markers are plotted. For example:

```
plot(x,y,'d')
```

Related Properties

When using the `plot` and `plot3` functions, you can also specify other characteristics of lines using graphics properties:

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

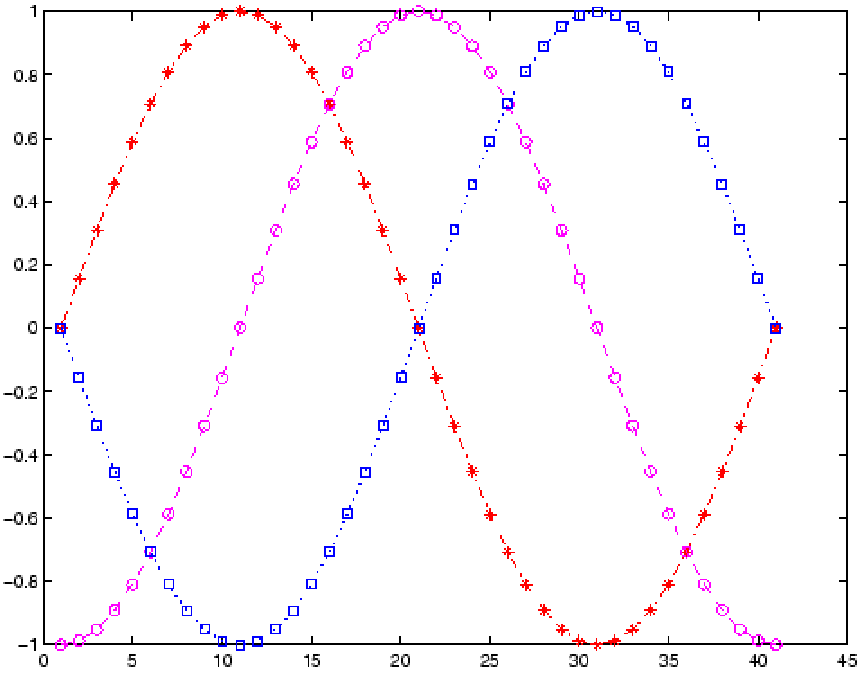
- `MarkerFaceColor` — Specifies the color of the face of filled markers.
- `MarkerSize` — Specifies the size of the marker in points (must be greater than 0).

In addition, you can specify the `LineStyle`, `Color`, and `Marker` properties instead of using the symbol string. This is useful if you want to specify a color that is not in the list by using RGB values. See Line Properties for details on these properties and `ColorSpec` for more information on color.

Examples

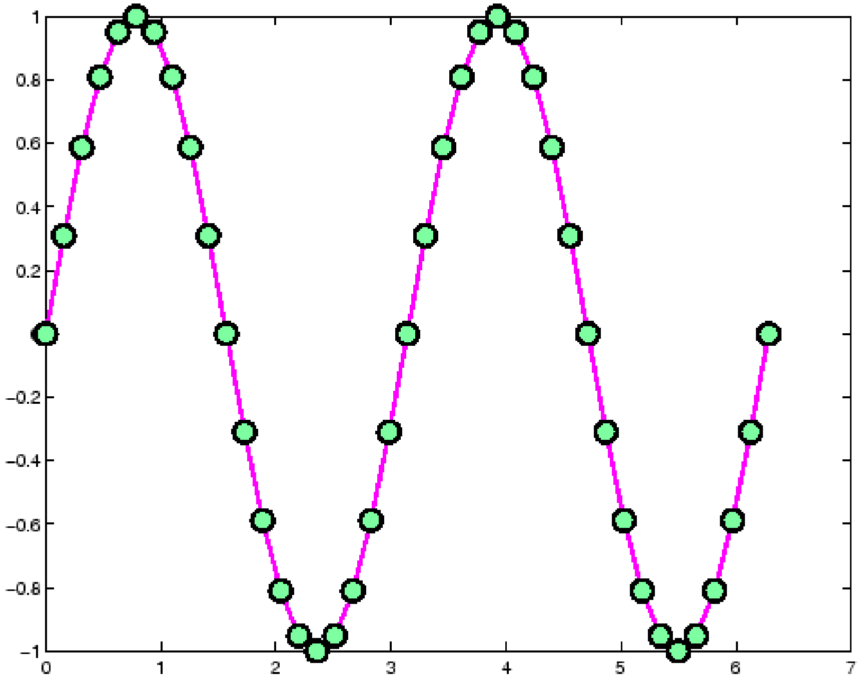
Plot the sine function over three different ranges using different line styles, colors, and markers.

```
t = 0:pi/20:2*pi;
plot(t,sin(t),'-.r*')
hold on
plot(t,sin(t-pi/2),'--mo')
plot(t,sin(t-pi),':bs')
hold off
```



Create a plot illustrating how to set line properties:

```
plot(t,sin(2*t),'-mo',...
                'LineWidth',2,...
                'MarkerEdgeColor','k',...
                'MarkerFaceColor',[.49 1 .63],...
                'MarkerSize',12)
```



See Also

`axes`, `line`, `plot`, `patch`, `set`, `surface`, `Line Properties`, `ColorSpec`

for information about defining an order for applying linestyles

for functions that use linespecs

"Basic Plots and Graphs" on page 1-91 for related functions

linkaxes

Purpose

Synchronize limits of specified 2-D axes

Syntax

```
linkaxes(axes_handles)
linkaxes(axes_handles,'option')
```

Description

Use `linkaxes` to synchronize the individual axis limits across several figures or subplots within a figure. Calling `linkaxes` makes all input axes have identical limits. Linking axes is best when you want to zoom or pan in one subplot and display the same range of data in another subplot.

`linkaxes(axes_handles)` links the *x*- and *y*-axis limits of the axes specified in the vector `axes_handles`. You can link any number of existing plots or subplots. The `axes_handles` input should be a vector of the handles for each plot or subplot. Entering an array of values results in an error message.

`linkaxes(axes_handles,'`*`option`*`')` links the axes' `axes_handles` according to the specified option. The *`option`* argument can be one of the following strings:

`x`	Link *x*-axis only.
`y`	Link *y*-axis only.
`xy`	Link *x*-axis and *y*-axis.
`off`	Remove linking.

See the `linkprop` function for more advanced capabilities that allow you to link object properties on any graphics object.

Remarks

The first axes you supply to `linkaxes` determines the *x*- and *y*-limits for all linked axes. This can cause plots to partly or entirely disappear if their limits or scaling are very different. To override this behavior, after calling `linkaxes`, specify the limits of the axes that you want to control with the `set` command, as shown in Example 3.

Note `linkaxes` is not designed to be transitive across multiple invocations. If you have three axes, `ax1`, `ax2`, and `ax3` and want to link them together, call `linkaxes` with `[ax1, ax2, ax3]` as the first argument. Linking `ax1` to `ax2`, then `ax2` to `ax3`, "unbinds" the `ax1`-`ax2` linkage.

Examples

You can use interactive zooming or panning (selected from the figure toolbar) to see the effect of axes linking. For example, pan in one graph and notice how the *x*-axis also changes in the other. The axes responds in the same way to `zoom` and `pan` directives you type in the Command Window.

Example 1

This example creates two subplots and links the *x*-axis limits of the two axes:

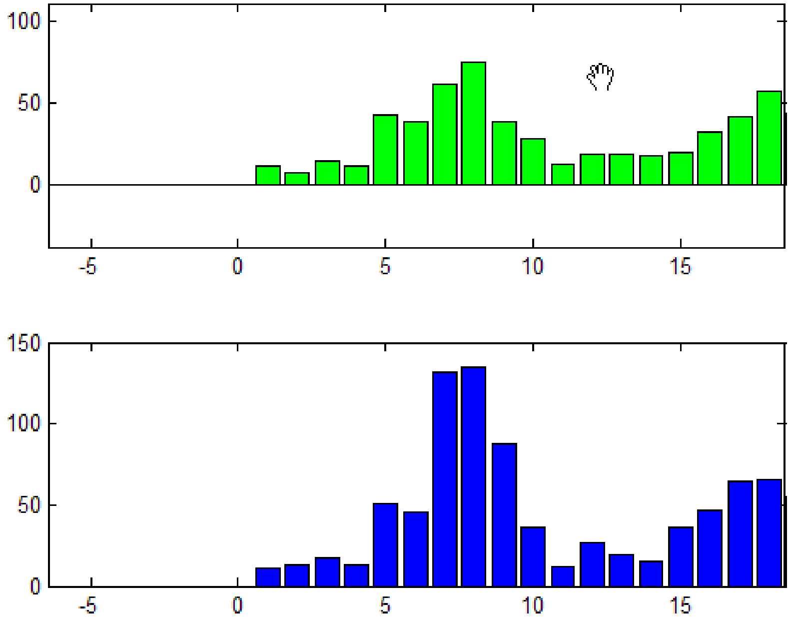
```
ax(1) = subplot(2,2,1);
plot(rand(1,10)*10,'Parent',ax(1));
ax(2) = subplot(2,2,2);
plot(rand(1,10)*100,'Parent',ax(2));
linkaxes(ax,'x');
```

Example 2

This example creates two figures and links the *x*-axis limits of the two axes. The illustration shows the effect of manually panning the top subplot:

```
load count.dat
figure; ax(1) = subplot(2,1,1);
h(1) = bar(ax(1),count(:,1),'g');
ax(2) = subplot(2,1,2);
h(2) = bar(ax(2),count(:,2),'b');
linkaxes(ax,'x');
```

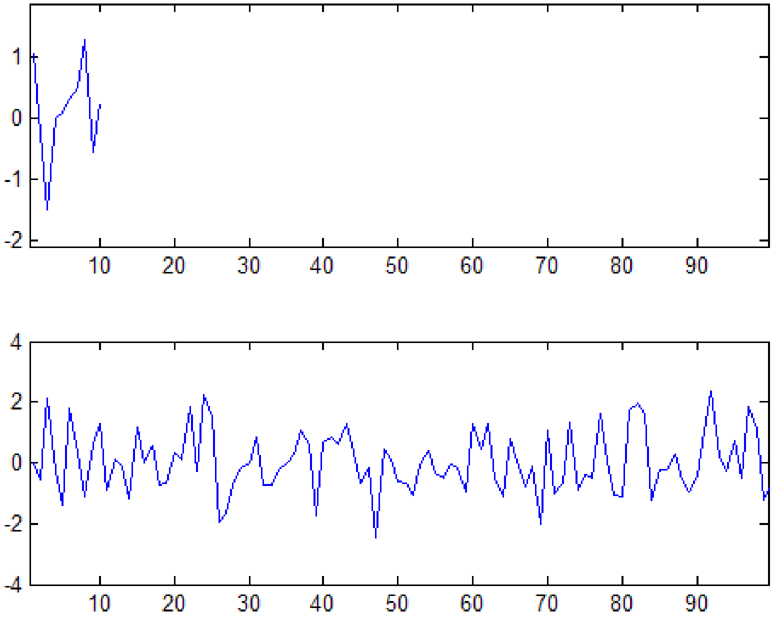
Choose the Pan tool (**Tools > Pan**) and drag the top axes. Both axes pans in step in *x*, but only the top one pans in *y*.



Example 3

Create two subplots containing data having different ranges. The first axes handle passed to `linkaxes` determines the data range for all other linked axes. In this example, calling `set` for the lower axes overrides the *x*-limits established by the call to `linkaxes`:

```
a1 = subplot(2,1,1);
plot(randn(10,1));         % Plot 10 numbers on top
a2 = subplot(2,1,2);
plot(a2,randn(100,1))      % Plot 100 numbers below
linkaxes([a1 a2], 'x');    % Link the axes; subplot 2 now out of range
set(a2,'xlimmode','auto'); % Now both axes run from 1-100 in x
                           % You could also use set(a2,'xlim',[1 100])
```



See Also `linkdata`, `linkprop`, `pan`, `zoom`

linkdata

Purpose

Automatically update graphs when variables change

GUI Alternatives

To turn data linking on or off, click the Data Linking tool in the figure toolbar. When on, an information bar appears below the figure's toolbar to identify and specify data sources for graphs.

Linked variables/expressions: y(:,1) vs. x

For details, see in the MATLAB Data Analysis documentation.

Syntax

```
linkdata on
linkdata off
linkdata
linkdata(figure_handle,...)
linkobj = linkdata(figure_handle)
```

Description

`linkdata on` turns on data linking for the current figure.

`linkdata off` turns data linking off.

`linkdata` by itself toggles the state of data linking.

`linkdata(figure_handle,...)` applies the function to the specified figure handle.

`linkobj = linkdata(figure_handle)` returns a *linkdata object* for the specified figure. The object has one read-only property, `Enable`, the string `'on'` or `'off'`, depending on the linked state of the figure.

Data linking connects graphs in figure windows to variables in the base or a function's workspace via their `XDataSource`, `YDataSource`, and `ZDataSource` properties. When you turn on data linking for a figure, variables in the current (base or caller) workspace are compared to the `XData`, `YData`, and `ZData` properties of graphs in the affected figure to try to match them. When a match is found, the appropriate `XDataSource`, `YDataSource` and/or `ZDataSource` for the graph are set to strings that name the matching variables.

Any subsequent changes to linked variables are then reflected in graphs that use them as data sources and in the Variable Editor, if the linked variables are displayed there. Conversely, any changes to plotted data values made at the command line, in the Variable Editor, or with the Brush tool (such as deleting or replacing data points), are immediately reflected in the workspace variables linked to the data points.

When a figure containing graphs is linked and any variable identified as `XDataSource`, `YDataSource`, and/or `ZDataSource` changes its values in the workspace, all graphs displaying it in that and other linked figures automatically update. This operation is equivalent to automatically calling the `refreshdata` function on the corresponding figure when a variable changes.

Linked figure windows identify themselves by the appearance of the Linked Plot information bar at the top of the window. When `linkdata` is `off` for a figure, the Linked Plot information bar is removed. If `linkdata` cannot unambiguously identify data sources for a graph in a linked figure, it reports this via the Linked Plot information bar, which gives the user an opportunity to identify data sources. The information bar displays a warning icon and a message, **Graphics have no data sources** and also prompts **Click here to fix it**. Clicking the word **here** opens the Specify Data Sources dialog box for specifying names and ranges of data sources for the figure.

Remarks

- "Types of Variables You Can Link" on page 2-2167
- "Restoring Links that Break" on page 2-2168
- "Linking Rapidly Changing Data" on page 2-2168
- "Linking Brushed Graphs" on page 2-2168

Types of Variables You Can Link

You can use `linkdata` to connect a graph with scalar, vector and matrix numeric variables of any class (including `complex`, if the graphing function can plot it) — essentially any data for which `isnumeric` equals `true`. See "Example 3" on page 2-2170 for instructions on linking complex variables. You can also link plots to numeric fields within structures. You can specify MATLAB expressions as data sources, for example, `sqrt(y)+1`.

Restoring Links that Break

Refreshing data on a linked plot fails if the strings in the `XDataSource`, `YDataSource`, or `ZDataSource` properties, when evaluated, are incompatible with what is in the current workspace, such that the corresponding `XData`, `YData`, or `ZData` are unable to respond. The visual appearance of the object in the graph is not affected by such failures, so graphic objects show no indication of broken links. Instead, a warning icon and the message **Failing links** appear on the Linked Plot information bar along with an **Edit** button that opens the Specify Data Sources dialog box.

Linking Rapidly Changing Data

`linkdata` buffers updates to data and dispatches them to plots at roughly half-second intervals. This makes data linking not suitable for smoothly animating changes in data values unless they are updated in loops that are forced to execute two times per second or less.

One consequence of buffering link updates is that `linkdata` might not detect changes in data streams it monitors. If you are running a function that uses `assignin` or `evalin` to update workspace variables, `linkdata` can sometimes fail to process updates that change values but not the size and class of workspace variables. Such failures only happen when the function itself updates the plot.

Linking Brushed Graphs

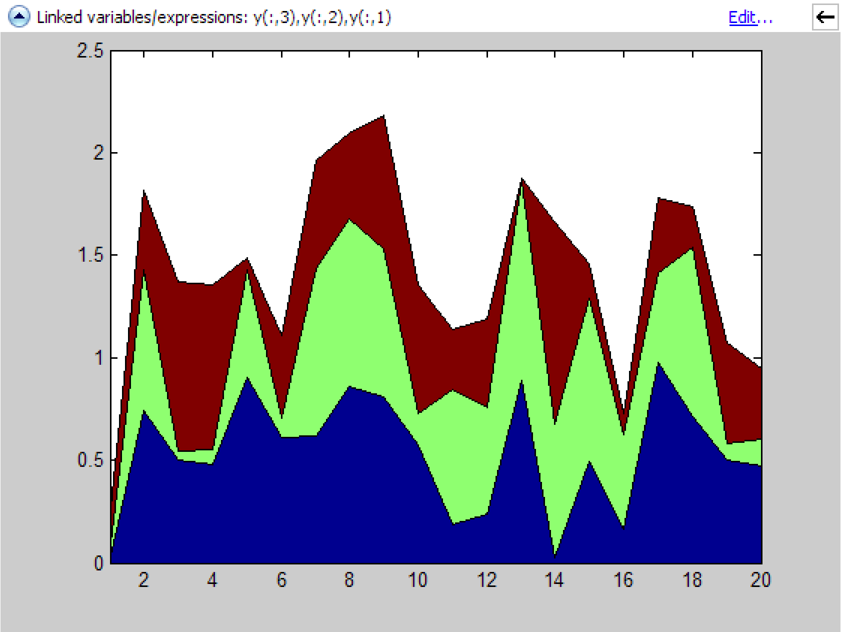
If you link data sources to graphs that have been brushed, their brushing marks can change or vanish. This is because the workspace variables in those graphs now dictate which, if any, observations are brushed, superseding any brushing annotations that were applied to their graphical data (`YData`, etc.). For more details, see in the `brush` reference page.

Examples

Example 1

Create two variables, graph them as areaseries, and link the plot to them:

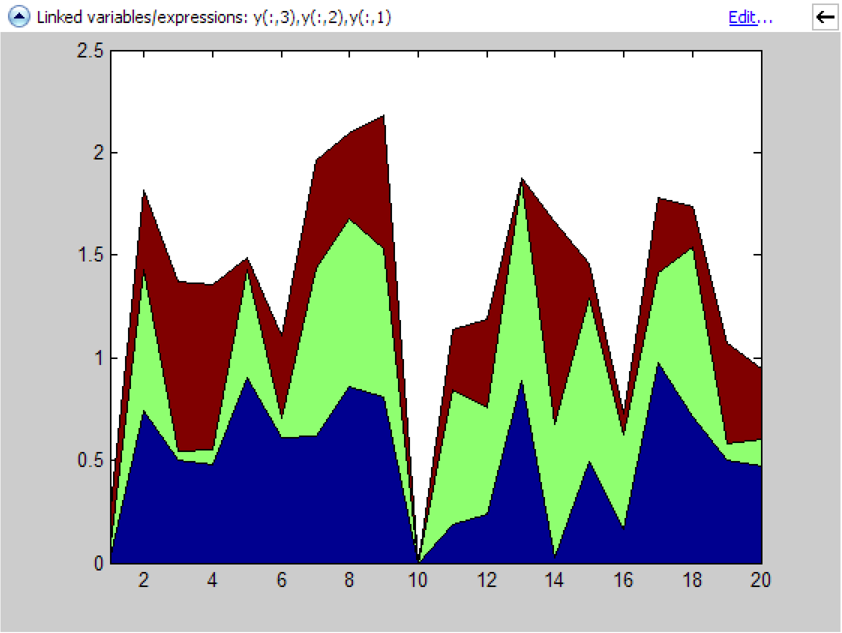
```
x = [1:20];
y = rand(20,3);
area(x,y)
linkdata on
```



Change values for linked variable `y` in the workspace:

```
y(10,:) = 0;
```

The area graph immediately updates.



Example 2

Delete a figure if it is not linked, based on a returned `linkdata` object:

```
ld = linkdata(fig)

ld =
     graphics.linkdata

if strcmp(ld.Enable,'off')
    delete(fig)
end
```

Example 3

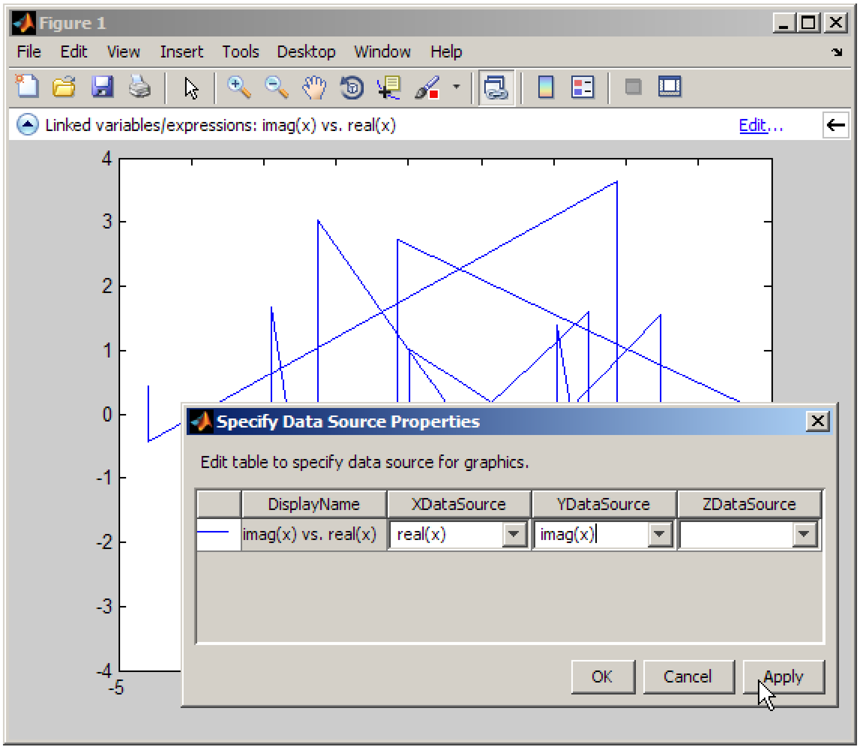
If a graphing function can display a complex variable, you can link such plots. To do so, you need to describe the data sources as expressions to separate the real and imaginary parts of the variable. For example,

```
x = eig(randn(20,20))
whos
  Name        Size       Bytes  Class     Attributes
  x          20x1          320  double    complex
```

yields a complex vector. You can use `plot` to display the real portion as *x* and the imaginary portion as *y*, then link the graph to the variable:

```
plot(x)
linkdata
```

However, `linkdata` cannot unambiguously identify the graph's data sources, and you must tell it by typing `real(x)` and `imag(x)` into the Specify Data Source Properties dialog box that displays when you click **fix it** in the Linked Plot information bar.



Linked variables/expressions: imag(x) vs. real(x)

Edit table to specify data source for graphics.

	DisplayName	XDataSource	YDataSource	ZDataSource
—	imag(x) vs. real(x)	real(x)	imag(x)	

To avoid having to type the data source names in the dialog box, you can specify them when you plot:

```
plot(x,'XDataSource','real(x)','YDataSource','imag(x)')
```

If you subsequently change values of `x` programmatically or manually, the plot updates accordingly.

Note Although you can use data brushing on linked plots of complex data, your brush marks only appear in the plot you are brushing, not in other plots or in the Variable Editor. This is because function calls, such as `real(x)` and `imag(x)`, that you specify as data sources are not interpreted when brushing graphed data.

See Also

`brush` , `linkaxes`, `linkprop`, `refreshdata`

Purpose

Keep same value for corresponding properties

Syntax

```
hlink = linkprop(obj_handles,'PropertyName')
hlink = linkprop(obj_handles,{'PropertyName1','PropertyName2',...})
```

Description

Use `linkprop` to maintain the same values for the corresponding properties of different objects.

`hlink = linkprop(obj_handles,'`*`PropertyName`*`')` maintains the same value for the property *`PropertyName`* on all objects whose handles appear in `obj_handles`. `linkprop` returns the link object in `hlink`. See "Link Object" on page 2-2174 for more information.

`hlink = linkprop(obj_handles,{'`*`PropertyName1`*`','`*`PropertyName2`*`',...})` maintains the same respective values for all properties passed as a cell array on all objects whose handles appear in `obj_handles`.

Note that the linked properties of all linked objects are updated immediately when `linkprop` is called. The first object in the list (`obj_handles`) determines the property values for the rest of the objects.

Link Object

The mechanism to link the properties of different graphics objects is stored in the link object, which is returned by `linkprop`. Therefore, the link object must exist within the context where you want property linking to occur (such as in the base workspace if users are to interact with the objects from the command line or figure tools).

The following list describes ways to maintain a reference to the link object.

- Return the link object as an output argument from a function and keep it in the base workspace while interacting with the linked objects.
- Make the `hlink` variable global.

- Store the `hlink` variable in an object's `UserData` property or in application data. See the "Examples" on page 2-2175 section for an example that uses application data.

Modifying Link Object

If you want to change either the graphics objects or the properties that are linked, you need to use the link object methods designed for that purpose. These methods are functions that operate only on link objects. To use them, you must first create a link object using `linkprop`.

Method	Purpose
`addtarget`	Add specified graphics object to the link object's targets.
`removetarget`	Remove specified graphics object from the link object's targets.
`addprop`	Add specified property to the linked properties.
`removeprop`	Remove specified property from the linked properties.

Method Syntax

```
addtarget(hlink,obj_handles)
removetarget(hlink,obj_handles)
addprop(hlink,'PropertyName')
removeprop(hlink,'PropertyName')
```

Arguments

- `hlink` — Link object returned by `linkprop`
- `obj_handles` — One or more graphic object handles
- *`PropertyName`* — Name of a property common to all target objects

Examples

This example creates four isosurface graphs of fluid flow data, each displaying a different isovalue. The `CameraPosition` and `CameraUpVector` properties of each subplot axes are linked so that the user can rotate all subplots in unison.

After running the example, select **Rotate 3D** from the figure **Tools** menu and observe how all subplots rotate together.

Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

The property linking code is in step 3.

1. Define the data using the `flow` M-file and specify property values for the isosurface (which is a patch object).

```
function linkprop_example
[x y z v] = flow;
isoval = [-3 -1 0 1];
props.FaceColor = [0 0 .5];
props.EdgeColor = 'none';
props.AmbientStrength = 1;
props.FaceLighting = 'gouraud';
```

2. Create four subplot axes and add an isosurface graph to each one. Add a title and set viewing and lighting parameters using a local function (`set_view`). (`subplot`, `patch`, `isosurface`, `title`, `num2str`)

```
for k = 1:4
    h(k) = subplot(2,2,k);
    patch(isosurface(x,y,z,v,isoval(k)),props)
    title(h(k),['Isovalue = ',num2str(k)])
    set_view(h(k))
end
```

3. Link the `CameraPosition` and `CameraTarget` properties of all subplot axes. Since this example function will have completed execution when the user is rotating the subplots, the link object is stored in the first subplot axes application data. See `setappdata` for more information on using application data.

```
hlink = linkprop(h,{'CameraPosition','CameraUpVector'});
key = 'graphics_linkprop';
% Store link object on first subplot axes
setappdata(h(1),key,hlink);
```

4. The following local function contains viewing and lighting commands issued on each axes. It is called with the creation of each subplot (`view`, `axis`, `camlight`).

```
function set_view(ax)
% Set the view and add lighting
view(ax,3); axis(ax,'tight','equal')
camlight left; camlight right
% Make axes invisible and title visible
axis(ax,'off')
set(get(ax,'title'),'Visible','on')
```

Linking an Additional Property

Suppose you want to add the axes `PlotBoxAspectRatio` to the linked properties in the previous example. You can do this by modifying the link object that is stored in the first subplot axes' application data.

1. First click the first subplot axes to make it the current axes (since its handle was saved only within the creating function). Then get the link object's handle from application data (`getappdata`).

```
hlink = getappdata(gca,'graphics_linkprop');
```

2. Use the `addprop` method to add a new property to the link object.

```
addprop(hlink,'PlotBoxAspectRatio')
```

Since `hlink` is a reference to the link object (i.e., not a copy), `addprop` can change the object that is stored in application data.

See Also

`getappdata`, `linkaxes`, `linkdata`, `setappdata`

linsolve

Purpose

Solve linear system of equations

Syntax

```
X = linsolve(A,B)
X = linsolve(A,B,opts)
```

Description

`X = linsolve(A,B)` solves the linear system `A*X = B` using LU factorization with partial pivoting when A is square and QR factorization with column pivoting otherwise. The number of rows of `A` must equal the number of rows of `B`. If `A` is m-by-n and `B` is m-by-k, then `X` is n-by-k. `linsolve` returns a warning if `A` is square and ill conditioned or if it is not square and rank deficient.

`[X, R] = linsolve(A,B)` suppresses these warnings and returns `R`, which is the reciprocal of the condition number of `A` if `A` is square, or the rank of `A` if `A` is not square.

`X = linsolve(A,B,opts)` solves the linear system `A*X = B` or `A'*X = B`, using the solver that is most appropriate given the properties of the matrix `A`, which you specify in `opts`. For example, if `A` is upper triangular, you can set `opts.UT = true` to make `linsolve` use a solver designed for upper triangular matrices. If `A` has the properties in `opts`, `linsolve` is faster than `mldivide`, because `linsolve` does not perform any tests to verify that `A` has the specified properties.

Notes If `A` does not have the properties that you specify in `opts`, `linsolve` returns incorrect results and does not return an error message. If you are not sure whether `A` has the specified properties, use `mldivide` instead.

For small problems, there is no speed benefit in using `linsolve` on triangular matrices as opposed to using the `mldivide` function.

The `TRANSA` field of the `opts` structure specifies the form of the linear system you want to solve:

- If you set `opts.TRANSA = false`, `linsolve(A,B,opts)` solves `A*X = B`.
- If you set `opts.TRANSA` = `true`, `linsolve(A,B,opts)` solves `A'*X = B`.

The following table lists all the field of `opts` and their corresponding matrix properties. The values of the fields of `opts` must be `logical` and the default value for all fields is `false`.

Field Name	Matrix Property
`LT`	Lower triangular
`UT`	Upper triangular
`UHESS`	Upper Hessenberg
`SYM`	Real symmetric or complex Hermitian
`POSDEF`	Positive definite
`RECT`	General rectangular
`TRANSA`	Conjugate transpose — specifies whether the function solves `A*X = B` or `A'*X = B`

The following table lists all combinations of field values in `opts` that are valid for `linsolve`. A true/false entry indicates that `linsolve` accepts either true or false.

LT	UT	UHESS	SYM	POSDEF	RECT	TRANSA
`true`	`false`	`false`	`false`	`false`	`true/false`	`true/false`
`false`	`true`	`false`	`false`	`false`	`true/false`	`true/false`
`false`	`false`	`true`	`false`	`false`	`false`	`true/false`
`false`	`false`	`false`	`true`	`true/false`	`false`	`true/false`
`false`	`false`	`false`	`false`	`false`	`true/false`	`true/false`

Example

The following code solves the system `A'x = b` for an upper triangular matrix `A` using both `mldivide` and `linsolve`.

```
A = triu(rand(5,3)); x = [1 1 1 0 0]'; b = A'*x;
y1 = (A')\b
opts.UT = true; opts.TRANSA = true;
y2 = linsolve(A,b,opts)

y1 =

    1.0000
    1.0000
    1.0000
         0
         0


y2 =

    1.0000
    1.0000
    1.0000
         0
         0
```

Note If you are working with matrices having different properties, it is useful to create an options structure for each type of matrix, such as `opts_sym`. This way you do not need to change the fields whenever you solve a system with a different type of matrix `A`.

See Also

`mldivide`

Purpose

Generate linearly spaced vectors

Syntax

```
y = linspace(a,b)
y = linspace(a,b,n)
```

Description

The `linspace` function generates linearly spaced vectors. It is similar to the colon operator ":", but gives direct control over the number of points.

`y = linspace(a,b)` generates a row vector `y` of 100 points linearly spaced between and including `a` and `b.`

`y = linspace(a,b,n)` generates a row vector `y` of `n` points linearly spaced between and including `a` and `b`. For `n < 2`, `linspace` returns `b`.

See Also

`logspace`

The colon operator `:`

list (RandStream)

Purpose Random number generator algorithms

Class `@RandStream`

Syntax `RandStream.list`

Description `RandStream.list` lists all the generator algorithms that may be used when creating a random number stream with `RandStream` or `RandStream.create`. The available generator algorithms and their properties are given in the following table.

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
mt19937ar	Mersenne twister (used by default stream at MATLAB startup)	No	$2^{19936}-1$
mcg16807	Multiplicative congruential generator	No	$2^{31}-2$
mlfg6331_64	Multiplicative lagged Fibonacci generator	Yes	2^{124}
mrg32k3a	Combined multiple recursive generator	Yes	2^{127}

Keyword	**Generator**	**Multiple Stream and Substream Support**	**Approximate Period In Full Precision**
shr3cong	Shift-register generator summed with linear congruential generator	No	2^{64}
swb2712	Modified subtract with borrow generator	No	2^{1492}

For a full description of the Mersenne twister algorithm, see http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.

All of the generator and transformation algorithms that were available in MATLAB versions 7.6 and earlier are available in the current version. To create random streams that are equivalent to the legacy generators without entering into legacy mode, use the following syntaxes:

Legacy mode	**RandStream syntax**
`rand('seed',0)`	`(RandStream('mcg16807', 'Seed',0))`
`rand('state'0)`	`(RandStream('swb2712','Seed',0))`
`rand('twister',5489)`	`(RandStream('mt19937ar', 'Seed', 5489))`
`randn('seed',0)`	`(RandStream('mcg16807', 'Seed',0))`
`randn('state',0)`	`(RandStream('shr3cong'))`

For more information on compatibility issues with MATLAB versions 7.6 and earlier, see in the documentation.

Purpose

Create and open list-selection dialog box

Syntax

```
[Selection,ok] = listdlg('ListString',S)
```

Description

`[Selection,ok] = listdlg('ListString',S)` creates a modal dialog box that enables you to select one or more items from a list. `Selection` is a vector of indices of the selected strings (in single selection mode, its length is 1). `Selection` is `[]` when `ok` is 0. `ok` is 1 if you click the **OK** button, or 0 if you click the **Cancel** button or close the dialog box. Double-clicking on an item or pressing **Return** when multiple items are selected has the same effect as clicking the **OK** button. The dialog box has a **Select all** button (when in multiple selection mode) that enables you to select all list items.

Inputs are in parameter/value pairs:

Parameter	Description
`'ListString'`	Cell array of strings that specify the list box items.
`'SelectionMode'`	String indicating whether one or many items can be selected: `'single'` or `'multiple'` (the default).
`'ListSize'`	List box size in pixels, specified as a two-element vector `[width height]`. Default is `[160 300]`.
`'InitialValue'`	Vector of indices of the list box items that are initially selected. Default is 1, the first item.
`'Name'`	String for the dialog box's title. Default is ”.
`'PromptString'`	String matrix or cell array of strings that appears as text above the list box. Default is {}.
`'OKString'`	String for the OK button. Default is `'OK'`.
`'CancelString'`	String for the Cancel button. Default is ’`Cancel`’.
`’uh'`	Uicontrol button height, in pixels. Default is 18.

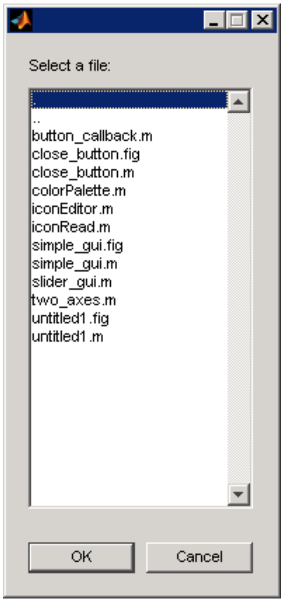
Parameter	Description
`'fus'`	Frame/uicontrol spacing, in pixels. Default is 8.
`'ffs'`	Frame/figure spacing, in pixels. Default is 8.

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

Example

This example displays a dialog box that enables the user to select a file from the current directory. The function returns a vector. Its first element is the index to the selected file; its second element is 0 if no selection is made, or 1 if a selection is made.

```
d = dir;
str = {d.name};
[s,v] = listdlg('PromptString','Select a file:',...
                'SelectionMode','single',...
                'ListString',str)
```



Select a file:

.
..
button_callback.m
close_button.fig
close_button.m
colorPalette.m
iconEditor.m
iconRead.m
simple_gui.fig
simple_gui.m
slider_gui.m
two_axes.m
untitled1.fig
untitled1.m

See Also `dialog`, `errordlg`, `helpdlg`, `inputdlg`, `msgbox`, `questdlg`, `warndlg`

`dir`, `figure`, `uiwait`, `uiresume`

for related functions

listfonts

Purpose

List available system fonts

Syntax

```
c = listfonts
c = listfonts(h)
```

Description

`c = listfonts` returns sorted list of available system fonts.

`c = listfonts(h)` returns sorted list of available system fonts and includes the `FontName` property of the object with handle `h`.

Remarks

Calling `listfonts` returns a list of all fonts residing on your system, possibly including fonts that cannot be used because they are bitmapped. You can instead use the `uisetfont` utility (a GUI) to preview fonts you might want to use; it only displays fonts that can be rendered in MATLAB figures and GUIs. Like `uisetfont`, the **Custom Fonts** pane of MATLAB Preferences also previews available fonts and only shows those that can be rendered.

Examples

Example 1

This example returns a list of available system fonts similar in format to the one shown.

```
list = listfonts

list =
    'Agency FB'
    'Algerian'
    'Arial'
    ...
    'ZapfChancery'
    'ZapfDingbats'
    'ZWAdobeF'
```

Example 2

This example returns a list of available system fonts with the value of the`FontName` property, for the object with handle `h`, sorted into the list.

```
 h = uicontrol('Style','text','String','My Font','FontName','MyFont
 list = listfonts(h)

list =
    'Agency FB'
    'Algerian'
    'Arial'
    ...
    'MyFont'
    ...
    'ZapfChancery'
    'ZapfDingbats'
    'ZWAdobeF'
```

See Also `uisetfont`

load

Purpose

Load workspace variables from disk

Syntax

```
load
load filename
load filename X Y Z ...
load filename -regexp expr1 expr2 ...
load -ascii filename
load -mat filename
S = load('arg1', 'arg2', 'arg3', ...)
```

Description

`load` loads all the variables from the MAT-file `matlab.mat`, if it exists, or returns an error if the file doesn't exist.

`load filename` loads all the variables from the file specified by `filename`. `filename` is an unquoted string specifying a file name, and can also include a file extension and a full or partial name. If `filename` has no extension, `load` looks for a file named `filename.mat` and treats it as a binary MAT-file. If `filename` has an extension other than `.mat`, `load` treats the file as ASCII data.

`load filename X Y Z ...` loads just the specified variables `X`, `Y`, `Z`, etc. from the MAT-file. The wildcard `'*'` loads variables that match a pattern (MAT-file only).

`load filename -regexp expr1 expr2 ...` loads those variables that match any of the given by `expr1`, `expr1`, etc.

`load -ascii filename` forces `load` to treat the file as an ASCII file, regardless of file extension. If the file is not numeric text, `load` returns an error.

`load -mat filename` forces `load` to treat the file as a MAT-file, regardless of file extension. If the file is not a MAT-file, `load` returns an error.

`S = load('arg1', 'arg2', 'arg3', ...)` calls `load` using MATLAB *function syntax*, (as opposed to the MATLAB *command syntax* that has been shown thus far). You can use function syntax with any form of the `load` command shown above, replacing `arg1`, `arg2`, etc. with the arguments shown. For example,

```
S = load('myfile.mat', '-regexp', '^Mon', '^Tue')
```

To specify a command line option, such as `-mat`, with the functional form, specify the option as a string argument, and include the hyphen. For example,

```
load('myfile.dat', '-mat')
```

Function syntax enables you to assign values returned by `load` to an output variable. You can also use function syntax when loading from a file having a name that contains space characters, or a filename that is stored in a variable.

If the file you are loading from is a MAT-file, then output `S` is a structure containing fields that match the variables retrieved. If the file contains ASCII data, then `S` is a double-precision array.

Remarks

For information on any of the following topics related to saving to MAT-files, see in the MATLAB Data Import and Export documentation:

-
-
-

You can also use the Current Folder browser to view the contents of a MAT-file without loading it — see .

MATLAB saves numeric data in MAT-files in the native byte format. The header of the MAT-file contains a 2-byte Endian Indicator that MATLAB uses to determine the byte format when loading the MAT-file. When MATLAB reads a MAT-file, it determines whether byte-swapping needs to be performed by the state of this indicator.

Examples

Example 1 — Loading From a Binary MAT-file

To see what is in the MAT-file prior to loading it, use `whos -file`:

```
whos -file mydata.mat
```

```
  Name           Size        Bytes  Class

  javArray      10x1                  java.lang.Double[][]
  spArray        5x5            84  double array (sparse)
  strArray       2x5           678  cell array
  x              3x2x2          96  double array
  y              4x5          1230  cell array
```

Clear the workspace and load it from MAT-file `mydata.mat`:

```
clear
load mydata

whos
  Name           Size        Bytes  Class

  javArray      10x1                java.lang.Double[][]
  spArray        5x5            84  double array (sparse)
  strArray       2x5           678  cell array
  x              3x2x2          96  double array
  y              4x5          1230  cell array
```

Example 2 — Loading a List of Variables

You can use a comma-separated list to pass the names of those variables you want to load from a file. This example generates a comma-separated list from a cell array

In this example, the file name is stored in a variable, `saved_file`. You must call `load` using the function syntax of the command if you intend to reference the file name through a variable:

```
saved_file = 'myfile.mat';
saved_file = 'ptarray.mat';
whos('-file', saved_file)

  Name          Size                 Bytes  Class

  AName         1x24            48  char array
```

```
  AVal          1x1               8  double array
  BName         1x24             48  char array
  BVal          1x1               8  double array
  CVal          5x5              84  double array (sparse)
  DArr          2x5             678  cell array

filevariables = {'AName', 'BVal', 'DArr'};
load(saved_file, filevariables{:});
```

The second part of this example generates a comma-separated list from the name field of a structure array, and loads the first ten variables from the specified file:

```
      saved_file = 'myfile.mat';
      vars = whos('-file', saved_file);
      load(saved_file, vars(1:10).name);
```

Example 3 — Loading From an ASCII File

Create several 4-column matrices and save them to an ASCII file:

```
a = magic(4);  b = ones(2, 4) * -5.7;  c = [8 6 4 2];
save -ascii mydata.dat
```

Clear the workspace and load it from the file `mydata.dat`. If the filename has an extension other than `.mat`, MATLAB assumes that it is ASCII:

```
clear
load mydata.dat
```

MATLAB loads all data from the ASCII file, merges it into a single matrix, and assigns the matrix to a variable named after the filename:

```
mydata
mydata =
   16.0000    2.0000    3.0000   13.0000
    5.0000   11.0000   10.0000    8.0000
    9.0000    7.0000    6.0000   12.0000
```

```
 4.0000   14.0000   15.0000    1.0000
-5.7000   -5.7000   -5.7000   -5.7000
-5.7000   -5.7000   -5.7000   -5.7000
 8.0000    6.0000    4.0000    2.0000
```

Example 4 — Using Regular Expressions

Using regular expressions, load from MAT-file `mydata.mat` those variables with names that begin with `Mon`, `Tue`, or `Wed`:

```
load('mydata', '-regexp', '^Mon|^Tue|^Wed');
```

Here is another way of doing the same thing. In this case, there are three separate expression arguments:

```
load('mydata', '-regexp', '^Mon', '^Tue', '^Wed');
```

See Also

`save`, `who`, `clear`, `uiimport`, `importdata`, `fileformats`, `type`, `spconvert`

Purpose

Initialize control object from file

Syntax

```
h.load('filename')
load(h, 'filename')
```

Description

`h.load('filename')` initializes the COM object associated with the interface represented by the MATLAB COM object `h` from file specified in the string `filename`. The file must have been created previously by serializing an instance of the same control.

`load(h, 'filename')` is an alternate syntax for the same operation.

Note The COM `load` function is only supported for controls at this time.

Remarks

COM functions are available on Microsoft Windows systems only.

Examples

Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.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 `save (COM)`, `actxcontrol`, `actxserver`, `release`, `delete (COM)`

load (serial)

Purpose Load serial port objects and variables into MATLAB workspace

Syntax

```
load filename
load filename obj1 obj2...
```

Description `load filename` returns all variables from the MAT-file specified by `filename` into the MATLAB workspace.

`load filename obj1 obj2...` returns the serial port objects specified by `obj1` `obj2` ... from the MAT-file `filename` into the MATLAB workspace.

`out = load('filename','obj1','obj2',...)` returns the specified serial port objects from the MAT-file `filename` as a structure to `out` instead of directly loading them into the workspace. The field names in `out` match the names of the loaded serial port objects.

Remarks Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

Example

Note This example is based on a Windows platform.

Suppose you create the serial port objects `s1` and `s2`, configure a few properties for `s1`, and connect both objects to their instruments:

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s1,'Parity','mark','DataBits',7);
fopen(s1);
fopen(s2);
```

Save `s1` and `s2` to the file `MyObject.mat`, and then load the objects back into the workspace:

```
save MyObject s1 s2;
```

```
load MyObject s1;
load MyObject s2;

get(s1, {'Parity', 'DataBits'})
ans =
    'mark'    [7]
get(s2, {'Parity', 'DataBits'})
ans =
    'none'    [8]
```

See Also

Functions

`save`

Properties

`Status`

Purpose Load shared library into MATLAB software

Syntax

```
loadlibrary('shrlib', 'hfile')
loadlibrary('shrlib', @protofile)
loadlibrary('shrlib', ..., 'options')
loadlibrary shrlib hfile options
[notfound, warnings] = loadlibrary('shrlib', 'hfile')
```

Description

`loadlibrary('shrlib', 'hfile')` loads the functions defined in header file `hfile` and found in shared library `shrlib` into MATLAB.

The `hfile` and `shrlib` file names are case sensitive. The name you use in `loadlibrary` must use the same case as the file on your system.

On Microsoft Windows systems, `shrlib` refers to the name of a Shared library (`.dll`) file. On Linux systems, it refers to the name of a shared object (`.so`) file. On Apple Macintosh systems, it refers to a dynamic shared library (`.dylib`). See “File Extensions for Libraries” on page 2-2199 for more information.

`loadlibrary('shrlib', @protofile)` uses the prototype M-file `protofile` in place of a header file in loading the library `shrlib`. The string `@protofile` specifies a function handle to the prototype M-file. (See the description of “Prototype M-Files” on page 2-2202 below).

Note The MATLAB Generic Shared Library interface does not support library functions that have function pointer inputs.

File Extensions for Libraries

If you do not include a file extension with the `shrlib` argument, `loadlibrary` attempts to find the library with either the appropriate platform MEX-file extension or the appropriate platform library extension (usually `.dll`, `.so`, or `.dylib`). For a list of file extensions, see .

If you do not include a file extension with the second argument, and this argument is not a function handle, `loadlibrary` uses `.h` for the extension.

`loadlibrary('shrlib', ..., 'options')` loads the library `shrlib` with one or more of the following *`options`*.

Option	Description
`addheader` `hfileN`	Loads the functions defined in the additional header file, `hfileN`. Note that each file specified by `addheader` must be referenced by a corresponding `#include` statement in the base header file. Specify the string `hfileN` as a file name without a file extension. MATLAB does not verify the existence of the header files and ignores any that are not needed. You can specify additional header files using the syntax:

```
loadlibrary shrlib hfile ...
   addheader hfile1 ...
   addheader hfile2 ...      % and so on
```

Option	Description
`alias` `name`	Associates the specified alias `name` with the library. All subsequent calls to MATLAB functions that reference this library must use this alias until the library is unloaded.
`includepath` `path`	Specifies an additional `path` in which to look for included header files.

Option	Description
`mfilename` `mfile`	Generates a prototype M-file `mfile` in the current directory. The `mfile` name must be different from the `shrlib` library name. You can use this file in place of a header file when loading the library. (See the following description of "Prototype M-Files" on page 2-2202).
`thunkfilename` `tfile`	Overrides the default thunk file name with `tfile`. For more information, see "Using loadlibrary on 64-Bit Platforms" on page 2-2203.

Only the **`alias`** option is available when loading using a prototype M-file.

If you have more than one library file of the same name, load the first using the library file name, and load the additional libraries using the **`alias`** option.

`loadlibrary shrlib hfile` *`options`* is the command format for this function.

`[notfound, warnings] = loadlibrary('shrlib', 'hfile')` returns warning information from the `shrlib` library file. `notfound` is a cell array of the names of functions found in the header file `hfile`, or any header added with the `addheader` option, but not found in the `shrlib` library. `warnings` contains a single character array of warnings produced while processing the header file `hfile`.

Remarks

How to Use the addheader Option

The `addheader` option enables you to add functions for MATLAB to load from those listed in header files included in the base header file (with a `#include` statement). For example, if your library header file contains the statement:

```
#include header2.h
```

then to load the functions in `header2.h`, you need to use `addheader` in the call to `loadlibrary`:

```
loadlibrary libname libname.h addheader header2.h
```

You can use the `addheader` option with a header file that lists function prototypes for only the functions that are needed by your library, and thereby avoid loading functions that you do not define in your library. To do this, you might need to create a header file that contains a subset of the functions listed in large header file.

addheader Syntax

When using `addheader` to specify which functions to load, ensure that there are `#include` statements in the base header file for each additional header file in the `loadlibrary` call. For example, to use the following statement:

```
loadlibrary mylib mylib.h addheader header2.h
```

the file `mylib.h` must contain this statement:

```
#include header2.h
```

Prototype M-Files

When you use the **`mfilename`** option with `loadlibrary`, MATLAB generates an M-file called a *prototype file.* Use this file on subsequent calls to `loadlibrary` in place of a header file.

Note The `mfile` name must be different from the `shrlib` library name.

Like a header file, the prototype file supplies MATLAB with function prototype information for the library. You can make changes to the prototypes by editing this file and reloading the library.

Here are some reasons for using a prototype file, along with the changes you would need to make to the file:

- You want to make temporary changes to signatures of the library functions.

  Edit the prototype file, changing the `fcns.LHS` or `fcns.RHS` field for that function. This changes the types of arguments on the left hand side or right hand side, respectively.

- You want to rename some of the library functions.

  Edit the prototype file, defining the `fcns.alias` field for that function.

- You expect to use only a small percentage of the functions in the library you are loading.

  Edit the prototype file, commenting out the unused functions. This reduces the amount of memory required for the library.

- You need to specify a number of include files when loading a particular library.

  Specify the full list of include files (plus the **`mfilename`** option) in the first call to `loadlibrary`. This puts all the information from the include files into the prototype file. After that, specify just the prototype file.

Using loadlibrary on 64-Bit Platforms

You must install a C compiler to use `loadlibrary` on a 64-bit platform and Perl must be available. The supported compilers are shown in the following table.

64-bit Platform	Required Compiler
Windows	Microsoft® Visual C++® 2005 SP1 Version 8.0 Professional Edition Microsoft Visual C++ 2008 SP1 Version 9.0 Professional Edition
Linux	gcc / g++ Version 4.1.1
Sun Solaris SPARC®	Sun Studio 12 cc / CC Version 5.9

MATLAB generates a *thunk file*, which is a compatibility layer to your 64-bit library. The name of the thunk file is:

```
BASENAME_thunk_COMPUTER.c
```

where *`BASENAME`* is either the name of the shared library or the *`mfilename`*, if specified. *`COMPUTER`* is the string returned by the `computer` function.

MATLAB compiles this file and creates the file:

```
BASENAME_thunk_COMPUTER.LIBEXT
```

where *`LIBEXT`* is the platform-dependent default shared library extension, for example, `dll` on Windows.

Examples

Load shrlibsample Example

Use `loadlibrary` to load the MATLAB sample shared library, `shrlibsample`:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h
```

Using alias Example

Load sample library `shrlibsample`, giving it an alias name of `lib`. Once you have set an alias, you need to use this name in all further interactions with the library for this session:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h alias lib

libfunctionsview lib

str = 'This was a Mixed Case string';
calllib('lib', 'stringToUpper', str)
ans =
   THIS WAS A MIXED CASE STRING
unloadlibrary lib
```

Using addpath Example

Load the library, specifying an additional path in which to search for included header files:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary('shrlibsample','shrlibsample.h','includepath', ...
          fullfile(matlabroot , 'extern', 'include'));
```

Using Prototype Example

Load the `libmx` library and generate a prototype M-file containing the prototypes defined in header file `matrix.h`:

```
hfile = [matlabroot '\extern\include\matrix.h'];
loadlibrary('libmx', hfile, 'mfilename', 'mxproto')

dir mxproto.m
   mxproto.m
```

Edit the generated file `mxproto.m` and locate the function `mxGetNumberOfDimensions`. Give it an alias of `mxGetDims` by adding this text to the line before `fcnNum` is incremented:

```
fcns.alias{fcnNum}='mxGetDims';
```

Here is the new function prototype. The change is shown in bold:

```
fcns.name{fcnNum}='mxGetNumberOfDimensions';
fcns.calltype{fcnNum}='cdecl';
fcns.LHS{fcnNum}='int32';
fcns.RHS{fcnNum}={'MATLAB array'};
fcns.alias{fcnNum}='mxGetDims'; % Alias defined
fcnNum=fcnNum+1; % Increment fcnNum
```

Unload the library and then reload it using the prototype M-file.

```
unloadlibrary libmx

loadlibrary('libmx', @mxproto)
```

Now call `mxGetNumberOfDimensions` using the alias function name:

```
y = rand(4, 7, 2);

calllib('libmx', 'mxGetDims', y)
ans =
     3

unloadlibrary libmx
```

See Also `unloadlibrary`, `libisloaded`, `libfunctions`,

Purpose

Modify load process for object

Syntax

`b = loadobj(a)`

Description

`b = loadobj(a)` is called by the `load` function if the class of `a` defines a `loadobj` method. `load` returns `b` as the value loaded from a MAT-file.

Define a `loadobj` method when objects of the class require special processing when loaded from MAT-files. If you define a `saveobj` method, then define a `loadobj` method to restore the object to the desired state. Define `loadobj` as a static method so it can accept as an argument whatever object or structure that you saved in the MAT-file. See .

When loading a subclass object, `load` calls only the subclass `loadobj` method. If a superclass defines a `loadobj` method, the subclass inherits this method. However, it is possible that the inherited method does not perform the necessary operations to load the subclass object. Consider overriding superclass `loadobj` methods.

If any superclass in a class hierarchy defines a `loadobj` method, then the subclass `loadobj` method must ensure that the subclass and superclass objects load properly. Ensure proper loading by calling the superclass `loadobj` (or other methods) from the subclass `loadobj` method. See .

Inputs

`a`

The input argument, `a`, can be:

- The object as loaded from the MAT-file.
- A structure created by `load` (if `load` cannot resolve the object).
- A `struct` returned by the `saveobj` method.

See Also

`load` | `save` | `saveobj`

Tutorials

-

Purpose

Natural logarithm

Syntax

```
Y = log(X)
```

Description

The `log` function operates element-wise on arrays. Its domain includes complex and negative numbers, which may lead to unexpected results if used unintentionally.

`Y = log(X)` returns the natural logarithm of the elements of `X`. For complex or negative z, where $z = x + y^*i$, the complex logarithm is returned.

```
log(z) = log(abs(z)) + i*atan2(y,x)
```

Examples

The statement `abs(log(-1))` is a clever way to generate π.

```
ans =

    3.1416
```

See Also

`exp`, `log10`, `log2`, `logm`, `reallog`

Purpose Common (base 10) logarithm

Syntax

```
Y = log10(X)
```

Description The `log10` function operates element-by-element on arrays. Its domain includes complex numbers, which may lead to unexpected results if used unintentionally.

`Y = log10(X)` returns the base 10 logarithm of the elements of `X`.

Examples

```
log10(realmax) is 308.2547
```

and

```
log10(eps) is -15.6536
```

See Also `exp`, `log`, `log2`, `logm`

Purpose Compute `log(1+x)` accurately for small values of `x`

Syntax

```
y = log1p(x)
```

Description `y = log1p(x)` computes `log(1+x)`, compensating for the roundoff in `1+x`. `log1p(x)` is more accurate than `log(1+x)` for small values of x. For small `x`, `log1p(x)` is approximately `x`, whereas `log(1+x)` can be zero.

See Also `log`, `expm1`

Purpose

Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

Syntax

```
Y = log2(X)
[F,E] = log2(X)
```

Description

`Y = log2(X)` computes the base 2 logarithm of the elements of `X`.

`[F,E] = log2(X)` returns arrays `F` and `E`. Argument `F` is an array of real values, usually in the range `0.5 <= abs(F) < 1`. For real `X`, `F` satisfies the equation: `X = F.*2.^E`. Argument `E` is an array of integers that, for real `X`, satisfy the equation: `X = F.*2.^E`.

Remarks

This function corresponds to the ANSI C function `frexp()` and the IEEE floating-point standard function `logb()`. Any zeros in `X` produce `F = 0` and `E = 0`.

Examples

For IEEE arithmetic, the statement `[F,E] = log2(X)` yields the values:

X	F	E
`1`	`1/2`	`1`
`pi`	`pi/4`	`2`
`-3`	`-3/4`	`2`
`eps`	`1/2`	`-51`
`realmax`	`1-eps/2`	`1024`
`realmin`	`1/2`	`-1021`

See Also

`log`, `pow2`

Purpose

Convert numeric values to logical

Syntax

```
K = logical(A)
```

Description

`K = logical(A)` returns an array that can be used for logical indexing or logical tests.

`A(B)`, where `B` is a logical array that is the same size as `A`, returns the values of `A` at the indices where the real part of `B` is nonzero.

`A(B)`, where `B` is a logical array that is smaller than `A`, returns the values of column vector `A(:)` at the indices where the real part of column vector `B(:)` is nonzero.

Remarks

Most arithmetic operations remove the logicalness from an array. For example, adding zero to a logical array removes its logical characteristic. `A = +A` is the easiest way to convert a logical array, `A`, to a numeric double array.

Logical arrays are also created by the relational operators (==,<,>,~, etc.) and functions like `any`, `all`, `isnan`, `isinf`, and `isfinite`.

Examples

Given `A = [1 2 3; 4 5 6; 7 8 9]`, the statement `B = logical(eye(3))` returns a logical array

```
B =
   1   0   0
   0   1   0
   0   0   1
```

which can be used in logical indexing that returns `A`'s diagonal elements:

```
A(B)

ans =
     1
     5
     9
```

However, attempting to index into `A` using the *numeric* array `eye(3)` results in:

```
A(eye(3))
??? Subscript indices must either be real positive integers or
logicals.
```

See Also `true`, `false`, `islogical`, logical operators (elementwise and short-circuit),

loglog

Purpose

Log-log scale plot

GUI Alternatives

To graph selected variables, use the Plot Selector 

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

Syntax

```
loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,LineSpec,...)
loglog(...,'PropertyName',PropertyValue,...)
h = loglog(...)
hlines = loglog('v6',...)
```

Description

`loglog(Y)` plots the columns of `Y` versus their index if `Y` contains real numbers. If `Y` contains complex numbers, `loglog(Y)` and `loglog(real(Y),imag(Y))` are equivalent. `loglog` ignores the imaginary component in all other uses of this function.

`loglog(X1,Y1,...)` plots all `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `loglog` plots the vector argument versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`loglog(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples, where `LineSpec` determines line type, marker symbol, and color of the plotted lines. You can mix `Xn,Yn,LineSpec` triples with `Xn,Yn` pairs, for example,

```
loglog(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

`loglog(...,'`*`PropertyName`*`',PropertyValue,...)` sets property values for all lineseries graphics objects created by `loglog`. See the `line` reference page for more information.

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

Backward-Compatible Version

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

Note The `v6` option enables users of MATLAB Version 7.x to create FIG-files that previous versions can open. It is obsolete and will be removed in a future MATLAB version.

See Plot Objects and Backward Compatibility for more information.

Remarks

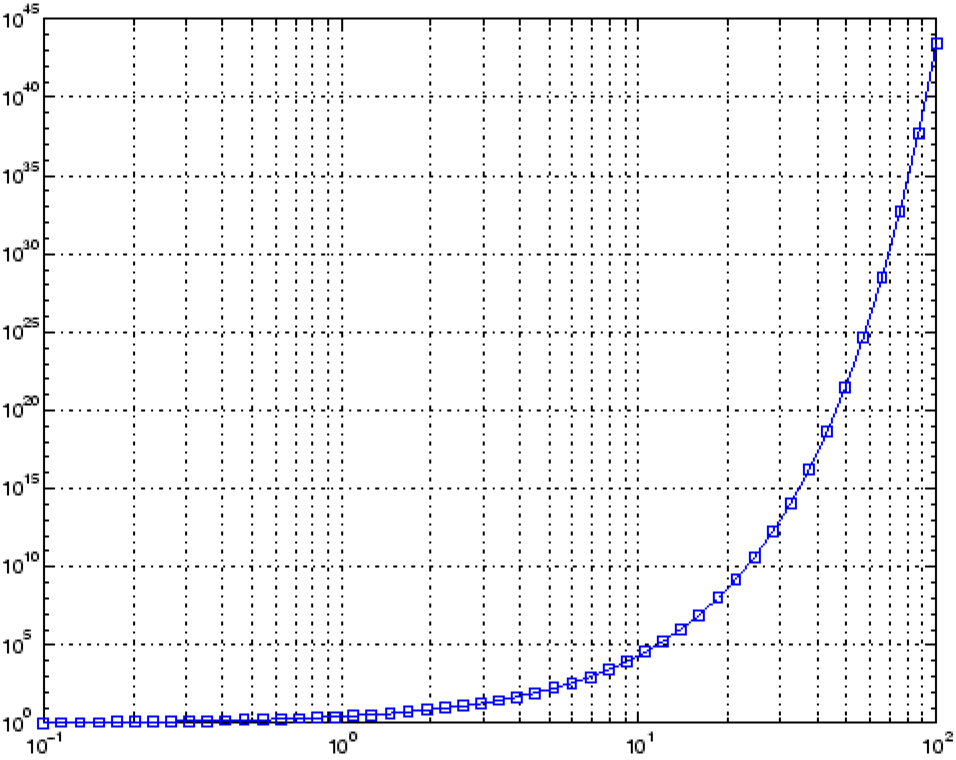
If you do not specify a color when plotting more than one line, `loglog` automatically cycles through the colors and line styles in the order specified by the current axes.

If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode will remain as it is and the new data will plot as linear.

Examples

Create a simple `loglog` plot with square markers.

```
x = logspace(-1,2);
loglog(x,exp(x),'-s')
grid on
```



See Also

`LineSpec`, `plot`, `semilogx`, `semilogy`

"Basic Plots and Graphs" on page 1-91 for related functions

Purpose

Matrix logarithm

Syntax

```
L = logm(A)
[L, exitflag] = logm(A)
```

Description

`L = logm(A)` is the principal matrix logarithm of A, the inverse of `expm(A)`. `L` is the unique logarithm for which every eigenvalue has imaginary part lying strictly between -π and π. If `A` is singular or has any eigenvalues on the negative real axis, the principal logarithm is undefined. In this case, `logm` computes a non-principal logarithm and returns a warning message.

`[L, exitflag] = logm(A)` returns a scalar `exitflag` that describes the exit condition of `logm`:

- If `exitflag = 0`, the algorithm was successfully completed.
- If `exitflag = 1`, too many matrix square roots had to be computed. However, the computed value of `L` might still be accurate. This is different from R13 and earlier versions that returned an expensive and often inaccurate error estimate as the second output argument.

The input `A` can have class `double` or `single`.

Remarks

If `A` is real symmetric or complex Hermitian, then so is `logm(A)`.

Some matrices, like `A = [0 1; 0 0]`, do not have any logarithms, real or complex, so `logm` cannot be expected to produce one.

Limitations

For most matrices:

```
logm(expm(A)) = A = expm(logm(A))
```

These identities may fail for some `A`. For example, if the computed eigenvalues of `A` include an exact zero, then `logm(A)` generates infinity. Or, if the elements of `A` are too large, `expm(A)` may overflow.

Examples

Suppose `A` is the 3-by-3 matrix

```
        1         1         0
        0         0         2
        0         0        -1
```

and `Y = expm(A)` is

```
Y =
    2.7183    1.7183    1.0862
         0    1.0000    1.2642
         0         0    0.3679
```

Then `A = logm(Y)` produces the original matrix `A`.

```
Y =
    1.0000    1.0000    0.0000
         0         0    2.0000
         0         0   -1.0000
```

But `log(A)` involves taking the logarithm of zero, and so produces

```
ans =
    0.0000                   0           -35.5119
     -Inf                 -Inf             0.6931
     -Inf                 -Inf             0.0000 + 3.1416i
```

Algorithm

The algorithm `logm` uses is described in [1].

See Also

`expm`, `funm`, `sqrtm`

References

[1] Davies, P. I. and N. J. Higham, “A Schur-Parlett algorithm for computing matrix functions,” *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Cheng, S. H., N. J. Higham, C. S. Kenney, and A. J. Laub, “Approximating the logarithm of a matrix to specified accuracy,” *SIAM J. Matrix Anal. Appl.*, Vol. 22, Number 4, pp. 1112-1125, 2001.

[3] Higham, N. J., “Evaluating Pade approximants of the matrix logarithm,” *SIAM J. Matrix Anal. Appl.*, Vol. 22, Number 4, pp. 1126-1135, 2001.

[4] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Johns Hopkins University Press, 1983, p. 384.

[5] Moler, C. B. and C. F. Van Loan, “Nineteen Dubious Ways to Compute the Exponential of a Matrix,” *SIAM Review* 20, 1978, pp. 801-836.

logspace

Purpose Generate logarithmically spaced vectors

Syntax

```
y = logspace(a,b)
y = logspace(a,b,n)
y = logspace(a,pi)
```

Description The `logspace` function generates logarithmically spaced vectors. Especially useful for creating frequency vectors, it is a logarithmic equivalent of `linspace` and the ":" or colon operator.

`y = logspace(a,b)` generates a row vector `y` of 50 logarithmically spaced points between decades `10^a` and `10^b`.

`y = logspace(a,b,n)` generates `n` points between decades `10^a` and `10^b`.

`y = logspace(a,pi)` generates the points between `10^a` and `pi`, which is useful for digital signal processing where frequencies over this interval go around the unit circle.

Remarks All the arguments to `logspace` must be scalars.

See Also `linspace`

The colon operator `:`

Purpose

Search for keyword in all help entries

GUI Alternatives

As an alternative to the `lookfor` function, use the Function Browser.

Syntax

```
lookfor topic
lookfor topic -all
```

Description

`lookfor topic` searches for the string `topic` in the first comment line (the H1 line) of the help text in all M-files found on the MATLAB search path. For all files in which a match occurs, `lookfor` displays the H1 line.

`lookfor topic -all` searches the entire first comment block of an M-file looking for `topic`.

Examples

For example:

```
lookfor inverse
```

finds at least a dozen matches, including H1 lines containing "inverse hyperbolic cosine," "two-dimensional inverse FFT," and "pseudoinverse." Contrast this with

```
which inverse
```

or

```
what inverse
```

These functions run more quickly, but probably fail to find anything because MATLAB does not have a function `inverse`.

In summary, `what` lists the functions in a given folder, `which` finds the folder containing a given function or file, and `lookfor` finds all functions in all folders that might have something to do with a given keyword.

Even more extensive than the `lookfor` function are the find features in the Current Folder browser. For example, you can look for all occurrences of a specified word in all the M-files in the current folder. For more information, see

See Also

`dir`, `doc`, `filebrowser`, `findstr`, `help`, `helpdesk`, `helpwin`, `regexp`, `what`, `which`, `who`

Topics in the User Guide:

-
-
-

Purpose

Convert string to lowercase

Syntax

```
t = lower('str')
B = lower(A)
```

Description

`t = lower('str')` returns the string formed by converting any uppercase characters in `str` to the corresponding lowercase characters and leaving all other characters unchanged.

`B = lower(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `lower` to each string within `A`.

Examples

`lower('MathWorks')` is `mathworks`.

Remarks

Character sets supported:

- PC: Latin-1 for the Microsoft Windows operating system
- Other: ISO Latin-1 (ISO 8859-1)

See Also

`upper`

Purpose Folder contents

Syntax `ls`

Graphical Interface As an alternative to the `ls` function, use the Current Folder browser.

Description `ls` lists the contents of the current folder. On UNIX[11] platforms, `ls` returns a character row vector of names separated by tab and space characters. On Microsoft Windows platforms, `ls` returns an `m`-by-`n` character array of names. `m` is the number of names and `n` is the number of characters in the longest name found. Names shorter than `n` characters are padded with space characters.

On UNIX platforms, you add any flags to `ls` that the operating system supports.

See Also `dir`, `pwd`

Topics in User Guide:

-
-

---

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

Purpose

Least-squares solution in presence of known covariance

Syntax

```
x = lscov(A,b)
x = lscov(A,b,w)
x = lscov(A,b,V)
x = lscov(A,b,V,alg)
[x,stdx] = lscov(...)
[x,stdx,mse] = lscov(...)
[x,stdx,mse,S] = lscov(...)
```

Description

`x = lscov(A,b)` returns the ordinary least squares solution to the linear system of equations `A*x = b`, i.e., `x` is the n-by-1 vector that minimizes the sum of squared errors `(b - A*x)'*(b - A*x)`, where `A` is m-by-n, and `b` is m-by-1. `b` can also be an m-by-k matrix, and `lscov` returns one solution for each column of `b`. When `rank(A) < n`, `lscov` sets the maximum possible number of elements of `x` to zero to obtain a "basic solution".

`x = lscov(A,b,w)`, where `w` is a vector length m of real positive weights, returns the weighted least squares solution to the linear system `A*x = b`, that is, `x` minimizes `(b - A*x)'*diag(w)*(b - A*x)`. `w` typically contains either counts or inverse variances.

`x = lscov(A,b,V)`, where `V` is an m-by-m real symmetric positive definite matrix, returns the generalized least squares solution to the linear system `A*x = b` with covariance matrix proportional to `V`, that is, `x` minimizes `(b - A*x)'*inv(V)*(b - A*x)`.

More generally, `V` can be positive semidefinite, and `lscov` returns `x` that minimizes `e'*e`, subject to `A*x + T*e = b`, where the minimization is over `x` and `e`, and `T*T' = V`. When `V` is semidefinite, this problem has a solution only if `b` is consistent with `A` and `V` (that is, `b` is in the column space of `[A T]`), otherwise `lscov` returns an error.

By default, `lscov` computes the Cholesky decomposition of `V` and, in effect, inverts that factor to transform the problem into ordinary least squares. However, if `lscov` determines that `V` is semidefinite, it uses an orthogonal decomposition algorithm that avoids inverting `V`.

`x = lscov(A,b,V,alg)` specifies the algorithm used to compute `x` when `V` is a matrix. `alg` can have the following values:

- `'chol'` uses the Cholesky decomposition of `V`.
- `'orth'` uses orthogonal decompositions, and is more appropriate when `V` is ill-conditioned or singular, but is computationally more expensive.

`[x,stdx] = lscov(...)` returns the estimated standard errors of `x`. When `A` is rank deficient, `stdx` contains zeros in the elements corresponding to the necessarily zero elements of `x`.

`[x,stdx,mse] = lscov(...)` returns the mean squared error.

`[x,stdx,mse,S] = lscov(...)` returns the estimated covariance matrix of `x`. When `A` is rank deficient, `S` contains zeros in the rows and columns corresponding to the necessarily zero elements of `x`. `lscov` cannot return `S` if it is called with multiple right-hand sides, that is, if `size(B,2) > 1`.

The standard formulas for these quantities, when `A` and `V` are full rank, are

- `x = inv(A'*inv(V)*A)*A'*inv(V)*B`
- `mse = B'*(inv(V) - inv(V)*A*inv(A'*inv(V)*A)*A'*inv(V))*B./(m-n)`
- `S = inv(A'*inv(V)*A)*mse`
- `stdx = sqrt(diag(S))`

However, `lscov` uses methods that are faster and more stable, and are applicable to rank deficient cases.

`lscov` assumes that the covariance matrix of `B` is known only up to a scale factor. `mse` is an estimate of that unknown scale factor, and `lscov` scales the outputs `S` and `stdx` appropriately. However, if `V` is known to be exactly the covariance matrix of `B`, then that scaling is unnecessary.

To get the appropriate estimates in this case, you should rescale `S` and `stdx` by `1/mse` and `sqrt(1/mse)`, respectively.

Algorithm

The vector `x` minimizes the quantity `(A*x-b)'*inv(V)*(A*x-b)`. The classical linear algebra solution to this problem is

```
x = inv(A'*inv(V)*A)*A'*inv(V)*b
```

but the `lscov` function instead computes the QR decomposition of `A` and then modifies `Q` by `V`.

Examples

Example 1 — Computing Ordinary Least Squares

The MATLAB backslash operator (\) enables you to perform linear regression by computing ordinary least-squares (OLS) estimates of the regression coefficients. You can also use `lscov` to compute the same OLS estimates. By using `lscov`, you can also compute estimates of the standard errors for those coefficients, and an estimate of the standard deviation of the regression error term:

```
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
X = [ones(size(x1)) x1 x2];
y = [.17 .26 .28 .23 .27 .34]';

a = X\y
a =
    0.1203
    0.3284
   -0.1312

[b,se_b,mse] = lscov(X,y)
b =
    0.1203
    0.3284
   -0.1312
se_b =
    0.0643
```

```
    0.2267
    0.1488
mse =
    0.0015
```

Example 2 — Computing Weighted Least Squares

Use `lscov` to compute a weighted least-squares (WLS) fit by providing a vector of relative observation weights. For example, you might want to downweight the influence of an unreliable observation on the fit:

```
w = [1 1 1 1 1 .1]';

[bw,sew_b,msew] = lscov(X,y,w)
bw =
    0.1046
    0.4614
   -0.2621
sew_b =
    0.0309
    0.1152
    0.0814
msew =
  3.4741e-004
```

Example 3 — Computing General Least Squares

Use `lscov` to compute a general least-squares (GLS) fit by providing an observation covariance matrix. For example, your data may not be independent:

```
V = .2*ones(length(x1)) + .8*diag(ones(size(x1)));

[bg,sew_b,mseg] = lscov(X,y,V)
bg =
    0.1203
    0.3284
   -0.1312
sew_b =
```

```
    0.0672
    0.2267
    0.1488
mseg =
    0.0019
```

Example 4 — Estimating the Coefficient Covariance Matrix

Compute an estimate of the coefficient covariance matrix for either OLS, WLS, or GLS fits. The coefficient standard errors are equal to the square roots of the values on the diagonal of this covariance matrix:

```
[b,se_b,mse,S] = lscov(X,y);

S
S =
    0.0041   -0.0130    0.0075
   -0.0130    0.0514   -0.0328
    0.0075   -0.0328    0.0221

[se_b sqrt(diag(S))]
ans =
    0.0643    0.0643
    0.2267    0.2267
    0.1488    0.1488
```

See Also

`lsqnonneg`, `qr`

The arithmetic operator \

Reference

[1] Strang, G., *Introduction to Applied Mathematics*, Wellesley-Cambridge, 1986, p. 398.

Purpose

Solve nonnegative least-squares constraints problem

Syntax

```
x = lsqnonneg(C,d)
x = lsqnonneg(C,d,x0)
x = lsqnonneg(C,d,x0,options)
[x,resnorm] = lsqnonneg(...)
[x,resnorm,residual] = lsqnonneg(...)
[x,resnorm,residual,exitflag] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)
```

Description

`x = lsqnonneg(C,d)` returns the vector `x` that minimizes `norm(C*x-d)` subject to `x >= 0`. `C` and `d` must be real.

`x = lsqnonneg(C,d,x0)` uses `x0` as the starting point if all `x0 >= 0`; otherwise, the default is used. The default start point is the origin (the default is used when `x0==[ ]` or when only two input arguments are provided).

`x = lsqnonneg(C,d,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `lsqnonneg` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`TolX`	Termination tolerance on `x`.

`[x,resnorm] = lsqnonneg(...)` returns the value of the squared 2-norm of the residual: `norm(C*x-d)^2`.

`[x,resnorm,residual] = lsqnonneg(...)` returns the residual, `d-C*x`.

`[x,resnorm,residual,exitflag] = lsqnonneg(...)` returns a value `exitflag` that describes the exit condition of `lsqnonneg`:

>0	Indicates that the function converged to a solution `x`.
0	Indicates that the iteration count was exceeded. Increasing the tolerance (`TolX` parameter in `options`) may lead to a solution.

`[x,resnorm,residual,exitflag,output] = lsqnonneg(...)` returns a structure `output` that contains information about the operation in the following fields:

`algorithm`	The algorithm used
`iterations`	The number of iterations taken
`message`	Exit message

`[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)` returns the dual vector (Lagrange multipliers) `lambda`, where `lambda(i)<=0` when `x(i)` is (approximately) `0`, and `lambda(i)` is (approximately) `0` when `x(i)>0`.

Examples

Compare the unconstrained least squares solution to the `lsqnonneg` solution for a 4-by-2 problem:

```
C = [
    0.0372    0.2869
    0.6861    0.7071
    0.6233    0.6245
    0.6344    0.6170];
d = [
    0.8587
    0.1781
    0.0747
    0.8405];
[C\d lsqnonneg(C,d)] =
    -2.5627         0
     3.1108    0.6929
[norm(C*(C\d)-d) norm(C*lsqnonneg(C,d)-d)] =
     0.6674 0.9118
```

The solution from `lsqnonneg` does not fit as well (has a larger residual), as the least squares solution. However, the nonnegative least squares solution has no negative components.

Algorithm

`lsqnonneg` uses the algorithm described in [1]. The algorithm starts with a set of possible basis vectors and computes the associated dual vector `lambda`. It then selects the basis vector corresponding to the maximum value in `lambda` in order to swap out of the basis in exchange for another possible candidate. This continues until `lambda <= 0`.

See Also

The arithmetic operator `\`, `optimset`

References

[1] Lawson, C.L. and R.J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, 1974, Chapter 23, p. 161.

Purpose

LSQR method

Syntax

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

Description

`x = lsqr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x` if `A` is consistent, otherwise it attempts to solve the least squares solution `x` that minimizes `norm(b-A*x)`. The `m`-by-`n` coefficient matrix `A` need not be square but it should be large and sparse. The column vector `b` must have length `m`. `A` can be a function handle `afun` such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`. See 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 `lsqr` converges, a message to that effect is displayed. If `lsqr` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

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

`lsqr(A,b,tol,maxit)` specifies the maximum number of iterations.

`lsqr(A,b,tol,maxit,M)` and `lsqr(A,b,tol,maxit,M1,M2)` use `n`-by-`n` preconditioner `M` or `M = M1*M2` and effectively solve the system `A*inv(M)*y = b` for `y`, where `y = M*x`. If `M` is `[]` then `lsqr` applies no preconditioner. `M` can be a function `mfun` such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`lsqr(A,b,tol,maxit,M1,M2,x0)` specifies the `n`-by-`1` initial guess. If `x0` is `[]`, then `lsqr` uses the default, an all zero vector.

`[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a convergence flag.

Flag	Convergence
0	`lsqr` converged to the desired tolerance `tol` within `maxit` iterations.
1	`lsqr` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`lsqr` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `lsqr` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if you specify the `flag` output.

`[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns an estimate of the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of the residual norm estimates at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of estimates of the scaled normal equations residual at each iteration: `norm((A*inv(M))'*(B-A*X))/norm(A*inv(M),'fro')`. Note that the estimate of `norm(A*inv(M),'fro')` changes, and hopefully improves, at each iteration.

Examples

Example 1

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = lsqr(A,b,tol,maxit,M1,M2);
```

displays the following message:

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in an M-file `run_lsqr` that

- Calls `lsqr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_lsqr` are available to `afun`.

The following shows the code for `run_lsqr`:

```
function x1 = run_lsqr
```

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = lsqr(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
          y(2:n) = y(2:n) - x(1:n-1);
       elseif strcmp(transp_flag,'notransp') % y = A*x
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - x(2:n);
       end
    end
end
```

When you enter

```
x1=run_lsqr;
```

MATLAB software displays the message

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

See Also

`bicg`, `bicgstab`, `cgs`, `gmres`, `minres`, `norm`, `pcg`, `qmr`, `symmlq`, `function_handle` (`@`)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "LSQR: An Algorithm for Sparse Linear Equations And Sparse Least Squares," *ACM Trans. Math. Soft.*, Vol.8, 1982, pp. 43-71.

Purpose

Test for less than

Syntax

```
A < B
lt(A, B)
```

Description

`A < B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is less than `B`, or set to logical 0 (`false`) where `A` is greater than or equal to `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLABsoftware returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`lt(A, B)` is called for the syntax `A < B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are less than the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A < B
ans =
     0     1     1     0     0     0
     1     0     1     0     0     0
     0     1     1     0     0     0
     1     0     0     1     0     1
```

```
     0     1     0     0     1     1
     1     0     0     0     1     0
```

See Also `gt`, `le`, `ge`, `ne`, `eq`, in the MATLAB Programming documentation

Purpose LU matrix factorization

Syntax

```
Y = lu(A)
[L,U] = lu(A)
[L,U,P] = lu(A)
[L,U,P,Q] = lu(A)
[L,U,P,Q,R] = lu(A)
[...] = lu(A,'vector')
[...] = lu(A,thresh)
[...] = lu(A,thresh,'vector')
```

Description The `lu` function expresses a matrix `A` as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The factorization is often called the *LU*, or sometimes the *LR*, factorization. `A` can be rectangular. For a full matrix `A`, `lu` uses the Linear Algebra Package (LAPACK) routines described in "Algorithm" on page 2-2246.

`Y = lu(A)` returns matrix `Y` that, for sparse `A`, contains the strictly lower triangular `L`, i.e., without its unit diagonal, and the upper triangular `U` as submatrices. That is, if `[L,U,P] = lu(A)`, then `Y = U+L-eye(size(A))`. For nonsparse `A`, `Y` is the output from the LAPACK `dgetrf` or `zgetrf` routine. The permutation matrix `P` is not returned.

`[L,U] = lu(A)` returns an upper triangular matrix in `U` and a permuted lower triangular matrix in `L` such that `A = L*U`. Return value `L` is a product of lower triangular and permutation matrices.

`[L,U,P] = lu(A)` returns an upper triangular matrix in `U`, a lower triangular matrix `L` with a unit diagonal, and a permutation matrix `P`, such that `L*U = P*A`. The statement `lu(A,'matrix')` returns identical output values.

`[L,U,P,Q] = lu(A)` for sparse nonempty `A`, returns a unit lower triangular matrix `L`, an upper triangular matrix `U`, a row permutation matrix `P`, and a column reordering matrix `Q`, so that `P*A*Q = L*U`. This syntax uses UMFPACK and is significantly more time and memory efficient than the other syntaxes, even when used with `colamd`. If `A` is empty or not sparse, `lu` displays an error message. The statement `lu(A,'matrix')` returns identical output values.

`[L,U,P,Q,R] = lu(A)` returns unit lower triangular matrix `L`, upper triangular matrix `U`, permutation matrices `P` and `Q`, and a diagonal scaling matrix `R` so that `P*(R\A)*Q = L*U` for sparse non-empty `A`. This uses UMFPACK as well. Typically, but not always, the row-scaling leads to a sparser and more stable factorization. Note that this factorization is the same as that used by sparse `mldivide` when UMFPACK is used. The statement `lu(A,'matrix')` returns identical output values.

`[...] = lu(A,'vector')` returns the permutation information in two row vectors `p` and `q`. You can specify from 1 to 5 outputs. Output `p` is defined as `A(p,:)=L*U`, output `q` is defined as `A(p,q)=L*U`, and output `R` is defined as `R(:,p)\A(:,q)=L*U`.

`[...] = lu(A,thresh)` controls pivoting in UMFPACK. This syntax applies to sparse matrices only. The `thresh` input is a one- or two-element vector of type `single` or `double` that defaults to [0.1, 0.001]. If `A` is a square matrix with a mostly symmetric structure and mostly nonzero diagonal, UMFPACK uses a symmetric pivoting strategy. For this strategy, the diagonal where

```
A(i,j) >= thresh(2) * max(abs(A(j:m,j)))
```

is selected. If the diagonal entry fails this test, a pivot entry below the diagonal is selected, using `thresh(1)`. In this case, `L` has entries with absolute value `1/min(thresh)` or less.

If A is not as described above, UMFPACK uses an asymmetric strategy. In this case, the sparsest row `i` where

```
A(i,j) >= thresh(1) * max(abs(A(j:m,j)))
```

is selected. A value of 1.0 results in conventional partial pivoting. Entries in `L` have an absolute value of `1/thresh(1)` or less. The second element of the `thresh` input vector is not used when UMFPACK uses an asymmetric strategy.

Smaller values of `thresh(1)` and `thresh(2)` tend to lead to sparser LU factors, but the solution can become inaccurate. Larger values can lead to a more accurate solution (but not always), and usually an increase in the total work and memory usage. The statement `lu(A,thresh,'matrix')` returns identical output values.

`[...]  = lu(A,thresh,'vector')` controls the pivoting strategy and also returns the permutation information in row vectors, as described above. The `thresh` input must precede `'vector'` in the input argument list.

Note In rare instances, incorrect factorization results in `P*A*Q` ≠ `L*U`. Increase `thresh`, to a maximum of `1.0` (regular partial pivoting), and try again.

Remarks

Most of the algorithms for computing LU factorization are variants of Gaussian elimination. The factorization is a key step in obtaining the inverse with `inv` and the determinant with `det`. It is also the basis for the linear equation solution or matrix division obtained with `\` and `/`.

Arguments

`A`	Rectangular matrix to be factored.
`thresh`	Pivot threshold for sparse matrices. Valid values are in the interval `[0,1]`. If you specify the fourth output `Q`, the default is `0.1`. Otherwise, the default is `1.0`.
`L`	Factor of `A`. Depending on the form of the function, `L` is either a unit lower triangular matrix, or else the product of a unit lower triangular matrix with `P'`.
`U`	Upper triangular matrix that is a factor of `A`.
`P`	Row permutation matrix satisfying the equation `L*U = P*A`, or `L*U = P*A*Q`. Used for numerical stability.

`Q`	Column permutation matrix satisfying the equation `P*A*Q = L*U`. Used to reduce fill-in in the sparse case.
`R`	Row-scaling matrix

Examples

Example 1

Start with

```
A = [ 1    2    3
      4    5    6
      7    8    0 ];
```

To see the LU factorization, call `lu` with two output arguments.

```
[L1,U] = lu(A)

L1 =
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
    1.0000         0         0

U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000
```

Notice that `L1` is a permutation of a lower triangular matrix: if you switch rows 2 and 3, and then switch rows 1 and 2, the resulting matrix is lower triangular and has `1`s on the diagonal. Notice also that `U` is upper triangular. To check that the factorization does its job, compute the product

```
L1*U
```

which returns the original `A`. The inverse of the example matrix, `X = inv(A)`, is actually computed from the inverses of the triangular factors

```
X = inv(U)*inv(L1)
```

Using three arguments on the left side to get the permutation matrix as well,

```
[L2,U,P] = lu(A)
```

returns a truly lower triangular `L2`, the same value of `U`, and the permutation matrix `P`.

```
L2 =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000

U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000

P =
    0    0    1
    1    0    0
    0    1    0
```

Note that `L2 = P*L1`.

```
P*L1

ans =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
```

To verify that `L2*U` is a permuted version of `A`, compute `L2*U` and subtract it from `P*A`:

```
P*A - L2*U

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

In this case, `inv(U)*inv(L)` results in the permutation of `inv(A)` given by `inv(P)*inv(A)`.

The determinant of the example matrix is

```
d = det(A)

d =    27
```

It is computed from the determinants of the triangular factors

```
d = det(L)*det(U)
```

The solution to $Ax = b$ is obtained with matrix division

```
x = A\b
```

The solution is actually computed by solving two triangular systems

```
y = L\b
x = U\y
```

Example 2

The 1-norm of their difference is within roundoff error, indicating that `L*U = P*B*Q`.

Generate a 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster-Fuller geodesic dome.

```
B = bucky;
```

Use the sparse matrix syntax with four outputs to get the row and column permutation matrices.

```
[L,U,P,Q] = lu(B);
```

Apply the permutation matrices to `B`, and subtract the product of the lower and upper triangular matrices.

```
Z = P*B*Q - L*U;
norm(Z,1)

ans =
  7.9936e-015
```

Example 3

This example illustrates the benefits of using the `'vector'` option. Note how much memory is saved by using the `lu(F,'vector')` syntax.

```
F = gallery('uniformdata',[1000 1000],0);
g = sum(F,2);
[L,U,P] = lu(F);
[L,U,p] = lu(F,'vector');
whos P p

Name           Size            Bytes  Class      Attributes
P           1000x1000        8000000  double
p              1x1000           8000  double
```

The following two statements are equivalent. The first typically requires less time:

```
x = U\(L\(g(p,:)));
y = U\(L\(P*g));
```

Algorithm

For full matrices `X`, `lu` uses the LAPACK routines listed in the following table.

	Real	**Complex**
`X double`	DGETRF	ZGETRF
`X single`	SGETRF	CGETRF

For sparse `X`, with four outputs, `lu` uses UMFPACK routines. With three or fewer outputs, `lu` uses its own sparse matrix routines.

See Also

`cond`, `det`, `inv`, `luinc`, `qr`, `rref`

The arithmetic operators `\` and `/`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

[2] Davis, T. A., *UMFPACK Version 4.6 User Guide* (`http://www.cise.ufl.edu/research/sparse/umfpack`), Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, 2002.

luinc

Purpose

Sparse incomplete LU factorization

Syntax

```
luinc(A,'0')
luinc(A,droptol)
luinc(A,options)
[L,U] = luinc(A,0)
[L,U] = luinc(A,options)
[L,U,P] = luinc(...)
```

Description

`luinc` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`luinc(A,'0')` computes the incomplete LU factorization of level `0` of a square sparse matrix. The triangular factors have the same sparsity pattern as the permutation of the original sparse matrix `A`, and their product agrees with the permuted `A` over its sparsity pattern. `luinc(A,'0')` returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost, but `nnz(luinc(A,'0')) = nnz(A)`, with the possible exception of some zeros due to cancellation.

`luinc(A,droptol)` computes the incomplete LU factorization of any sparse matrix using the drop tolerance specified by the non-negative scalar `droptol`. The result is an approximation of the complete LU factors returned by `lu(A)`. For increasingly smaller values of the drop tolerance, this approximation improves until the drop tolerance is `0`, at which time the complete LU factorization is produced, as in `lu(A)`.

As each column `j` of the triangular incomplete factors is being computed, the entries smaller in magnitude than the local drop tolerance (the product of the drop tolerance and the norm of the corresponding column of `A`)

```
droptol*norm(A(:,j))
```

are dropped from the appropriate factor.

The only exceptions to this dropping rule are the diagonal entries of the upper triangular factor, which are preserved to avoid a singular factor.

`luinc(A,options)` computes the factorization with up to four options. These options are specified by fields of the input structure `options`. The fields must be named exactly as shown in the table below. You can include any number of these fields in the structure and define them in any order. Any additional fields are ignored.

Field Name	Description
`droptol`	Drop tolerance of the incomplete factorization.
`milu`	If `milu` is `1`, `luinc` produces the modified incomplete LU factorization that subtracts the dropped elements in any column from the diagonal element of the upper triangular factor. The default value is `0`.
`udiag`	If `udiag` is `1`, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is `0`.
`thresh`	Pivot threshold between `0` (forces diagonal pivoting) and `1`, the default, which always chooses the maximum magnitude entry in the column to be the pivot. `thresh` is described in greater detail in the `lu` reference page.

`luinc(A,options)` is the same as `luinc(A,droptol)` if options has `droptol` as its only field.

`[L,U] = luinc(A,0)` returns the product of permutation matrices and a unit lower triangular matrix in `L` and an upper triangular matrix in `U`. The exact sparsity patterns of `L`, `U`, and `A` are not comparable but the number of nonzeros is maintained with the possible exception of some zeros in `L` and `U` due to cancellation:

```
nnz(L)+nnz(U) = nnz(A)+n, where A is n-by-n.
```

The product `L*U` agrees with `A` over its sparsity pattern. `(L*U).*spones(A)-A` has entries of the order of `eps`.

`[L,U] = luinc(A,options)` returns a permutation of a unit lower triangular matrix in `L` and an upper triangular matrix in `U`. The product `L*U` is an approximation to `A`. `luinc(A,options)` returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost.

`[L,U,P] = luinc(...)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`.

`[L,U,P] = luinc(A,'0')` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U` and a permutation matrix in `P`. `L` has the same sparsity pattern as the lower triangle of permuted `A`

```
spones(L) = spones(tril(P*A))
```

with the possible exceptions of `1`s on the diagonal of `L` where `P*A` may be zero, and zeros in `L` due to cancellation where `P*A` may be nonzero. `U` has the same sparsity pattern as the upper triangle of `P*A`

```
spones(U) = spones(triu(P*A))
```

with the possible exceptions of zeros in `U` due to cancellation where `P*A` may be nonzero. The product `L*U` agrees within rounding error with the permuted matrix `P*A` over its sparsity pattern. `(L*U).*spones(P*A)-P*A` has entries of the order of `eps`.

`[L,U,P] = luinc(A,options)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`. The nonzero entries of `U` satisfy

```
abs(U(i,j)) >= droptol*norm((A:,j)),
```

with the possible exception of the diagonal entries, which were retained despite not satisfying the criterion. The entries of `L` were tested against the local drop tolerance before being scaled by the pivot, so for nonzeros in `L`

```
abs(L(i,j)) >= droptol*norm(A(:,j))/U(j,j).
```

The product `L*U` is an approximation to the permuted `P*A`.

Remarks

These incomplete factorizations may be useful as preconditioners for solving large sparse systems of linear equations. The lower triangular factors all have `1`s along the main diagonal but a single `0` on the diagonal of the upper triangular factor makes it singular. The incomplete factorization with a drop tolerance prints a warning message if the upper triangular factor has zeros on the diagonal. Similarly, using the `udiag` option to replace a zero diagonal only gets rid of the symptoms of the problem but does not solve it. The preconditioner may not be singular, but it probably is not useful and a warning message is printed.

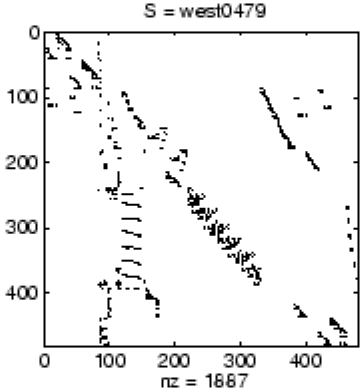
Limitations

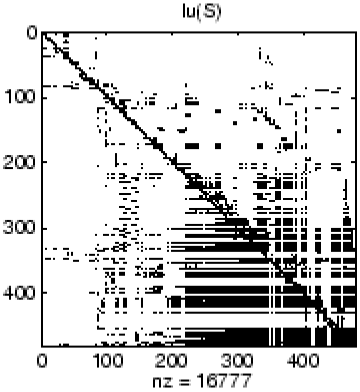
`luinc(X,'0')` works on square matrices only.

Examples

Start with a sparse matrix and compute its LU factorization.

```
load west0479;
S = west0479;
[L,U] = lu(S);
```



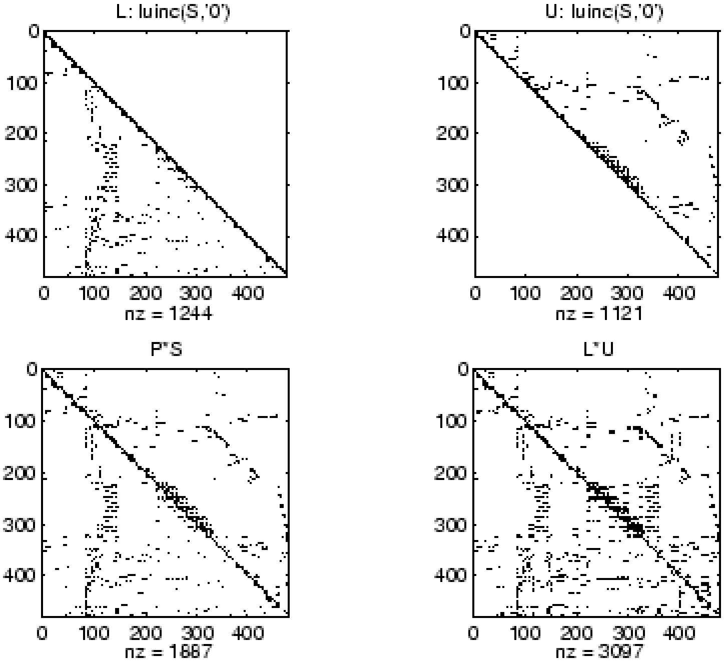


Compute the incomplete LU factorization of level 0.

```
[L,U,P] = luinc(S,'0');
D = (L*U).*spones(P*S)-P*S;
```

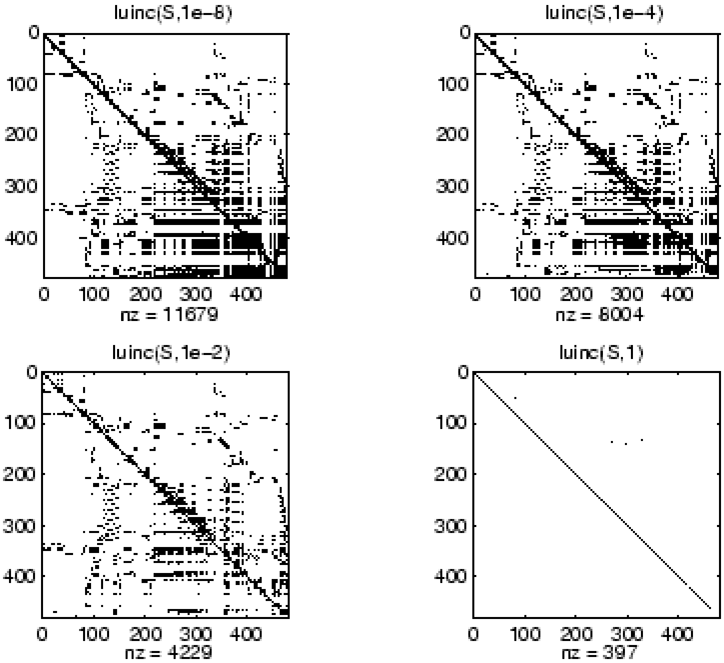
`spones(U)` and `spones(triu(P*S))` are identical.

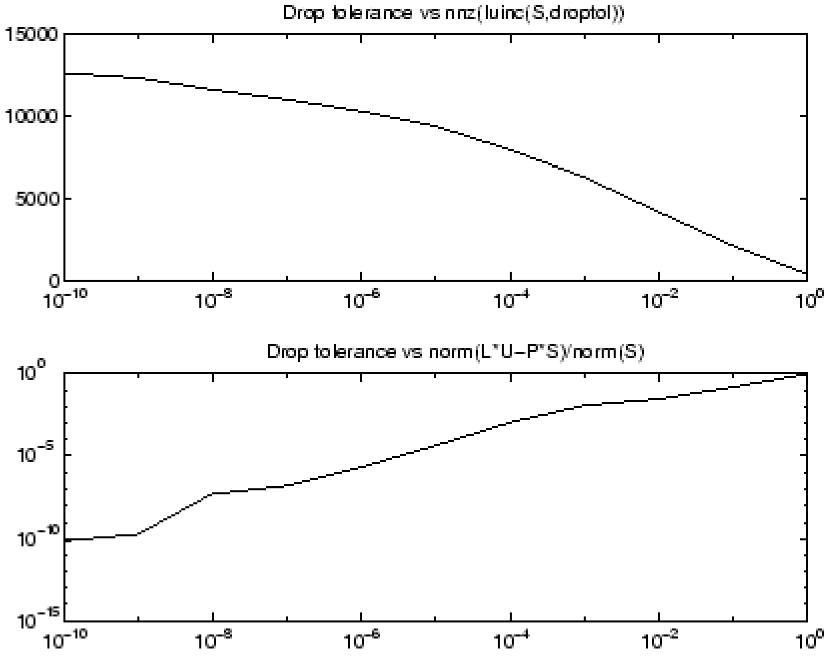
`spones(L)` and `spones(tril(P*S))` disagree at 73 places on the diagonal, where `L` is `1` and `P*S` is `0`, and also at position (206,113), where `L` is `0` due to cancellation, and `P*S` is `-1`. `D` has entries of the order of eps.



```
[IL0,IU0,IP0] = luinc(S,0);
[IL1,IU1,IP1] = luinc(S,1e-10);
                .
                .
                .
```

A drop tolerance of `0` produces the complete LU factorization. Increasing the drop tolerance increases the sparsity of the factors (decreases the number of nonzeros) but also increases the error in the factors, as seen in the plot of drop tolerance versus `norm(L*U-P*S,1)/norm(S,1)` in the second figure below.


luinc(S,1e-8)
0
100
200
300
400
0
100
200
300
400
nz = 11679
luinc(S,1e-4)
0
100
200
300
400
0
100
200
300
400
nz = 8004
luinc(S,1e-2)
0
100
200
300
400
0
100
200
300
400
nz = 4229
luinc(S,1)
0
100
200
300
400
0
100
200
300
400
nz = 397



Algorithm

`luinc(A,'0')` is based on the “KJI” variant of the LU factorization with partial pivoting. Updates are made only to positions which are nonzero in `A`.

`luinc(A,droptol)` and `luinc(A,options)` are based on the column-oriented `lu` for sparse matrices.

See Also

`bicg`, `cholinc`,`ilu`, `lu`

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

Purpose

Magic square

Syntax

`M = magic(n)`

Description

`M = magic(n)` returns an `n`-by-`n` matrix constructed from the integers `1` through `n^2` with equal row and column sums. The order n must be a scalar greater than or equal to `3`.

Remarks

A magic square, scaled by its magic sum, is doubly stochastic.

Examples

The magic square of order 3 is

```
M = magic(3)

M =

    8    1    6
    3    5    7
    4    9    2
```

This is called a magic square because the sum of the elements in each column is the same.

```
sum(M) =

    15    15    15
```

And the sum of the elements in each row, obtained by transposing twice, is the same.

```
sum(M')' =

    15
    15
    15
```

This is also a special magic square because the diagonal elements have the same sum.

```
sum(diag(M)) =

     15
```

The value of the characteristic sum for a magic square of order `n` is

```
sum(1:n^2)/n
```

which, when `n = 3`, is `15`.

Algorithm

There are three different algorithms:

- `n` odd
- `n` even but not divisible by four
- `n` divisible by four

To make this apparent, type

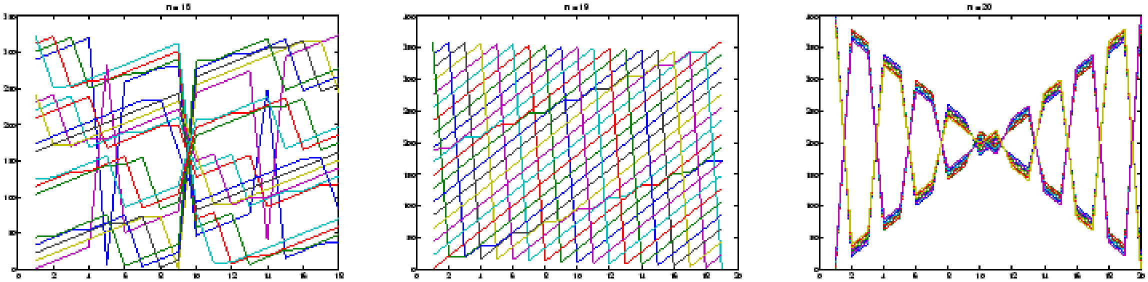
```
for n = 3:20
    A = magic(n);
    r(n) = rank(A);
end
```

For `n` odd, the rank of the magic square is `n`. For `n` divisible by 4, the rank is `3`. For `n` even but not divisible by 4, the rank is `n/2 + 2`.

```
[(3:20)',r(3:20)']
ans =
     3     3
     4     3
     5     5
     6     5
     7     7
     8     3
     9     9
    10     7
    11    11
```

```
12     3
13    13
14     9
15    15
16     3
17    17
18    11
19    19
20     3
```

Plotting A for n = 18, 19, 20 shows the characteristic plot for each category.



Limitations

If you supply `n` less than `3`, `magic` returns either a nonmagic square, or else the degenerate magic squares `1` and `[]`.

See Also

`ones`, `rand`

makehgtform

Purpose

Create 4-by-4 transform matrix

Syntax

```
M = makehgtform
M = makehgtform('translate',[tx ty tz])
M = makehgtform('scale',s)
M = makehgtform('scale',[sx,sy,sz])
M = makehgtform('xrotate',t)
M = makehgtform('yrotate',t)
M = makehgtform('zrotate',t)
M = makehgtform('axisrotate',[ax,ay,az],t)
```

Description

Use `makehgtform` to create transform matrices for translation, scaling, and rotation of graphics objects. Apply the transform to graphics objects by assigning the transform to the `Matrix` property of a parent hgtransform object. See Examples for more information.

`M = makehgtform` returns an identity transform.

`M = makehgtform('translate',[tx ty tz])` or M = makehgtform('translate',tx,ty,tz) returns a transform that translates along the *x*-axis by `tx`, along the *y*-axis by `ty`, and along the *z*-axis by `tz`.

`M = makehgtform('scale',s)` returns a transform that scales uniformly along the *x*-, *y*-, and *z*-axes.

`M = makehgtform('scale',[sx,sy,sz])` returns a transform that scales along the *x*-axis by `sx`, along the *y*-axis by `sy`, and along the *z*-axis by `sz`.

`M = makehgtform('xrotate',t)` returns a transform that rotates around the *x*-axis by `t` radians.

`M = makehgtform('yrotate',t)` returns a transform that rotates around the *y*-axis by `t` radians.

`M = makehgtform('zrotate',t)` returns a transform that rotates around the *z*-axis by `t` radians.

`M = makehgtform('axisrotate',[ax,ay,az],t)` Rotate around axis `[ax ay az]` by `t` radians.

Note that you can specify multiple operations in one call to `makehgtform` and the MATLAB software returns a transform matrix that is the result of concatenating all specified operations. For example,

```
m = makehgtform('xrotate',pi/2,'yrotate',pi/2);
```

is the same as

```
m = makehgtform('xrotate',pi/2)*makehgtform('yrotate',pi/2);
```

See Also

`hggroup`, `hgtransform`

Tomas Moller and Eric Haines, *Real-Time Rendering*, A K Peters, Ltd., 1999 for more information about transforms.

in MATLAB Graphics documentation for more information and examples.

`Hgtransform Properties` for property descriptions.

Purpose

Construct `containers.Map` object

Syntax

```
M = containers.Map
M = containers.Map(keys, values)
M = containers.Map(keys, values, 'uniformvalues', tf)
```

Description

The `Map` object is a data structure that is a container for other data. A `Map` container is similar to an array except for the means by which you index into the data stored inside the map. Instead of being restricted to the use of integer array indices 1 through N, you select elements of a `Map` container using indices of various data types, (strings, for example). A map consists of `keys` (the indices) and data `values` and is a handle object.

`M = containers.Map` constructs a `Map` container object `M` to hold data values that you can easily reference using keys that you establish. `M` is a handle object with properties `Count`, `KeyType`, and `ValueType`. These properties represent the number of keys in the map, the type of these keys, and the type of the values assigned to those keys respectively.

When you call the `containers.Map` constructor with no input arguments, MATLAB constructs an empty `Map` object, setting the `Count` , `KeyType`, and `'ValueType'` properties to `0`, `char`, and `'any'`, respectively.

`M = containers.Map(keys, values)` constructs a `Map` object `M` that contains one or more keys and a value for each of these keys, as specified in the `keys` and `values` arguments. A `value` is some unit of data that you want stored in the `Map` object, and a `key` is a unique reference to that data. Valid data types for the `keys` argument are any real-valued scalars or character arrays. This includes `char`, `double`, `int32`, `uint32`, `int64`, or `uint64`. Valid values are the same, plus the string `'any'`.

To specify multiple keys, make the `keys` argument a 1-by-`n` cell array, where `n` is the number of keys to be stored in the map. Elements of this cell array should belong to the same type. You can specify values of mixed numeric, or numeric and logical, types, without generating an error. If you do this, the `Map` constructor converts all elements to the type of the leftmost element in the `keys` cell array. To specify multiple values, use a 1-by-`n` cell array, where `n` is equal to `numel` (`keys`). There must be exactly one value for each `key` argument.

Object `M` has properties `Count`, `KeyType`, and `ValueType`. `Count` is a string that contains the number of key-value pairs in the `Map` object once the map has been constructed. `KeyType` is a character array containing the data type of the keys in the map. All keys belong to the same data type. `ValueType` is a character array containing the data type of the values in the map. If these values are of different data types, `ValueType` is set to the string `'any'`.

`M = containers.Map(keys, values, 'uniformvalues', tf)` constructs Map `M` in which all `values` are required to be of the same type when `uniformvalues` is set to logical 1 (`true`). If they are not, MATLAB throws an error. If you want to be able to store values of mixed types, set `uniformvalues` to logical 0 (`false`). This flag is only needed if you want to override the default.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

Properties

Property	Description
`Count`	Unsigned 64-bit integer that represents the total number of key-value pairs contained in the `Map` object when the initial object is constructed.
`KeyType`	Character array that indicates the data type of all keys contained in the `Map` object.
`ValueType`	Character array that indicates the data type of all values contained in the `Map` object. If not all values have the same data type, then `ValueType` is `'any'`.

Methods

Method	Description
`isKey`	Check if `containers.Map` object contains key.

Method	Description
`keys`	Return all keys of `containers.Map` object.
`size`	Return size of `containers.Map` object.
`length`	Return length of `containers.Map` object.
`values`	Return all values of `containers.Map` object in cell array.
`remove`	Remove key-value pairs from `containers.Map`.

Examples

Example 1 — Constructing a New Map Object

Construct a one-member `Map` object, `US_Capitals`:

```
US_Capitals = containers.Map('Arizona', 'Phoenix');
```

To call methods of the class, just use the method name followed by the name of the `Map` object in parentheses. Call the `keys` and `values` methods of the `US_Capitals` object you just constructed:

```
keys(US_Capitals), values(US_Capitals)
ans =
    'Arizona'
ans =
    'Phoenix'
```

Example 2 — Finding Map Properties

List the properties of the `US_Capitals` object:

```
properties(US_Capitals)

Properties for class containers.Map:
    Count
    KeyType
    ValueType
```

Examine each property:

```
US_Capitals.Count,
ans =
     1

US_Capitals.KeyType
ans =
     char

US_Capitals.ValueType
ans =
     char
```

Example 3 — Storing Multiple Keys and Values

Construct a new `Map` object and store 6 key and value parameters in it. Specify multiple keys and values by listing them in a cell array as shown here:

```
US_Capitals = containers.Map( ...
{'Arizona', 'Nebraska', 'Nevada', ...       % 6 States
 'New York', 'Georgia', 'Alaska'}, ...
{'Phoenix', 'Lincoln', 'Carson City', ...   % 6 Capitals
 'Albany', 'Atlanta', 'Juneau'})

US_Capitals =
  containers.Map handle
  Package: containers

  Properties:
        Count: 6
      KeyType: 'char'
    ValueType: 'char'
  Methods, Events, Superclasses
```

Example 4 — Displaying the Map Contents

Use the `keys` and `values` methods to see the mapping order defined within the map. The `keys` method lists all keys of the character array type in alphabetical order. The `values` method lists the value associated with each of these keys. These are listed in an order determined by their associated keys:

```
keys(US_Capitals),  values(US_Capitals)
ans =
    'Alaska'  'Arizona'  'Georgia'  'Nebraska'  'Nevada'

  'New York'
ans =
   'Juneau'  'Phoenix'  'Atlanta'  'Lincoln'  'Carson City'
   'Albany'
```

When using the `values` method, you can either list all values in the map, as shown above, or list only those values that belong to those keys you specify in the command:

```
values(US_Capitals, {'Arizona', 'New York', 'Nebraska'})
ans =
    'Phoenix'    'Albany'    'Lincoln'
```

Example 5— Adding More Keys and Values Later

Once the object has been created, store two additional keys (`Vermont` and `Oregon`), and their related values (`Montpelier` and `Salem`) in it. You do not need to call the constructor this time as you are adding to an existing `Map` object:

```
US_Capitals('Vermont') = 'Montpelier';
US_Capitals('Oregon') = 'Salem';
keys(US_Capitals)
ans =
    'Alaska'    'Arizona'    'Georgia'    'Nebraska'
    'Nevada'    'New York'    'Oregon'    'Vermont'
```

Note that the `Count` property has gone from 6 to 8:

```
US_Capitals.Count
ans =
    8
```

If you want to add more than one key-value pair at a time, you can concatenate your existing map with a new map that contains the new keys and values that you want to add. Only vertical concatenation is allowed. This example adds four more state/capital pairs to the `US_Capitals` map in just one concatenation operation. The `US_Capitals` map now contains twelve key-value pairs:

```
newSta = {'New Jersey', 'Ohio', 'Delaware', 'Montana'};
newCap = {'Trenton', 'Columbus', 'Dover', 'Helena'};
newMap = containers.Map(newSta, newCap);

% Construct a new map. Concatenate it to the existing one.
 US_Capitals = [US_Capitals; newMap]
US_Capitals =
  containers.Map handle
  Package: containers

  Properties:
        Count: 12
      KeyType: 'char'
    ValueType: 'char'
  Methods, Events, Superclasses
```

Example 6— Looking Up Values with the Map

Use the map to find the capital cities of two states in the US:

```
S1 = 'Alaska';   S2 = 'Arizona'
sprintf('\nThe capitals of %s and %s are %s and %s.', ...
        S1, S2, US_Capitals(S1), US_Capitals(S2))

ans =
The capitals of Alaska and Arizona are Juneau and Phoenix.
```

Example 7— Removing Keys and Values

To remove a key-value pair, use the `remove` method of the `Map` class, as shown here:

```
keys(US_Capitals)
ans =
  Columns 1 through 6
    'Alaska'    'Arizona'    'Delaware'    'Georgia'    'Montana'    '
  Columns 7 through 12
    'Nevada'    'New Jersey'    'New York'    'Ohio'    'Oregon'    'V

remove(US_Capitals, {'Nebraska', 'Nevada', 'New York'});

keys(US_Capitals)
ans =
  Columns 1 through 6
    'Alaska'    'Arizona'    'Delaware'    'Georgia'    'Montana'    '
  Columns 7 through 9
    'Ohio'    'Oregon'    'Vermont'
```

Removing keys and their values from the `Map` object also decrements the setting of the `Count` property.

See Also

`keys(Map)`, `values(Map)`, `size(Map)`, `length(Map)`, `isKey(Map)`, `remove(Map)`, `handle`

Purpose

Divide matrix into cell array of matrices

Syntax

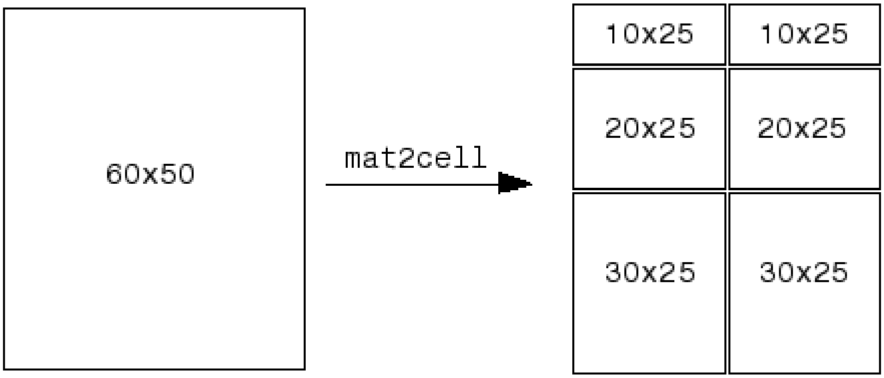
```
c = mat2cell(x, m, n)
c = mat2cell(x, d1, d2, ..., dn)
c = mat2cell(x, r)
```

Description

`c = mat2cell(x, m, n)` divides the two-dimensional matrix `x` into adjacent submatrices, each contained in a cell of the returned cell array `c`. Vectors `m` and `n` specify the number of rows and columns, respectively, to be assigned to the submatrices in `c`.

The example shown below divides a 60-by-50 matrix into six smaller matrices. The MATLAB software returns the new matrices in a 3-by-2 cell array:

```
mat2cell(x, [10 20 30], [25 25])
```



The sum of the element values in `m` must equal the total number of rows in `x`. And the sum of the element values in `n` must equal the number of columns in `x`.

The elements of `m` and `n` determine the size of each cell in `c` by satisfying the following formula for `i = 1:length(m)` and `j = 1:length(n)`:

```
size(c{i,j}) == [m(i) n(j)]
```

`c = mat2cell(x, d1, d2, ..., dn)` divides the multidimensional array `x` and returns a multidimensional cell array of adjacent submatrices of `x`. Each of the vector arguments `d1` through `dn` should sum to the respective dimension sizes of `x` such that, for `p = 1:n`,

```
size(x,p) == sum(dp)
```

The elements of `d1` through `dn` determine the size of each cell in `c` by satisfying the following formula for `ip = 1:length(dp)`:

```
size(c{i1,i2,...,in}) == [d1(i1) d2(i2) ... dn(in)]
```

If `x` is an empty array, `mat2cell` returns an empty cell array. This requires that all `dn` inputs that correspond to the zero dimensions of `x` be equal to `[]`.

For example,

```
a = rand(3,0,4);
c = mat2cell(a, [1 2], [], [2 1 1]);
```

`c = mat2cell(x, r)` divides an array `x` by returning a single-column cell array containing full rows of `x`. The sum of the element values in vector `r` must equal the number of rows of `x`.

The elements of `r` determine the size of each cell in `c`, subject to the following formula for `i = 1:length(r)`:

```
size(c{i},1) == r(i)
```

Remarks

`mat2cell` supports all array types.

Examples

Divide matrix `X` into 2-by-3 and 2-by-2 matrices contained in a cell array:

```
X = [1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15; 16 17 18 19 20]
X =
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
```

```
    16    17    18    19    20

C = mat2cell(X, [2 2], [3 2])
C =
    [2x3 double]    [2x2 double]
    [2x3 double]    [2x2 double]

C{1,1}                           C{1,2}
ans =                            ans =
     1     2     3                    4     5
     6     7     8                    9    10

C{2,1}                           C{2,2}
ans =                            ans =
    11    12    13                   14    15
    16    17    18                   19    20
```

See Also

`cell2mat`, `num2cell`

mat2str

Purpose

Convert matrix to string

Syntax

```
str = mat2str(A)
str = mat2str(A,n)
str = mat2str(A, 'class')
str = mat2str(A, n, 'class')
```

Description

`str = mat2str(A)` converts matrix `A` into a string. This string is suitable for input to the `eval` function such that `eval(str)` produces the original matrix to within 15 digits of precision.

`str = mat2str(A,n)` converts matrix `A` using `n` digits of precision.

`str = mat2str(A, 'class')` creates a string with the name of the class of `A` included. This option ensures that the result of evaluating `str` will also contain the class information.

`str = mat2str(A, n, 'class')` uses `n` digits of precision and includes the class information.

Limitations

The `mat2str` function is intended to operate on scalar, vector, or rectangular array inputs only. An error will result if `A` is a multidimensional array.

Examples

Example 1

Consider the matrix

```
x = [3.85 2.91; 7.74 8.99]
x =
    3.8500    2.9100
    7.7400    8.9900
```

The statement

```
A = mat2str(x)
```

produces

```
A =
   [3.85 2.91;7.74 8.99]
```

where `A` is a string of 21 characters, including the square brackets, spaces, and a semicolon.

`eval(mat2str(x))` reproduces `x`.

Example 2

Create a 1-by-6 matrix of signed 16-bit integers, and then use `mat2str` to convert the matrix to a 1-by-33 character array, `A`. Note that output string `A` includes the class name, `int16`:

```
x1 = int16([-300 407 213 418 32 -125]);

A = mat2str(x1, 'class')
A =
   int16([-300 407 213 418 32 -125])

class(A)
ans =
   char
```

Evaluating the string `A` gives you an output `x2` that is the same as the original `int16` matrix:

```
x2 = eval(A);

if isnumeric(x2) && isa(x2, 'int16') && all(x2 == x1)
   disp 'Conversion back to int16 worked'
end

Conversion back to int16 worked
```

See Also

`num2str`, `int2str`, `str2num`, `sprintf`, `fprintf`

material

Purpose Control reflectance properties of surfaces and patches

Syntax

```
material shiny
material dull
material metal
material([ka kd ks])
material([ka kd ks n])
material([ka kd ks n sc])
material default
```

Description `material` sets the lighting characteristics of `surface` and `patch` objects.

`material shiny` sets the reflectance properties so that the object has a high specular reflectance relative to the diffuse and ambient light, and the color of the specular light depends only on the color of the light source.

`material dull` sets the reflectance properties so that the object reflects more diffuse light and has no specular highlights, but the color of the reflected light depends only on the light source.

`material metal` sets the reflectance properties so that the object has a very high specular reflectance, very low ambient and diffuse reflectance, and the color of the reflected light depends on both the color of the light source and the color of the object.

`material([ka kd ks])` sets the ambient/diffuse/specular strength of the objects.

`material([ka kd ks n])` sets the ambient/diffuse/specular strength and specular exponent of the objects.

`material([ka kd ks n sc])` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects.

`material default` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects to their defaults.

Remarks

The `material` command sets the `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, and `SpecularColorReflectance` properties of all `surface` and `patch` objects in the axes. There must be visible `light` objects in the `axes` for lighting to be enabled. Look at the `materal.m` M-file to see the actual values set (enter the command `type material`).

See Also

`light`, `lighting`, `patch`, `surface`

Lighting as a Visualization Tool for more information on lighting

“Lighting” on page 1-106 for related functions

matlabcolon (matlab:)

Purpose Run specified function via hyperlink

Syntax

```
disp('<a href="matlab: stmnt_1; stmnt_n;">hyperlink_text</a>')
```

Description `matlab:` executes `stmnt_1` through `stmnt_n` when you click (or press **Ctrl+Enter**) in `hyperlink_text`. This must be used with another function, such as `disp`, where `disp` creates and displays underlined and colored `hyperlink_text` in the Command Window. Use `disp`, `error`, `fprintf`, `help`, or `warning` functions to display the hyperlink. The `hyperlink_text` is interpreted as HTML—you might need to use HTML character entity references or ASCII values for some special characters. Include the full hypertext string, from `'<a href=` to `</a>'` within a single line, that is, do not continue a long string on a new line. No spaces are allowed after the opening < and before the closing >. A single space is required between `a` and `href`.

Remarks The `matlab:` function behaves differently with `diary`, `notebook`, `type`, and similar functions than might be expected. For example, if you enter the following statement

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

the diary file, when viewed in a text editor, shows

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
<a href="matlab:magic(4)">Generate magic square</a>
```

If you view the output of `diary` in the Command Window, the Command Window interprets the `<a href ...>` statement and does display it as a hyperlink.

Examples

Single Function

The statement

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

displays

in the Command Window. When you click the link `Generate magic square`, the MATLAB software runs `magic(4)`.

Multiple Functions

You can include multiple functions in the statement, such as

```
disp('<a href="matlab:  x=0:1:8;y=sin(x);plot(x,y)">Plot
x,y</a>')
```

which displays

Plot x,y

in the Command Window. When you click the link, MATLAB runs

```
x = 0:1:8;
y = sin(x);
plot(x,y)
```

Clicking the Hyperlink Again

After running the statements in the hyperlink `Plot x,y` defined in the previous example, "Multiple Functions" on page 2-2275, you can subsequently redefine `x` in the base workspace, for example, as

```
x = -2*pi:pi/16:2*pi;
```

If you then click the hyperlink, `Plot x,y`, it changes the current value of `x` back to

```
0:1:8
```

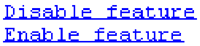
because the `matlab:` statement defines `x` in the base workspace. In the `matlab:` statement that displayed the hyperlink, `Plot x,y`, `x` was defined as 0:1:8.

Presenting Options

Use multiple `matlab:` statements in an M-file to present options, such as

```
disp('<a href = "matlab:state = 0">Disable feature</a>')
disp('<a href = "matlab:state = 1">Enable feature</a>')
```

The Command Window displays



```
Disable feature
Enable feature
```

and depending on which link is clicked, sets `state` to `0` or `1`.

Special Characters

MATLAB correctly interprets most strings that includes special characters, such as a greater than sign (>). For example, the following statement includes a `>`

```
disp('<a href="matlab:str = ''Value > 0''">Positive</a>')
```

and generates the following hyperlink.

```
Positive
```

Some symbols might not be interpreted correctly and you might need to use the HTML character entity reference for the symbol. For example, an alternative way to run the same statement is to use the `>` character entity reference instead of the > symbol:

```
disp('<a href="matlab:str = ''Value &gt; 0''">Positive</a>')
```

Instead of the HTML character entity reference, you can use the ASCII value for the symbol. For example, the greater than sign, >, is ASCII 62. The above example becomes

```
disp('<a href="matlab:str=[''Value '' char(62) '' 0'']">Positive</a>')
```

Here are some values for common special characters.

Character	**HTML Character Entity Reference**	**ASCII Value**
>	>	62
<	<	60
&	&	38
"	"	34

For a list of all HTML character entity references, see `http://www.w3.org/`.

Links from M-File Help

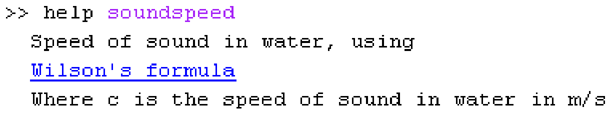
For functions you create, you can include `matlab:` links within the M-file help, but you do not need to include a `disp` or similar statement because the `help` function already includes it for displaying hyperlinks. Use the links to display additional help in a browser when the user clicks them. The M-file `soundspeed` contains the following statements:

```
function c=soundspeed(s,t,p)

% Speed of sound in water, using
% <a href="matlab: web('http://www.zu.edu')">Wilson's formula</a>
% Where c is the speed of sound in water in m/s
```

etc.

Run `help soundspeed` and MATLAB displays the following in the Command Window.



```
>> help soundspeed
  Speed of sound in water, using
  Wilson's formula
  Where c is the speed of sound in water in m/s
```

When you click the link `Wilson's formula`, MATLAB displays the HTML page `http://www.zu.edu` in the Web browser. Note that this URL is only an example and is intentionally invalid.

See Also `disp`, `error`, `fprintf`, `input`, `run`, `warning`

Purpose

Startup M-file for MATLAB program

Description

At startup time, MATLAB automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`. On multiuser or networked systems, `matlabrc.m` is reserved for use by the system manager. The file `matlabrc.m` invokes the file `startup.m` if it exists on the search path MATLAB uses.

As an individual user, you can create a startup file in your own MATLAB directory. Use the startup file to define physical constants, engineering conversion factors, graphics defaults, or anything else you want predefined in your workspace.

Algorithm

Only `matlabrc` is actually invoked by MATLAB at startup. However, `matlabrc.m` contains the statements

```
if exist('startup') == 2
    startup
end
```

that invoke `startup.m`. Extend this process to create additional startup M-files, if required.

Remarks

You can also start MATLAB using options you define at the Command Window prompt or in your Microsoft Windows shortcut for MATLAB.

Examples

Turning Off the Figure Window Toolbar

If you do not want the toolbar to appear in the figure window, remove the comment marks from the following line in the `matlabrc.m` file, or create a similar line in your own `startup.m` file.

```
%   set(0,'defaultfiguretoolbar','none')
```

See Also

`matlabroot`, `quit`, `restoredefaultpath`, `startup`

Startup Options in the MATLAB Desktop Tools and Development Environment documentation

matlabroot

Purpose Root folder

Syntax

```
matlabroot
mr = matlabroot
```

Description `matlabroot` returns the name of the folder where the MATLAB software is installed. Use `matlabroot` to create a path to MATLAB and toolbox folders that does not depend on a specific platform, MATLAB version, or installation location.

`mr = matlabroot` returns the name of the folder in which the MATLAB software is installed and assigns it to `mr`.

Remarks

matlabroot as Folder Name

The term *`matlabroot`* also refers to the folder where MATLAB files are installed. For example, "save to *`matlabroot`*`/toolbox/local`" means save to the `toolbox/local` folder in the MATLAB root folder.

Using $matlabroot as a Literal

In some files, such as `info.xml` and `classpath.txt`, `$matlabroot` is literal. In those files, MATLAB interprets `$matlabroot` as the full path to the MATLAB root folder. For example, including the line

```
$matlabroot/toolbox/local/myfile.jar
```

in `classpath.txt`, adds `myfile.jar`, which is located in the `toolbox/local` folder, to `classpath.txt`.

Sometimes, particularly in older code examples, the term `$matlabroot` or `$MATLABROOT` is not meant to be interpreted literally but is used to represent the value returned by the `matlabroot` function.

matlabroot on Macintosh Platforms

In R2008b (V7,7) and more recent versions, running `matlabroot` on Apple Macintosh platforms returns

```
/Applications/MATLAB_R2008b.app
```

In versions prior to R2008b (V7.7), such as R2008a (V7.6), running `matlabroot` on Macintosh platforms returns, for example

```
/Applications/MATLAB_R2008a
```

When you use GUIs on Macintosh platforms, you cannot directly view the contents of the MATLAB root folder. For more information, see .

Use in Compiled Mode

To return the path to the executable in compiled mode, use the MATLAB® Compiler™ `ctfroot` function, or the MATLAB `toolboxdir` function. For details, see the MATLAB or MATLAB Compiler documentation.

Examples

Get the location where MATLAB is installed:

```
matlabroot
```

MATLAB returns:

```
C:\Program Files\MATLAB\R2009a
```

Produce a full path to the `toolbox/matlab/general` folder that is correct for the platform on which it is executed:

```
fullfile(matlabroot,'toolbox','matlab','general')
```

Change the current folder to the MATLAB root folder:

```
cd(matlabroot)
```

To add the folder `myfiles` to the MATLAB search path, run

```
addpath([matlabroot '/toolbox/local/myfiles'])
```

See Also

`ctfroot` (in MATLAB Compiler product), `fullfile`, `path`, `toolboxdir`

matlab (UNIX)

Purpose Start MATLAB program (UNIX platforms)

Syntax

```
matlab helpOption
matlab envDispOption
matlab archOption
matlab dispOption
matlab modeOption
matlab -c licensefile
matlab -r matlab_command
matlab -logfile filename
matlab -mwvisual visualid
matlab -nosplash
matlab -singleCompThread
matlab -debug
matlab -Ddebugger options
```

Note You can enter more than one of these options in the same `matlab` command. If you use `-Ddebugger` to start MATLAB in debug mode, the first option in the command must be `-Ddebugger`.

Description

`matlab` is a Bourne shell script that starts the MATLAB executable on UNIX[12] platforms. (In this document, `matlab` refers to this script; MATLAB refers to the application program). Before actually initiating the execution of MATLAB, this script configures the run-time environment by

- Determining the MATLAB root directory
- Determining the host machine architecture
- Processing any command line options
- Reading the MATLAB startup file, `.matlab7rc.sh`

---

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

- Setting MATLAB environment variables

There are two ways in which you can control the way the `matlab` script works:

- By specifying command line options
- By assigning values in the MATLAB startup file, `.matlab7rc.sh`

Specifying Options at the Command Line

Options that you can enter at the command line are as follows:

`matlab helpOption` displays information that matches the specified `helpOption` argument without starting MATLAB. `helpOption` can be any one of the keywords shown in the table below. Enter only one `helpOption` keyword in a `matlab` command.

Values for helpOption

Option	Description
`-help`	Display `matlab` command usage.
`-h`	The same as **`-help`**.

`matlab envDispOption` displays the values of environment variables passed to MATLAB or their values just prior to exiting MATLAB. `envDispOption` can be either one of the options shown in the table below.

Values for envDispOption

Option	Description
`-n`	Display all the final values of the environment variables and arguments passed to the MATLAB executable as well as other diagnostic information. Does not start MATLAB.
`-e`	Display *all* environment variables and their values just prior to exiting. This argument must have been parsed before exiting for anything to be displayed. The last possible exiting point is just before the MATLAB image would have been executed and a status of `0` is returned. If the exit status is not `0` on return, then the variables and values may not be correct. Does not start MATLAB.

`matlab archOption` starts MATLAB and assumes that you are running on the system architecture specified by `arch`, or using the MATLAB version specified by `variant`, or both. The values for the `archOption` argument are shown in the table below. Enter only one of these options in a `matlab` command.

Values for archOption

Option	Description
`-arch`	Run MATLAB assuming this architecture rather than the actual architecture of the machine you are using. Replace the term `arch` with a string representing a recognized system architecture.

Values for archOption (Continued)

Option	Description
`v=variant`	Execute the version of MATLAB found in the directory `bin/$ARCH/variant` instead of `bin/$ARCH`. Replace the term `variant` with a string representing a MATLAB version.
`v=arch/variant`	Execute the version of MATLAB found in the directory `bin/arch/variant` instead of `bin/$ARCH`. Replace the terms `arch` and `variant` with strings representing a specific architecture and MATLAB version.

`matlab dispOption` starts MATLAB using one of the display options shown in the table below. Enter only one of these options in a `matlab` command.

Values for dispOption

Option	Description
`-display` `xDisp`	Send X commands to X Window Server display `xDisp`. This supersedes the value of the `DISPLAY` environment variable.
`-nodisplay`	Start the Sun Microsystems JVM software, but do not start the MATLAB desktop. Do not display any X commands, and ignore the `DISPLAY` environment variable,

`matlab modeOption` starts MATLAB without its usual desktop component. Enter only one of the options shown below.

Values for modeOption

Option	Description
`-desktop`	Allow the MATLAB desktop to be started by a process without a controlling terminal. This is usually a required command line argument when attempting to start MATLAB from a window manager menu or desktop icon.
`-nodesktop`	Start MATLAB without bringing up the MATLAB desktop. The JVM software *is* started. Use the current window in the operating system to enter commands. Use this option to run without an X-window, for example, in VT100 mode, or in batch processing mode. Note that if you pipe to MATLAB using the `>` constructor, the `nodesktop` option is used automatically. With `nodesktop`, MATLAB does not save statements to the Command History. With `nodesktop`, you can still use most development environment tools by starting them via a function. For example, use `preferences` to open the Preferences dialog box and `helpbrowser` to open the Help browser. Do not use `nodesktop` to provide a Command Window-only interface; instead, select **Desktop > Desktop Layout > Command Window Only**.
`-nojvm`	Start MATLAB without the JVM software. Use the current window to enter commands. The MATLAB desktop does not open. Any tools that require Java software, such as the desktop tools, cannot be used. Handle Graphics and related functionality are not supported; MATLAB produces a warning when you use them.

`matlab` **`-c`** `licensefile` starts MATLAB using the specified license file. The `licensefile` argument can have the form `port@host` or it can be a colon-separated list of license filenames. This option causes the `LM_LICENSE_FILE` and `MLM_LICENSE_FILE` environment variables to be ignored.

`matlab` **`-r`** `matlab_command` starts MATLAB and executes the specified MATLAB `command`.

`matlab` **`-logfile`** `filename` starts MATLAB and makes a copy of any output to the command window in file log. This includes all crash reports.

`matlab` **`-mwvisual`** `visualid` starts MATLAB and uses `visualid` as the default X visual for figure windows. `visualid` is a hexadecimal number that can be found using `xdpyinfo`.

`matlab` **`-nosplash`** starts MATLAB but does not display the splash screen during startup.

`matlab` **`-singleCompThread`** limits MATLAB to a single computational thread. By default, MATLAB makes use of the multithreading capabilities of the computer on which it is running. For more information about multithreading, see .

`matlab` **`-debug`** starts MATLAB and displays debugging information that can be useful, especially for X based problems. This option should be used only when working with a Technical Support Representative from The MathWorks, Inc.

`matlab` **`-D`**`debugger options` starts MATLAB in debug mode, using the named `debugger` (e.g., `dbx`, `gdb`, `xdb`, `cvd`). A full path can be specified for debugger.

The `options` argument can include *only* those options that follow the debugger name in the syntax of the actual debug command. For most debuggers, there is a very limited number of such options. Options that would normally be passed to the MATLAB executable should be used as parameters of a command inside the debugger (like `run`). They should not be used when running the `matlab` script.

If any other `matlab` command options are placed before the **`-D`**`debugger` argument, they will be handled as if they were part of the options after the **`-D`**`debugger` argument and will be treated as illegal options by most debuggers. The `MATLAB_DEBUG` environment variable is set to the filename part of the debugger argument.

To customize your debugging session, use a startup file. See your debugger documentation for details.

Note For certain debuggers like `gdb`, the `SHELL` environment variable is *always* set to `/bin/sh`.

Specifying Options in the MATLAB Startup File

The `.matlab7rc.sh` shell script contains definitions for a number of variables that the `matlab` script uses. These variables are defined within the `matlab` script, but can be redefined in `.matlab7rc.sh`. When invoked, `matlab` looks for the first occurrence of `.matlab7rc.sh` in the current directory, in the home directory (`$HOME`), and in the *`matlabroot`*`/bin` directory, where the template version of `.matlab7rc.sh` is located.

You can edit the template file to redefine information used by the `matlab` script. If you do not want your changes applied systemwide, copy the edited version of the script to your current or home directory. Ensure that you edit the section that applies to your machine architecture.

The following table lists the variables defined in the`.matlab7rc.sh` file. See the comments in the `.matlab7rc.sh` file for more information about these variables.

Variable	Definition and Standard Assignment Behavior
`ARCH`	The machine architecture. The value `ARCH` passed with the `-arch` or `-arch/ext` argument to the script is tried first, then the value of the environment variable `MATLAB_ARCH` is tried next, and finally it is computed. The first one that gives a valid architecture is used.
`AUTOMOUNT_MAP`	Path prefix map for automounting. The value set in `.matlab7rc.sh` (initially by the installer) is used unless the value differs from that determined by the script, in which case the value in the environment is used.
`DISPLAY`	The hostname of the X Window display MATLAB uses for output. The value of `Xdisplay` passed with the `-display` argument to the script is used; otherwise, the value in the environment is used. `DISPLAY` is ignored by MATLAB if the `-nodisplay` argument is passed.

Variable	Definition and Standard Assignment Behavior
`LD_LIBRARY_PATH`	Final Load library path. The name `LD_LIBRARY_PATH` is platform dependent. The final value is normally a colon-separated list of four sublists, each of which could be empty. The first sublist is defined in `.matlab7rc.sh` as `LDPATH_PREFIX`. The second sublist is computed in the script and includes directories inside the MATLAB root directory and relevant Sun Microsystems Java directories. The third sublist contains any nonempty value of `LD_LIBRARY_PATH` from the environment possibly augmented in `.matlab7rc.sh`. The final sublist is defined in `.matlab7rc.sh` as `LDPATH_SUFFIX`.
`LM_LICENSE_FILE`	The FLEX lm license variable. The license file value passed with the `-c` argument to the script is used; otherwise it is the value set in `.matlab7rc.sh`. In general, the final value is a colon-separated list of license files and/or `port@host` entries. The shipping `.matlab7rc.sh` file starts out the value by prepending `LM_LICENSE_FILE` in the environment to a default `license.file`. Later in the `matlab` script, if the `-c` option is not used, the *`matlabroot`*`/etc` directory is searched for the files that start with `license.dat.DEMO`. These files are assumed to contain demo licenses and are added automatically to the end of the current list.

Variable	**Definition and Standard Assignment Behavior**
`MATLAB`	The MATLAB root directory. The default computed by the script is used unless `MATLABdefault` is reset in `.matlab7rc.sh`. Currently `MATLABdefault` is not reset in the shipping `.matlab7rc.sh`.
`MATLAB_DEBUG`	Normally set to the name of the debugger. The `-Ddebugger` argument passed to the script sets this variable. Otherwise, a nonempty value in the environment is used.
`MATLAB_JAVA`	The path to the root of the Java Runtime Environment. The default set in the script is used unless `MATLAB_JAVA` is already set. Any nonempty value from `.matlab7rc.sh` is used first, then any nonempty value from the environment. Currently there is no value set in the shipping `.matlab67rc.sh`, so that environment alone is used.
`MATLABPATH`	The MATLAB search path. The final value is a colon-separated list with the `MATLABPATH` from the environment prepended to a list of computed defaults. You can add subdirectories of `userpath` to theMATLAB search path upon startup. See `userpath` for details.

Variable	Definition and Standard Assignment Behavior
`SHELL`	The shell to use when the "`!`" or `unix` command is issued in MATLAB. This is taken from the environment unless `SHELL` is reset in `.matlab7rc.sh`. Note that an additional environment variable called `MATLAB_SHELL` takes precedence over `SHELL`. MATLAB checks internally for `MATLAB_SHELL` first and, if empty or not defined, then checks `SHELL`. If `SHELL` is also empty or not defined, MATLAB uses `/bin/sh`. The value of `MATLAB_SHELL` should be an absolute path, i.e. `/bin/sh`, not simply `sh`. Currently, the shipping `.matlab7rc.sh` file does not reset `SHELL` and also does not reference or set `MATLAB_SHELL`.
`TOOLBOX`	Path of the toolbox directory. A nonempty value in the environment is used first. Otherwise, *`matlabroot`*`/toolbox`, computed by the script, is used unless `TOOLBOX` is reset in `.matlab7rc.sh`. Currently `TOOLBOX` is not reset in the shipping `.matlab7rc.sh`.

Variable	**Definition and Standard Assignment Behavior**
`XAPPLRESDIR`	The X application resource directory. A nonempty value in the environment is used first unless `XAPPLRESDIR` is reset in `.matlab7rc.sh`. Otherwise, *`matlabroot`*`/X11/app-defaults`, computed by the script, is used.
`XKEYSYMDB`	The X keysym database file. A nonempty value in the environment is used first unless `XKEYSYMDB` is reset in `.matlab7rc.sh`. Otherwise, *`matlabroot`*`/X11/app-defaults/XKeysymDB`, computed by the script, is used. The `matlab` script determines the path of the MATLAB root directory as one level up the directory tree from the location of the script. Information in the `AUTOMOUNT_MAP` variable is used to fix the path so that it is correct to force a mount. This can involve deleting part of the pathname from the front of the MATLAB root path. The MATLAB variable is then used to locate all files within the MATLAB directory tree.

The `matlab` script determines the path of the MATLAB root directory by looking up the directory tree from the *`matlabroot`*`/bin` directory (where the `matlab` script is located). The `MATLAB` variable is then used to locate all files within the MATLAB directory tree.

You can change the definition of `MATLAB` if, for example, you want to run a different version of MATLAB or if, for some reason, the path determined by the `matlab` script is not correct. (This can happen when certain types of automounting schemes are used by your system.)

`AUTOMOUNT_MAP` is used to modify the MATLAB root directory path. The pathname that is assigned to `AUTOMOUNT_MAP` is deleted from the front of the MATLAB root path. (It is unlikely that you will need to use this option.)

See Also

`matlab (Windows)`, `mex`

, , and in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Start MATLAB program (Windows platforms)

Syntax

```
matlab helpOption
matlab -automation
matlab -c licensefile
matlab -logfile filename
matlab -nosplash
matlab -noFigureWindows
matlab -r "statement"
matlab -regserver
matlab -sd "startdir"
matlab shieldOption
matlab -singleCompThread
matlab -unregserver
matlab -wait
```

Note You can enter more than one of these options in the same `matlab` command.

Description

`matlab` is a script that runs the main MATLAB executable on Microsoft Windows platforms. (In this document, the term `matlab` refers to the script, and MATLAB refers to the main executable). Before actually initiating the execution of MATLAB, it configures the run-time environment by

- Determining the MATLAB root directory
- Determining the host machine architecture
- Selectively processing command line options with the rest passed to MATLAB.
- Setting certain MATLAB environment variables

There are two ways in which you can control the way `matlab` works:

- By specifying command line options
- By setting environment variables before calling the program

Specifying Options at the Command Line

Options that you can enter at the command line are as follows:

`matlab` *`helpOption`* displays information that matches the specified *`helpOption`* argument without starting MATLAB. *`helpOption`* can be any one of the keywords shown in the table below. Enter only one *`helpOption`* keyword in a `matlab` statement.

Values for helpOption

Option	Description
`-help`	Display `matlab` command usage.
`-h`	The same as **`-help`**.
`-?`	The same as **`-help`**.

`matlab` **`-automation`** starts MATLAB as an automation server. The server window is minimized, and the MATLAB splash screen does not display on startup.

`matlab` **`-c`** `licensefile` starts MATLAB using the specified license file. The `licensefile` argument can have the form `port@host`. This option causes MATLAB to ignore the `LM_LICENSE_FILE` and `MLM_LICENSE_FILE` environment variables.

`matlab` **`-logfile`** `filename` starts MATLAB and makes a copy of any output to the Command Window in `filename`. This includes all crash reports.

`matlab` **`-nosplash`** starts MATLAB, but does not display the splash screen during startup.

`matlab` **`-noFigureWindows`** starts MATLAB, but disables the display of any figure windows in MATLAB.

`matlab` **`-r`** `"statement"` starts MATLAB and executes the specified MATLAB `statement`. Any required file must be on the MATLAB search path or in the startup directory.

`matlab` **`-regserver`** registers MATLAB as a Component Object Model (COM) server.

`matlab` **`-sd`** `"`*`startdir`*`"` specifies the startup directory for MATLAB (the current directory in MATLAB after startup). The **`-sd`** option has been deprecated. For information about alternatives, see .

`matlab` *`shieldOption`* provides the specified level of protection of the address space used by MATLAB during startup on Windows 32–bit platforms. It attempts to help ensure the largest contiguous block of memory available after startup, which is useful for processing large data sets. The *`shieldOption`* does this by ensuring resources such as DLLs, are loaded into locations that will not fragment the address space. With *`shieldOption`* set to a value other than `none`, address space is protected up to or after the processing of `matlabrc`. Use higher levels of protection to secure larger initial blocks of contiguous memory, however a higher level might not always provide a larger size block and might cause startup problems. Therefore, start with a lower level of protection, and if successful, try the next higher level. You can use the `memory` function after startup to see the size of the largest contiguous block of memory; this helps you determine the actual effect of the *`shieldOption`* setting you used. If your `matlabrc` (or `startup.m`) requires significant memory, a higher level of protection for *`shieldOption`* might cause startup to fail; in that event try a lower level. Values for *`shieldOption`* can be any one of the keywords shown in the table below.

Option	Description
`-shield minimum`	This is the default setting. It protects the range `0x50000000` to `0x70000000` during MATLAB startup until just before startup processes `matlabrc`. It ensures there is at least approximately 500 MB of contiguous memory up to this point. Start with this, the default value. To use the default, do not specify a shield option upon startup. If MATLAB fails to start successfully using the default option, **`-shield minimum`**, instead use **`-shield none`**. If MATLAB starts successfully with the default value for *`shieldOption`* and you want to try to ensure an even larger contiguous block after startup, try using the **`-shield medium`** option.
`-shield medium`	This protects the same range as for minimum, `0x50000000` to `0x70000000`, but protects the range until just after startup processes `matlabrc`. It ensures there is at least approximately 500 MB of contiguous memory up to this point. If MATLAB fails to start successfully with the **`-shield medium`** option, instead use the default option (**`-shield minimum`**). If MATLAB starts successfully with the **`-shield medium`** option and you want to try to ensure an even larger contiguous block after startup, try using the **`-shield maximum`** option.

Option	Description
`-shield maximum`	This protects the maximum possible range, which can be up to approximately 1.5 GB, until just after startup processes `matlabrc`. If MATLAB fails to start successfully with the **`-shield maximum`** option, instead use the **`-shield medium`** option.
`-shield none`	This completely disables address shielding. Use this if MATLAB fails to start successfully with the default (**`-shield minimum`**) option.

`matlab` **`-singleCompThread`** limits MATLAB to a single computational thread. By default, MATLAB makes use of the multithreading capabilities of the computer on which it is running. For more information about multithreading, see

`matlab` **`-unregserver`** removes all MATLAB COM server entries from the registry.

`matlab` **`-wait`** MATLAB is started by a separate starter program which normally launches MATLAB and then immediately quits. Using this option tells the starter program not to quit until MATLAB has terminated. This option is useful when you need to process the results from MATLAB in a script. Calling MATLAB with this option blocks the script from continuing until the results are generated.

Setting Environment Variables

You can set the following environment variables before starting MATLAB.

Variable Name	Description
`LM_LICENSE_FILE`	This variable specifies the License File to use. If you use the **`-c`** argument to specify the License File it overrides this variable. The value of this variable can be a list of License Files, separated by semi-colons, or `port@host` entries.

See Also `matlab (UNIX)`, `mex`, `userpath`

, and in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Largest elements in array

Syntax

```
C = max(A)
C = max(A,B)
C = max(A,[],dim)
[C,I] = max(...)
```

Description

`C = max(A)` returns the largest elements along different dimensions of an array.

If `A` is a vector, `max(A)` returns the largest element in `A`.

If `A` is a matrix, `max(A)` treats the columns of `A` as vectors, returning a row vector containing the maximum element from each column.

If `A` is a multidimensional array, `max(A)` treats the values along the first non-singleton dimension as vectors, returning the maximum value of each vector.

`C = max(A,B)` returns an array the same size as `A` and `B` with the largest elements taken from `A` or `B`. The dimensions of `A` and `B` must match, or they may be scalar.

`C = max(A,[],dim)` returns the largest elements along the dimension of `A` specified by scalar `dim`. For example, `max(A,[],1)` produces the maximum values along the first dimension (the rows) of `A`.

`[C,I] = max(...)` finds the indices of the maximum values of `A`, and returns them in output vector `I`. If there are several identical maximum values, the index of the first one found is returned.

Remarks

For complex input `A`, `max` returns the complex number with the largest complex modulus (magnitude), computed with `max(abs(A))`. Then computes the largest phase angle with `max(angle(x))`, if necessary.

The `max` function ignores `NaN`s.

See Also

`isnan`, `mean`, `median`, `min`, `sort`

max (timeseries)

Purpose

Maximum value of `timeseries` data

Syntax

```
ts_max = max(ts)
ts_max = max(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_max = max(ts)` returns the maximum value in the time-series data. When `ts.Data` is a vector, ts_max is the maximum value of `ts.Data` values. When `ts.Data` is a matrix, ts_max is a row vector containing the maximum value of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `max` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_max = max(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`.
  When you specify `'time'`, larger time values correspond to larger weights.

Examples

The following example illustrates how to find the maximum values in multivariate time-series data.

1. Load a 24-by-3 data array.

   ```
   load count.dat
   ```

2. Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,[1:24],'Name','CountPerSecond')
```

3 Find the maximum in each data column for this `timeseries` object.

```
max(count_ts)

ans =

   114   145   257
```

The maximum is found independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `min (timeseries)`, `median (timeseries)`, `mean (timeseries)`, `std (timeseries)`, `timeseries`, `var (timeseries)`

MaximizeCommandWindow

Purpose

Open Automation server window

Syntax

MATLAB Client

```
h.MaximizeCommandWindow
MaximizeCommandWindow(h)
```

IDL Method Signature

```
HRESULT MaximizeCommandWindow(void)
```

Microsoft Visual Basic Client

```
MaximizeCommandWindow
```

Description

`h.MaximizeCommandWindow` displays the window for the server attached to handle `h`, and makes it the currently active window on the desktop.

`MaximizeCommandWindow(h)` is an alternate syntax.

`MaximizeCommandWindow` restores the window to the size it had at the time it was minimized, not to the maximum size on the desktop. If the server window was not previously in a minimized state, `MaximizeCommandWindow` does nothing.

Examples

From a MATLAB client, modify the size of the command window in a MATLAB Automation server:

```
h = actxserver('matlab.application');
h.MinimizeCommandWindow;
% Now return the server window to its former state on
% the desktop and make it the currently active window.
h.MaximizeCommandWindow;
```

From a Visual Basic client, modify the size of the command window in a MATLAB Automation server:

```
Dim Matlab As Object
```

```
Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow

'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

See Also

`MinimizeCommandWindow`

How To

-
-

maxNumCompThreads

Purpose

Control maximum number of computational threads

Note `maxNumCompThreads` will be removed in a future version. You can set the `-singleCompThread` option when starting MATLAB to limit MATLAB to a single computational thread. By default, MATLAB makes use of the multithreading capabilities of the computer on which it is running.

Syntax

```
N = maxNumCompThreads
LASTN = maxNumCompThreads(N)
LASTN = maxNumCompThreads('automatic')
```

Description

`N = maxNumCompThreads` returns the current maximum number of computational threads `N`.

`LASTN = maxNumCompThreads(N)` sets the maximum number of computational threads to `N`, and returns the previous maximum number of computational threads, `LASTN`.

`LASTN = maxNumCompThreads('automatic')` sets the maximum number of computational threads using what the MATLAB software determines to be the most desirable. It additionally returns the previous maximum number of computational threads, `LASTN`.

Currently, the maximum number of computational threads is equal to the number of computational cores on your machine.

Note Setting the maximum number of computational threads using `maxNumCompThreads` does not propagate to your next MATLAB session.

Purpose Average or mean value of array

Syntax

```
M = mean(A)
M = mean(A,dim)
```

Description `M = mean(A)` returns the mean values of the elements along different dimensions of an array.

If `A` is a vector, `mean(A)` returns the mean value of `A`.

If `A` is a matrix, `mean(A)` treats the columns of `A` as vectors, returning a row vector of mean values.

If `A` is a multidimensional array, `mean(A)` treats the values along the first non-singleton dimension as vectors, returning an array of mean values.

`M = mean(A,dim)` returns the mean values for elements along the dimension of `A` specified by scalar `dim`. For matrices, `mean(A,2)` is a column vector containing the mean value of each row.

Examples

```
A = [1 2 3; 3 3 6; 4 6 8; 4 7 7];
mean(A)
ans =
    3.0000    4.5000    6.0000

mean(A,2)
ans =
    2.0000
    4.0000
    6.0000
    6.0000
```

See Also `corrcoef`, `cov`, `max`, `median`, `min`, `mode`, `std`, `var`

mean (timeseries)

Purpose Mean value of `timeseries` data

Syntax

```
ts_mn = mean(ts)
ts_mn = mean(ts,'PropertyName1',PropertyValue1,...)
```

Description `ts_mn = mean(ts)` returns the mean value of `ts.Data`. When `ts.Data` is a vector, ts_mn is the mean value of `ts.Data` values. When `ts.Data` is a matrix, ts_mn is a row vector containing the mean value of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `mean` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_mn = mean(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`.
  When you specify `'time'`, larger time values correspond to larger weights.

Examples The following example illustrates how to find the mean values in multivariate time-series data.

1. Load a 24-by-3 data array.

   ```
   load count.dat
   ```

2. Create a `timeseries` object with 24 time values.

   ```
   count_ts = timeseries(count,[1:24],'Name','CountPerSecond')
   ```

3 Find the mean of each data column for this `timeseries` object.

```
mean(count_ts)

ans =

   32.0000   46.5417   65.5833
```

The mean is found independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `max (timeseries)`, `min (timeseries)`, `median (timeseries)`, `std (timeseries)`, `timeseries`, `var (timeseries)`

median

Purpose

Median value of array

Syntax

```
M = median(A)
M = median(A,dim)
```

Description

`M = median(A)` returns the median values of the elements along different dimensions of an array. `A` should be of type `single` or `double`.

If `A` is a vector, `median(A)` returns the median value of `A`.

If `A` is a matrix, `median(A)` treats the columns of `A` as vectors, returning a row vector of median values.

If `A` is a multidimensional array, `median(A)` treats the values along the first nonsingleton dimension as vectors, returning an array of median values.

`M = median(A,dim)` returns the median values for elements along the dimension of `A` specified by scalar `dim`.

Examples

```
A = [1 2 4 4; 3 4 6 6; 5 6 8 8; 5 6 8 8];
median(A)

ans =

     4     5     7     7

median(A,2)

ans =

     3
     5
     7
     7
```

See Also

`corrcoef`, `cov`, `max`, `mean`, `min`, `mode`, `std`, `var`

Purpose

Median value of `timeseries` data

Syntax

```
ts_med = median(ts)
ts_med = median(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_med = median(ts)` returns the median value of `ts.Data`. When `ts.Data` is a vector, ts_med is the median value of `ts.Data` values. When `ts.Data` is a matrix, ts_med is a row vector containing the median value of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `median` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_med = median(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`.
  When you specify `'time'`, larger time values correspond to larger weights.

Examples

The following example illustrates how to find the median values in multivariate time-series data.

1. Load a 24-by-3 data array.

   ```
   load count.dat
   ```

2. Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,[1:24],'Name','CountPerSecond')
```

3 Find the median of each data column for this `timeseries` object.

```
median(count_ts)

ans =

   23.5000   36.0000   39.0000
```

The median is found independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `max (timeseries)`, `min (timeseries)`, `mean (timeseries)`, `std (timeseries)`, `timeseries`, `var (timeseries)`

Purpose

Construct memmapfile object

Syntax

```
m = memmapfile(filename)
m = memmapfile(filename, prop1, value1, prop2, value2, ...)
```

Description

`m = memmapfile(filename)` constructs an object of the `memmapfile` class that maps file `filename` to memory using the default property values. The `filename` input is a quoted string that specifies the path and name of the file to be mapped into memory. `filename` must include a filename extension if the name of the file being mapped has an extension. The `filename` argument cannot include any wildcard characters (e.g., `*` or `?`), is case sensitive on The Open Group UNIX platforms, but is not case sensitive on Microsoft Windows platforms.

`m = memmapfile(filename, prop1, value1, prop2, value2, ...)` constructs an object of the `memmapfile` class that maps file `filename` into memory and sets the properties of that object that are named in the argument list (`prop1`, `prop2`, etc.) to the given values (`value1`, `value2`, etc.). All property name arguments must be quoted strings (e.g., `'Writable'`). Any properties that are not specified are given their default values.

Optional properties are shown in the table below and are described in the sections that follow.

Property	Description	Data Type	Default
`Format`	Format of the contents of the mapped region, including data type, array shape, and variable or field name by which to access the data	`char` array or N-by-3 `cell` array	`uint8`

Property	**Description**	**Data Type**	**Default**
`Offset`	Number of bytes from the start of the file to the start of the mapped region. This number is zero-based. That is, offset 0 represents the start of the file.	`double`	`0`
`Repeat`	Number of times to apply the specified format to the mapped region of the file	`double`	`Inf`
`Writable`	Type of access allowed to the mapped region	`logical`	false

There are three different ways you can specify a value for the `Format` property. See the following sections in the MATLAB Data Import and Export documentation for more information:

-
-
-

Any of the following data types can be used when you specify a `Format` value. The default type is `uint8`.

Format String	Data Type Description
`'int8'`	Signed 8-bit integers
`'int16'`	Signed 16-bit integers
`'int32'`	Signed 32-bit integers
`'int64'`	Signed 64-bit integers
`'uint8'`	Unsigned 8-bit integers
`'uint16'`	Unsigned 16-bit integers
`'uint32'`	Unsigned 32-bit integers
`'uint64'`	Unsigned 64-bit integers
`'single'`	32-bit floating-point
`'double'`	64-bit floating-point

Remarks

You can only map an existing file. You cannot create a new file and map that file to memory in one operation. Use the MATLAB file I/O functions to create the file before attempting to map it to memory.

Once `memmapfile` locates the file, MATLAB stores the absolute pathname for the file internally, and then uses this stored path to locate the file from that point on. This enables you to work in other directories outside your current work directory and retain access to the mapped file.

Once a `memmapfile` object has been constructed, you can change the value of any of its properties. Use the `objname.property` syntax in assigning the new value. To set a new offset value for memory map object `m`, type

```
m.Offset = 2048;
```

Property names are not case sensitive. For example, MATLAB considers `m.Offset` to be the same as `m.offset`.

Examples

To run the following examples, create a file in your current folder called `records.dat`. Example 2 expects that `records.dat` contains

5000 double-precision values. The following code shows one method to create this file:

```
randData = gallery('uniformdata', [5000, 1], 0, 'double');

fid = fopen('records.dat','w');
fwrite(fid, randData, 'double');
fclose(fid);
```

Example 1

To construct a map for `records.dat`, type the following:

```
m = memmapfile('records.dat');
```

MATLAB constructs an instance of the `memmapfile` class, assigns it to the variable `m`, and maps the entire `records.dat` file to memory, setting all properties of the object to their default values. In this example, the command maps the entire file as a sequence of unsigned 8-bit integers and gives the caller read-only access to its contents.

Example 2

To construct a map using nondefault values for the `Offset`, `Format`, and `Writable` properties, type the following, enclosing all property names in single quotation marks:

```
m = memmapfile('records.dat',         ...
               'Offset', 1024,        ...
               'Format', 'uint32',    ...
               'Writable', true);
```

Type the object name to see the current settings for all properties:

```
m

m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: true
      Offset: 1024
```

```
    Format: 'uint32'
    Repeat: Inf
      Data: 9744x1 uint32 array
```

Example 3

Construct a `memmapfile` object for the entire file `records.dat` and set the `Format` property for that object to `uint64`. Any read or write operations made via the memory map will read and write the file contents as a sequence of unsigned 64-bit integers:

```
m = memmapfile('records.dat', 'Format', 'uint64');
```

Example 4

Construct a `memmapfile` object for a region of `records.dat` such that the contents of the region are handled by MATLAB as a 4-by-10-by-18 array of unsigned 32-bit integers, and can be referenced in the structure of the returned object using the field name `x`:

```
m = memmapfile('records.dat',    ...
               'Offset', 1024,   ...
               'Format', {'uint32' [4 10 18] 'x'});

A = m.Data(1).x;

whos A
  Name      Size                    Bytes  Class

  A        4x10x18                  2880   uint32 array
```

Example 5

Map a file to three different data types: `int16`, `uint32`, and `single`. The `int16` data is mapped as a 2-by-2 matrix that can be accessed using the field name `model`. The `uint32` data is a scalar value accessed as field `serialno`. The `single` data is a 1-by-3 matrix named `expenses`. Repeat the pattern 1000 times.

Each of the fields belongs to the 1000-by-1 structure array `m.Data`:

```
m = memmapfile('records.dat',                          ...
               'Format', {                             ...
                  'int16'  [2 2] 'model';              ...
                  'uint32' [1 1] 'serialno';           ...
                  'single' [1 3] 'expenses'},          ...
               'Repeat', 1000);
```

See Also `disp(memmapfile)`, `get(memmapfile)`

Purpose

Display memory information

Syntax

```
memory
userview = memory
[userview systemview] = memory
```

Description

`memory` displays information showing how much memory is available and how much the MATLAB software is currently using. The information displayed at your computer screen includes the following items, each of which is described in a section below:

- "Maximum Possible Array" on page 2-2320
- "Memory Available for All Arrays" on page 2-2321
- "Memory Used By MATLAB" on page 2-2322
- "Total Physical Memory (RAM)" on page 2-2322

`userview = memory` returns user-focused information on memory use in structure `userview`. The information returned in `userview` includes the following items, each of which is described in a section below:

- "Maximum Possible Array" on page 2-2320
- "Memory Available for All Arrays" on page 2-2321
- "Memory Used By MATLAB" on page 2-2322

`[userview systemview] = memory` returns both user- and system-focused information on memory use in structures `userview` and `systemview`, respectively. The `userview` structure is described in the command syntax above. The information returned in `systemview` includes the following items, each of which is described in a section below:

- "Virtual Address Space" on page 2-2323
- "System Memory" on page 2-2323

- "Physical Memory" on page 2-2324

Output

Each of the sections below describes a value that is displayed or returned by the `memory` function.

Maximum Possible Array

Maximum Possible Array is the size of the largest contiguous free memory block. As such, it is an upper bound on the largest single array MATLAB can create at this time.

MATLAB derives this number from the smaller of the following two values:

- The largest contiguous memory block found in the MATLAB virtual address space
- The total available system memory

To see how many array elements this number represents, divide by the number of bytes in the array class. For example, for a `double` array, divide by 8. The actual number of elements MATLAB can create is always fewer than this number.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	String labelled `Maximum possible array:`
`user = memory`	Structure field `user.MaxPossibleArrayBytes`

All values are double-precision and in units of bytes.

Footnotes

When you enter the `memory` command without specifying any outputs, MATLAB may also display one of the following footnotes. 32-bit systems show either the first or second footnote; 64-bit systems show only the second footnote:

`Limited by contiguous virtual address space available.`

There is sufficient system memory to allow mapping of all virtual addresses in the largest available block of the MATLAB process. The maximum amount of total MATLAB virtual address space is either 2 GB or 3 GB, depending on whether the /3GB switch is in effect or not.

`Limited by System Memory (physical + swap file) available.`

There is insufficient system memory to allow mapping of all virtual addresses in the largest available block of the MATLAB process.

Memory Available for All Arrays

Memory Available for All Arrays is the total amount of memory available to hold data. The amount of memory available is guaranteed to be at least as large as this field.

MATLAB derives this number from the smaller of the following two values:

- The total available MATLAB virtual address space
- The total available system memory

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	String labelled `Memory available for all arrays:`
`user = memory`	Structure field `user.MemAvailableAllArrays`

Footnotes

When you enter the `memory` command without specifying any outputs, MATLAB may also display one of the following footnotes. 32-bit systems show either the first or second footnote; 64-bit systems show only the latter footnote:

`Limited by virtual address space available.`

> There is sufficient system memory to allow mapping of all available virtual addresses in the MATLAB process virtual address space to system memory. The maximum amount of total MATLAB virtual address space is either 2 GB or 3 GB, depending on whether the /3GB switch is in effect or not.

`Limited by System Memory (physical + swap file) available.`

> There is insufficient system memory to allow mapping of all available virtual addresses in the MATLAB process.

Memory Used By MATLAB

Memory Used By MATLAB is the total amount of system memory reserved for the MATLAB process. It is the sum of the physical memory and potential swap file usage.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	String labelled `Memory used by MATLAB:`
`user = memory`	Structure field `user.MemUsedMATLAB`

Total Physical Memory (RAM)

Physical Memory (RAM) is the total physical memory (or RAM) in the computer.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. See the table below.

Command	**Returned in**
`memory`	String labelled `Physical Memory (RAM):`

Virtual Address Space

Virtual Address Space is the amount of available and total virtual memory for the MATLAB process. MATLAB returns the information in two fields of the return structure: `Available` and `Total`.

Command	**Return Value**	**Returned in Structure Field**
`[user,sys] = memory`	Available memory	`sys.VirtualAddressSpace.Available`
	Total memory	`sys.VirtualAddressSpace.Total`

You can monitor the difference:

```
VirtualAddressSpace.Total - VirtualAddressSpace.Available
```

as the Virtual Bytes counter in the WindowsPerformance program. (e.g., Windows XP Control Panel/Administrative Tool/Performance program).

System Memory

System Memory is the amount of available system memory on your computer system. This number includes the amount of available physical memory and the amount of available swap file space on the computer running MATLAB. MATLAB returns the information in the `SystemMemory` field of the return structure.

Command	**Return Value**	**Returned in Structure Field**
`[user,sys] = memory`	Available memory	`sys.SystemMemory`

This is the same as the difference:

```
limit - total (in bytes)
```

found in the Windows Task Manager: Performance/Commit Charge.

Physical Memory

Physical Memory is the available and total amounts of physical memory (RAM) on the computer running MATLAB. MATLAB returns the information in two fields of the return structure: `Available` and `Total`.

Command	Value	Returned in Structure Field
`[user,sys] = memory`	Available memory	`sys.PhysicalMemory.Available`
	Total memory	`sys.PhysicalMemory.Total`

Available physical memory is the same as:

```
Available (in bytes)
```

found in the Windows Task Manager: Performance/Physical Memory

The total physical memory is the same as

```
Total (in bytes)
```

found in the Windows Task Manager: Performance/Physical Memory

You can use the amount of available physical memory as a measure of how much data you can access quickly.

Remarks

The `memory` function is currently available on Microsoft Windows systems only. Results vary, depending on the computer running MATLAB, the load on that computer, and what MATLAB is doing at the time.

Details on Memory Used By MATLAB

MATLAB computes the value for Memory Used By MATLAB by walking the MATLAB process memory structures and summing all the sections that have physical storage allocated in memory or in the paging file on disk.

Using the Windows Task Manager, you have for the MATLAB.exe image:

```
Mem Usage < MemUsedMATLAB < Mem Usage + VM Size (in bytes)
```

where both of the following are true:

- `Mem Usage` is the working set size in kilobytes.
- `VM Size` is the page file usage, or private bytes, in kilobytes.

The working set size is the portion of the MATLAB virtual address space that is *currently* resident in RAM and can be referenced without a memory page fault. The page file usage gives the portion of the MATLAB virtual address space that requires a backup that doesn't already exist. Another name for page file usage is *private bytes*. It includes all MATLAB variables and workspaces. Since some of the pages in the page file may also be part of the working set, this sum is an overestimate of `MemUseMATLAB`. Note that there are virtual pages in the MATLAB process space that already have a backup. For example, code loaded from EXEs and DLLs and memory-mapped files. If any part of those files is in memory when the memory builtin is called, that memory will be counted as part of `MemUsedMATLAB`.

Reserved Addresses

Reserved addresses are addresses sets aside in the process virtual address space for some specific future use. These reserved addresses reduce the size of `MemAvailableAllArrays` and can reduce the size of the current or future value of `MaxPossibleArrayBytes`.

Example 1 — Java Virtual Machine (JVM)

At MATLAB startup, part of the MATLAB virtual address space is reserved by the Java Virtual Machine (JVM) and cannot be used for storing MATLAB arrays.

Example 2 — Standard Windows Heap Manager

MATLAB, by default, uses the standard Windows heap manager except for a set of small preselected allocation sizes. One characteristic of this heap manager is that its behavior depends upon whether the requested allocation is less than or greater than the fixed number of 524,280 bytes. For, example, if you create a sequence of MATLAB arrays, each less then 524,280 bytes, and then clear them all, the `MemUsedMATLAB` value before and after shows little change, and the `MemAvailableAllArrays` value is now smaller by the total space allocated.

The result is that, instead of globally freeing the extra memory, the memory becomes reserved. It can *only* be reused for arrays less than 524,280 bytes. You cannot reclaim this memory for a larger array except by restarting MATLAB.

Examples

Display memory statistics on a 32-bit Windows system:

```
memory

Maximum possible array:           677 MB (7.101e+008 bytes) *
Memory available for all arrays: 1601 MB (1.679e+009 bytes) **
Memory used by MATLAB:            446 MB (4.681e+008 bytes)
Physical Memory (RAM):           3327 MB (3.489e+009 bytes)

  *  Limited by contiguous virtual address space available.
  ** Limited by virtual address space available.
```

Return in the structure `userview`, information on the largest array MATLAB can create at this time, how much memory is available to hold data, and the amount of memory currently being used by your MATLAB process:

```
userview = memory

userview =
   MaxPossibleArrayBytes: 710127616
   MemAvailableAllArrays: 1.6792e+009
           MemUsedMATLAB: 468127744
```

Assign the output to two structures, `user` and `sys`, to obtain the information shown here:

```
[user sys] = memory;

%  --- Largest array MATLAB can create ---
user.MaxPossibleArrayBytes
ans =
   710127616

%  --- Memory available for data ---
user.MemAvailableAllArrays
ans =
   1.6797e+009

%  --- Memory used by MATLAB process ---
user.MemUsedMATLAB
ans =
   467603456

%  --- Virtual memory for MATLAB process ---
sys.VirtualAddressSpace
ans =
   Available: 1.6797e+009
       Total: 2.1474e+009

%  --- Physical memory and paging file ---
sys.SystemMemory
ans =
   Available: 4.4775e+009

%   --- Computer's physical memory ---
sys.PhysicalMemory
ans =
   Available: 2.3941e+009
       Total: 3.4889e+009
```

See Also

`clear`, `pack`, `whos`, `inmem`, `save`, `load`, `mlock`, `munlock`

menu

Purpose

Generate menu of choices for user input

Syntax

```
choice = menu('mtitle','opt1','opt2',...,'optn')
choice = menu('mtitle',options)
```

displays the

Description

`choice = menu('mtitle','opt1','opt2',...,'optn')` menu whose title is in the string variable `'mtitle'` and whose choices are string variables `'opt1'`, `'opt2'`, and so on. The menu opens in a modal dialog box. `menu` returns the number of the selected menu item, or 0 if the user clicks the close button on the window.

`choice = menu('mtitle',options)` , where options is a 1-by-N cell array of strings containing the menu choices.

If the user's terminal provides a graphics capability, `menu` displays the menu items as push buttons in a figure window (Example 1). Otherwise. they will be given as a numbered list in the Command Window (Example 2).

Remarks

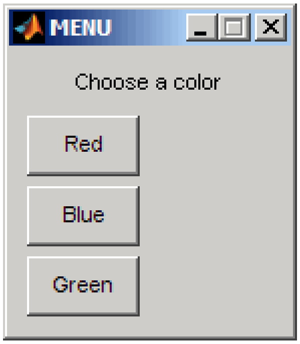
To call `menu` from a uicontrol or other ui object, set that object's `Interruptible` property to `'on'`. For more information, see Uicontrol Properties.

Examples

Example 1

On a system with a display, `menu` displays choices as buttons in a dialog box:

`choice = menu('Choose a color','Red','Green','Blue')` displays



The number entered by the user in response to the prompt is returned as `choice` (i.e., `choice = 2` implies that the user selected `Blue`).

After input is accepted, the dialog box closes, returning the output in `choice`. You can use `choice` to control the color of a graph:

```
t = 0:.1:60;
s = sin(t);
color = ['r','g','b']
plot(t,s,color(choice))
```

Example 2

On a system without a display, `menu` displays choices in the Command Window:

```
choice = menu('Choose a color','Red','Blue','Green')
```

displays the text

```
----- Choose a color -----
1) Red
2) Blue
3) Green
Select a menu number:
```

See Also

`guide`, `input`, `uicontrol`, `uimenu`

mesh, meshc, meshz

Purpose

Mesh plots

GUI Alternatives

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

Syntax

```
mesh(X,Y,Z)
mesh(Z)
mesh(...,C)
mesh(...,'PropertyName',PropertyValue,...)
mesh(axes_handles,...)
meshc(...)
meshz(...)
h = mesh(...)
hsurface = mesh('v6',...) hsurface = meshc('v6',...),
```

Description

`mesh`, `meshc`, and `meshz` create wireframe parametric surfaces specified by `X`, `Y`, and `Z`, with color specified by `C`.

`mesh(X,Y,Z)` draws a wireframe mesh with color determined by `Z` so color is proportional to surface height. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, $(X(j), Y(i), Z(i,j))$ are the intersections of the wireframe grid lines; `X` and `Y` correspond to the columns and rows of `Z`, respectively. If `X` and `Y` are matrices, $(X(i,j), Y(i,j), Z(i,j))$ are the intersections of the wireframe grid lines.

`mesh(Z)` draws a wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`mesh(...,C)` draws a wireframe mesh with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`mesh(...,'`*`PropertyName`*`',PropertyValue,...)` sets the value of the specified surface property. Multiple property values can be set with a single statement.

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

`meshc(...)` draws a contour plot beneath the mesh.

`meshz(...)` draws a curtain plot (i.e., a reference plane) around the mesh.

`h = mesh(...)`, h = meshc(...), and h = meshz(...) return a handle to a surfaceplot graphics object.

Backward-Compatible Version

`hsurface = mesh('v6',...)` `hsurface = meshc('v6',...)`, and `hsurface` = meshc('v6',...) returns the handles of surface objects instead of surfaceplot objects for compatibility with MATLAB 6.5 and earlier.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

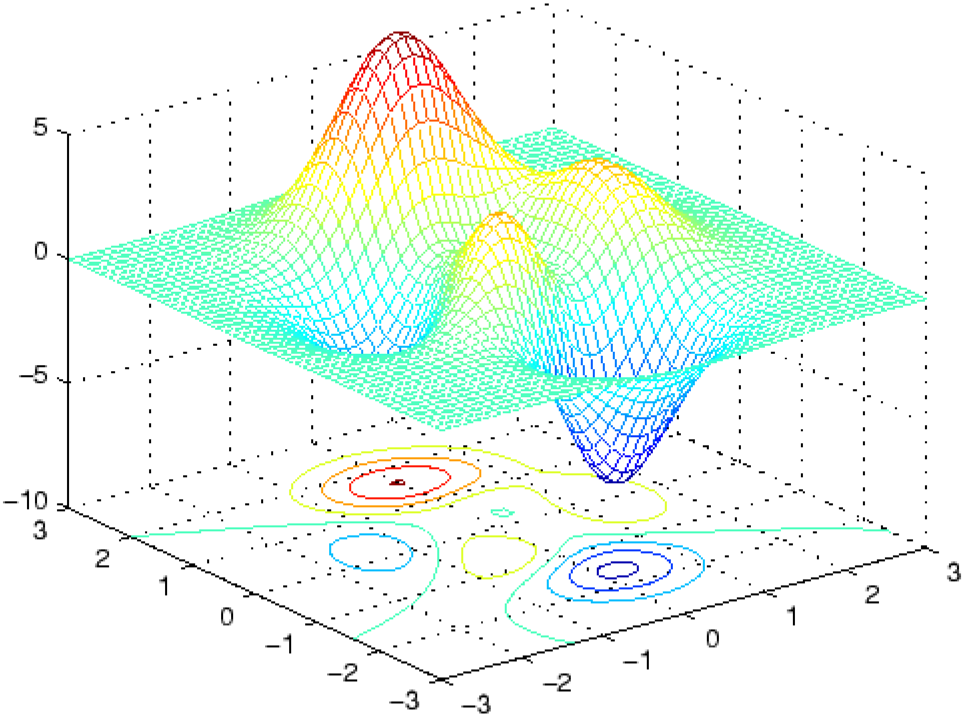
`mesh`, `meshc`, and `meshz` do not accept complex inputs.

A mesh is drawn as a surface graphics object with the viewpoint specified by `view(3)`. The face color is the same as the background color (to simulate a wireframe with hidden-surface elimination), or `none` when drawing a standard see-through wireframe. The current colormap determines the edge color. The `hidden` command controls the simulation of hidden-surface elimination in the mesh, and the `shading` command controls the shading model.

Examples

Produce a combination mesh and contour plot of the `peaks` surface:

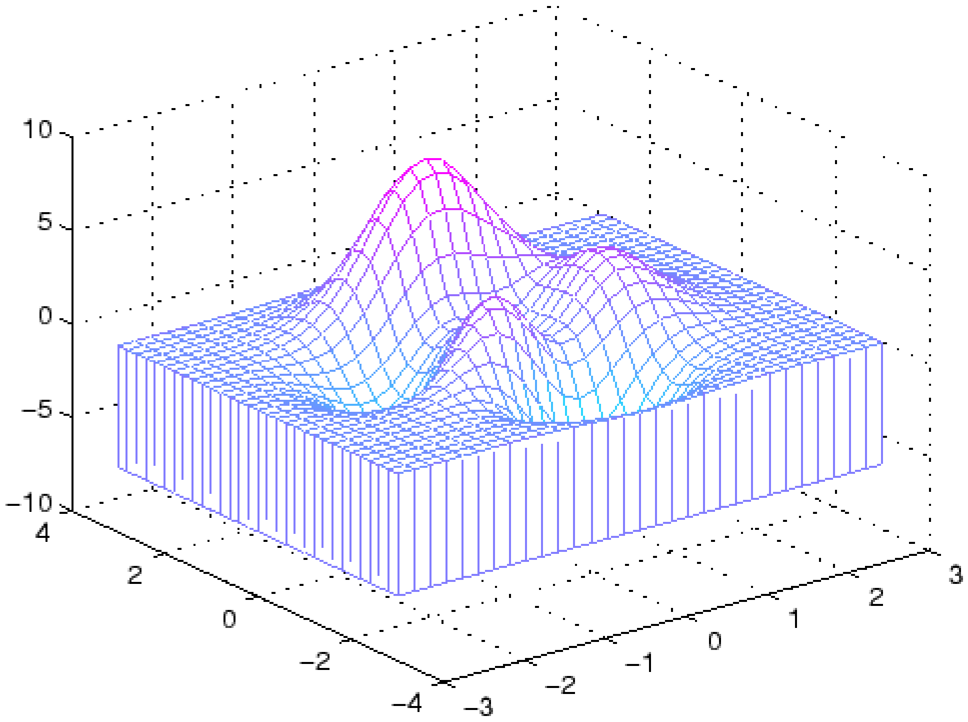
```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshc(X,Y,Z);
axis([-3 3 -3 3 -10 5])
```



Generate the curtain plot for the `peaks` function:

```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
```

```
meshz(X,Y,Z)
```



Algorithm

The range of `X`, `Y`, and `Z`, or the current settings of the axes `XLimMode`, `YLimMode`, and `ZLimMode` properties, determine the axis limits. `axis` sets these properties.

The range of `C`, or the current settings of the axes `CLim` and `CLimMode` properties (also set by the `caxis` function), determine the color scaling. The scaled color values are used as indices into the current colormap.

The mesh rendering functions produce color values by mapping the *z* data values (or an explicit color array) onto the current colormap. The MATLAB default behavior is to compute the color limits automatically using the minimum and maximum data values (also set using `caxis` `auto`). The minimum data value maps to the first color value in the colormap and the maximum data value maps to the last color value in the colormap. MATLAB performs a linear transformation on the intermediate values to map them to the current colormap.

`meshc` calls `mesh`, turns `hold` on, and then calls `contour` and positions the contour on the *x*-*y* plane. For additional control over the appearance of the contours, you can issue these commands directly. You can combine other types of graphs in this manner, for example `surf` and `pcolor` plots.

`meshc` assumes that `X` and `Y` are monotonically increasing. If `X` or `Y` is irregularly spaced, `contour3` calculates contours using a regularly spaced contour grid, then transforms the data to `X` or `Y`.

See Also

`contour`, `hidden`, `meshgrid`, `surface`, `surf`, `surfc`, `surfl`, `waterfall`

"Surface and Mesh Creation" on page 1-102 for related functions

`Surfaceplot Properties` for a list of surfaceplot properties

The functions `axis`, `caxis`, `colormap`, `hold`, `shading`, and `view` all set graphics object properties that affect `mesh`, `meshc`, and `meshz`.

For a discussion of parametric surfaces plots, refer to `surf`.

Purpose

Generate `X` and `Y` arrays for 3-D plots

Syntax

```
[X,Y] = meshgrid(x,y)
[X,Y] = meshgrid(x)
[X,Y,Z] = meshgrid(x,y,z)
```

Description

`[X,Y] = meshgrid(x,y)` transforms the domain specified by vectors `x` and `y` into arrays `X` and `Y`, which can be used to evaluate functions of two variables and three-dimensional mesh/surface plots. The rows of the output array `X` are copies of the vector `x`; columns of the output array `Y` are copies of the vector `y`.

`[X,Y] = meshgrid(x)` is the same as `[X,Y] = meshgrid(x,x)`.

`[X,Y,Z] = meshgrid(x,y,z)` produces three-dimensional arrays used to evaluate functions of three variables and three-dimensional volumetric plots.

Remarks

The `meshgrid` function is similar to `ndgrid` except that the order of the first two input and output arguments is switched. That is, the statement

```
[X,Y,Z] = meshgrid(x,y,z)
```

produces the same result as

```
[Y,X,Z] = ndgrid(y,x,z)
```

Because of this, `meshgrid` is better suited to problems in two- or three-dimensional Cartesian space, while `ndgrid` is better suited to multidimensional problems that aren't spatially based.

`meshgrid` is limited to two- or three-dimensional Cartesian space.

Examples

```
[X,Y] = meshgrid(1:3,10:14)

X =

     1     2     3
     1     2     3
```

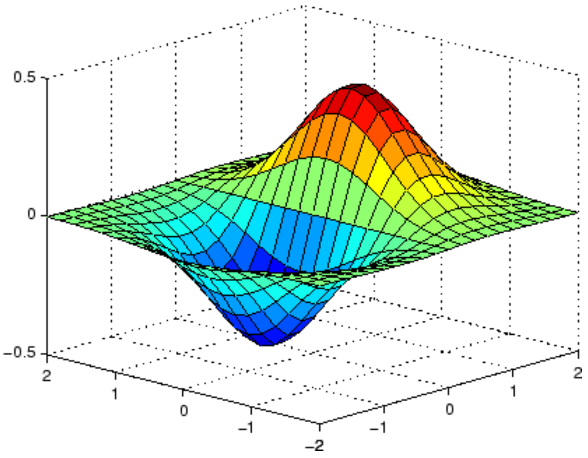
```
     1     2     3
     1     2     3
     1     2     3

Y =

    10    10    10
    11    11    11
    12    12    12
    13    13    13
    14    14    14
```

The following example shows how to use meshgrid to create a surface plot of a function.

```
[X,Y] = meshgrid(-2:.2:2, -2:.2:2);
Z = X .* exp(-X.^2 - Y.^2);
surf(X,Y,Z)
```



See Also

`griddata`, `mesh`, `ndgrid`, `slice`, `surf`

Purpose

`meta.class` class describes MATLAB classes

Description

Instances of the `meta.class` class contains information about MATLAB classes. The read/write properties of the `meta.class` class correspond to class attributes and are set only from within class definitions on the `classdef` line. You can query the read–only properties of the `meta.class` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class).

You cannot instantiate a `meta.class` object directly. You can construct a `meta.class` object from an instance of a class or using the class name:

- `metaclass` — returns a `meta.class` object representing the object passed as an argument.
- `?`*`classname`* — returns a `meta.class` object representing the named class.
- `fromName` — static method returns a `meta.class` object representing the named class.

For example, the `metaclass` function returns the `meta.class` object representing `myclass`.

```
ob = myclass;
obmeta = metaclass(ob);
obmeta.Name
ans =
myclass
```

You can use the class name to obtain the `meta.class` object:

```
obmeta = ?myclass;
```

You can also use the `fromName` static method:

```
obmeta = meta.class.fromName('myclass');
```

Properties

Property	Purpose
`ConstructOnLoad` attribute, default = `false`	If `true`, the class constructor is called automatically when loading an object from a MAT-file. Therefore, the construction must be implemented so that calling it with no arguments does not produce an error. See
`ContainingPackage` read only	A `meta.package` object describing the package within which this class is contained, or an empty object if this class is not in a package. See .
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used
`Events` read only	A cell array of `meta.event` objects describing each event defined by this class, including all inherited events. See .
`Hidden` attribute, default = `false`	If set to `true`, the class does not appear in the output of MATLAB commands or tools that display class names.
`InferiorClasses` attribute, default = `{}`	A cell array of `meta.class` objects defining the precedence of classes represented by the list as inferior to this class. See

Property	Purpose
`Methods` read only	A cell array of `meta.method` objects describing each method defined by this class, including all inherited public and protected methods. See .
`Name` read only	Name of the class associated with this `meta.class` object (`char` array)
`Properties` read only	A cell array of `meta.property` objects describing each property defined by this class, including all inherited public and protected properties. See .
`Sealed` attribute, default = `false`	If `true`, the class can be not be specialized with subclasses.
`SuperClasses` read only	A cell array of `meta.class` objects describing each superclass from which this class is derived. See .

Methods

Method	Purpose
`fromName`	Returns the `meta.class` object associated with the specified class name.
`tf = eq(cls)`	Equality function (`a == b`). Use to test if two variables refer to equal classed (classes that contain exactly the same list of elements).
`tf = ne(cls)`	Not equal function (`a ~= b`). Use to test if two variables refer to different meta–classes.

Method	Purpose
`tf = lt(clsa,clsb)`	Less than function (`clsa < clsb`). Use to determine if `clsa` is a strict subclass of `clsb` (i.e., `clsb < clsb` is `false`).
`tf = le(clsa,clsb)`	Less than or equal to function (`clsa <= clsb`). Use to determine if `clsa` is a subclass of `clsb`.
`tf = gt(clsa,clsb)`	Greater than function (`clsb > clsa`). Use to determine if `clsb` is a strict superclass of `clsa` (i.e., `clsb > clsb` is `false`).
`tf = ge(clsa,clsb)`	Greater than or equal to function (`clsb >= clsa`). Use to determine if `clsb` is a superclass of `clsa`.

See Also

See , `fromName`, `meta.property`, `meta.method`, `meta.event`, `meta.package`

meta.class.fromName

Purpose

Return `meta.class` object associated with named class

Syntax

`mcls = meta.class.fromName('className')`

Description

`mcls = meta.class.fromName('className')` is a static method that returns the `meta.class` object associated with the class `className`. Note that you can also use the `?` operator to obtain the `meta.class` object for a class name:

```
mcls = ?className;
```

The equivalent call to `fromName` is:

```
mcls = meta.class.fromName('className');
```

See Also

`meta.class`

meta.DynamicProperty

Purpose

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

Description

The `meta.DynamicProperty` class contains descriptive information about dynamic properties that you have added to an instance of a MATLAB classes. The MATLAB class must be a subclass of `dynamicprops`. The properties of the `meta.DynamicProperty` class correspond to property attributes that you specify from within class definitions. Dynamic properties are not defined in `classdef` blocks, but you can set their attributes by setting the `meta.DynamicProperty` object properties.

You add a dynamic property to an object using the `addprop` method of the `dynamicprops` class. The `addprop` method returns a `meta.DynamicProperty` instance representing the new dynamic property. You can modify the properties of the `meta.DynamicProperty` object to set the attributes of the dynamic property or to add set and get access methods, which would be defined in the `classdef` for regular properties.

You cannot instantiate the `meta.DynamicProperty` class. You must use `addprop` to obtain a `meta.DynamicProperty` object.

To remove the dynamic property, call the `delete` handle class method on the `meta.DynamicProperty` object.

Obtain a `meta.DynamicProperty` object from the `addprops` method, which returns an array of `meta.DynamicProperty` objects, one for each dynamic property.

See for more information.

Properties

Property	Purpose
`Name`	Name of the property.
`Description`	Can contain text
`DetailedDescription`	Can contain text

Property	Purpose
`Abstract` attribute, default = false	If `true`, the property has no implementation, but a concrete subclass must redefine this property without `Abstract` being set to `true`. • Abstract properties cannot define set or get access methods. See • Abstract properties cannot define initial values. • All subclasses must specify the same values as the superclass for the property `SetAccess` and `GetAccess` attributes. • `Abstract=true` should be used with the class attribute `Sealed=false` (the default).
`constant` attribute, default = `false`	Set to `true` if you want only one value for this property in all instances of the class. • Subclasses inherit constant properties, but cannot change them. • `Constant` properties cannot be `Dependent` • `SetAccess` is ignored. See
`GetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or derived classes `private` – access by class members only

Property	Purpose
`SetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or derived classes `private` – access by class members only
`Dependent` attribute, default = `false`	If `false`, property value is stored in object. If `true`, property value is not stored in object and the set and get functions cannot access the property by indexing into the object using the property name. See
`Transient` attribute, default = `false`	If `true`, property value is not saved when object is saved to a file. See for more about saving objects.
`Hidden` attribute, default = `false`	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to `properties`, etc.).
`GetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See
`SetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See
`GetMethod`	Function handle of the get method associated with this property. Empty if there is no get method specified. See

Property	Purpose
`SetMethod`	Function handle of the set method associated with this property. Empty if there is no set method specified. See
`DefiningClass`	The `meta.class` object representing the class that defines this property.

Events

See for information on using property events.

Event Name	Purpose
`PreGet`	Event occurs just before property is queried.
`PostGet`	Event occurs just after property has been queried
`PreSet`	Event occurs just before this property is modified
`PostSet`	Event occurs just after this property has been modified
`ObjectBeingDestroyed`	Inherited from `handle`

See Also

`addprop`, `handle`

meta.event

Purpose

`meta.event` class describes MATLAB class events

Description

The `meta.event` class provides information about MATLAB class events. The read/write properties of the `meta.event` class correspond to event attributes and are specified only from within class definitions.

You can query the read-only properties of the `meta.event` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class defining the event).

You cannot instantiate a `meta.event` object directly. Obtain a `meta.event` object from the `meta.class` `Events` property, which contains a cell array of `meta.event` objects, one for each event defined by the class. For example:

```
mco = ?classname;
eventcell = mco.Events;
eventcell{1}.Name; % name of first event
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	**Purpose**
`Name` read only	Name of the event.
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used
`Hidden`	If `true`, the event does not appear in the list of events returned by the `events` function (or other event listing functions or viewers)

Property	Purpose
`ListenAccess`	Determines where you can create listeners for the event. • `public` — unrestricted access • `protected` — access from methods in class or derived classes • `private` — access by class methods only (not from derived classes)
`NotifyAccess`	Determines where code can trigger the event. • `public` — any code can trigger event • `protected` — can trigger event from methods in class or derived classes • `private` — can trigger event by class methods only (not from derived classes)
`DefiningClass`	The `meta.class` object representing the class that defines this event.

See Also

`meta.class`, `meta.property`, `meta.method`, `metaclass`

meta.method

Purpose

`meta.method` class describes MATLAB class methods

Description

The `meta.method` class provides information about the methods of MATLAB classes. The read/write properties of the `meta.method` class correspond to method attributes and are specified only from within class definitions.

You can query the read-only properties of the `meta.method` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class defining a method).

You cannot instantiate a `meta.method` object directly. Obtain a `meta.method` object from the `meta.class` `Methods` property, which contains a cell array of `meta.method` objects, one for each class method. For example:

```
mco = ?classname;
methodcell = mco.Methods;
methodcell{1}.Name; % name of first method
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Abstract`	If `true`, the method has no implementation. The method has a syntax line that can include arguments, which subclasses use when implementing the method. • Subclasses are not required to define the same number of input and output arguments. • The method can have comments after the `function` line • Does not contain `function` or `end` keywords, only the function syntax (e.g., `[a,b] = myMethod(x,y)`)
`Access` attribute, default = `public`	Determines what code can call this method. • `public` — unrestricted access • `protected` — access from methods in class
`DefiningClass`	The `meta.class` object representing the class that defines this method.
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used
`Hidden` attribute, default = `false`	When `false`, the method name shows in the list of methods displayed using the `methods` or `methodsview` commands. If set to `true`, the method name is not included in these listings.
`Name` read only	Name of the method.

Property	**Purpose**
`Sealed` attribute, default = `false`	If `true`, the method cannot be redefined in a subclass. Attempting to define a method with the same name in a subclass causes an error.
`Static` attribute, default = `false`	Set to `true` to define a method that does not depend on an object of the class and does not require an object argument. You must use class name to call the method: *`classname.methodname`* See

See Also `meta.class`, `meta.property`, `meta.event`, `metaclass`

Purpose

`meta.package` class describes MATLAB packages

Description

The `meta.package` class contains information about MATLAB packages.

You cannot instantiate a `meta.package` object directly. Obtain a `meta.package` object from the `meta.class` `ContainingPackage` property, which contains a `meta.package` object, or an empty object, if the class is not in a package.

Properties

Property	Purpose
`Name` read only	Name of the package associated with this `meta.package` object
`Packages` read only	List of packages that are scoped to this package. A cell array of `meta.package` objects.
`Classes` read only	List of classes that are scoped to this package. A cell array of `meta.class` objects.
`Functions` read only	List of functions that are scoped to this package. A cell array of function handles.
`ContainingPackage` read only	A `meta.package` object describing the package within which this package is contained, or an empty object if this package is not nested.

Methods

Method	Purpose
`fromName`	Static method returns a `meta.package` object for a specified package name.
`getAllPackages`	Static method returns a cell array of `meta.package` objects representing all top-level packages.

See Also

See

`meta.class`, `meta.property`, `meta.method`, `meta.event`

meta.package.fromName

Purpose Return `meta.package` object for specified package

Syntax

```
mpkg = meta.package.fromName('pkgname')
```

Description `mpkg = meta.package.fromName('pkgname')` is a static method that returns the `meta.package` object associated with the named package. If `pkgname` is a nested package, then you must provide the fully qualified name (e.g., `'pkgname1.pkgname2'`).

Examples List the classes in the `event` package:

```
mev = meta.package.fromName('event');
for k=1:length(mev.Classes)
   disp(mev.Classes{k}.Name)
end
event.EventData
event.PropertyEvent
event.listener
event.proplistener
```

See Also `meta.package`, `meta.package.getAllPackages`

meta.package.getAllPackages

Purpose

Get all top-level packages

Syntax

```
P = meta.package.getAllPackages
```

Description

`P = meta.package.getAllPackages` is a static method that returns a cell array of `meta.package` objects representing all the top-level packages that are visible on the MATLAB path or defined as top-level built-in packages. You can access subpackages using the `Packages` property of each `meta.package` object.

Note that the time required to find all the packages on the path might be excessively long in some cases. You should therefore avoid using this method in any code where execution time is a consideration. `getAllPackages` is generally intended for interactive use only.

See Also

`meta.package`, `meta.package.fromName`

meta.property

Purpose

`meta.property` class describes MATLAB class properties

Description

The `meta.property` class provides information about the properties of MATLAB classes. The read/write properties of the `meta.property` class correspond to property attributes and are specified only from within your class definitions.

You can query the read-only properties of the `meta.property` object to obtain information that is specified syntactically by the class (for example, to obtain the function handle of a property's set access method).

You cannot instantiate a `meta.property` object directly. Obtain a `meta.property` object from the `meta.class` `Properties` property, which contains a cell array of `meta.property` objects, one for each class property. For example:

```
mco = ?classname;
propcell = mco.Properties;
propcell{1}.Name; % name of first property
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Name` read only	Name of the property.
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used

Property	Purpose
`AbortSet` attribute, default = false	If `true`, and this property belongs to a handle class, then MATLAB does not set the property value if the new value is the same as the current value. This prevents the triggering of property `PreSet` and `PostSet` events. See
`Abstract` attribute, default = false	If `true`, the property has no implementation, but a concrete subclass must redefine this property without `Abstract` being set to `true`. • Abstract properties cannot define set or get access methods. See • Abstract properties cannot define initial values. • All subclasses must specify the same values as the superclass for the property `SetAccess` and `GetAccess` attributes. • `Abstract=true` should be used with the class attribute `Sealed=false` (the default).
`constant` attribute, default = `false`	Set to `true` if you want only one value for this property in all instances of the class. • Subclasses inherit constant properties, but cannot change them. • `Constant` properties cannot be `Dependent` • `SetAccess` is ignored. See

Property	Purpose
`GetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or derived classes `private` – access by class members only
`SetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or derived classes `private` – access by class members only
`Dependent` attribute, default = `false`	If `false`, property value is stored in object. If `true`, property value is not stored in object and the set and get functions cannot access the property by indexing into the object using the property name. See
`Transient` attribute, default = `false`	If `true`, property value is not saved when object is saved to a file. See for more about saving objects.
`Hidden` attribute, default = `false`	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to `properties`, etc.).
`GetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See
`SetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See

Property	**Purpose**
`GetMethod` read only	Function handle of the get method associated with this property. Empty if there is no get method specified. See
`SetMethod` read only	Function handle of the set method associated with this property. Empty if there is no set method specified. See
`DefiningClass`	The `meta.class` object representing the class that defines this property.

Events

See for information on using property events.

Event Name	**Purpose**
`PreGet`	Event occurs just before property is queried.
`PostGet`	Event occurs just after property has been queried
`PreSet`	Event occurs just before this property is modified
`PostSet`	Event occurs just after this property has been modified

See Also

`meta.class`, `meta.method`, `meta.event`, `metaclass`

metaclass

Purpose

Obtain `meta.class` object

Syntax

```
mc = metaclass(object)
mc = ?classname
```

Description

`mc = metaclass(object)` returns the `meta.class` object for the class of `object`. The `object` input argument can be a scalar or an array of objects. However, `metaclass` always returns a scalar `meta.class` object.

`mc = ?`*`classname`* returns the `meta.class` object for the class with name *`classname`*. The `?` operator works only with a class name, not an object.

If you pass a class name as a string to the `metaclass` function, it returns the `meta.class` object for the `char` class. Use the `?` operator or the `meta.class.fromName` method to obtain the `meta.class` object from a class name.

Examples

Return the `meta.class` object for an instance of the `MException` class:

```
obj = MException('Msg:ID','MsgTxt');
mc = metaclass(obj);
```

Use the `?` operator to get the `meta.class` object for the `hgsetget` class:

```
mc = ?hgsetget;
```

See Also

`meta.class` | `meta.class.fromName`

Tutorials

-

Purpose

Class method names

Syntax

```
methods('classname')
methods(...,'-full')
m = methods(...)
```

Description

`methods('classname')` displays the names of the methods for the class `classname`. If `classname` is a MATLAB or Java class, then `methods` displays only public methods, including those methods inherited from superclasses.

`methods(...,'-full')` displays a full description of the methods, including inheritance information and, for MATLAB and Java methods, method attributes and signatures. `methods` does not remove duplicate method names with different signatures. Do not use this option with classes defined before MATLAB 7.6.

`m = methods(...)` returns the method names in a cell array of strings.

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

Examples

Retrieve the names of the static methods in class `MException`:

```
methods('MException')

Methods for class MException:

addCause       getReport      ne             throw
eq             isequal        rethrow        throwAsCaller

Static methods:

last
```

See Also

`methodsview` | `properties` | `events` | `what` | `which`

Tutorials

-

methodsview

Purpose View class methods

Syntax

```
methodsview packagename.classname
methodsview classname
methodsview(object)
```

Description

`methodsview packagename.classname` displays information about the methods in the class, `classname`. If the class is in a package, include `packagename`. If `classname` is a MATLAB or Sun Java class, `methodsview` lists only public methods, including those methods inherited from superclasses.

`methodsview classname` displays information describing the class `classname`.

`methodsview(object)` displays information about the methods of the class of `object`.

`methodsview` creates a window that displays the methods defined in the specified class. `methodsview` provides additional information like arguments, returned values, and superclasses. It also includes method qualifiers (for example, `abstract` or `synchronized`) and possible exceptions thrown.

Examples

List information on all methods in the `java.awt.MenuItem` class:

```
methodsview java.awt.MenuItem
```

MATLAB displays this information in a new window.

See Also

`methods` | `import` | `class` | `javaArray`

Purpose Compile MEX-function from C/ C++ or Fortran source code

Syntax

```
mex -help
mex -setup
mex filenames
mex options filenames
```

Description `mex` **`-help`** displays the M-file help for `mex`.

`mex` **`-setup`** lets you select or change the compiler configuration. MATLAB software searches for installed compilers and allows you to choose an options file as the default for future invocations of `mex`. For more information, see and `mex.getCompilerConfigurations`.

`mex` *`filenames`* compiles and links one or more C/C++ or Fortran source files specified in `filenames` into a shared library called a binary MEX-file from MATLAB.

`mex` *`options filenames`* compiles and links one or more source files specified in `filenames` using one or more of the specified command-line `options`.

The MEX-file has a platform-dependent extension. Use the `mexext` function to return the extension for the current machine or for all supported platforms.

`filenames` can be any combination of source files, object files, and library files. Include both the file name and the file extension in `filenames`. A non-source-code `filenames` parameter is passed to the linker without being compiled.

All valid command-line `options` are shown in the MEX Script Switches on page 2-2362 table. These options are available on all platforms except where noted.

`mex` also can build executable files for stand alone MATLAB engine and MAT-file applications. For more information, see “Engine/MAT Stand Alone Application Details” on page 2-2367.

You can run `mex` from the MATLAB Command Prompt, the Microsoft Windows Command Prompt, or the UNIX[13] shell. `mex` is a script named `mex.bat` on Windows systems and `mex` on UNIX systems. It is located in the `*matlabroot*/bin` directory.

The first file listed in `filenames` becomes the name of the binary MEX-file. You can list other source, object, or library files as additional `filenames` parameters to satisfy external references.

`mex` uses an options file to specify variables and values that are passed as arguments to the compiler, linker, and other tools (e.g., the resource linker on Windows systems). For more information, see "Options File Details" on page 2-2367. The default name for the options file is `mexopts.bat` (Windows systems) or `mexopts.sh` (UNIX systems).

Command-line options to `mex` may supplement or override contents of the options file. For more information, see "Override Option Details" on page 2-2367.

For an up-to-date list of supported compilers, see the Supported and Compatible Compilers Web page.

MEX Script Switches

Switch	Function
`@*rsp_file*`	(Windows systems only) Include the contents of the text file `*rsp_file*` as command-line arguments to `mex`.

---

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

MEX Script Switches (Continued)

Switch	Function
`-`*`arch`*	Build an output file for architecture *`arch`*. To determine the value for *`arch`*, type `computer('arch')` at the MATLAB Command Prompt on the target machine. Valid values for *`arch`* depend on the architecture of the build platform. You can get this information from the **Help** menu, as described in in the Desktop Tools and Development Environment documentation.
`-argcheck`	(C functions only) Add argument checking. This adds code so arguments passed incorrectly to MATLAB API functions cause assertion failures.
`-c`	Compile only. Creates an object file, but not a binary MEX-file.
`-compatibleArrayDims`	Build a binary MEX-file using the MATLAB Version 7.2 array-handling API, which limits arrays to 2^31-1 elements. This option is the default, but in the future the `-largeArrayDims` option will be the default.
`-cxx`	(UNIX systems only) Use the C++ linker to link the MEX-file if the first source file is in C and there are one or more C++ source or object files. This option overrides the assumption that the first source file in the list determines which linker to use.

MEX Script Switches (Continued)

Switch	Function
`-D`*`name`*	Define a symbol name to the C preprocessor. Equivalent to a `#define` *`name`* directive in the source. Do not add a space after this switch.
`-D`*`name=value`*	Define a symbol name and value to the C preprocessor. Equivalent to a `#define` *`name value`* directive in the source. Do not add a space after this switch.
`-f` *`optionsfile`*	Specify location and name of options file to use. Overrides the `mex` default-options-file search mechanism.
`-fortran`	(UNIX systems only) Specify that the gateway routine is in Fortran. This option overrides the assumption that the first source file in the list determines which linker to use.
`-g`	Create a binary MEX-file containing additional symbolic information for use in debugging. This option disables the `mex` default behavior of optimizing built object code (see the `-O` option).
`-h[elp]`	Print help for `mex`.
`-I`*`pathname`*	Add *`pathname`* to the list of directories to search for `#include` files. Do not add a space after this switch.

MEX Script Switches (Continued)

Switch	Function
`-inline`	Inline matrix accessor functions (`mx*`). This option is deprecated and will be removed in a future release. The generated MEX-function may not be compatible with future versions of MATLAB.
`-l`*`name`*	Link with object library. On Windows systems, *`name`* expands to *`name`*`.lib` or `lib`*`name`*`.lib` and on UNIX systems, to `lib`*`name`*`.so` or `lib`*`name`*`.dylib`. Do not add a space after this switch.
`-L`*`directory`*	Add *`directory`* to the list of directories to search for libraries specified with the `-l` option. The `-L` option must precede the `-l` option. On UNIX systems, you must also set the run-time library path, as explained in . Do not add a space after this switch.
`-largeArrayDims`	Build a binary MEX-file using the MATLAB large-array-handling API. This API can handle arrays with more than 2^31–1 elements when compiled on 64-bit platforms. (See also the `-compatibleArrayDims` option.)
`-n`	No execute mode. Print any commands that `mex` would otherwise have executed, but do not actually execute any of them.

MEX Script Switches (Continued)

Switch	Function
`-O`	Optimize the object code. Optimization is enabled by default and by including this option on the command line. If the `-g` option appears without the `-O` option, optimization is disabled.
`-outdir` *`dirname`*	Place all output files in directory *`dirname`*.
`-output` *`resultname`*	Create binary MEX-file named *`resultname`*. Automatically appends the appropriate MEX-file extension. Overrides the default MEX-file naming mechanism.
`-setup`	Specify the compiler options file to use when calling the `mex` function. When you use this option, all other command-line options are ignored.
`-U`*`name`*	Remove any initial definition of the C preprocessor symbol *`name`*. (Inverse of the `-D` option.) Do not add a space after this switch.
`-v`	Verbose mode. Print the values for important internal variables after the options file is processed and all command-line arguments are considered. Prints each compile step and final link step fully evaluated.
`name=value`	Override an options file variable for variable *`name`*. For examples, see Override Option Details in the Remarks section of the `mex` reference page.

Remarks

Options File Details

MATLAB provides template options files for the compilers that are supported by `mex`. These templates are located in the *`matlabroot`*`\bin\win32\mexopts` or the *`matlabroot`*`\bin\win64\mexopts` directories on Windows systems, or the *`matlabroot`*`/bin` directory on UNIX systems. These template options files are used by the `-setup` option to define the selected default options file.

Override Option Details

Use the *`name`*`=`*`value`* command-line argument to override a variable specified in the options file at the command line. When using this option, you may need to use the shell's quoting syntax to protect characters such as spaces, which have a meaning in the shell syntax.

This option is processed after the options file is processed and all command line arguments are considered.

On Windows platforms, at either the MATLAB prompt or the DOS prompt, use double quotes ("). For example:

```
mex -v COMPFLAGS="$COMPFLAGS -Wall" LINKFLAGS="$LINKFLAGS /VERBOSE"
```

At the MATLAB command line on UNIX platforms, use double quotes ("). Use the backslash (\) escape character before the dollar sign ($). For example:

```
mex -v CFLAGS="\$CFLAGS -Wall" LDFLAGS="\$LDFLAGS-w" yprime.c
```

At the shell command line on UNIX platforms, use single quotes ('). For example:

```
mex -v CFLAGS='$CFLAGS -Wall' LDFLAGS='$LDFLAGS -w' yprime.c
```

Engine/MAT Stand Alone Application Details

`mex` can build executable files for stand alone MATLAB engine and MAT-file applications. For these applications, `mex` does not use the default options file; you must use the `-f` option to specify an options file.

The options files used to generate stand alone MATLAB engine and MAT-file executables are named `*engmatopts.bat` on Windows systems, or `engopts.sh` and `matopts.sh` on UNIX systems, and are located in the same directory as the template options files referred to above in Options File Details.

Examples

Compiling a C File

The following command compiles `yprime.c`:

```
mex yprime.c
```

Using Verbose Mode

When debugging, it is often useful to use verbose mode, as well as include symbolic debugging information:

```
mex -v -g yprime.c
```

Overriding Command Line Options

For examples, see “Override Option Details” on page 2-2367.

See Also

`computer`, `dbmex`, `inmem`, `loadlibrary`, `mexext`, `pcode`, `prefdir`, `system`, `mex.getCompilerConfigurations`

Purpose

Get compiler configuration information for building MEX-files

Syntax

```
cc = mex.getCompilerConfigurations()
cc = mex.getCompilerConfigurations('lang')
cc = mex.getCompilerConfigurations('lang','list')
```

Description

`cc = mex.getCompilerConfigurations()` returns a `mex.CompilerConfiguration` object `cc` containing information about the selected compiler configuration used by `mex`. The selected compiler is the one you choose when you run the `mex -setup` command. For details about the `mex.CompilerConfiguration` class, see "mex.CompilerConfiguration" on page 2-2370.

`cc = mex.getCompilerConfigurations('`*`lang`*`')` returns an array of `mex.CompilerConfiguration` objects `cc` containing information about the selected configuration for the given *`lang`*. If the language of the selected complier is different from *`lang`*, then `cc` is empty.

Language *`lang`* is a string with one of the following values:

- **`'Any'`** — All supported languages. This is the default value.
- **`'C'`** — All C compiler configurations, including C++ configurations.
- **`'C++'`** or **`'CPP'`** — All C++ compiler configurations.
- **`'Fortran'`** — All Fortran compiler configurations.

`cc = mex.getCompilerConfigurations('`*`lang`*`','`*`list`*`')` returns an array of `mex.CompilerConfiguration` objects `cc` containing information about configurations for the given language and the given *`list`*. Values for *`list`* are:

- **`'Selected'`** — The compiler you choose when you run `mex-setup`. This is the default value.
- **`'Installed'`** — All supported compilers `mex` finds installed on your system.

- `'Supported'` — All compilers supported in the current release. For an up-to-date list of supported compilers, see the Supported and Compatible Compilers Web page.

Classes

mex.CompilerConfiguration

The `mex.CompilerConfiguration` class contains the following read-only properties about compiler configurations.

Property	Purpose
`Name`	Name of the compiler
`Manufacturer`	Name of the manufacturer of the compiler
`Language`	Compiler language
`Version`	(Windows platforms only) Version of the compiler
`Location`	(Windows platforms only) Directory where compiler is installed
`Details`	A `mex.CompilerConfigurationDetails` object containing specific information about build options. For details about this class, see "mex.CompilerConfigurationDetails" on page 2-2370.

mex.CompilerConfigurationDetails

The `mex.CompilerConfigurationDetails` class provides information about the command options used by the compiler, linker and other build programs used to create MEX-files. These properties are read-only.

Property
`CompilerExecutable`
`CompilerFlags`
`OptimizationFlags`

Property
`DebugFlags`
`LinkerExecutable`
`LinkerFlags`
`LinkerOptimizationFlags`
`LinkerDebugFlags`

Examples

Selected Compiler Example

```
myCompiler = mex.getCompilerConfigurations()
```

MATLAB software displays information similar to the following (depending on your architecture, your version of MATLAB, and what you selected when you ran `mex -setup`):

```
myCompiler =

  mex.CompilerConfiguration
  Package: mex

  Properties:
            Name: 'Microsoft Visual C++ 2005'
    Manufacturer: 'Microsoft'
        Language: 'C++'
         Version: '8.0'
        Location: '%VS80COMNTOOLS%\..\..'
         Details: [1x1 mex.CompilerConfigurationDetails]

  Methods
```

Supported Compiler Configurations Example

```
allCC = mex.getCompilerConfigurations('Any','Supported')
```

MATLAB displays information similar to the following:

```
allCC =

  1x11 mex.CompilerConfiguration
  Package: mex

  Properties:
    Name
    Manufacturer
    Language
    Version
    Location
    Details

  Methods
```

This version of MATLAB supports eleven configurations, hence, `allCC` is a 1-by-11 matrix.

Supported C Compilers Example

To see what C compilers MATLAB supports, type:

```
cLanguageCC = mex.getCompilerConfigurations('C','Supported')
```

MATLAB displays the following information (the number of compilers for your version of MATLAB may be different):

```
cLanguageCC =

  1x9 mex.CompilerConfiguration
  Package: mex

  Properties:
    Name
    Manufacturer
    Language
    Version
```

```
      Location
      Details

    Methods
```

To display the compiler names, type:

```
format compact
cLanguageCC.Name
```

MATLAB displays information similar to the following:

```
ans =
Intel C++
ans =
Lcc-win32
ans =
Microsoft Visual C++
ans =
Microsoft Visual C++ 2003
ans =
Microsoft Visual C++ 2005 Express Edition
ans =
Microsoft Visual C++ 2005
ans =
Microsoft Visual C++ 2008
ans =
Open WATCOM C/C++
ans =
Open WATCOM C/C++
```

Example — Viewing Build Options for a C Compiler

To see what build options MATLAB uses with a particular C compiler, create an array `CC` of all supported C compiler configurations:

```
CC = mex.getCompilerConfigurations('C','Supported');
disp('Compiler Name')
```

```
for i = 1:3; disp(CC(i).Name); end;
```

MATLAB displays a list similar to:

```
Intel C++
Lcc-win32
Microsoft Visual C++
```

To see the build options for the Microsoft Visual C++ compiler, type:

```
CC(3).Details
```

MATLAB displays information similar to the following (output is formatted):

```
ans =
  mex.CompilerConfigurationDetails
  Package: mex

  Properties:
         CompilerExecutable: 'cl'
              CompilerFlags: '-c -Zp8 -G5 -W3 -EHs
                             -DMATLAB_MEX_FILE -nologo /MD'
          OptimizationFlags: '-O2 -Oy- -DNDEBUG'
                 DebugFlags: '-Zi
                             -Fd"%OUTDIR%%MEX_NAME%.pdb"'
           LinkerExecutable: 'link'
                LinkerFlags: [1x258 char]
    LinkerOptimizationFlags: ''
           LinkerDebugFlags: '/debug'
  Methods
```

See Also

`mex`

Purpose

Capture error information

Syntax

`exception = MException(msgIdent, msgString, v1, v2, ...)`

Description

`exception = MException(msgIdent, msgString, v1, v2, ...)` captures information about a specific error that has occurred and stores it in error record *`exception`*. Information stored in the record includes a message identifier *`msgIdent`* and an error message string *`msgString`*. Optional arguments v_1, v_2, `...` represent additional values you would like MATLAB to add to the error message string at run time.

Message identifier *`msgIdent`* is a character string composed of two substrings, the *`component`* and the *`mnemonic`*, separated by a colon (e.g., *`component`*`:`*`mnemonic`*). The purpose of the identifier is to better identify the source of the error. See the documentation on for more information.

Message string *`msgString`* is a character string that informs the user about the cause of the error and can also suggest how to correct the faulty condition. *`msgString`* can include predefined escape sequences, such as `\n` for newline, and conversion specifiers, such as `%d` for a decimal number.

Inputs v_1, v_2, `...` represent numeric values or substrings that are to replace conversion specifiers used in the `msgString` input. The format is the same as that used with the `sprintf` function. v_1 replaces the first conversion specifier in *`msgString`*, v_2 replaces the second, and so on. For example, if *`msgString`* is “Error on line `%d`, command `%s`”, then v_1 is the line number at which the error was detected, and v_2 is the command that failed. The v_n arguments replace the conversion specifiers at the time of execution.

The *`exception`* output is an object of the `MException` class. `MException` is the constructor for this class. In addition to calling the constructor directly, you can also create an object of `MException` with any of the following functions: `error`, `assert`, `throw`, `rethrow`, and `throwAsCaller`. See the documentation and figure in the section for more information on this class.

Properties

The `MException` object has four properties: `identifier`, `message`, `stack`, and `cause`. Click any of the links below to find out more about `MException` properties:

Property	Description
`identifier`	Identifies the error.
`message`	Formatted error message that is displayed.
`stack`	Structure containing stack trace information such as M-file function name and line number where the `MException` was thrown.
`cause`	Cell array of `MException` that caused this exception to be created.

Methods

The `MException` object has the following methods. Click any of the links below to find out more about `MException` methods:

Method	Description
`addCause`	Appends an `MException` to the `cause` field of another `MException.`
`getReport`	Returns a formatted message string based on the current exception that uses the same format as errors thrown by internal MATLAB code.
`last`	Returns an `MException` object for the most recently thrown exception.
`rethrow`	Reissues an exception that has been caught, causing the program to stop.
`throw`	Issues an exception from the currently running M-file.
`throwAsCaller`	Issues an exception from the currently running M-file, also omitting the current stack frame from the `stack` field of the `MException`.

Remarks

Valid escape sequences for the `msgString` argument are `\b`, `\f`, `\n`, `\r`, `\t`, and `\x` or `\` when followed by a valid hexadecimal or octal number, respectively. Following a backslash in the `msgString` with any other character causes MATLAB to issue a warning. Conversion specifiers are similar to those used in the C programming language and in the `sprintf` function.

All string input arguments must be enclosed in single quotation marks. If `msgString` is an empty string, the `error` command has no effect.

Examples

Example 1 — Formatted Messages

If your message string requires formatting specifications like those used with the `sprintf` function, you can use this syntax to compose the error message string:

```
exception = MException(msgIdent, msgString, v1, v2, ...)
```

For example,

```
exception = MException('AcctError:Incomplete', ...
      'Field ''%s.%s'' is not defined.', ...
      'Accounts', 'ClientName');

exception.message
ans =
    Field 'Accounts.ClientName' is not defined.
```

Example 2 — Error Recovery

This example reads the contents of an image file. The attempt to open and then read the file is done in a `try` block. If either the open or read fails, the program catches the resulting exception and saves the `MException` object in the variable `exception1`.

The `catch` block in this example checks to see if the specified file could not be found. If this is the case, the program allows for the possibility that a common variation of the file name extension (e.g., `jpeg` instead of `jpg`) was used by retrying the operation with a modified extension.

This is done using a `try-catch` statement that is nested within the original `try-catch`.

```
function d_in = read_image(filename)
[path name ext] = fileparts(filename);
try
   fid = fopen(filename, 'r');
   d_in = fread(fid);
catch exception_1
   % Get last segment of the error message identifier.
   idSegLast = regexp(exception_1.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(exception_1);
      end

      % Try again, with modifed filenames.
      try
         fid = fopen(filename, 'r');
         d_in = fread(fid);
      catch exception_2
```

```
            fprintf('Unable to access file %s\n', filename);
            exception_2 = addCause(exception_2, exception_1);
            rethrow(exception_2)
         end
      end
   end
```

Example 3 — Nested try-catch

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

```
function data = read_it(filename);
try
   fid = fopen(filename, 'r');
   data = fread(fid);
catch exception_1
   if strcmp(exception_1.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(exception_1);
      end
      addpath(newdir);
      try
         fid = fopen(filename, 'r');
         data = fread(fid);
      catch exception_2
         exception_3 = addCause(exception_2, exception_1)
         throw(exception_3);
      end
      rmpath(newdir);
   end
```

```
end
fclose(fid);
```

If you run this function in a `try`-`catch` block at the command line, you can look at the `MException` object by assigning it to a variable (`e`) with the `catch` command.

See Also

, `try`, `catch`, `error`, `assert`, `throw(MException)`, `rethrow(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `getReport(MException)`,`last(MException)`, `dbstack`

Purpose

Binary MEX-file name extension

Syntax

```
ext = mexext
extlist = mexext('all')
```

Description

`ext = mexext` returns the file name extension for the current platform.

`extlist = mexext('all')` returns a `struct` with fields `arch` and `ext` describing MEX-file name extensions for the all platforms.

Remarks

For a table of file extensions, see .

Examples

Find the MEX-file extension for the system you are currently working on:

```
ext = mexext

ext =
   mexw32
```

Find the MEX-file extension for an Apple Macintosh system:

```
extlist = mexext('all');

for k=1:length(extlist)
   if strcmp(extlist(k).arch, 'maci')
   disp(sprintf('Arch: %s      Ext: %s', ...
            extlist(k).arch, extlist(k).ext))
end, end

Arch: maci       Ext: mexmaci
```

See Also

`mex`

mfilename

Purpose Name of currently running M-file

Syntax

```
mfilename
p = mfilename('fullpath')
c = mfilename('class')
```

Description `mfilename` returns a string containing the name of the most recently invoked M-file. When called from within an M-file, it returns the name of that M-file, allowing an M-file to determine its name, even if the filename has been changed.

`p = mfilename('fullpath')` returns the full path and name of the M-file in which the call occurs, not including the filename extension.

`c = mfilename('class')` in a method, returns the class of the method, not including the leading `@` sign. If called from a nonmethod, it yields the empty string.

Remarks If `mfilename` is called with any argument other than the above two, it behaves as if it were called with no argument.

When called from the command line, `mfilename` returns an empty string.

To get the names of the callers of an M-file, use `dbstack` with an output argument.

See Also `dbstack`, `function`, `nargin`, `nargout`, `inputname`

Purpose

Download file from FTP server

Syntax

```
mget(f,'filename')
mget(f,'dirname')
mget(...,'target')
```

Description

`mget(f,'filename')` retrieves `filename` from the FTP server `f` into the MATLAB current directory, where `f` was created using `ftp`.

`mget(f,'dirname')` retrieves the directory `dirname` and its contents from the FTP server `f` into the MATLAB current directory, where `f` was created using `ftp`. You can use a `wildcard` (`*`) in `dirname`.

`mget(...,'target')` retrieves the specified items from the FTP server `f`, where `f` was created using `ftp`, into the local directory specified by `target`, where `target` is an absolute path name.

Examples

Connect to an FTP server and retrieve the file `README` into the current MATLAB directory.

```
ftpobj = ftp('ftp.mathworks.com');
mget(ftpobj, 'README');
close(ftpobj);
```

See Also

`cd( ftp)`, `ftp`, `mput`

min

Purpose

Smallest elements in array

Syntax

```
C = min(A)
C = min(A,B)
C = min(A,[],dim)
[C,I] = min(...)
```

Description

`C = min(A)` returns the smallest elements along different dimensions of an array.

If `A` is a vector, `min(A)` returns the smallest element in `A`.

If `A` is a matrix, `min(A)` treats the columns of `A` as vectors, returning a row vector containing the minimum element from each column.

If `A` is a multidimensional array, `min` operates along the first nonsingleton dimension.

`C = min(A,B)` returns an array the same size as `A` and `B` with the smallest elements taken from `A` or `B`. The dimensions of `A` and `B` must match, or they may be scalar.

`C = min(A,[],dim)` returns the smallest elements along the dimension of `A` specified by scalar `dim`. For example, `min(A,[],1)` produces the minimum values along the first dimension (the rows) of `A`.

`[C,I] = min(...)` finds the indices of the minimum values of `A`, and returns them in output vector `I`. If there are several identical minimum values, the index of the first one found is returned.

Remarks

For complex input `A`, `min` returns the complex number with the smallest complex modulus (magnitude), computed with `min(abs(A))`. Then computes the smallest phase angle with `min(angle(x))`, if necessary.

The `min` function ignores `NaN`s.

See Also

`max`, `mean`, `median`, `sort`

Purpose

Minimum value of `timeseries` data

Syntax

```
ts_min = min(ts)
ts_min = min(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_min = min(ts)` returns the minimum value in the time-series data. When `ts.Data` is a vector, ts_min is the minimum value of `ts.Data` values. When `ts.Data` is a matrix, ts_min is a row vector containing the minimum value of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `min` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_min = min(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`.
  When you specify `'time'`, larger time values correspond to larger weights.

Examples

The following example illustrates how to find the minimum values in multivariate time-series data.

1. Load a 24-by-3 data array.

   ```
   load count.dat
   ```

2. Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,[1:24],'Name','CountPerSecond')
```

3 Find the minimum in each data column for this `timeseries` object.

```
min(count_ts)

ans =

     7     9     7
```

The minimum is found independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `max (timeseries)`, `median (timeseries)`, `mean (timeseries)`, `std (timeseries)`, `timeseries`, `var (timeseries)`

Purpose

Minimize size of Automation server window

Syntax

MATLAB Client

```
h.MinimizeCommandWindow
MinimizeCommandWindow(h)
```

IDL Method Signature

```
HRESULT MinimizeCommandWindow(void)
```

Microsoft Visual Basic Client

```
MinimizeCommandWindow
```

Description

`h.MinimizeCommandWindow` minimizes the window for the server attached to handle `h`, and makes it inactive.

`MinimizeCommandWindow(h)` is an alternate syntax.

If the server window was already in a minimized state, `MinimizeCommandWindow` does nothing.

Examples

From a MATLAB client, modify the size of the command window in a MATLAB Automation server:

```
h = actxserver('matlab.application');
h.MinimizeCommandWindow;
% Now return the server window to its former state on
% the desktop and make it the currently active window.
h.MaximizeCommandWindow;
```

From a Visual Basic client, modify the size of the command window in a MATLAB Automation server:

```
Dim Matlab As Object

Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow
```

```
'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

See Also

`MaximizeCommandWindow`

How To

-
-

Purpose

Minimum residual method

Syntax

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

Description

`x = minres(A,b)` attempts to find a minimum norm residual solution `x` to the system of linear equations `A*x=b`. The `n`-by-`n` coefficient matrix `A` must be symmetric but need not be positive definite. It should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x)` returns `A*x`. See 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 `minres` converges, a message to that effect is displayed. If `minres` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

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

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

`minres(A,b,tol,maxit,M)` and `minres(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(sqrt(M))*A*inv(sqrt(M))*y = inv(sqrt(M))*b` for `y` and then return `x = inv(sqrt(M))*y`. If `M` is `[]` then `minres` applies no preconditioner. `M` can be a function handle `mfun`, such that `mfun(x)` returns `M\x`.

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

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

Flag	Convergence
0	`minres` converged to the desired tolerance `tol` within `maxit` iterations.
1	`minres` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`minres` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `minres` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = minres(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

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

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

`[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...)` also returns a vector of estimates of the Conjugate Gradients residual norms at each iteration.

Examples

Example 1

```
n = 100; on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M1 = spdiags(4*on,0,n,n);

x = minres(A,b,tol,maxit,M1);
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in an M-file `run_minres` that

- Calls `minres` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_minres` are available to `afun`.

The following shows the code for `run_minres`:

```
function x1 = run_minres
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M = spdiags(4*on,0,n,n);
x1 = minres(@afun,b,tol,maxit,M);

       function y = afun(x)
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
```

```
        y(1:n-1) = y(1:n-1) - 2 * x(2:n);
      end
end
```

When you enter

```
x1=run_minres;
```

MATLAB software displays the message

```
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

Example 3

Use a symmetric indefinite matrix that fails with `pcg`.

```
A = diag([20:-1:1, -1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
```

displays the following message:

```
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, `minres` can handle the indefinite matrix `A`.

```
x = minres(A,b,1e-6,40);
minres converged at iteration 39 to a solution with relative
residual 1.3e-007
```

See Also

`bicg`, `bicgstab`, `cgs`, `cholinc`, `gmres`, `lsqr`, `pcg`, `qmr`, `symmlq`

`function_handle` (`@`), `mldivide` (`\`)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

mislocked

Purpose Determine whether M-file or MEX-file cannot be cleared from memory

Syntax

```
mislocked
mislocked(fun)
```

Description `mislocked` by itself returns logical `1` (`true`) if the currently running M-file or MEX-file is locked, and logical `0` (`false`) otherwise.

`mislocked(`*`fun`*`)` returns logical `1` (`true`) if the function named *`fun`* is locked in memory, and logical `0` (`false`) otherwise. Locked M-files and MEX-files cannot be removed with the `clear` function.

See Also `mlock`, `munlock`

Purpose

Make new folder

Graphical Interface

As an alternative to `mkdir`, use the Current Folder browser.

Syntax

```
mkdir('folderName')
mkdir('parentFolder','folderName')
status = mkdir(...)
[status,message,messageid] = mkdir(...)
```

Description

`mkdir('folderName')` creates the folder `folderName`, where `folderName` can be an absolute or a relative path.

`mkdir('parentFolder','folderName')` creates the folder `folderName` in `parentFolder`, where `parentFolder` is an absolute or relative path. If `parentFolder` does not exist, MATLAB attempts to create it. See the Remarks section.

`status = mkdir(...)` creates the specified folder. When the operation is successful, it returns a `status` of logical 1. When the operation is unsuccessful, it returns logical 0.

`[status,message,messageid] = mkdir(...)` creates the specified folder, and returns the status, message string, and MATLAB message ID. The value given to `status` is logical `1` for success, and logical `0` for error.

Remarks

If an argument specifies a path that includes one or more nonexistent folders, MATLAB attempts to create the nonexistent folder. For example, for

```
mkdir('myFolder\folder1\folder2\targetFolder')
```

if `folder1` does not exist, MATLAB creates `folder1`, creates `folder2` within `folder1`, and creates `targetFolder` within `folder2`.

Examples

Creating a Subfolder in the Current Folder

Create a subfolder called `newdir` in the current folder:

```
mkdir('newdir')
```

Creating a Subfolder in the Specified Parent Folder

Create a subfolder called `newFolder` in the folder `testdata`, using a relative path, where `newFolder` is at the same level as the current folder:

```
mkdir('../testdata','newFolder')
```

Returning Status When Creating a Folder

In this example, the first attempt to create `newFolder` succeeds, returning a status of `1`, and no error or warning message or message identifier:

```
[s, mess, messid] = mkdir('../testdata', 'newFolder')
s =
   1
mess =
   ''
messid =
   ''
```

Attempt to create the same folder again. `mkdir` again returns a success status, and also a warning and message identifier informing you that the folder exists:

```
[s,mess,messid] = mkdir('../testdata','newFolder')
s =
   1
mess =
   Directory "newFolder" already exists.
messid =
   MATLAB:MKDIR:DirectoryExists
```

See Also

`copyfile`, `cd`, `dir`, `ls`, `movefile`, `rmdir`

Purpose Create new directory on FTP server

Syntax `mkdir(f,'dirname')`

Description `mkdir(f,'dirname')` creates the directory `dirname` in the current directory of the FTP server `f`, where `f` was created using `ftp`, and `dirname` is a path name relative to the current directory on `f`.

Examples Connect to hypothetical server `testsite` and view the contents.

```
test=ftp('ftp.testsite.com');
dir(test)
```

Suppose that the contents include the following:

```
.            ..       otherfile.m         testfolder
```

Create the folder `newfolder` in `testfolder` and close the connection.

```
mkdir(test,'testfolder/newfolder');
close(test);
```

See Also `dir (ftp)`, `ftp`, `rmdir (ftp)`

mkpp

Purpose

Make piecewise polynomial

Syntax

```
pp = mkpp(breaks,coefs)
pp = mkpp(breaks,coefs,d)
```

Description

`pp = mkpp(breaks,coefs)` builds a piecewise polynomial `pp` from its breaks and coefficients. `breaks` is a vector of length `L+1` with strictly increasing elements which represent the start and end of each of `L` intervals. `coefs` is an `L`-by-`k` matrix with each row `coefs(i,:)` containing the coefficients of the terms, from highest to lowest exponent, of the order `k` polynomial on the interval `[breaks(i),breaks(i+1)]`.

`pp = mkpp(breaks,coefs,d)` indicates that the piecewise polynomial `pp` is `d`-vector valued, i.e., the value of each of its coefficients is a vector of length `d`. `breaks` is an increasing vector of length `L+1`. `coefs` is a `d`-by-`L`-by-`k` array with `coefs(r,i,:)` containing the `k` coefficients of the `i`th polynomial piece of the `r`th component of the piecewise polynomial.

Use `ppval` to evaluate the piecewise polynomial at specific points. Use `unmkpp` to extract details of the piecewise polynomial.

Note. The *order* of a polynomial tells you the number of coefficients used in its description. A *k*th order polynomial has the form

$$c_1 x^{k-1} + c_2 x^{k-2} + \ldots + c_{k-1} x + c_k$$

It has *k* coefficients, some of which can be 0, and maximum exponent *k*-1. So the order of a polynomial is usually one greater than its degree. For example, a cubic polynomial is of order 4.

Examples

The first plot shows the quadratic polynomial

$$1 - \left(\frac{x}{2} - 1\right)^2 = \frac{-x^2}{4} + x$$

shifted to the interval [-8,-4]. The second plot shows its negative

$$\left(\frac{x}{2}-1\right)^2-1=\frac{x^2}{4}-x$$

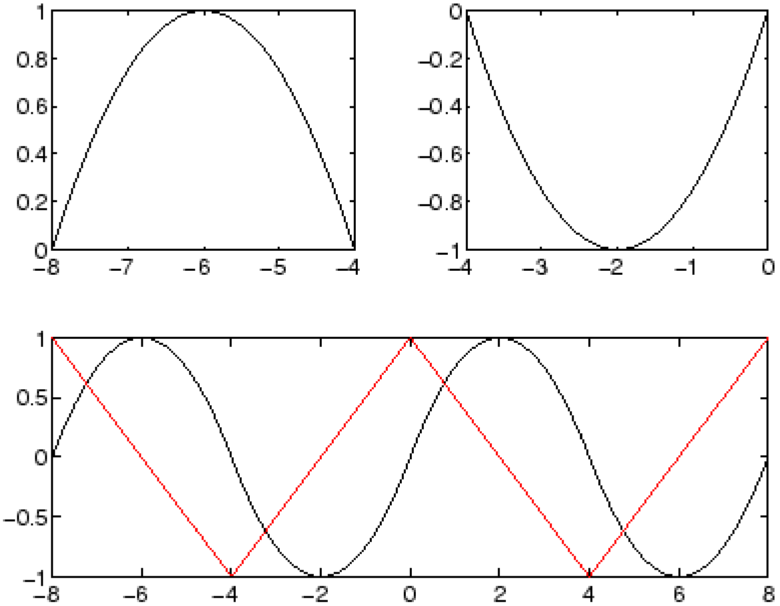
but shifted to the interval [-4,0].

The last plot shows a piecewise polynomial constructed by alternating these two quadratic pieces over four intervals. It also shows its first derivative, which was constructed after breaking the piecewise polynomial apart using `unmkpp`.

```
subplot(2,2,1)
cc = [-1/4 1 0];
pp1 = mkpp([-8 -4],cc);
xx1 = -8:0.1:-4;
plot(xx1,ppval(pp1,xx1),'k-')

subplot(2,2,2)
pp2 = mkpp([-4 0],-cc);
xx2 = -4:0.1:0;
plot(xx2,ppval(pp2,xx2),'k-')

subplot(2,1,2)
pp = mkpp([-8 -4 0 4 8],[cc;-cc;cc;-cc]);
xx = -8:0.1:8;
plot(xx,ppval(pp,xx),'k-')
[breaks,coefs,l,k,d] = unmkpp(pp);
dpp = mkpp(breaks,repmat(k-1:-1:1,d*l,1).*coefs(:,1:k-1),d);
hold on, plot(xx,ppval(dpp,xx),'r-'), hold off
```



See Also `ppval`, `spline`, `unmkpp`

Purpose Left or right matrix division

Syntax

```
mldivide(A,B)     A\B
mrdivide(B,A)     B/A
```

Description `mldivide(A,B)` and the equivalent `A\B` perform matrix left division (back slash). `A` and `B` must be matrices that have the same number of rows, unless `A` is a scalar, in which case `A\B` performs element-wise division — that is, `A\B = A.\B`.

If `A` is a square matrix, `A\B` is roughly the same as `inv(A)*B`, except it is computed in a different way. If `A` is an `n`-by-`n` matrix and `B` is a column vector with `n` elements, or a matrix with several such columns, then `X = A\B` is the solution to the equation $AX = B$ (see "Algorithm" on page 2-2405 for details). A warning message is displayed if `A` is badly scaled or nearly singular.

If `A` is an `m`-by-`n` matrix with `m ~= n` and `B` is a column vector with `m` components, or a matrix with several such columns, then `X = A\B` is the solution in the least squares sense to the under- or overdetermined system of equations $AX = B$. In other words, `X` minimizes `norm(A*X - B)`, the length of the vector $AX - B$. The rank `k` of `A` is determined from the QR decomposition with column pivoting (see "Algorithm" on page 2-2405 for details). The computed solution `X` has at most `k` nonzero elements per column. If `k < n`, this is usually not the same solution as `x = pinv(A)*B`, which returns a least squares solution.

`mrdivide(B,A)` and the equivalent `B/A` perform matrix right division (forward slash). `B` and `A` must have the same number of columns.

If `A` is a square matrix, `B/A` is roughly the same as `B*inv(A)`. If `A` is an `n`-by-`n` matrix and `B` is a row vector with `n` elements, or a matrix with several such rows, then `X = B/A` is the solution to the equation $XA = B$ computed by Gaussian elimination with partial pivoting. A warning message is displayed if `A` is badly scaled or nearly singular.

If `B` is an `m`-by-`n` matrix with `m ~= n` and `A` is a column vector with `m` components, or a matrix with several such columns, then `X = B/A` is the solution in the least squares sense to the under- or overdetermined system of equations $XA = B$.

Note Matrix right division and matrix left division are related by the equation `B/A = (A'\B')'`.

Least Squares Solutions

If the equation $Ax = b$ does not have a solution (and A is not a square matrix), `x = A\b` returns a *least squares solution* — in other words, a solution that minimizes the length of the vector $Ax - b$, which is equal to `norm(A*x - b)`. See “Example 3” on page 2-2404 for an example of this.

Examples

Example 1

Suppose that `A` and `b` are the following.

```
A = magic(3)

A =

     8     1     6
     3     5     7
     4     9     2

b = [1;2;3]

b =

     1
     2
     3
```

To solve the matrix equation $Ax = b$, enter

```
x=A\b
```

```
x =

    0.0500
    0.3000
    0.0500
```

You can verify that `x` is the solution to the equation as follows.

```
A*x

ans =

    1.0000
    2.0000
    3.0000
```

Example 2 — A Singular

If `A` is singular, `A\b` returns the following warning.

```
Warning: Matrix is singular to working precision.
```

In this case, $Ax = b$ might not have a solution. For example,

```
A = magic(5);
A(:,1) = zeros(1,5); % Set column 1 of A to zeros
b = [1;2;5;7;7];
x = A\b
Warning: Matrix is singular to working precision.

ans =

   NaN
   NaN
   NaN
   NaN
   NaN
```

If you get this warning, you can still attempt to solve $Ax = b$ using the pseudoinverse function `pinv`.

```
x = pinv(A)*b

x =

        0
   0.0209
   0.2717
   0.0808
  -0.0321
```

The result `x` is least squares solution to $Ax = b$. To determine whether `x` is a exact solution — that is, a solution for which $Ax - b = 0$ — simply compute

```
A*x-b

ans =

  -0.0603
   0.6246
  -0.4320
   0.0141
   0.0415
```

The answer is not the zero vector, so `x` is not an exact solution.

, in the online MATLAB Mathematics documentation, provides more examples of solving linear systems using `pinv`.

Example 3

Suppose that

```
A = [1 0 0;1 0 0];
b = [1; 2];
```

Note that $Ax = b$ cannot have a solution, because `A*x` has equal entries for any `x`. Entering

```
x = A\b
```

returns the least squares solution

```
x =

    1.5000
         0
         0
```

along with a warning that `A` is rank deficient. Note that `x` is not an exact solution:

```
A*x-b

ans =

    0.5000
   -0.5000
```

Data Type Support

When computing `X = A\B` or `X = A/B`, the matrices `A` and `B` can have data type `double` or `single`. The following rules determine the data type of the result:

- If both `A` and `B` have type `double`, `X` has type `double`.
- If either `A` or `B` has type single, `X` has type `single`.

Algorithm

The specific algorithm used for solving the simultaneous linear equations denoted by `X = A\B` and `X = B/A` depends upon the structure of the coefficient matrix `A`. To determine the structure of `A` and select the appropriate algorithm, MATLAB software follows this precedence:

1. **If A is sparse and diagonal**, `X` is computed by dividing by the diagonal elements of `A`.

2. **If A is sparse, square, and banded**, then banded solvers are used. Band density is (# nonzeros in the band)/(# nonzeros in a full band). Band density = `1.0` if there are no zeros on any of the three diagonals.

   - If `A` is real and tridiagonal, i.e., band density = `1.0`, and `B` is real with only one column, `X` is computed quickly using Gaussian elimination without pivoting.

   - If the tridiagonal solver detects a need for pivoting, or if `A` or `B` is not real, or if `B` has more than one column, but `A` is banded with band density greater than the `spparms` parameter `'bandden'` (default = `0.5`), then `X` is computed using the Linear Algebra Package (LAPACK) routines in the following table.

	Real	**Complex**
`A` and `B` `double`	DGBTRF, DGBTRS	ZGBTRF, ZGBTRS
`A` or `B` `single`	SGBTRF, SGBTRS	CGBTRF, CGBTRS

3. **If A is an upper or lower triangular matrix**, then `X` is computed quickly with a backsubstitution algorithm for upper triangular matrices, or a forward substitution algorithm for lower triangular matrices. The check for triangularity is done for full matrices by testing for zero elements and for sparse matrices by accessing the sparse data structure.

   If A is a full matrix, computations are performed using the Basic Linear Algebra Subprograms (BLAS) routines in the following table.

	Real	**Complex**
`A and B` `double`	DTRSV, DTRSM	ZTRSV, ZTRSM
`A` or `B` `single`	STRSV, STRSM	CTRSV, CTRSM

4. **If A is a permutation of a triangular matrix**, then `X` is computed with a permuted backsubstitution algorithm.

5 **If A is symmetric, or Hermitian, and has real positive diagonal elements**, then a Cholesky factorization is attempted (see `chol`). If `A` is found to be positive definite, the Cholesky factorization attempt is successful and requires less than half the time of a general factorization. Nonpositive definite matrices are usually detected almost immediately, so this check also requires little time.

If successful, the Cholesky factorization for full `A` is

```
A = R'*R
```

where `R` is upper triangular. The solution `X` is computed by solving two triangular systems,

```
X = R\(R'\B)
```

Computations are performed using the LAPACK routines in the following table.

	Real	**Complex**
`A` and `B` `double`	DLANSY, DPOTRF, DPOTRS, DPOCON	ZLANHE, ZPOTRF, ZPOTRS, ZPOCON
`A` or `B` `single`	SLANSY, SPOTRF, SPOTRS, SDPOCON	CLANHE, CPOTRF, CPOTRS, CPOCON

6 **If `A` is sparse**, then MATLAB software uses CHOLMOD to compute `X`. The computations result in

```
P'*A*P = R'*R
```

where `P` is a permutation matrix generated by `amd`, and `R` is an upper triangular matrix. In this case,

```
X = P*(R\(R'\(P'*B)))
```

7. **If `A` is not sparse but is symmetric**, and the Cholesky factorization failed, then MATLAB solves the system using a symmetric, indefinite factorization. That is, MATLAB computes the factorization `P'*A*P=L*D*L'`, and computes the solution `X` by `X=P*(L'\(D\(L\(P*B))))`. Computations are performed using the LAPACK routines in the following table:

	Real	**Complex**
`A` and `B` `double`	DLANSY, DSYTRF, DSYTRS, DSYCON	ZLANHE, ZHETRF, ZHETRS, ZHECON
`A` or `B` `single`	SLANSY, SSYTRF, SSYTRS, SSYCON	CLANHE, CHETRF, CHETRS, CHECON

8. **If A is Hessenberg**, but not sparse, it is reduced to an upper triangular matrix and that system is solved via substitution.

9. **If A is square** and does not satisfy criteria 1 through 6, then a general triangular factorization is computed by Gaussian elimination with partial pivoting (see `lu`). This results in

```
A = L*U
```

where `L` is a permutation of a lower triangular matrix and `U` is an upper triangular matrix. Then `X` is computed by solving two permuted triangular systems.

```
X = U\(L\B)
```

If `A` is not sparse, computations are performed using the LAPACK routines in the following table.

	Real	**Complex**
`A` and `B` `double`	DLANGE, DGESV, DGECON	ZLANGE, ZGESV, ZGECON
`A` or `B` `single`	SLANGE, SGESV, SGECON	CLANGE, CGESV, CGECON

If `A` is sparse, then UMFPACK is used to compute `X`. The computations result in

```
P*(R\A)*Q = L*U
```

where

- `P` is a row permutation matrix
- `R` is a diagonal matrix that scales the rows of `A`
- `Q` is a column reordering matrix.

Then `X = Q*(U\L\(P*(R\B)))`.

Note The factorization P*(R\A)*Q = L*U differs from the factorization used by the function `lu`, which does not scale the rows of A.

---

10 **If A is not square**, then Householder reflections are used to compute an orthogonal-triangular factorization.

```
A*P = Q*R
```

where `P` is a permutation, `Q` is orthogonal and `R` is upper triangular (see `qr`). The least squares solution `X` is computed with

```
X = P*(R\(Q'*B))
```

If `A` is sparse, MATLAB computes a least squares solution using the sparse `qr` factorization of `A`.

If `A` is full, MATLAB uses the LAPACK routines listed in the following table to compute these matrix factorizations.

	Real	**Complex**
`A` and `B` `double`	DGEQP3, DORMQR, DTRTRS	ZGEQP3, ZORMQR, ZTRTRS
`A` or `B` `single`	SGEQP3, SORMQR, STRTRS	CGEQP3, CORMQR, CTRTRS

Note To see information about choice of algorithm and storage allocation for sparse matrices, set the `spparms` parameter `'spumoni' = 1`.

Note `mldivide` and `mrdivide` are not implemented for sparse matrices `A` that are complex but not square.

See Also

Arithmetic Operators, `linsolve`, `ldivide`, `rdivide`

Purpose

Check M-files for possible problems

GUI Alternatives

From the Current Folder browser, click the **Actions** button ⚙, and then select **Reports > M-Lint Code Check Report**. See also in the Editor.

Syntax

```
mlint('filename')
mlint('filename','-config=settings.txt')
mlint('filename','-config=factory')
inform=mlint('filename','-struct')
msg=mlint('filename','-string')
[inform,filepaths]=mlint('filename')
inform=mlint('filename','-id')
inform=mlint('filename','-fullpath')
inform=mlint('filename','-notok'
mlint('filename','-cyc')
mlint('filename','-eml')
%#eml
%#ok
```

Description

`mlint('filename')` displays M-Lint information about `filename`, where the information reports potential problems and opportunities for code improvement, referred to as suspicious constructs. The line number in the message is a hyperlink that opens the file in the Editor, scrolled to that line. If `filename` is a cell array, information is displayed for each file. For `mlint(F1,F2,F3,...)`, where each input is a character array, MATLAB software displays information about each input file name. You cannot combine cell arrays and character arrays of file names. Note that the exact text of the `mlint` messages is subject to some change between versions.

`mlint('filename','-config=settings.txt')` overrides the default M-lint active settings file with the M-Lint settings that enable or suppress messages as indicated in the specified `settings.txt` file.

Note If used, you must specify the full path to the `settings.txt` file specified with the `-config` option.

For information about creating a `settings.txt` file, see . If you specify an invalid file, `mlint` returns a message indicating that it cannot open or read the file you specified. In that case, `mlint` uses the factory default settings.

`mlint('filename','`**`-config`**`=factory')` ignores all settings files and uses the factory default M-lint preference settings.

`inform=mlint('filename','`**`-struct`**`')` returns the M-Lint information in a structure array whose length is the number of suspicious constructs found. The structure has the fields that follow.

Field	Description
`message`	Message describing the suspicious construct that M-Lint caught.
`line`	Vector of M-file line numbers to which the message refers.
`column`	Two-column array of M-file columns (column extents) to which the message applies. The first column of the array specifies the column in the Editor where the M-Lint message begins. The second column of the array specifies the column in the Editor where the M-Lint message ends. There is one row in the two-column array for each occurrence of an M-Lint message.

If you specify multiple file names as input, or if you specify a cell array as input, `inform` contains a cell array of structures.

`msg=mlint('filename','`**`-string`**`')` returns the M-Lint information as a string to the variable `msg`. If you specify multiple file names as input, or if you specify a cell array as input, `msg` contains a string where each file's information is separated by 10 equal sign characters (=), a space, the file name, a space, and 10 equal sign characters.

If you omit the **`-struct`** or **`-string`** argument and you specify an output argument, the default behavior is **`-struct`**. If you omit the argument and there are no output arguments, the default behavior is to display the information to the command line.

`[inform,filepaths]=mlint('filename')` additionally returns `filepaths`, the absolute paths to the file names, in the same order as you specified them.

`inform=mlint('filename','`**`-id`**`')` requests the message ID from M-Lint, where ID is a string of the form `ABC...`. When returned to a structure, the output also has the `id` field, which is the ID associated with the message.

`inform=mlint('filename','`**`-fullpath`**`')` assumes that the input file names are absolute paths, so that M-Lint does not try to locate them.

`inform=mlint('filename','`**`-notok`**`'`) runs `mlint` for all lines in `filename`, even those lines that end with the `mlint` suppression syntax, `%#ok`.

`mlint('filename','`**`-cyc`**`')` displays the McCabe complexity (also referred to as cyclomatic complexity) of each function in the file. Higher McCabe complexity values indicate higher complexity, and there is some evidence to suggest that programs with higher complexity values are more likely to contain errors. Frequently, you can lower the complexity of a function by dividing it into smaller, simpler functions. In general, smaller complexity values indicate programs that are easier to understand and modify. Some people advocate splitting up programs that have a complexity rating over 10.

`mlint('filename','`**`-eml`**`')` enables Embedded MATLAB™ messages for display in the Command Window.

If you include `%#eml` anywhere within an M-file, except within a comment, it causes `mlint` to behave as though you specified `eml` for that file. For more information, see . MATLAB comments can follow the `%#eml` directive.

If you include `%#ok` at the end of a line in an M-file, `mlint` ignores that line. `mlint` ignores specified messages *id1* through *idn* on a given line when `%#ok<` *id1,id2,...idn*`>` appears at the end of that line. `mlint` ignores specified messages `1` through *`n`* throughout the file when `%#ok<*`*id1,*id2,...*idn*`>` appears at the end of a line. To determine the id for a given message, use the following command, where *`filename`* is the name of the file that elicits the message:

```
mlint filename -id
```

For information on adding the `%#ok` directive using the Editor context menu, see .

Examples

The following examples use `lengthofline.m`, which is a sample M-file with code that can be improved. You can find it in *`matlabroot`*`/help/techdoc/matlab_env/examples`. If you want to run the examples, save a copy of `lengthofline.m` to a location on your MATLAB path.

Running mlint on a File with No Options

To run `mlint` on the example file, `lengthofline.m`, run

```
mlint('lengthofline')
```

MATLAB displays M-Lint messages for `lengthofline.m` in the Command Window:

```
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
                Type 'doc struct' for more information.
```

```
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop.Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary. Use parentheses
to group, if needed.
```

For details about these messages and how to improve the code, see in the MATLAB Desktop Tools and Development Environment documentation.

Running mlint with Options to Show IDs and Return Results to a Structure

To store the results to a structure and include message IDs, run

```
inform=mlint('lengthofline', '-id')
```

MATLAB returns

```
inform =

19x1 struct array with fields:
    message
    line
    column
    id
```

To see values for the first message, run

```
inform(1)
```

MATLAB displays

```
ans = 

    message: 'The value assigned here to variable 'nothandle' might never be used.'
       line: 22
     column: [1 9]
         id: 'NASGU'
```

Here, the message is for the value that appears on line 22 that extends from column 1–9 in the M-file.`NASGU` is the ID for the message `'The value assigned here to variable 'nothandle' might never be used.'`.

Suppressing Specific Messages with mlint

When you add `%#ok` to a line, it suppresses all `mlint` messages for that line. However, suppose there are multiple messages in a line and you want to suppress some, but not all of them. Or, suppose you want to suppress a specific message, but not all messages that might arise in the future due to changes you make to that line. Use the `%#ok` syntax in conjunction with message IDs.

This example uses the following code, `displayAnonymousFunction.m`:

```
function displayAnonymousFunction
% mini tutorial on anonymous function handles.

disp('  ');
disp('  Here is an example of an anonymous function that');
disp('  retrieves the last modified date of a given file:');
disp('  ');
fileDate = @(f)getfield(dir(f),'date')

disp('  ');
disp('  You can call it by passing a filename into the ');
disp('  function_handle variable.  We will use the currently');
disp('  running M-file for example purposes:');
disp('  ');
```

```
thisFile = which(mfilename('fullpath'))

disp('  ');
disp('  Now call the anonymous function handle as you would');
disp('  call any function or function_handle: fileDate(thisFile)');
disp('  ');
fileDate(thisFile)
```

Run `mlint` with the `-id` option on `displayAnonymousFunction.m`:

```
mlint('displayAnonymousFunction','-id')
```

Results displayed to the Command Window show two messages for line 8:

```
L 8 (C 10): NOPRT: Terminate statement with semicolon to suppress output (in functions).
L 8 (C 16-23): GFLD: Use dynamic fieldnames with structures instead of GETFIELD.
               Type 'doc struct' for more information.
```

To suppress the first message on the line (about using a semicolon), use its message ID, `NOPRT`, with the `%#ok` syntax as shown here:

```
fileDate = @(f)getfield(dir(f),'date') %#ok<NOPRT>
```

When you run `mlint` for `displayAnonymousFunction.m`, only one message now displays for line 8.

To suppress multiple specific messages for a line, separate message IDs with commas in the `%#ok` syntax:

```
fileDate = @(f)getfield(dir(f),'date') %#ok<NOPRT,GFLD>
```

Now when you run `mlint` for `displayAnonymousFunction.m`, no messages display for line 8.

Suppressing Specific Messages Throughout a File with mlint

To suppress a specific message throughout a file, use the `%#ok` syntax in conjunction with a message ID preceded by an asterisk (*).

Run `mlint` with the `-id` option on the original `displayAnonymousFunction.m` code presented in the previous example:

```
mlint('displayAnonymousFunction','-id')
```

Results displayed to the Command Window show two messages for line 8:

```
L 8 (C 10): NOPRT: Terminate statement with semicolon to suppress output (in functions).
L 8 (C 16-23): GFLD: Use dynamic fieldnames with structures instead of GETFIELD.
                     Type 'doc struct' for more information.
```

To suppress the semicolon message throughout the file, use its message ID, `NOPRT`, with an asterisk in the `%#ok` syntax as shown here:

```
fileDate = @(f)getfield(dir(f),'date') %#ok<*NOPRT>
```

When you run `mlint` for `displayAnonymousFunction.m`, the semicolon message is suppressed throughout the file and only one message displays for line 8.

To suppress multiple specific messages throughout a file, separate message IDs with commas in the `%#ok` syntax, and precede each message ID with an asterisk:

```
fileDate = @(f)getfield(dir(f),'date') %#ok<*NOPRT,*GFLD>
```

Now when you run `mlint` for `displayAnonymousFunction.m`, both the `NOPRT` and`GFLD` messages are suppressed throughout the file.

Error Message: An M-Lint message Was Once Suppressed Here, But the Message No Longer Appears

This examples shows how to interpret the message, “An M-Lint message was once suppressed here, but the message no longer appears.”

Suppose you direct `mlint` to ignore line 15, in the M-file, `displayAnonymousFunction.m` (the code for which is presented in the third example in this section) by adding `%#ok` to the end of line 15:

```
thisFile = which(mfilename('fullpath') %#ok
```

When you run `mlint` for `displayAnonymousFunction.m`, typically no message is shown for line 15, because it contains the `%#ok` message suppression syntax. However, there are some exceptions, as follows:

- If you change the code so that it would not elicit the message, "`Terminate statement with semicolon to suppress output (in functions)`" if you removed the `%#ok` directive
- If you disable the message in M-Lint preferences after you add the `%#ok` directive
- If the rules M-Lint uses for generating the message change

If any one of these cases is true for line 15, then the following message now appears at line 15:

```
An M-Lint message was once suppressed here, but the message no longer appears.
```

To remove this message, use the context menu and select **Remove the Message Suppression**. The `%#ok` directive is removed and now no M-Lint messages appear for line 15 of `displayAnonymousFunction.m`.

Displaying McCabe Complexity with mlint

To display the McCabe complexity of an M-File, run `mlint` with the `-cyc` option, as shown in the following example (assuming you have saved `lengthofline.m` to a local folder).

```
mlint lengthofline.m -cyc
```

Results displayed in the Command Window show the McCabe complexity of the file, followed by the M-File messages, as shown here:

```
L 1 (C 23-34): The McCabe complexity of 'lengthofline' is 12.
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP.
```

```
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD. Type 'doc struct' for mor
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary.  Use parentheses to group, if needed.
```

See Also

`mlintrpt`, `profile`

Purpose

Run `mlint` for file or folder, reporting results in browser

GUI Alternatives

From the Current Folder browser, click the **Actions** button ⚙, and then select **Reports > M-Lint Code Check Report**. See also the automatic in the Editor.

Syntax

```
mlintrpt
mlintrpt('filename','file')
mlintrpt('dirname','dir')
mlintrpt('filename','file','settings.txt')
mlintrpt('dirname','dir','settings.txt')
```

Description

`mlintrpt` scans all M-files in the current folder for M-Lint messages and reports the results in a MATLAB Web browser.

`mlintrpt('filename','file')` scans the M-file `filename` for messages and reports results. You can omit `'file'` in this form of the syntax because it is the default.

`mlintrpt('dirname','dir')` scans the specified folder. Here, `dirname` can be in the current folder or can be a full path.

`mlintrpt('filename','file','settings.txt')` applies the M-Lint settings to enable or suppress messages as indicated in the specified `settings.txt` file. For information about creating a `settings.txt` file, select **File > Preferences > M-Lint**, and click **Help**.

`mlintrpt('dirname','dir','settings.txt')` applies the M-Lint settings indicated in the specified `settings.txt` file.

Note If you specify a `settings.txt` file, you must specify the full path to the file.

Examples

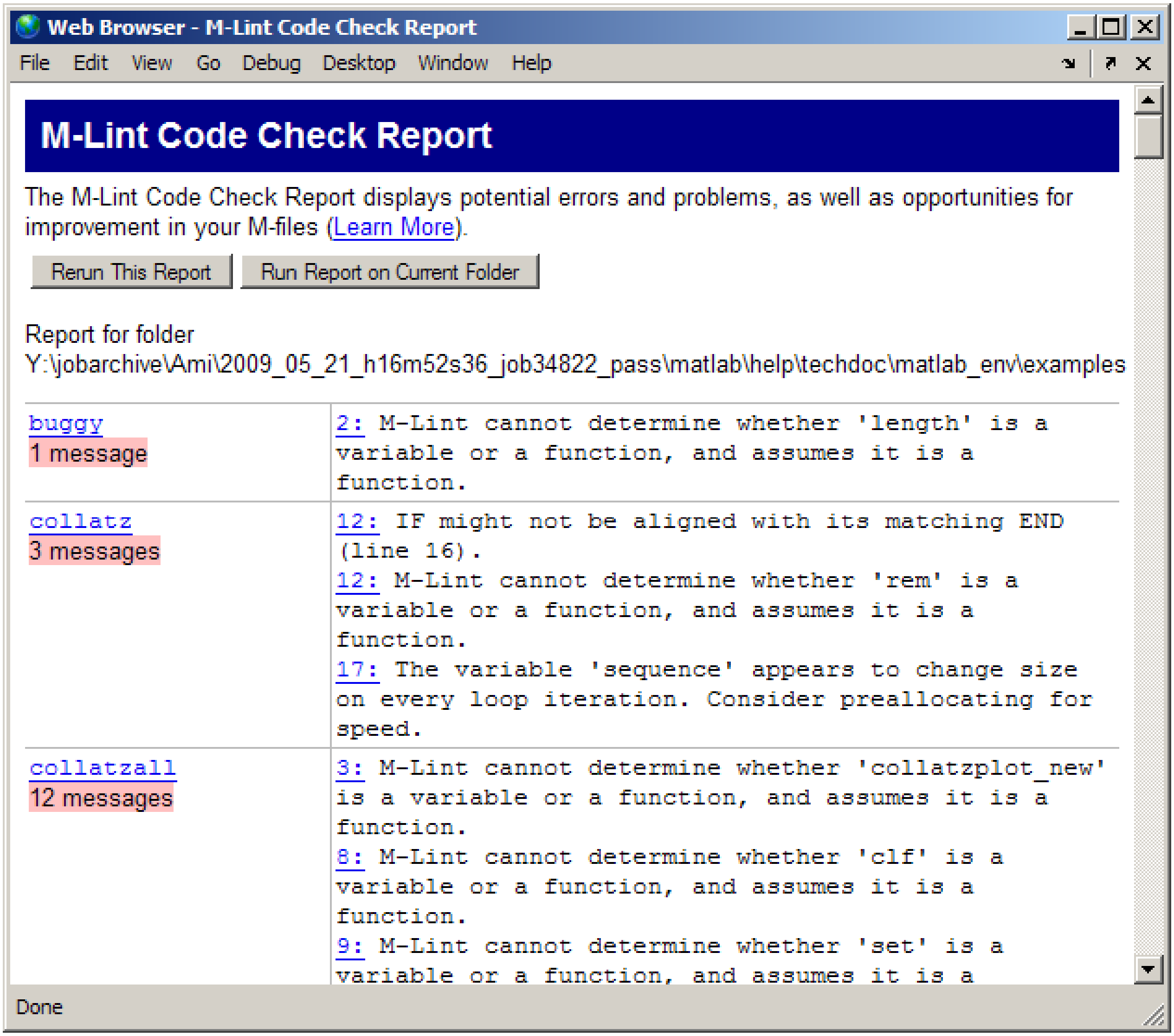
`lengthofline.m` is an example M-file with code that can be improved. It is found in *`matlabroot`*`/matlab/help/techdoc/matlab_env/examples`.

Run Report for All Files in a Directory

Run

```
mlintrpt(fullfile(matlabroot,'help','techdoc','matlab_env','examples'),'dir')
```

and MATLAB displays a report of potential problems and improvements for all M-files in the `examples` folder.



M-Lint Code Check Report

The M-Lint Code Check Report displays potential errors and problems, as well as opportunities for improvement in your M-files (Learn More).

Report for folder
Y:\jobarchive\Ami\2009_05_21_h16m52s36_job34822_pass\matlab\help\techdoc\matlab_env\examples

buggy 1 message	2: M-Lint cannot determine whether 'length' is a variable or a function, and assumes it is a function.
collatz 3 messages	12: IF might not be aligned with its matching END (line 16). 12: M-Lint cannot determine whether 'rem' is a variable or a function, and assumes it is a function. 17: The variable 'sequence' appears to change size on every loop iteration. Consider preallocating for speed.
collatzall 12 messages	3: M-Lint cannot determine whether 'collatzplot_new' is a variable or a function, and assumes it is a function. 8: M-Lint cannot determine whether 'clf' is a variable or a function, and assumes it is a function. 9: M-Lint cannot determine whether 'set' is a variable or a function, and assumes it is a

For details about these messages and how to improve the code, see in the MATLAB Desktop Tools and Development Environment documentation.

Run Report Using M-Lint Preference Settings

In **File > Preferences > M-Lint**, save preference settings to a file, for example, `MLintNoSemis.txt`. To apply those settings when you run `mlintrpt`, use the `file` option and supply the full path to the settings file name as shown in this example:

```
mlintrpt('lengthofline.m', 'file', ...
'C:\WINNT\Profiles\me\Application Data\MathWorks\MATLAB\R2007a\MLintNoSemis.txt')
```

Alternatively, use `fullfile` if the settings file is stored in the preferences folder:

```
 mlintrpt('lengthofline.m', 'file', fullfile(prefdir,'MLintNoSemis.txt'))
```

Assuming that in that example `MLintNoSemis.txt` file, the setting for `Terminate statement with semicolon to suppress output` has been disabled, the results of `mlintrpt` for `lengthofline` do not show that message for line 49.

When `mlintrpt` cannot locate the settings file, the first message in the report is

```
0: Unable to open or read the configuration file 'MLintNoSemis.txt'--using default settings.
```

See Also

`mlint`

Purpose

Prevent clearing M-file or MEX-file from memory

Syntax

```
mlock
```

Description

`mlock` locks the currently running M-file or MEX-file in memory so that subsequent `clear` functions do not remove it.

Use the `munlock` function to return the file to its normal, clearable state.

Locking an M-file or MEX-file in memory also prevents any `persistent` variables defined in the file from getting reinitialized.

Examples

The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
  .
  .
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked('testfun')
ans =
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock('testfun')

mislocked('testfun')
ans =
     0
```

See Also

`mislocked`, `munlock`, `persistent`

mmfileinfo

Purpose

Information about multimedia file

Syntax

`info = mmfileinfo(filename)`

Description

Note You can use `mmfileinfo` only on Linux, Macintosh, and Microsoft Windows operating systems.

`info = mmfileinfo(filename)` returns a structure, *`info`*, with fields containing information about the contents of the multimedia file identified by *`filename`*. The *`filename`* input is a string enclosed in single quotation marks.

If *`filename`* is a URL, `mmfileinfo` might take a long time to return because it must first download the file. For large files, downloading can take several minutes. To avoid blocking the MATLAB command line while this processing takes place, download the file before calling `mmfileinfo`.

The *`info`* structure contains the following fields, listed in the order they appear in the structure.

Field	Description
`Filename`	String indicating the name of the file.
`Duration`	Length of the file in seconds.
`Audio`	Structure containing information about the audio data in the file. See "Audio Data" on page 2-2427 for more information about this data structure.
`Video`	Structure containing information about the video data in the file. See "Video Data" on page 2-2427 for more information about this data structure.

Audio Data

The `Audio` structure contains the following fields, listed in the order they appear in the structure. If the file does not contain audio data, the fields in the structure are empty.

Field	Description
`Format`	Text string, indicating the audio format.
`NumberOfChannels`	Number of audio channels.

Video Data

The `Video` structure contains the following fields, listed in the order they appear in the structure.

Field	Description
`Format`	Text string, indicating the video format.
`Height`	Height of the video frame.
`Width`	Width of the video frame.

Examples

This example gets information about the contents of a file containing audio data.

```
info = mmfileinfo('my_audio_data.mp3')

info =

    Filename: 'my_audio_data.mp3'
    Duration: 1.6030e+002
       Audio: [1x1 struct]
       Video: [1x1 struct]
```

To look at the information returned about the audio data in the file, examine the fields in the `Audio` structure.

```
audio_data = info.Audio

audio_data =

              Format: 'MPEGLAYER3'
    NumberOfChannels: 2
```

Because the file contains only audio data, the fields in the `Video` structure are empty.

```
info.Video

ans =

    Format: ''
    Height: []
     Width: []
```

Purpose

Create multimedia reader object for reading video files

Syntax

```
obj = mmreader(filename)
obj = mmreader(filename, 'P1', V1, 'P2', V2,...)
```

Description

`obj = mmreader(filename)` constructs a multimedia reader object, *`obj`*, that can read video data from a multimedia file named *`filename`*. The `mmreader` function searches for the file on the MATLAB path, and generates an error if it cannot construct the object for any reason.

`mmreader` supports the following file formats:

Platform	**Supported File Formats**
Windows, Macintosh, and Linux	Motion JPEG 2000 (`.mj2`)
Windows	AVI (`.avi`), MPEG-1 (`.mpg`), Windows Media Video (`.wmv`, `.asf`, `.asx`), and any format supported by Microsoft DirectShow.
Macintosh	AVI (`.avi`), MPEG-1 (`.mpg`), MPEG-4 (`.mp4`, `.m4v`), Apple QuickTime Movie (`.mov`), and any format supported by QuickTime as listed on .
Linux	Any format supported by your installed plug-ins for GStreamer 0.10 or above, as listed on , including AVI (`.avi`) and Ogg Theora (`.ogg`).

There are no restrictions on file extensions. For more information, see in the MATLAB Data Import and Export documentation

`obj = mmreader(filename, 'P1', V1, 'P2', V2,...)` constructs a multimedia object and assigns values *`V1`*, *`V2`*, etc. to the respective specified properties *`P1`*, *`P2`*, etc. If you specify an invalid property name or property value, MATLAB throws an error and does not create the object. Property value pairs can be in any format supported by the `set` function: parameter-value string pairs, structures, or parameter-value cell array pairs. The `mmreader` object supports the following properties:

Property	**Description**	**Read-Only**	**Default Value**
`BitsPerPixel`	Bits per pixel of the video data	Yes	
`Duration`	Total length of file in seconds	Yes	
`FrameRate`	Frame rate of the video in frames per second	Yes	
`Height`	Height of the video frame in pixels	Yes	
`Name`	Name of the file from which the reader object was created	Yes	
`NumberOfFrames`	Total number of frames in the video stream	Yes	
`Path`	String containing the full path to the file associated with the reader	Yes	
`Tag`	Generic string for you to set	No	`''`
`Type`	Class name of the object	Yes	`mmreader`
`UserData`	Generic field for any user-defined data	No	`[]`

Property	Description	Read-Only	Default Value
`VideoFormat`	String indicating the video format as it is represented in MATLAB, e.g., `RGB24`	Yes	
`Width`	Width of the video frame in pixels	Yes	

Remarks

Some files store video at a variable frame rate, including many Windows Media Video files. For these files, `mmreader` cannot determine the number of frames until you read the last frame. When you construct the object, `mmreader` returns a warning and does not set the `NumberOfFrames` property.

To count the number of frames in a variable frame rate file, call the `read` function to read the last frame of the file. For example:

```
vidObj = mmreader('varFrameRateFile.wmv');
lastFrame = read(vidObj, inf);
numFrames = vidObj.NumberOfFrames;
```

Because `mmreader` must decode all video data to count the frames reliably, the call to `read` sometimes takes a long time to run. For more information, see in the MATLAB Data Import and Export documentation.

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)`, `mmfileinfo`, `read (mmreader)`, `set (hgsetget)`

Purpose

Determine whether `mmreader` is available on current platform

Syntax

```
supported = mmreader.isPlatformSupported()
```

Description

`supported = mmreader.isPlatformSupported()` returns `true` if `mmreader` can read at least one file format on the current platform, or `false` otherwise. For a list of supported file formats, and the requirements to read these formats on each platform, see `mmreader`.

Note Because `mmreader` can read Motion JPEG 2000 files on Windows, Macintosh, and Linux platforms, `mmreader.isPlatformSupported` always returns `true` for those platforms.

See Also

`mmreader`

mod

Purpose

Modulus after division

Syntax

`M = mod(X,Y)`

Description

`M = mod(X,Y)` if `Y ~= 0`, returns `X - n.*Y` where `n = floor(X./Y)`. If `Y` is not an integer and the quotient `X./Y` is within roundoff error of an integer, then `n` is that integer. The inputs `X` and `Y` must be real arrays of the same size, or real scalars.

The following are true by convention:

- `mod(X,0)` is `X`
- `mod(X,X)` is `0`
- `mod(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `Y`.

Remarks

`rem(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `X`.

`mod(X,Y)` and `rem(X,Y)` are equal if `X` and `Y` have the same sign, but differ by `Y` if `X` and `Y` have different signs.

The `mod` function is useful for congruence relationships: *x and y are congruent (mod m)* if and only if `mod(x,m) == mod(y,m)`.

Examples

```
mod(13,5)
ans =
     3

mod([1:5],3)
ans =
     1     2     0     1     2

mod(magic(3),3)
ans =
     2     1     0
     0     2     1
     1     0     2
```

See Also `rem`

mode

Purpose

Most frequent values in array

Syntax

```
M = mode(X)
M = mode(X, dim)
[M,F] = mode(X, ...)
[M,F,C] = mode(X, ...)
```

Description

`M = mode(X)` for vector `X` computes the sample mode `M`, (i.e., the most frequently occurring value in `X`). If `X` is a matrix, then `M` is a row vector containing the mode of each column of that matrix. If `X` is an N-dimensional array, then `M` is the mode of the elements along the first nonsingleton dimension of that array.

When there are multiple values occurring equally frequently, `mode` returns the smallest of those values. For complex inputs, this is taken to be the first value in a sorted list of values.

`M = mode(X, dim)` computes the mode along the dimension `dim` of `X`.

`[M,F] = mode(X, ...)` also returns array `F`, each element of which represents the number of occurrences of the corresponding element of `M`. The `M` and `F` output arrays are of equal size.

`[M,F,C] = mode(X, ...)` also returns cell array `C`, each element of which is a sorted vector of all values that have the same frequency as the corresponding element of `M`. All three output arrays `M`, `F`, and `C` are of equal size.

Remarks

The `mode` function is most useful with discrete or coarsely rounded data. The mode for a continuous probability distribution is defined as the peak of its density function. Applying the `mode` function to a sample from that distribution is unlikely to provide a good estimate of the peak; it would be better to compute a histogram or density estimate and calculate the peak of that estimate. Also, the `mode` function is not suitable for finding peaks in distributions having multiple modes.

Examples

Example 1

Find the mode of the 3-by-4 matrix shown here:

```
X = [3 3 1 4; 0 0 1 1; 0 1 2 4]
X =
     3     3     1     4
     0     0     1     1
     0     1     2     4

mode(X)
ans =
     0     0     1     4
```

Find the mode along the second (row) dimension:

```
mode(X, 2)
ans =
     3
     0
     0
```

Example 2

Find the mode of a continuous variable grouped into bins:

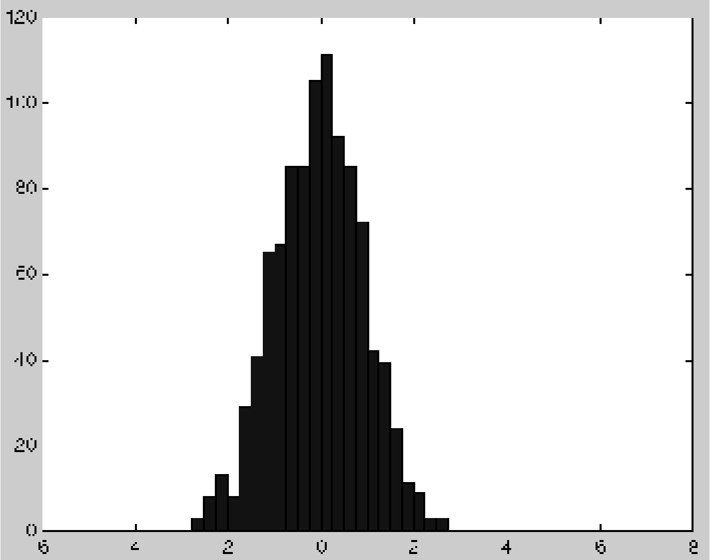
```
randn('state', 0);       % Reset the random number generator

y = randn(1000,1);
edges = -6:.25:6;
[n,bin] = histc(y,edges);

m = mode(bin)
m =
    22

edges([m, m+1])
ans =
   -0.7500   -0.5000

hist(y,edges+.125)
```



See Also `mean`, `median`, `hist`, `histc`

Purpose

Control paged output for Command Window

Syntax

```
more on
more off
more(n)
A = more(state)
```

Description

`more on` enables paging of the output in the MATLAB Command Window. MATLAB displays output one page at a time. Use the keys defined in the table below to control paging.

`more off` disables paging of the output in the MATLAB Command Window.

`more(n)` defines the length of a page to be `n` lines.

`A = more(state)` returns in `A` the number of lines that are currently defined to be a page. The `state` input can be one of the quoted strings `'on'` or `'off'`, or the number of lines to set as the new page length.

By default, the length of a page is equal to the number of lines available for display in the MATLAB Command Window. Manually changing the size of the command window adjusts the page length accordingly.

If you set the page length to a specific value, MATLAB uses that value for the page size, regardless of the size of the command window. To have MATLAB return to matching page size to window size, type `more off` followed by `more on`.

To see the status of `more`, type `get(0,'More')`. MATLAB returns either `on` or `off` indicating the `more` status. You can also set status for `more` by using `set(0,'More', 'status')`, where `'status'` is either `'on'` or `'off'`.

When you have enabled `more` and are examining output, you can do the following.

Press the...	To...
Return key	Advance to the next line of output.

Press the...	**To...**
Space bar	Advance to the next page of output.
Q (for quit) key	Terminate display of the text. Do not use **Ctrl+C** to terminate `more` or you might generate error messages in the Command Window.

`more` is in the **`off`** state, by default.

See Also

`diary`

Purpose

Move or resize control in parent window

Syntax

```
V = h.move(position)
V = move(h, position)
```

Description

`V = h.move(position)` moves the control to the position specified by the `position` argument. When you use move with only the handle argument, `h`, it returns a four-element vector indicating the current position of the control.

`V = move(h, position)` is an alternate syntax.

The position argument is a four-element vector specifying the position and size of the control in the parent figure window. The elements of the vector are:

```
[x, y, width, height]
```

where `x` and `y` are offsets, in pixels, from the bottom left corner of the figure window to the same corner of the control, and `width` and `height` are the size of the control itself.

Examples

This example moves the control:

```
f = figure('Position', [100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200], f);
pos = h.move([50 50 200 200])
pos =
    50    50   200   200
```

The next example resizes the control to always be centered in the figure as you resize the figure window. Start by creating the script `resizectrl.m` that contains

```
% Get the new position and size of the figure window
  fpos = get(gcbo, 'position');

% Resize the control accordingly
```

```
    h.move([0 0 fpos(3) fpos(4)]);
```

Now execute the following in MATLAB or in an M-file:

```
f = figure('Position', [100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200]);
set(f, 'ResizeFcn', 'resizectrl');
```

As you resize the figure window, notice that the circle moves so that it is always positioned in the center of the window.

See Also

`set (COM)`, `get (COM)`

Purpose

Move file or folder

Graphical Interface

As an alternative to the `movefile` function, use the Current Folder browser.

Syntax

```
movefile('source')
movefile('source','destination')
movefile('source','destination','f')
[status,message,messageid]=movefile(...)
```

Description

`movefile('source')` moves the file or folder named `source` to the current folder, where `source` is the absolute or relative path name for the folder or file. To move multiple files or folders, use one or more wildcard characters, (`*`), after the last file separator in `source`. The `source` argument permits an `*` in a path string. `movefile` does not preserve the archive attribute of `source`.

`movefile('source','destination')` moves the file or folder named `source` to the location `destination`, where `source` and `destination` are the absolute or relative paths for the folder or file. To move multiple files or folders, you can use one or more wildcard characters, `*`, after the last file separator in `source`. You cannot use a wildcard character in `destination`. To rename a file or folder when moving it, make `destination` a different name than `source`, and specify only one file for `source`. When `source` and `destination` have the same location, `movefile` renames `source` to `destination`.

`movefile('source','destination','f')` moves the file or folder named `source` to the location `destination`, regardless of the read-only attribute of `destination`.

`[status,message,messageid]=movefile(...)` moves the file or folder named `source` to the location `destination`, returning the status, a message, and the MATLAB message ID. Here, `status` is logical `1` for success or logical `0` for error. `movefile` requires only one output argument.

Examples

Moving a File to the Current Folder

Move the file `myfiles/myfunction.m` to the current folder:

```
movefile('myfiles/myfunction.m')
```

Move `projects/myfiles` and its contents to the current folder, when the current folder is `projects/testcases`:

```
movefile('../myfiles')
```

Renaming a File in the Current Folder

Rename `myfunction.m` to `oldfunction.m`:

```
movefile('myfunction.m','oldfunction.m')
```

Using a Wildcard to Move All Matching Files

Move all files in the folder `myfiles`, whose names begin with `my`, to the current folder:

```
movefile('myfiles/my*')
```

Moving a File to a Different Folder

Move the file `myfunction.m` from the current folder to the folder `projects`, where `projects` and the current folder are at the same level:

```
movefile('myfunction.m','../projects')
```

Moving a Folder Down One Level

Move the folder `projects/testcases` and all its contents down a level in `projects` to `projects/myfiles`:

```
movefile('projects/testcases','projects/myfiles/')
```

Moving a File to Read-Only Folder and Renaming the File

Move the file `myfile.m` from the current folder to `d:/work/restricted`, assigning it the name `test1.m`, where `restricted` is a read-only folder:

```
movefile('myfile.m','d:/work/restricted/test1.m','f')
```

The read-only file `myfile.m` is no longer in the current folder. The file `test1.m` is in `d:/work/restricted` and is read only.

Returning Status When Moving Files

Move all files in the folder `myfiles` whose names start with `new` to the current folder, when there is an error. You mistype `new*` mistyped as `nex*` and no items in the current folder start with `nex*`:

```
[s,mess,messid]=movefile('myfiles/nex*')

s =
     0

mess =

A duplicate filename exists, or the file cannot be found.

messid =

MATLAB:MOVEFILE:OSError
```

See Also

`cd`, `copyfile`, `delete`, `dir`, `fileattrib`, `ls`, `mkdir`, `rmdir`

movegui

Purpose

Move GUI figure to specified location on screen

Syntax

```
movegui(h,'position')
movegui(position)
movegui(h)
movegui
```

Description

`movegui(h,'position')` moves the figure identified by handle `h` to the specified screen location, preserving the figure's size. The *`position`* argument is a string or a two-element vector, as defined in the following tables.

`movegui(position)` moves the callback figure (`gcbf`) or the current figure (`gcf`) to the specified position.

`movegui(h)` moves the figure identified by the handle `h` to the `onscreen` position.

`movegui` moves the callback figure (`gcbf`) or the current figure (`gcf`) to the `onscreen` position. You can specify `'movegui'` as a `CreateFcn` callback for a figure. It ensures after you save it, the figure appears on screen when you reload it, regardless of its saved position. See the following example.

When it is a string, *`position`* is one of the following descriptors.

Position String	Description
`north`	Top center edge of screen
`south`	Bottom center edge of screen
`east`	Right center edge of screen
`west`	Left center edge of screen
`northeast`	Top right corner of screen
`northwest`	Top left corner of screen
`southeast`	Bottom right corner of screen

Position String	Description
`southwest`	Bottom left corner
`center`	Centered on screen
`onscreen`	Nearest location to current location that is entirely on screen

You can also specify the *`position`* argument as a two-element vector, `[h,v]`. Depending on sign, `h` specifies the figure's offset from the left or right edge of the screen, and `v` specifies the figure's offset from the top or bottom of the screen, in pixels. The following table summarizes the possible values.

`h` (for `h >= 0`)	Offset of left side from left edge of screen
`h` (for `h < 0`)	Offset of right side from right edge of screen
`v` (for `v >= 0`)	Offset of bottom edge from bottom of screen
`v` (for `v < 0`)	Offset of top edge from top of screen

When you apply `movegui` to a maximized figure window, the window can shrink in size by a few pixels. On Microsoft Windows platforms, using it to move a maximized window toward the Windows task bar creates a gap on the opposite side of the screen about as wide as the task bar.

`GUIDE` and `openfig` call `movegui` when loading figures to ensure they are visible.

Examples

Use `movegui` to ensure that a saved GUI appears on screen you reload it, regardless of the target computer screen size and resolution. Create a figure that is off the screen, assign `movegui` as its `CreateFcn` callback, save it, and then reload the figure.

```
f = figure('Position',[10000,10000,400,300]);
```

```
% The figure does not display because it is created offscreen
set(f,'CreateFcn','movegui')
hgsave(f,'onscreenfig')
close(f)
f2 = hgload('onscreenfig');
% The reloaded figure is now visible
```

Alternatives

See Also

`guide` | `openfig`

Tutorials

-
-

How To

-

Purpose

Play recorded movie frames

Syntax

```
movie(M)
movie(M,n)
movie(M,n,fps)
movie(h,...)
movie(h,M,n,fps,loc)
```

Description

The `movie` function plays the movie defined by a matrix whose columns are movie frames (usually produced by `getframe`).

`movie(M)` plays the movie in matrix `M` once, using the current axes as the default target. If you want to play the movie in the figure instead of the axes, specify the figure handle (or `gcf`) as the first argument: `movie(`*`figure_handle`*`,...)`. `M` must be an array of movie frames (usually from `getframe`).

`movie(M,n)` plays the movie `n` times. If `n` is negative, each cycle is shown forward then backward. If `n` is a vector, the first element is the number of times to play the movie, and the remaining elements make up a list of frames to play in the movie.

For example, if `M` has four frames then `n = [10 4 4 2 1]` plays the movie ten times, and the movie consists of frame `4` followed by frame `4` again, followed by frame `2` and finally frame `1`.

`movie(M,n,fps)` plays the movie at `fps` frames per second. The default is 12 frames per second. Computers that cannot achieve the specified speed play as fast as possible.

`movie(h,...)` plays the movie centered in the figure or axes identified by the handle `h`.

`movie(h,M,n,fps,loc)` specifies `loc`, a four-element location vector, `[x y 0 0]`, where the lower left corner of the movie frame is anchored (only the first two elements in the vector are used). The location is relative to the lower left corner of the figure or axes specified by `handle h` and in units of pixels, regardless of the object's `Units` property.

Remarks

The `movie` function uses a default figure size of 560-by-420 and does not resize figures to fit movies with larger or smaller frames. To accommodate other frame sizes, you can resize the figure to fit the movie, as shown in the second example below.

`movie` only accepts 8-bit image frames; it does not accept 16-bit grayscale or 24–bit truecolor image frames.

Buffering the movie places all frames in memory. As a result, on Microsoft Windows and perhaps other platforms, a long movie (on the order of several hundred frames) can exhaust memory, depending on system resources. In such cases an error message is issued that says

```
??? Error using ==> movie
Could not create movie frame
```

You can abort a movie by typing **Ctrl-C**.

Examples

Example 1: Animate the `peaks` function as you scale the values of `Z`:

```
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
% Record the movie
for j = 1:20
 surf(sin(2*pi*j/20)*Z,Z)
 F(j) = getframe;
end
% Play the movie ten times
movie(F,10)
```

Example 2: Specify figure when calling movie to fit the movie to the figure:

```
r = subplot(2,1,1)
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
```

```
s = subplot(2,1,2)
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
% Record the movie
for j = 1:20
 axes(r)
    surf(sin(2*pi*j/20)*Z,Z)
 axes(s)
 surf(sin(2*pi*(j+5)/20)*Z,Z)
    F(j) = getframe(gcf);
 pause(.0333)
end
% Play the movie; note that it does not fit the figure properly:
h2 = figure;
movie(F,10)
% Use the figure handle to make the frames fit:
movie(h2,F,10)
```

Example 3: With larger frames, first adjust the figure's size to fit the movie:

```
figure('position',[100 100 850 600])
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
% Record the movie
for j = 1:20
 surf(sin(2*pi*j/20)*Z,Z)
 F(j) = getframe;
end
[h, w, p] = size(F(1).cdata);  % use 1st frame to get dimensions
hf = figure;
% resize figure based on frame's w x h, and place at (150, 150)
set(hf, 'position', [150 150 w h]);
axis off
% tell movie command to place frames at bottom left
```

```
movie(hf,F,4,30,[0 0 0 0]);
```

See Also

`aviread`, `getframe`, `frame2im`, `im2frame`

“Animation” on page 1-96 for related functions

See Example – Visualizing an FFT as a Movie for another example

Purpose

Create Audio/Video Interleaved (AVI) movie from MATLAB movie

Syntax

```
movie2avi(mov, filename)
movie2avi(mov, filename, param, value, param, value...)
```

Description

`movie2avi(mov, filename)` creates the AVI movie `filename` from the MATLAB movie `mov`. The `filename` input is a string enclosed in single quotes.

`movie2avi(mov, filename, param, value, param, value...)` creates the AVI movie `filename` from the MATLAB movie `mov` using the specified parameter settings.

Parameter	Value	Default
`'colormap'`	An `m`-by-3 matrix defining the colormap to be used for indexed AVI movies, where `m` must be no greater than 256 (236 if using `Indeo` compression). This parameter can be specified only when the `'compression'` parameter is set to `'MSVC'`, `'RLE'`, or `'None'`	There is no default colormap.

Parameter	Value	Default
`'compression'`	A text string specifying the compression codec to use. On Microsoft Windows operating systems: • `'Indeo3'` • `'Indeo5'` • `'Cinepak'` • `'MSVC'` • `'RLE'` • `'None'` • To use a custom compression codec on Windows systems, specify the four-character code that identifies the codec (typically included in the codec documentation). The `movie2avi` function reports an error if it can not find the specified custom compressor. On UNIX operating systems: • `'None'`	`'Indeo5'` on Windows systems. `'None'` on UNIX systems.
`'fps'`	A scalar value specifying the speed of the AVI movie in frames per second (fps).	15 fps
`'keyframe'`	For compressors that support temporal compression, this is the number of key frames per second.	2.1429 key frames per second.

Parameter	Value	Default
`'quality'`	A number between 0 and 100 the specifies the desired quality of the output. Higher numbers result in higher video quality and larger file sizes. Lower numbers result in lower video quality and smaller file sizes. This parameter has no effect on uncompressed movies.	75
`'videoname'`	A descriptive name for the video stream. This parameter must be no greater than 64 characters long.	The default is the `filename`.

See Also

`avifile`, `mmreader`, `mmfileinfo`, `movie`

Purpose

Upload file or directory to FTP server

Syntax

```
mput(f,'filename')
mput(ftp,'directoryname')
mput(f,'wildcard')
```

Description

`mput(f,'filename')` uploads `filename` from the MATLAB current directory to the current directory of the FTP server `f`, where `filename` is a file, and where `f` was created using `ftp`. You can use a wildcard (`*`) in `filename`. MATLAB returns a cell array listing the full path to the uploaded files on the server.

`mput(ftp,'directoryname')` uploads the directory `directoryname` and its contents. MATLAB returns a cell array listing the full path to the uploaded files on the server.

`mput(f,'wildcard')` uploads a set of files or directories specified by a wildcard. MATLAB returns a cell array listing the full path to the uploaded files on the server.

See Also

`ftp`, `mget`, `mkdir (ftp)`, `rename`

Purpose

Create and open message box

Syntax

```
h = msgbox(Message)
h = msgbox(Message,Title)
h = msgbox(Message,Title,Icon)
h = msgbox(Message,Title,'custom',IconData,IconCMap)
h = msgbox(...,CreateMode)
```

Description

`h = msgbox(Message)` creates a message dialog box that automatically wraps `Message` to fit an appropriately sized figure. `Message` is a string vector, string matrix, or cell array. `msgbox` returns the handle of the message box in `h`.

`h = msgbox(Message,Title)` specifies the title of the message box.

`h = msgbox(Message,Title,Icon)` specifies which icon to display in the message box. `Icon` is `'none'`, `'error'`, `'help'`, `'warn'`, or `'custom'`. The default is `'none'`.

Error Icon

Help Icon

Warning Icon

`h = msgbox(Message,Title,'custom',IconData,IconCMap)` defines a customized icon. `IconData` contains image data defining the icon. `IconCMap` is the colormap used for the image.

`h = msgbox(...,`*`CreateMode`*`)` specifies whether the message box is modal or nonmodal. Optionally, it can also specify an interpreter for `Message` and `Title`.

If *`CreateMode`* is a string, it must be one of the values shown in the following table.

CreateMode Value	Description
`'modal'`	Replaces the message box having the specified `Title`, that was last created or clicked on, with a modal message box as specified. All other message boxes with the same title are deleted. The message box which is replaced can be either modal or nonmodal.
`'non-modal'` (default)	Creates a new nonmodal message box with the specified parameters. Existing message boxes with the same title are not deleted.
`'replace'`	Replaces the message box having the specified `Title`, that was last created or clicked on, with a nonmodal message box as specified. All other message boxes with the same title are deleted. The message box which is replaced can be either modal or nonmodal.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function.

If you open a dialog with `errordlg`, `msgbox`, or `warndlg` using `'CreateMode','modal'` and a non-modal dialog created with any of these functions is already present and *has the same name as the modal dialog*, the non-modal dialog closes when the modal one opens.

For more information about modal dialog boxes, see `WindowStyle` in the Figure Properties.

If *`CreateMode`* is a structure, it can have fields `WindowStyle` and `Interpreter`. The `WindowStyle` field must be one of the values in the table above. `Interpreter` is one of the strings`'tex'` or `'none'`. The default value for `Interpreter` is `'none'`.

See Also

`dialog`, `errordlg`, `helpdlg`, `inputdlg`, `listdlg`, `questdlg`, `warndlg`

`figure`, `textwrap`, `uiwait`, `uiresume`

for related functions

Purpose

Matrix multiplication

Syntax

```
C = A*B
```

Description

`C = A*B` is the linear algebraic product of the matrices `A` and `B`. If `A` is an m-by-p and `B` is a p-by-n matrix, the `i,j` entry of `C` is defined by

$$C(i, j) = \sum_{k=1}^{p} A(i, k)B(k, j)$$

The product `C` is an m-by-n matrix. For nonscalar `A` and `B`, the number of columns of `A` must equal the number of rows of `B`. You can multiply a scalar by a matrix of any size.

The preceding definition says that `C(i,j)` is the inner product of the `i`th row of `A` with the `j`th column of `B`. You can write this definition using the MATLAB colon operator as

```
C(i,j) = A(i,:)*B(:,j)
```

where `A(i,:)` is the `i`th row of `A` and `B(:,j)` is the `j`th row of `B`.

Note If `A` is an m-by-0 empty matrix and `B` is a 0-by-n empty matrix, where m and n are positive integers, `A*B` is an m-by-n matrix of all zeros.

Examples

Example 1

If `A` is a row vector and `B` is a column vector with the same number of elements as `A`, `A*B` is simply the inner product of `A` and `B`. For example,

```
A = [5 3 2 6]

A =

     5     3     2     6
```

```
B = [-4 9 0 1]'

B =

    -4
     9
     0
     1
A*B

ans =

    13
```

Example 2

```
A = [1 3 5; 2 4 7]

A =

     1     3     5
     2     4     7
B = [-5 8 11; 3 9 21;4 0 8]

B =

    -5     8    11
     3     9    21
     4     0     8
```

The product of `A` and `B` is

```
C = A*B

C =

    24    35   114
    30    52   162
```

Note that the second row of A is

```
A(2,:)

ans =

     2     4     7
```

while the third column of B is

```
B(:,3)

ans =

    11
    21
     8
```

The inner product of `A(2,:)` and `B(:,3)` is

```
A(2,:)*B(:,3)

ans =

   162
```

which is the same as `C(2,3)`.

Algorithm

`mtimes` uses the following Basic Linear Algebra Subroutines (BLAS):

- `DDOT`
- `DGEMV`
- `DGEMM`
- `DSYRK`
- `DSYRZK`

For inputs of type `single`, mtimes using corresponding routines that begin with “`S`” instead of “`D`”.

See Also Arithmetic Operators

mu2lin

Purpose Convert mu-law audio signal to linear

Syntax `y = mu2lin(mu)`

Description `y = mu2lin(mu)` converts mu-law encoded 8-bit audio signals, stored as “flints” in the range `0` ≤ `mu` ≤ `255`, to linear signal amplitude in the range `-s < Y < s` where `s = 32124/32768 ~= .9803`. The input `mu` is often obtained using `fread(...,'uchar')` to read byte-encoded audio files. "Flints" are MATLAB integers — floating-point numbers whose values are integers.

See Also `auread`, `lin2mu`

Purpose

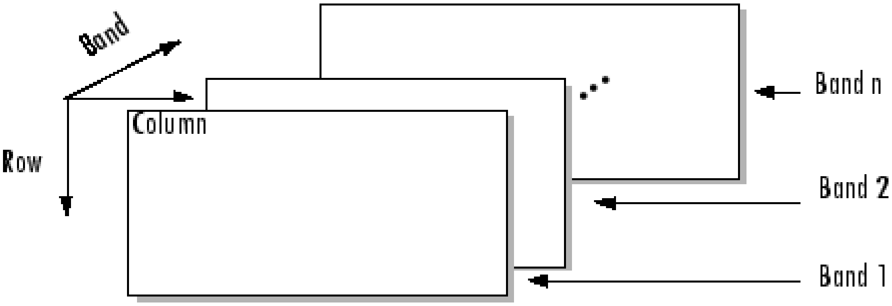
Read band-interleaved data from binary file

Syntax

```
X = multibandread(filename, size, precision, offset,
   interleave,  byteorder)
X = multibandread(...,subset1,subset2,subset3)
```

Description

`X = multibandread(filename, size, precision, offset, interleave, byteorder)` reads band-sequential (BSQ), band-interleaved-by-line (BIL), or band-interleaved-by-pixel (BIP) data from the binary file `filename`. The `filename` input is a string enclosed in single quotes. This function defines *band* as the third dimension in a 3-D array, as shown in this figure.



You can use the parameters to `multibandread` to specify many aspects of the read operation, such as which bands to read. See "Parameters" on page 2-2466 for more information.

`X` is a 2-D array if only one band is read; otherwise it is 3-D. `X` is returned as an array of data type `double` by default. Use the `precision` parameter to map the data to a different data type.

`X = multibandread(...,subset1,subset2,subset3)` reads a subset of the data in the file. You can use up to three subsetting parameters to specify the data subset along row, column, and band dimensions. See "Subsetting Parameters" on page 2-2467 for more information.

Note In addition to BSQ, BIL, and BIP files, multiband imagery may be stored using the TIFF file format. In that case, use the `imread` function to import the data.

Parameters

This table describes the arguments accepted by `multibandread`.

Argument	**Description**
`filename`	String containing the name of the file to be read.
`size`	Three-element vector of integers consisting of `[height, width, N]`, where • `height` is the total number of rows • `width` is the total number of elements in each row • `N` is the total number of bands. This will be the dimensions of the data if it is read in its entirety.
`precision`	String specifying the format of the data to be read, such as `'uint8'`, `'double'`, `'integer*4'`, or any of the other precisions supported by the `fread` function. Note: You can also use the `precision` parameter to specify the format of the output data. For example, to read `uint8` data and output a `uint8` array, specify a precision of `'uint8=>uint8'` (or `'*uint8'`). To read `uint8` data and output it in the MATLAB software in single precision, specify `'uint8=>single'`. See `fread` for more information.

Argument	Description
`offset`	Scalar specifying the zero-based location of the first data element in the file. This value represents the number of bytes from the beginning of the file to where the data begins.
`interleave`	String specifying the format in which the data is stored • `'bsq`' — Band-Sequential • `'bil'`— Band-Interleaved-by-Line • `'bip'`— Band-Interleaved-by-Pixel For more information about these interleave methods, see the `multibandwrite` reference page.
`byteorder`	String specifying the byte ordering (machine format) in which the data is stored, such as • `'ieee-le'` — Little-endian • `'ieee-be'` — Big-endian See `fopen` for a complete list of supported formats.

Subsetting Parameters

You can specify up to three subsetting parameters. Each subsetting parameter is a three-element cell array, `{`*`dim`*`, `*`method`*`, index}`, where

Parameter	Description
`dim`	Text string specifying the dimension to subset along. It can have any of these values: • `'Column'` • `'Row'` • `'Band'`

Parameter	**Description**
`method`	Text string specifying the subsetting method. It can have either of these values: • `'Direct'` • `'Range'` If you leave out this element of the subset cell array, `multibandread` uses `'Direct'` as the default.
`index`	If `method` is `'Direct'`, `index` is a vector specifying the indices to read along the Band dimension. If `method` is `'Range'`, `index` is a three-element vector of `[start, increment, stop]` specifying the range and step size to read along the dimension specified in `dim`. If `index` is a two-element vector, `multibandread` assumes that the value of `increment` is `1`.

Examples

Example 1

Setup initial parameters for a data set.

```
rows=3; cols=3; bands=5;
filename = tempname;
```

Define the data set.

```
fid = fopen(filename, 'w', 'ieee-le');
fwrite(fid, 1:rows*cols*bands, 'double');
fclose(fid);
```

Read every other band of the data using the Band-Sequential format.

```
im1 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bsq', 'ieee-le', ...
```

```
                          {'Band', 'Range', [1 2 bands]} )
```

Read the first two rows and columns of data using Band-Interleaved-by-Pixel format.

```
im2 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bip', 'ieee-le', ...
                    {'Row', 'Range', [1 2]}, ...
                    {'Column', 'Range', [1 2]} )
```

Read the data using Band-Interleaved-by-Line format.

```
im3 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bil', 'ieee-le')
```

Delete the file created in this example.

```
delete(filename);
```

Example 2

Read `int16` BIL data from the FITS file `tst0012.fits`, starting at byte 74880.

```
im4 = multibandread('tst0012.fits', [31 73 5], ...
                    'int16', 74880, 'bil', 'ieee-be', ...
                    {'Band', 'Range', [1 3]} );
im5 = double(im4)/max(max(max(im4)));
imagesc(im5);
```

See Also

`fread`, `fwrite`, `imread`, `memmapfile`, `multibandwrite`

multibandwrite

Purpose

Write band-interleaved data to file

Syntax

```
multibandwrite(data,filename,interleave)
multibandwrite(data,filename,interleave,start,totalsize)
multibandwrite(...,param,value...)
```

Description

`multibandwrite(data,filename,interleave)` writes `data`, a two- or three-dimensional numeric or logical array, to the binary file specified by `filename`. The `filename` input is a string enclosed in single quotes. The length of the third dimension of `data` determines the number of bands written to the file. The bands are written to the file in the form specified by `interleave`. See "Interleave Methods" on page 2-2472 for more information about this argument.

If filename already exists, `multibandwrite` overwrites it unless you specify the optional `offset` parameter. See the last alternate syntax for `multibandwrite` for information about other optional parameters.

`multibandwrite(data,filename,interleave,start,totalsize)` writes `data` to the binary file `filename` in chunks. In this syntax, `data` is a subset of the complete data set.

`start` is a 1-by-3 array `[firstrow firstcolumn firstband]` that specifies the location to start writing data. `firstrow` and `firstcolumn` specify the location of the upper left image pixel. `firstband` gives the index of the first band to write. For example, `data(I,J,K)` contains the data for the pixel at `[firstrow+I-1, firstcolumn+J-1]` in the `(firstband+K-1)`-th band.

`totalsize` is a 1-by-3 array, `[totalrows,totalcolumns,totalbands]`, which specifies the full, three-dimensional size of the data to be written to the file.

Note In this syntax, you must call `multibandwrite` multiple times to write all the data to the file. The first time it is called, `multibandwrite` writes the complete file, using the fill value for all values outside the data subset. In each subsequent call, `multibandwrite` overwrites these fill values with the data subset in `data`. The parameters `filename`, `interleave`, `offset`, and `totalsize` must remain constant throughout the writing of the file.

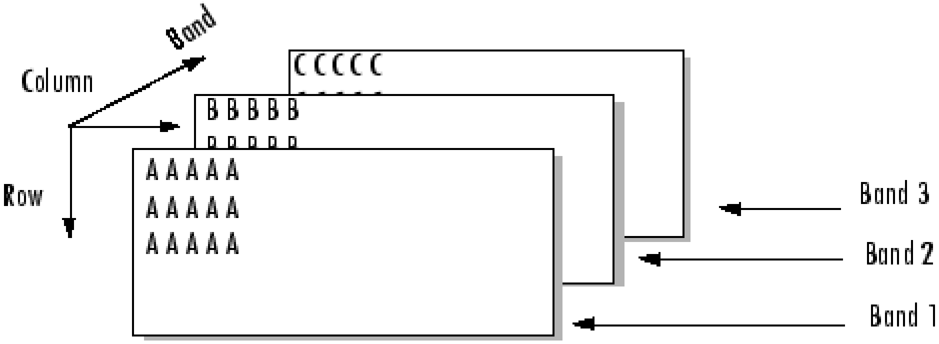
`multibandwrite(...,param,value...)` writes the multiband data to a file, specifying any of these optional parameter/value pairs.

Parameter	Description
`'precision'`	String specifying the form and size of each element written to the file. See the help for `fwrite` for a list of valid values. The default precision is the class of the data.
`'offset'`	The number of bytes to skip before the first data element. If the file does not already exist, `multibandwrite` writes ASCII null values to fill the space. To specify a different fill value, use the parameter `'fillvalue'`. This option is useful when you are writing a header to the file before or after writing the data. When writing the header to the file after the data is written, open the file with `fopen` using `'r+'` permission.

Parameter	Description
`'machfmt'`	String to control the format in which the data is written to the file. Typical values are `'ieee-le'` for little endian and `'ieee-be'` for big endian. See the help for `fopen` for a complete list of available formats. The default machine format is the local machine format.
`'fillvalue'`	A number specifying the value to use in place of missing data. `'fillvalue'` can be a single number, specifying the fill value for all missing data, or a 1-by-Number-of-bands vector of numbers specifying the fill value for each band. This value is used to fill space when data is written in chunks.

Interleave Methods

`interleave` is a string that specifies how `multibandwrite` interleaves the bands as it writes data to the file. If `data` is two-dimensional, `multibandwrite` ignores the `interleave` argument. The following table lists the supported methods and uses this example multiband file to illustrate each method.



Supported methods of interleaving bands include those listed below.

Method	String	Description	Example
Band-Interleaved-by-Line	`'bil'`	Write an entire row from each band	AAAAABBBBBCCCCC AAAAABBBBBCCCCC AAAAABBBBBCCCCC
Band-Interleaved-by-Pixel	`'bip'`	Write a pixel from each band	ABCABCABCABCABC...
Band-Sequential	`'bsq'`	Write each band in its entirety	AAAAA AAAAA AAAAA BBBBB BBBBB BBBBB CCCCC CCCCC CCCCC

Examples

Note To run these examples successfully, you must be in a writable directory.

Example 1

Write all data (interleaved by line) to the file in one call.

```
data = reshape(uint16(1:600), [10 20 3]);
multibandwrite(data,'data.bil','bil');
```

Example 2

Write the bands (interleaved by pixel) to the file in separate calls.

```
totalRows    = size(data, 1);
totalColumns = size(data, 2);
totalBands   = size(data, 3);
for i = 1:totalBands
   bandData = data(:, :, i);
   multibandwrite(bandData, 'data.bip', 'bip', [1 1 i],...
                  [totalColumns, totalRows, totalBands]);
end
```

Example 3

Write a single-band tiled image with one call for each tile. This is only useful if a subset of each band is available at each call to `multibandwrite`.

```
numBands = 1;
dataDims = [1024 1024 numBands];
data = reshape(uint32(1:(1024 * 1024 * numBands)), dataDims);

for band = 1:numBands
   for row = 1:2
      for col = 1:2

         subsetRows = ((row - 1) * 512 + 1):(row * 512);
         subsetCols = ((col - 1) * 512 + 1):(col * 512);

         upperLeft = [subsetRows(1), subsetCols(1), band];
         multibandwrite(data(subsetRows, subsetCols, band), ...
                        'banddata.bsq', 'bsq', upperLeft, dataDims);

      end
   end
```

```
    end
```

See Also `multibandread`, `fwrite`, `fread`

munlock

Purpose Allow clearing M-file or MEX-file from memory

Syntax

```
munlock
munlock fun
munlock('fun')
```

Description `munlock` unlocks the currently running M-file or MEX-file in memory so that subsequent `clear` functions can remove it.

`munlock fun` unlocks the M-file or MEX-file named `fun` from memory. By default, these files are unlocked so that changes to the file are picked up. Calls to munlock are needed only to unlock M-files or MEX-files that have been locked with `mlock`.

`munlock('fun')` is the function form of `munlock`.

Examples The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
  .
  .
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked testfun
ans =
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock testfun

mislocked testfun
ans =
```

```
       0
```

See Also

`mlock`, `mislocked`, `persistent`

namelengthmax

Purpose

Maximum identifier length

Syntax

```
len = namelengthmax
```

Description

`len = namelengthmax` returns the maximum length allowed for MATLAB identifiers. MATLAB identifiers are

- Variable names
- Function and subfunction names
- Structure fieldnames
- Object names
- M-file names
- MEX-file names
- MDL-file names

Rather than hard-coding a specific maximum name length into your programs, use the `namelengthmax` function. This saves you the trouble of having to update these limits should the identifier length change in some future MATLAB release.

Examples

Call `namelengthmax` to get the maximum identifier length:

```
maxid = namelengthmax
maxid =
    63
```

See Also

`isvarname`, `genvarname`

Purpose Not-a-Number

Syntax `NaN`

Description `NaN` returns the IEEE arithmetic representation for Not-a-Number (`NaN`). These result from operations which have undefined numerical results.

`NaN('double')` is the same as `NaN` with no inputs.

`NaN('single')` is the single precision representation of `NaN`.

`NaN(n)` is an `n`-by-`n` matrix of `NaN`s.

`NaN(m,n)` or `NaN([m,n])` is an `m`-by-`n` matrix of `NaN`s.

`NaN(m,n,p,...)` or `NaN([m,n,p,...])` is an `m`-by-`n`-by-`p`-by-... array of `NaN`s.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`NaN(...,classname)` is an array of `NaN`s of class specified by `classname`. `classname` must be either `'single'` or `'double'`.

Examples These operations produce `NaN`:

- Any arithmetic operation on a `NaN`, such as `sqrt(NaN)`
- Addition or subtraction, such as magnitude subtraction of infinities as `(+Inf)+(-Inf)`
- Multiplication, such as `0*Inf`
- Division, such as `0/0` and `Inf/Inf`
- Remainder, such as `rem(x,y)` where `y` is zero or `x` is infinity

Remarks

Because two `NaN`s are not equal to each other, logical operations involving `NaN`s always return false, except ~= (not equal). Consequently,

```
NaN ~= NaN
ans =
     1
NaN == NaN
ans =
     0
```

and the `NaN`s in a vector are treated as different unique elements.

```
unique([1 1 NaN NaN])
ans =
     1 NaN NaN
```

Use the `isnan` function to detect `NaN`s in an array.

```
isnan([1 1 NaN NaN])
ans =
     0     0     1     1
```

See Also

`Inf`, `isnan`

Purpose

Validate number of input arguments

Syntax

```
msgstring = nargchk(minargs, maxargs, numargs)
msgstring = nargchk(minargs, maxargs, numargs, 'string')
msgstruct = nargchk(minargs, maxargs, numargs, 'struct')
```

Description

Use `nargchk` inside an M-file function to check that the desired number of input arguments is specified in the call to that function.

`msgstring = nargchk(minargs, maxargs, numargs)` returns an error message string `msgstring` if the number of inputs specified in the call `numargs` is less than `minargs` or greater than `maxargs`. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty matrix.

It is common to use the `nargin` function to determine the number of input arguments specified in the call.

`msgstring = nargchk(minargs, maxargs, numargs, 'string')` is essentially the same as the command shown above, as `nargchk` returns a string by default.

`msgstruct = nargchk(minargs, maxargs, numargs, 'struct')` returns an error message structure `msgstruct` instead of a string. The fields of the return structure contain the error message string and a message identifier. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty structure.

When too few inputs are supplied, the message string and identifier are

```
   message: 'Not enough input arguments.'
identifier: 'MATLAB:nargchk:notEnoughInputs'
```

When too many inputs are supplied, the message string and identifier are

```
   message: 'Too many input arguments.'
identifier: 'MATLAB:nargchk:tooManyInputs'
```

Remarks

`nargchk` is often used together with the `error` function. The `error` function accepts either type of return value from `nargchk`: a message string or message structure. For example, this command provides the `error` function with a message string and identifier regarding which error was caught:

```
error(nargchk(2, 4, nargin, 'struct'))
```

If `nargchk` detects no error, it returns an empty string or structure. When `nargchk` is used with the `error` function, as shown here, this empty string or structure is passed as an input to `error`. When `error` receives an empty string or structure, it simply returns and no error is generated.

Examples

Given the function `foo`,

```
function f = foo(x, y, z)
error(nargchk(2, 3, nargin))
```

Then typing `foo(1)` produces

```
Not enough input arguments.
```

See Also

`nargoutchk`, `nargin`, `nargout`, `varargin`, `varargout`, `error`

Purpose

Number of function arguments

Syntax

```
nargin
nargin(fun)
nargout
nargout(fun)
```

Description

In the body of a function M-file, `nargin` and `nargout` indicate how many input or output arguments, respectively, a user has supplied. Outside the body of a function M-file, `nargin` and `nargout` indicate the number of input or output arguments, respectively, for a given function. The number of arguments is negative if the function has a variable number of arguments.

`nargin` returns the number of input arguments specified for a function.

`nargin(fun)` returns the number of declared inputs for the function `fun`. If the function has a variable number of input arguments, `nargin` returns a negative value. `fun` may be the name of a function, or the name of that map to specific functions.

`nargout` returns the number of output arguments specified for a function.

`nargout(fun)` returns the number of declared outputs for the function `fun`. `fun` may be the name of a function, or the name of that map to specific functions.

Examples

This example shows portions of the code for a function called `myplot`, which accepts an optional number of input and output arguments:

```
function [x0, y0] = myplot(x, y, npts, angle, subdiv)
% MYPLOT  Plot a function.
% MYPLOT(x, y, npts, angle, subdiv)
%     The first two input arguments are
%     required; the other three have default values.
 ...
if nargin < 5, subdiv = 20; end
if nargin < 4, angle = 10; end
```

```
if nargin < 3, npts = 25; end
 ...
if nargout == 0
     plot(x, y)
else
     x0 = x;
     y0 = y;
end
```

See Also `inputname`, `varargin`, `varargout`, `nargchk`, `nargoutchk`

Purpose

Validate number of output arguments

Syntax

```
msgstring = nargoutchk(minargs, maxargs, numargs)
msgstring = nargoutchk(minargs, maxargs, numargs, 'string')
msgstruct = nargoutchk(minargs, maxargs, numargs, 'struct')
```

Description

Use `nargoutchk` inside an M-file function to check that the desired number of output arguments is specified in the call to that function.

`msgstring = nargoutchk(minargs, maxargs, numargs)` returns an error message string `msgstring` if the number of outputs specified in the call, `numargs`, is less than `minargs` or greater than `maxargs`. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargoutchk` returns an empty matrix.

It is common to use the `nargout` function to determine the number of output arguments specified in the call.

`msgstring = nargoutchk(minargs, maxargs, numargs, 'string')` is essentially the same as the command shown above, as `nargoutchk` returns a string by default.

`msgstruct = nargoutchk(minargs, maxargs, numargs, 'struct')` returns an error message structure `msgstruct` instead of a string. The fields of the return structure contain the error message string and a message identifier. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargoutchk` returns an empty structure.

When too few outputs are supplied, the message string and identifier are

```
   message: 'Not enough output arguments.'
identifier: 'MATLAB:nargoutchk:notEnoughOutputs'
```

When too many outputs are supplied, the message string and identifier are

```
   message: 'Too many output arguments.'
identifier: 'MATLAB:nargoutchk:tooManyOutputs'
```

Remarks

`nargoutchk` is often used together with the `error` function. The `error` function accepts either type of return value from `nargoutchk`: a message string or message structure. For example, this command provides the `error` function with a message string and identifier regarding which error was caught:

```
error(nargoutchk(2, 4, nargout, 'struct'))
```

If `nargoutchk` detects no error, it returns an empty string or structure. When `nargoutchk` is used with the `error` function, as shown here, this empty string or structure is passed as an input to `error`. When `error` receives an empty string or structure, it simply returns and no error is generated.

Examples

You can use `nargoutchk` to determine if an M-file has been called with the correct number of output arguments. This example uses `nargout` to return the number of output arguments specified when the function was called. The function is designed to be called with one, two, or three output arguments. If called with no arguments or more than three arguments, `nargoutchk` returns an error message:

```
function [s, varargout] = mysize(x)
msg = nargoutchk(1, 3, nargout);
if isempty(msg)
    nout = max(nargout, 1) - 1;
    s = size(x);
    for k = 1:nout, varargout(k) = {s(k)}; end
else
    disp(msg)
end
```

See Also

`nargchk`, `nargout`, `nargin`, `varargout`, `varargin`, `error`

Purpose

Convert numeric bytes to Unicode characters

Syntax

```
unicodestr = native2unicode(bytes)
unicodestr = native2unicode(bytes, encoding)
```

Description

`unicodestr = native2unicode(bytes)` takes a vector containing numeric values in the range [0,255] and converts these values as a stream of 8-bit bytes to Unicode characters. The stream of bytes is assumed to be in the MATLAB default character encoding scheme. Return value `unicodestr` is a `char` vector that has the same general array shape as `bytes`.

`unicodestr = native2unicode(bytes, encoding)` does the conversion with the assumption that the byte stream is in the character encoding scheme specified by the string `encoding`. `encoding` must be the empty string (`''`) or a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. For common names and aliases, see the Web site `http://www.iana.org/assignments/character-sets`. If `encoding` is unspecified or is the empty string (`''`), the MATLAB default encoding scheme is used.

Note If `bytes` is a `char` vector, it is returned unchanged.

Examples

This example begins with a vector of bytes in an unknown character encoding scheme. The user-written function `detect_encoding` determines the encoding scheme. If successful, it returns the encoding scheme name or alias as a string. If unsuccessful, it throws an error represented by an MException object, `ME`. The example calls `native2unicode` to convert the bytes to Unicode characters:

```
try
   enc = detect_encoding(bytes);
    str = native2unicode(bytes, enc);
    disp(str);
```

```
catch ME
    rethrow(ME);
end
```

Note that the computer must be configured to display text in a language represented by the detected encoding scheme for the output of `disp(str)` to be correct.

See Also `unicode2native`

Purpose

Binomial coefficient or all combinations

Syntax

```
C = nchoosek(n,k)
C = nchoosek(v,k)
```

Description

`C = nchoosek(n,k)` where `n` and `k` are nonnegative integers, returns $n!/((n-k)!\ k!)$. This is the number of combinations of *n* things taken *k* at a time.

`C = nchoosek(v,k)`, where `v` is a row vector of length `n`, creates a matrix whose rows consist of all possible combinations of the *n* elements of `v` taken *k* at a time. Matrix `C` contains $n!/((n-k)!\ k!)$ rows and *k* columns.

Inputs `n`, `k`, and `v` support classes of `float double` and `float single`.

Examples

The command `nchoosek(2:2:10,4)` returns the even numbers from two to ten, taken four at a time:

```
     2     4     6     8
     2     4     6    10
     2     4     8    10
     2     6     8    10
     4     6     8    10
```

Limitations

When `C = nchoosek(n,k)` has a large coefficient, a warning will be produced indicating possible inexact results. In such cases, the result is only accurate to 15 digits for double-precision inputs, or 8 digits for single-precision inputs.

`C = nchoosek(v,k)` is only practical for situations where `n` is less than about 15.

See Also

`perms`

ndgrid

Purpose

Generate arrays for N-D functions and interpolation

Syntax

```
[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)
[X1,X2,...] = ndgrid(x)
```

Description

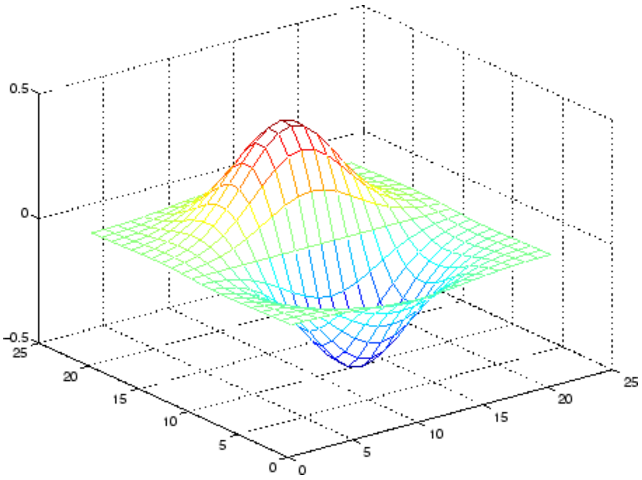
`[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)` transforms the domain specified by vectors `x1,x2,x3...` into arrays `X1,X2,X3...` that can be used for the evaluation of functions of multiple variables and multidimensional interpolation. The `i`th dimension of the output array `Xi` are copies of elements of the vector `xi`.

`[X1,X2,...] = ndgrid(x)` is the same as `[X1,X2,...] = ndgrid(x,x,...)`.

Examples

Evaluate the function $x_1 e^{-x_1^2 - x_2^2}$ over the range $-2 < x_1 < 2, -2 < x_2 < 2$.

```
[X1,X2] = ndgrid(-2:.2:2, -2:.2:2);
Z = X1 .* exp(-X1.^2 - X2.^2);
mesh(Z)
```



Remarks

The `ndgrid` function is like `meshgrid` except that the order of the first two input arguments are switched. That is, the statement

```
[X1,X2,X3] = ndgrid(x1,x2,x3)
```

produces the same result as

```
[X2,X1,X3] = meshgrid(x2,x1,x3)
```

Because of this, `ndgrid` is better suited to multidimensional problems that aren't spatially based, while `meshgrid` is better suited to problems in two- or three-dimensional Cartesian space.

See Also

`meshgrid`, `interpn`

ndims

Purpose Number of array dimensions

Syntax `n = ndims(A)`

Description `n = ndims(A)` returns the number of dimensions in the array `A`. The number of dimensions in an array is always greater than or equal to `2`. Trailing singleton dimensions are ignored. A singleton dimension is any dimension for which `size(A,dim) = 1`.

Algorithm `ndims(x)` is `length(size(x))`.

See Also `size`

Purpose

Test for inequality

Syntax

```
A ~= B
ne(A, B)
```

Description

`A ~= B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` and `B` are unequal, or logical 0 (`false`) where they are equal. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLAB software returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`ne(A, B)` is called for the syntax `A ~= B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are not equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(magic(3), 2, 2);

A ~= B
ans =
     1     0     0     1     1     1
     0     1     0     1     1     1
     1     0     0     1     1     1
     0     1     1     1     1     1
     1     0     1     1     1     1
```

```
     0     1     1     1     1     1
```

See Also

`eq`, `le`, `ge`, `lt`, `gt`, relational operators

Purpose

Point closest to specified location

Syntax

```
PI = nearestNeighbor(DT, QX)
PI = nearestNeighbor(DT, QX,QY)
PI = nearestNeighbor(DT, QX,QY,QZ)
```

Description

`PI = nearestNeighbor(DT, QX)` returns the index of nearest point in `DT.X` for each query point location in `QX`.

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

Note Note: `nearestNeighbor` is not supported for 2-D triangulations that have constrained edges.

Inputs

`DT`	Delaunay triangulation.
`QX`	The matrix `QX` is of size `mpts`-by-`ndim`, `mpts` being the number of query points and `ndim` the dimension of the space where the points reside.

Outputs

`PI`	`PI` is a column vector of point indices that index into the points `DT.X`. The length of `PI` is equal to the number of query points `mpts`.

Examples

Create a Delaunay triangulation:

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

Create query points:

```
qrypts = [0.25 0.25; 0.5 0.5];
```

Find the nearest neighbors to the query points:

```
pid = nearestNeighbor(dt, qrypts)
```

See Also

`DelaunayTri.pointLocation`

Purpose

Compare `MException` objects for inequality

Syntax

```
eObj1 ~= eObj2
```

Description

`eObj1 ~= eObj2` tests `MException` objects `eObj1` and `eObj2` for inequality, returning logical 1 (`true`) if the two objects are not identical, otherwise returning logical 0 (`false`).

See Also

`try`, `catch`, `error`, `assert`, `MException`, `isequal(MException)`, `eq(MException)`, `getReport(MException)`, `disp(MException)`, `throw(MException)`, `rethrow(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `last(MException)`

TriRep.neighbors

Purpose Simplex neighbor information

Syntax `SN = neighbors(TR, SI)`

Description `SN = neighbors(TR, SI)` returns the simplex neighbor information for the specified simplices `SI`.

Inputs

`TR`	Triangulation representation.
`SI`	`SI` is a column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the neighbor information for the entire triangulation is returned, where the neighbors associated with simplex `i` are defined by the `i`'th row of `SN`.

Outputs

`SN`	`SN` is an `m`-by-`n` matrix, where `m = length(SI)` , the number of specified simplices, and `n` is the number of neighbors per simplex. Each row `SN(i,:)` represents the neighbors of the simplex `SI(i)`. By convention, the simplex opposite `vertex(j)` of simplex `SI(i)` is `SN(i,j)`. If a simplex has one or more boundary facets, the nonexistent neighbors are represented by `NaN`.

Definitions A *simplex* is a triangle/tetrahedron or higher-dimensional equivalent. A *facet* is an edge of a triangle or a face of a tetrahedron.

Examples

Example 1

Load a 3-D triangulation and use `TriRep` to compute the neighbors of all tetrahedra.

```
load tetmesh
```

```
trep = TriRep(tet, X)
nbrs = neighbors(trep)
```

Example 2

Query a 2-D triangulation created using `DelaunayTri`.

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

Find the neighbors of the first triangle:

```
n1 = neighbors(dt, 1)
```

See Also

`DelaunayTri`

Purpose

Summary of functions in MATLAB .NET interface

Description

Use the following functions to bring assemblies from the Microsoft .NET Framework into the MATLAB environment. The functions are implemented as a package called NET. To use these functions, prefix the function name with package name 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

See Also

Purpose

Make .NET assembly visible to MATLAB

Syntax

`asminfo = NET.addAssembly(assemblyname)`

Description

`asminfo = NET.addAssembly(assemblyname)` loads .NET assembly `assemblyname` into MATLAB. The `assemblyname` argument is a string representing the name of a global assembly, a string representing the full path of a private assembly, or an instance of `System.Reflection.AssemblyName` class. Returns `NET.Assembly` object `asminfo`.

For more information about assemblies, see . To find information about the `System.Reflection.AssemblyName` class, see .

Examples

Load an assembly located in the Global Assembly Cache (GAC) by specifying the short name:

```
asm = NET.addAssembly('System.Windows.Forms');
import System.Windows.Forms.*;
MessageBox.Show('Simple Message Box')
```

Use pseudocode to load a private assembly `MLDotNetTest.dll` in the `c:\work` directory:

```
NET.addAssembly('c:\work\MLDotNetTest.dll');
```

See Also

`NET.Assembly`

NET.Assembly class

Purpose	Members of .NET assembly
Description	`NET.Assembly` object returns names of the members of an assembly.
Construction	The `NET.addAssembly` function creates an instance of this class.

Properties

`AssemblyHandle`

> Instance of `System.Reflection.Assembly` class of the added assembly.

`Classes`

> `nClassx1` cell array of class names of the assembly, where `nClass` is the number of classes

`Enums`

> `nEnumx1` cell array of enums of the assembly, where `nEnum` is the number of enums

`Structures`

> `nStructx1` cell array of structures of the assembly, where `nStruct` is the number of structures

`GenericTypes`

> `nGenTypex1` cell array of generic types of the assembly, where `nGenType` is the number of generic types

`Interfaces`

> `nInterfacex1` cell array of interface names of the assembly, where `nInterface` is the number of interfaces

`Delegates`

> `nDelegatex1` cell array of delegates of the assembly, where `nDelegate` is the number of delegates

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

`NET.addAssembly`

How To

-

NET.convertArray

Purpose Convert numeric MATLAB array to .NET array

Syntax `arrObj = NET.convertArray(V, 'arrType', [m,n])`

Description `arrObj = NET.convertArray(V, 'arrType', [m,n])` converts a MATLAB array `V` to a .NET array. Optional value `arrType` is a string representing a namespace-qualified .NET array type. Use optional values `m,n` to convert a MATLAB vector to a two-dimensional .NET array (either 1-by-n or m-by-1). If `V` is a MATLAB vector and you do not specify the number of dimensions and their sizes, the output `arrObj` is a one-dimensional .NET array.

If you do not specify `arrType`, MATLAB converts the type according to the .

Examples

Convert MATLAB Array Example

Convert an array of type `double`.

```
A =[1 2 3 4];
arr = NET.convertArray(A);
class(A)
class(arr)
```

MATLAB displays:

```
ans =
double
ans =
System.Double[]
```

Convert to System.String .NET Type Example

To convert a `char` array into a `System.String` object, type:

```
strArray = System.String('sample string');
```

Convert to Explicit .NET Type Example

Explicitly convert an array of `double` to an array of .NET type `System.Int32`.

```
A =[1 2 3 4];
arr =NET.convertArray(A, 'System.Int32');
class(A)
class(arr)
```

MATLAB displays:

```
ans =
double
ans =
System.Int32[]
```

Convert Multidimensional Array Example

```
A =[1 2 3 4; 5 6 7 8];
arr =NET.convertArray(A);
class(arr)
```

MATLAB displays:

```
ans =
System.Double[,]
```

Convert MATLAB Vector Example

Use the optional array dimension parameter to create a two-dimensional array from a MATLAB vector. To create a 4-by-1 array, type:

```
A =[1 2 3 4];
arr2 = NET.convertArray(A, 'System.Double', [4,1]);
arr2.GetLength(0)
arr2.GetLength(1)
```

MATLAB displays:

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

To create a 1-by-4 array, type:

```
arr3 = NET.convertArray(A, 'System.Double', [1,4]);
arr3.GetLength(0)
arr3.GetLength(1)
```

MATLAB displays:

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

See Also `NET.createArray`

Purpose Create single or multidimensional .NET array

Syntax

```
array = NET.createArray(typeName, [m,n,p,...])
array = NET.createArray(typeName, m,n,p,...)
```

Description `array = NET.createArray(typeName, [m,n,p,...])` creates an `m`-by-`n`-by-`p`-by-`...` MATLAB `array` of type `typeName`, which is either a fully qualified .NET array type name (`namespace` and array type name) or an instance of the `NET.GenericClass` class, in case of arrays of generic type. `m,n,p,...` are the number of elements in each dimension of the array.

`array = NET.createArray(typeName, m,n,p,...)` alternative syntax for creating an array.

You cannot specify the lower bound of an array or create a jagged array.

Examples

Create One-Dimensional Array Example

Create a single dimensional, zero-based array of strings:

```
strArray = NET.createArray('System.String', 3);
class(strArray)
```

MATLAB displays:

```
System.String[]
```

For information about accessing and setting values, see .

Create Two-Dimensional Array Example

Create a two dimensional array:

```
strArray = NET.createArray('System.String', [2,3]);
class(strArray)
```

MATLAB displays:

```
System.String[,]
```

Alternatively, to create the same array type:

```
strArray2 = NET.createArray('System.String',2,3);
```

See Also

`NET.createGeneric`, `NET.GenericClass`

Purpose

Create instance of specialized .NET generic type

Syntax

```
genObj = createGeneric(className, paramTypes,
   varargin ctorArgs)
```

Description

`genObj = createGeneric(className, paramTypes, varargin ctorArgs)` creates an instance `genObj` of generic type `className`.

Inputs

`className`	Fully qualified string with the generic type name.
`paramTypes`	Allowed cell types are: strings with fully qualified parameter type names and instances of the `NET.GenericClass` class when parameterization with another parameterized type is needed.
`ctorArgs`	Optional, variable length (0 to N) list of constructor arguments matching the arguments of the .NET generic class constructor intended to be invoked.

Outputs

`genObj`	Handle to the specialized generic class instance.

Examples

Create an instance of `System.Collections.Generic.List` of `System.Double` values with initial storage capacity for 10 elements.

```
dblLst = NET.createGeneric('System.Collections.Generic.List', ...
  {'System.Double'}, 10);
```

Create an instance of `System.Collections.Generic.List` of `System.Collections.Generic.KeyValuePair` generic associations where `Key` is of `System.Int32` type and `Value` is a `System.String` class with initial storage capacity for 10 key-value pairs.

```
kvpType = NET.GenericClass('System.Collections.Generic.KeyValuePair',.
  'System.Int32', 'System.String');
kvpList = NET.createGeneric('System.Collections.Generic.List',...
  { kvpType }, 10);
```

See Also `NET.GenericClass`

Purpose

Represent parameterized generic type definitions

Description

Instances of this class are used by the `NET.createGeneric` function when creation of generic specialization requires parameterization with another parameterized type.

Construction

NET.GenericClass	Constructor for NET.GenericClass class

Methods

This class has no methods.

Properties

This class has no properties.

See Also

`NET.createGeneric`, `NET.createArray`, `NET.invokeGenericMethod`

NET.GenericClass

Purpose

Constructor for NET.GenericClass class

Syntax

`genType = NET.GenericClass (`*`className`*`, varargin paramTypes)`

Description

`genType = NET.GenericClass (`*`className`*`, varargin paramTypes)` where *`className`* is a fully qualified string with the generic type name, and `paramTypes` is an optional, variable length (1 to N) list of types for the generic class parameterization. Allowed argument types are: strings with fully qualified parameter type name and instances of the `NET.GenericClass` class when deeper nested parameterization with another parameterized type is needed.

Examples

Create an instance of `System.Collections.Generic.List` of `System.Collections.Generic.KeyValuePair` generic associations where `Key` is of `System.Int32` type and `Value` is a `System.String` class with initial storage capacity for 10 key-value pairs.

```
kvpType = NET.GenericClass('System.Collections.Generic.KeyValuePair',.
  'System.Int32', 'System.String');
kvpList = NET.createGeneric('System.Collections.Generic.List',...
  { kvpType }, 10);
```

See Also

`NET.createGeneric`, `NET.createArray`

Purpose

Invoke generic method of object

Syntax

```
[varargout] = NET.invokeGenericMethod(obj,
   'genericMethodName', paramTypes, args, ...)
```

Description

`[varargout] = NET.invokeGenericMethod(obj, 'genericMethodName', paramTypes, args, ...)` calls instance or static generic method *`genericMethodName`*.

Inputs

`obj`	Allowed argument types are: • Instances of class containing the generic method • Strings with fully qualified class name, if calling static generic methods • Instances of `NET.GenericClass` definitions, if calling static generic methods of a generic class
`genericMethodName`	Generic method name to invoke
`paramTypes`	Cell vector (1 to N) with the types for generic method parameterization, where allowed cell types are: • Strings with fully qualified parameter type name. • Instances of `NET.GenericClass` definitions, if using nested parameterization with another parameterized type
`args`	Optional, variable length (0 to N) list of method arguments

Outputs

`varargout`	Variable-length output argument list, `varargout`, from method *`genericMethodName`*

Examples

Use the following syntax to call a generic method that takes two parameterized types and returns a parameterized type:

```
a = NET.invokeGenericMethod(obj, ...
  'myGenericSwapMethod', ...
  {'System.Double', 'System.Double'}, ...
  5, 6);
```

See Also

`NET.GenericClass`, `NET.createGeneric`, `varargout`

Purpose

.NET exception

Description

Construct a `NET.NetException` object to handle .NET errors.

Construction

`NE = NET.NetException(msgID, errMsg, netObj)` constructs an object `NE` of class `NET.NetException` and assigns to that object a message identifier `msgID` and error message string `errMsg`. The `System.Exception` object that caused the exception is `netObj`. Derived from the `MException` class.

Properties

In addition to the base class `MException` properties, `NET.NetException` includes:

`ExceptionObject`

`System.Exception` class causing the error.

Methods

Inherited Methods

See the methods of the base class `MException`.

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Display error information after trying to load an unknown assembly:

```
try
  NET.addAssembly('C:\Work\invalidfile.dll')
catch e
  e.message;
  if(isa(e, 'NET.NetException'))
    eObj = e.ExceptionObject
  end
end
```

MATLAB displays:

```
ans =
Message: Could not load file or assembly
  'file:///C:\Work\invalidfile.dll' or
  one of its dependencies. The system cannot
  find the file specified.
Source: mscorlib
HelpLink:

eObj =
  System.IO.FileNotFoundException handle
  Package: System.IO

  Properties:
           Message: [1x1 System.String]
          FileName: [1x1 System.String]
         FusionLog: [1x1 System.String]
              Data: [1x1 System.Collections.ListDictionaryInternal]
    InnerException: []
        TargetSite: [1x1 System.Reflection.RuntimeMethodInfo]
        StackTrace: [1x1 System.String]
          HelpLink: []
            Source: [1x1 System.String]
  Methods, Events, Superclasses
```

See Also

`MException`

How To

- Class Attributes
- Property Attributes

Purpose Static property or field name

Syntax

```
NET.setStaticProperty('propName', value)
```

Description `NET.setStaticProperty('propName', value)` sets the static property or field name specified in the string `propName` to the given `value`.

Examples To set the `myStaticProperty` in the given class and namespace, use the syntax:

```
NET.setStaticProperty('MyTestObject.MyClass.myStaticProperty', 5);
```

Purpose

Summary of MATLAB Network Common Data Form (netCDF) capabilities

Description

MATLAB provides access to more than 30 functions in the Network Common Data Form (netCDF) interface. This interface provides an API that you can use to enable reading data from and writing data to netCDF files (known as *datasets* in netCDF terminology).

To use these functions, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Note For information about MATLAB support for the Common Data Format (CDF), which is a completely separate, incompatible format, see and .

In most cases, the syntax of the MATLAB function is similar to the syntax of the netCDF library function. The functions are implemented as a package called `netcdf`. To use these functions, prefix the function name with package name `netcdf`. For example, to call the netCDF library routine used to open existing netCDF files, use the following MATLAB syntax:

```
ncid = netcdf.open( ncfile, mode );
```

netCDF Library Functions

The following tables list all of the functions in the MATLAB netCDF package, grouped by category.

File Operations

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

Purpose

Revert recent netCDF file definitions

Syntax

`netcdf.abort(ncid)`

Description

`netcdf.abort(ncid)` reverts a netCDF file to its previous state, backing out any definitions made since the file last entered define mode. A file enters define mode when you create it (using `netcdf.create`) or when you explicitly enter define mode (using `netcdf.redef`). Once you leave define mode (using `netcdf.endDef`), you cannot revert the definitions you made while in define mode. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`. A call to `netcdf.abort` closes the file.

This function corresponds to the `nc_abort` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example creates a new file, performs an operation on the file, and then reverts the file back to its original state. To run this example, you must have write permission in your current directory.

```
% Create a netCDF file
ncid = netcdf.create('foo.nc','NC_NOCLOBBER');

% Perform an operation, such as defining a dimension.
dimid = netcdf.defDim(ncid, 'lat', 50);

% Revert the file back to its previous state.
netcdf.abort(ncid)

% Verify that the file is now closed.
dimid = netcdf.defDim(ncid, 'lat', 50); % should fail
??? Error using ==> netcdflib
NetCDF: Not a valid ID

Error in ==> defDim at 22
```

```
dimid = netcdflib('def_dim', ncid,dimname,dimlen);
```

See Also `netcdf.create`, `netcdf.endDef`, `netcdf.reDef`

Purpose

Close netCDF file

Syntax

`netcdf.close(ncid)`

Description

`netcdf.close(ncid)` terminates access to the netCDF file identified by `ncid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_close` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example creates a new netCDF file, and then closes the file. You must have write permission in your current directory to run this example.

```
ncid = netcdf.open('foo.nc','NC_WRITE')

netcdf.close(ncid)
```

See Also

`netcdf.create`, `netCDF.open`

netcdf.copyAtt

Purpose

Copy attribute to new location

Syntax

`netcdf.copyAtt(ncid_in,varid_in,attname,ncid_out,varid_out)`

Description

`netcdf.copyAtt(ncid_in,varid_in,attname,ncid_out,varid_out)` copies an attribute from one variable to another, possibly across files. `ncid_in` and `ncid_out` are netCDF file identifiers returned by `netcdf.create` or `netcdf.open`. `varid_in` identifies the variable with an attribute that you want to copy. `varid_out` identifies the variable to which you want to associate a copy of the attribute.

This function corresponds to the `nc_copy_att` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example makes a copy of the attribute associated with the first variable in the netCDF example file, `example.nc`, in a new file. To run this example, you must have write permission in your current directory.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get identifier for a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Create new netCDF file.
ncid2 = netcdf.create('foo.nc','NC_NOCLOBBER');

% Define a dimension in the new file.
dimid2 = netcdf.defDim(ncid2,'x',50);

% Define a variable in the new file.
varid2 = netcdf.defVar(ncid2,'myvar','double',dimid2);

% Copy the attribute named 'description' into the new file,
% associating the attribute with the new variable.
```

```
netcdf.copyAtt(ncid,varid,'description',ncid2,varid2);
%
% Check the name of the attribute in new file.
attname = netcdf.inqAttName(ncid2,varid2,0)

attname =

description
```

See Also `netcdf.inqAtt`, `netcdf.inqAttID`, `netcdf.inqAttName`, `netcdf.putAtt`, `netcdf.renameAtt`

Purpose Create new netCDF dataset

Syntax

```
ncid = netcdf.create(filename, mode)
[chunksize_out, ncid]=netcdf.create(filename,mode,initsz,
   chunksize)
```

Description `ncid = netcdf.create(filename, mode)` creates a new netCDF file according to the file creation mode. The return value, `ncid`, is a file ID. The type of access is described by the mode parameter, which can have any of the following values.

Value	Description
`'NC_NOCLOBBER'`	Prevent overwriting of existing file with the
`'NC_SHARE'`	Allow synchronous file updates.
`'NC_64BIT_OFFSET'`	Allow easier creation of files and variables which are larger than two gigabytes.

Note You can specify the mode as a numeric value, retrieved using the `netcdf.getConstant` function. To specify more than one mode, use a bitwise-OR of the numeric values of the modes.

`[chunksize_out, ncid]=netcdf.create(filename,mode,initsz,chunksize)` creates a new netCDF file, but with additional performance tuning parameters. `initsz` sets the initial size of the file. `chunksize` can affect I/O performance. The actual value chosen by the netCDF library might not correspond to the input value.

This function corresponds to the `nc_create` and `nc__create` functions in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example creates a netCDF dataset named `foo.nc`, only if no other file with the same name exists in the current directory. To run this example, you must have write permission in your current directory.

```
ncid = netcdf.create('foo.nc','NC_NOCLOBBER');
```

See Also

`netcdf.getConstant`, `netcdf.open`

netcdf.defDim

Purpose

Create netCDF dimension

Syntax

`dimid = netcdf.defDim(ncid,dimname,dimlen)`

Description

`dimid = netcdf.defDim(ncid,dimname,dimlen)` creates a new dimension in the netCDF file specified by `ncid`, where`dimname` is a character string that specifies the name of the dimension and `dimlen` is a numeric value that specifies its length. To define an unlimited dimension, specify the predefined constant `'NC_UNLIMITED'` for `dimlen`, using `netcdf.getConstant` to retrieve the value.

`netcdf.defDim` returns `dimid`, a numeric ID corresponding to the new dimension.

This function corresponds to the `nc_def_dim` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example creates a new file and defines two dimensions in the file. One dimension is an unlimited dimension. To run this example, you must have write permission in your current directory.

```
% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_NOCLOBBER')

% Define a dimension.
latdimID = netcdf.defDim(ncid,'lat',50);

% Define an unlimited dimension.
londimID = netcdf.defDim(ncid,'lon',netcdf.getConstant('NC_UNLIMITED'))
```

See Also

`netcdf.getConstant`

Purpose Create netCDF variable

Syntax `varid = netcdf.defVar(ncid,varname,xtype,dimids)`

Description `varid = netcdf.defVar(ncid,varname,xtype,dimids)` creates a new variable in the dataset identified by `ncid`.

`varname` is a character string that specifies the name of the variable. `xtype` can be either a character string specifying the data type of the variable, such as `'double'`, or it can be the numeric equivalent returned by the `netcdf.getConstant` function. `dimids` specifies a list of dimension IDs.

`netcdf.defVar` returns `varid`, a numeric identifier for the new variable.

This function corresponds to the `nc_def_var` function in the netCDF library C API. Because MATLAB uses FORTRAN-style ordering, the fastest-varying dimension comes first and the slowest comes last. Any unlimited dimension is therefore last in the list of dimension IDs. This ordering is the reverse of that found in the C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples This example creates a new netCDF file, defines a dimension in the file, and then defines a variable on that dimension. (In netCDF files, you must create a dimension before you can create a variable.) To run this example, you must have write permission in your current directory.

```
% Create  netCDF file.
ncid = netcdf.create('foo.nc','NC_NOCLOBBER');
%
% Define a dimension in the new file.
dimid = netcdf.defDim(ncid,'x',50);

% Define a variable in the new file.
varid = netcdf.defVar(ncid,'myvar','double',dimid)
```

See Also `netCDF.getConstant`, `netCDF.inqVar`, `netCDF.putVar`

netcdf.delAtt

Purpose

Delete netCDF attribute

Syntax

`netcdf.delAtt(ncid,varid,attName)`

Description

`netcdf.delAtt(ncid,varid,attName)` deletes the attribute identified by the text string `attName`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` is a numeric value that identifies the variable. To delete a global attribute, use `netcdf.getConstant('GLOBAL')` for the `varid`. You must be in define mode to delete an attribute.

This function corresponds to the `nc_del_att` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example opens a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open a netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Determine number of global attributes in file.
[numdims numvars numatts unlimdimID] = netcdf.inq(ncid);

numatts =

   1

% Get name of attribute; it is needed for deletion.
attname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)

% Put file in define mode to delete an attribute.
netcdf.reDef(ncid);
```

```
% Delete the global attribute in the netCDF file.
netcdf.delAtt(ncid,netcdf.getConstant('GLOBAL'),attname);

% Verify that the global attribute was deleted.
[numdims numvars numatts unlimdimID] = netcdf.inq(ncid);

numatts =

   0
```

See Also `netcdf.getConstant`, `netcdf.inqAttName`

netcdf.endDef

Purpose

End netCDF file define mode

Syntax

```
netcdf.endDef(ncid)
netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)
```

Description

`netcdf.endDef(ncid)` takes a netCDF file out of define mode and into data mode. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)` takes a netCDF file out of define mode, specifying four additional performance tuning parameters. For example, one reason for using the performance parameters is to reserve extra space in the netCDF file header using the `h_minfree` parameter:

```
ncid = netcdf.endDef(ncid,20000,4,0,4);
```

This reserves 20,000 bytes in the header, which can be used later when adding attributes. This can be extremely efficient when working with very large files. To understand how to use these performance tuning parameters, see the netCDF library documentation.

This function corresponds to the `nc_enddef` and `nc__enddef` functions in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

When you create a file using `netcdf.create`, the functions opens the file in define mode. This example uses `netcdf.endDef` to take the file out of define mode.

```
% Create a netCDF file.
ncid = netcdf.create('foo.c','NC_NOCLOBBER');

% Define a dimension.
dimid = netcdf.defDim(ncid, 'lat', 50);

% Leave define mode.
netcdf.endDef(ncid)
```

```
% Test if still in define mode.
dimid = netcdf.defDim(ncid, 'lon', 50);  % should fail
??? Error using ==> netcdflib
NetCDF: Operation not allowed in data mode

Error in ==> defDim at 22
dimid = netcdflib('def_dim', ncid,dimname,dimlen);
```

See Also

`netcdf.create`, `netcdf.reDef`

netcdf.getAtt

Purpose

Return netCDF attribute

Syntax

```
attrvalue = netcdf.getAtt(ncid,varid,attname)
attrvalue = netcdf.getAtt(ncid,varid,attname,output_datatype)
```

Description

`attrvalue = netcdf.getAtt(ncid,varid,attname)` returns `attrvalue`, the value of the attribute specified by the text string `attname`. When it chooses the data type of `attrvalue`, MATLAB attempts to match the netCDF class of the attribute. For example, if the attribute has the netCDF data type `NC_INT`, MATLAB uses the `int32` class for the output data. If an attribute has the netCDF data type `NC_BYTE`, the class of the output data is `int8` value.

`attrvalue = netcdf.getAtt(ncid,varid,attname,output_datatype)` returns `attrvalue`, the value of the attribute specified by the text string `attname`, using the output class specified by `output_datatype`. You can specify any of the following strings for the output data type.

`'int'`	`'double'`	`'int16'`
`'short'`	`'single'`	`'int8'`
`'float'`	`'int32'`	`'uint8'`

This function corresponds to several attribute I/O functions in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and gets the value of the attribute associated with the first variable. The example also gets the value of the global variable in the file.

```
% Open a netCDF file.
ncid = netcdf.open('example.nc','NC_NOWRITE');
```

```
% Get name of first variable.
[varname vartype vardimIDs varatts] = netcdf.inqVar(ncid,0);

% Get ID of variable, given its name.
varid = netcdf.inqVarID(ncid,varname);

% Get attribute name, given variable id.
attname = netcdf.inqAttName(ncid,varid,0);

% Get value of attribute.
attval = netcdf.getAtt(ncid,varid,attname);

% Get name of global attribute
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0

% Get value of global attribute.
gattval = netcdf.getAtt(ncid,netcdf.getConstant('NC_GLOBAL'),gattna

gattval =

09-Jun-2008
```

See Also

`netcdf.inqAtt`,`netcdf.putAtt`

netcdf.getConstant

Purpose

Return numeric value of named constant

Syntax

`val = netcdf.getConstant(param_name)`

Description

`val = netcdf.getConstant(param_name)` returns the numeric value corresponding to the name of a constant defined by the netCDF library. For example, `netcdf.getConstant('NC_NOCLOBBER')` returns the numeric value corresponding to the netCDF constant `NC_NOCLOBBER`.

The value for `param_name` can be either upper- or lowercase, and does not need to include the leading three characters `'NC_'`. To retrieve a list of all the names defined by the netCDF library, use the `netcdf.getConstantNames` function.

This function has no direct equivalent in the netCDF C interface. To learn about netCDF, see the information contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Determine contents of the file.
[ndims nvars natts dimm] = netcdf.inq(ncid);

% Get name of global attribute.
% Note: You must use netcdf.getConstant to specify NC_GLOBAL.
attname = netcdf.inqattname(ncid,netcdf.getConstant('NC_GLOBAL'),0)

attname =

creation_date
```

See Also

`netcdf.getConstantNames`

Purpose Return list of constants known to netCDF library

Syntax

```
val = netcdf.getConstantNames(param_name)
```

Description `val = netcdf.getConstantNames(param_name)` returns a list of names of netCDF library constants, definitions, and enumerations. When these strings are supplied as actual parameters to MATLAB netCDF package functions, the functions automatically convert the constant to the appropriate numeric value.

This MATLAB function has no direct equivalent in the netCDF C interface. To learn about netCDF, see the information contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

```
nc_constants = netcdf.getConstantNames

nc_constants =

    'NC2_ERR'
    'NC_64BIT_OFFSET'
    'NC_BYTE'
    'NC_CHAR'
    'NC_CLOBBER'
    'NC_DOUBLE'
    'NC_EBADDIM'
    'NC_EBADID'
    'NC_EBADNAME'
    'NC_EBADTYPE'
  ...
```

See Also `netCDF.getConstantNames`

netcdf.getVar

Purpose Return data from netCDF variable

Syntax

```
data = netcdf.getVar(ncid,varid)
data = netcdf.getVar(ncid,varid,start)
data = netcdf.getVar(ncid,varid,start,count)
data = netcdf.getVar(ncid,varid,start,count,stride)
data = netcdf.getVar(...,output_type)
```

Description

`data = netcdf.getVar(ncid,varid)` returns `data`, the value of the variable specified by `varid`. MATLAB attempts to match the class of the output data to netCDF class of the variable.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`data = netcdf.getVar(ncid,varid,start)` returns a single value starting at the specified index, `start`.

`data = netcdf.getVar(ncid,varid,start,count)` returns a contiguous section of a variable. `start` specifies the starting point and `count` specifies the amount of data to return.

`data = netcdf.getVar(ncid,varid,start,count,stride)` returns a subset of a section of a variable. `start` specifies the starting point, `count` specifies the extent of the section, and `stride` specifies which values to return.

`data = netcdf.getVar(...,output_type)` specifies the data type of the return value `data`. For example, to read in an entire integer variable as double precision, use:

```
data=netcdf.getVar(ncid,varid,'double');
```

You can specify any of the following strings for the output data type.

`'int'`	`'double'`	`'int16'`
`'short'`	`'single'`	`'int8'`
`'float'`	`'int32'`	`'uint8'`

This function corresponds to several functions in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and uses several functions to get the value of a variable.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get the name of the first variable.
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

% Get variable ID of the first variable, given its name.
varid = netcdf.inqVarID(ncid,varname);

% Get the value of the first variable, given its ID.
data = netcdf.getVar(ncid,varid)

data =

  6.0221e+023
```

See Also

`netcdf.create` ,`netcdf.inqVarID`, `netcdf.open`

netcdf.inq

Purpose Return information about netCDF file

Syntax `[ndims,nvars,ngatts,unlimdimid] = netcdf.inq(ncid)`

Description `[ndims,nvars,ngatts,unlimdimid] = netcdf.inq(ncid)` returns the number of dimensions, variables, and global attributes in a netCDF file. The function also returns the ID of the dimension defined with unlimited length, if one exists.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`. You can call `netcdf.inq` in either define mode or data mode.

This function corresponds to the `nc_inq` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples This example opens the example netCDF file included with MATLAB, `example.nc`, and uses the `netcdf.inq` function to get information about the contents of the file.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE')

% Get information about the contents of the file.
[numdims, numvars, numglobalatts, unlimdimID] = netcdf.inq(ncid)

numdims =

     4


numvars =

     4
```

```
numglobalatts =

     1


unlimdimID =

     3
```

See Also

`netcdf.create`, `netcdf.open`

netcdf.inqAtt

Purpose Return information about netCDF attribute

Syntax `[xtype,attlen] = netcdf.inqAtt(ncid,varid,attname)`

Description `[xtype,attlen] = netcdf.inqAtt(ncid,varid,attname)` returns the data type, `xtype`, and length, `attlen`, of the attribute identified by the text string `attname`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` identifies the variable that the attribute is associated with. To get information about a global attribute, specify `netcdf.getConstant('NC_GLOBAL')` in place of `varid`.

This function corresponds to the `nc_inq_att` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples This example opens the example netCDF file included with MATLAB, `example.nc`, and gets information about an attribute in the file.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NOWRITE');

% Get identifier of a variable in the file, given its name.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Get attribute name, given variable id and attribute number.
attname = netcdf.inqAttName(ncid,varid,0);

% Get information about the attribute.
[xtype,attlen] = netcdf.inqAtt(ncid,varid,'description')

xtype =

     2
```

```
attlen =

    31

% Get name of global attribute
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0

% Get information about global attribute.
[gxtype gattlen] = netcdf.inqAtt(ncid,netcdf.getConstant('NC_GLOBAL

gxtype =

     2


gattlen =

    11
```

See Also

`netcdf.inqAttID` `netcdf.inqAttName`

netcdf.inqAttID

Purpose Return ID of netCDF attribute

Syntax `attnum = netcdf.inqAttID(ncid,varid,attname)`

Description `attnum = netcdf.inqAttID(ncid,varid,attname)` retrieves `attnum`, the identifier of the attribute specified by the text string `attname`.

`varid` specifies the variable the attribute is associated with.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_attid` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples This example opens the netCDF example file included with MATLAB, `example.nc`.

```
% Open the netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get the identifier of a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Retrieve the identifier of the attribute associated with the variabl
attid = netcdf.inqAttID(ncid,varid,'description');
```

See Also `netcdf.inqAttnetcdf.inqAttName`

Purpose

Return name of netCDF attribute

Syntax

`attname = netcdf.inqAttName(ncid,varid,attnum)`

Description

`attname = netcdf.inqAttName(ncid,varid,attnum)` returns `attname`, a text string specifying the name of an attribute.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` is a numeric identifier of a variable in the file. If you want to get the name of a global attribute in the file, use `netcdf.getConstant('NC_GLOBAL')` in place of `attnum` is a zero-based numeric value specifying the attribute, with 0 indicating the first attribute, 1 the second attribute, and so on.

This function corresponds to the `nc_inq_attname` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get identifier of a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number')

% Get the name of the attribute associated with the variable.
attname = netcdf.inqAttName(ncid,varid,0)

attname =

description

% Get the name of the global attribute associated with the variable
```

```
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)

gattname =

creation_date
```

See Also `netcdf.inqAtt` `netcdf.inqAttID`

Purpose

Return netCDF dimension name and length

Syntax

`[dimname, dimlen] = netcdf.inqDim(ncid,dimid)`

Description

`[dimname, dimlen] = netcdf.inqDim(ncid,dimid)` returns the name, `dimname`, and length, `dimlen`, of the dimension specified by `dimid`. If `ndims` is the number of dimensions defined for a netCDF file, each dimension has an ID between 0 and `ndims-1`. For example, the dimension identifier of the first dimension is 0, the second dimension is 1, and so on.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_dim` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

The example opens the example netCDF file include with MATLAB, `example.nc`.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name and length of first dimension
[dimname, dimlen] = netcdf.inqDim(ncid,0)

dimname =

x


dimlen =

    50
```

See Also

`netcdf.inqDimID`

netcdf.inqDimID

Purpose

Return dimension ID

Syntax

`dimid = netcdf.inqDimID(ncid,dimname)`

Description

`dimid = netcdf.inqDimID(ncid,dimname)` returns `dimid`, the identifier of the dimension specified by the character string `dimname`. You can use the `netcdf.inqDim` function to retrieve the dimension name. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_dimid` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name and length of first dimension
[dimname, dimlen] = netcdf.inqDim(ncid,0);

% Retrieve identifier of dimension.
dimid = netcdf.inqDimID(ncid,dimname)

dimid =

     0
```

See Also

`netcdf.inqDim`

netcdf.inqLibVers

Purpose

Return netCDF library version information

Syntax

`libvers = netcdf.inqLibVers`

Description

`libvers = netcdf.inqLibVers` returns a string identifying the version of the netCDF library.

This function corresponds to the `nc_inq_libvers` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

```
libvers = netcdf.inqLibVers

libvers =

3.6.2
```

netcdf.inqVar

Purpose Return information about variable

Syntax `[varname,xtype,dimids,natts] = netcdf.inqVar(ncid,varid)`

Description `[varname,xtype,dimids,natts] = netcdf.inqVar(ncid,varid)` returns the name, data type, dimensions IDs, and the number of attributes associated with the variable identified by `varid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_var` function in the netCDF library C API. Because MATLAB uses FORTRAN-style ordering, however, the order of the dimension IDs is reversed relative to what would be obtained from the C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples This example opens the example netCDF file included with MATLAB, `example.nc`, and gets information about a variable in the file.

```
% Open the example netCDF file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get information about third variable in the file.
[varname, xtype, dimids, numatts] = netcdf.inqVar(ncid,2)

varname =

peaks


xtype =

     5


dimids =
```

```
          0     1


     numatts =

          1     1
```

See Also `netcdf.create` ,`netcdf.inqVarID`, `netcdf.open`

netcdf.inqVarID

Purpose

Return ID associated with variable name

Syntax

`varid = netcdf.inqVarID(ncid,varname)`

Description

`varid = netcdf.inqVarID(ncid,varname)` returns `varid`, the ID of a netCDF variable specified by the text string, `varname`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_varid` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and uses several inquiry functions to get the ID of the first variable.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get information about first variable in the file.
[varname, xtype, dimids, atts] = netcdf.inqVar(ncid,0);

% Get variable ID of the first variable, given its name
varid = netcdf.inqVarID(ncid,varname)

varid =

     0
```

See Also

`netcdf.create`, `netcdf.inqVar`, `netcdf.open`

Purpose

Open netCDF file

Syntax

```
ncid = netcdf.open(filename, mode)
[chosen_chunksize, ncid] = netcdf.open(filename, mode,
   chunksize)
```

Description

`ncid = netcdf.open(filename, mode)` opens an existing netCDF file and returns a netCDF ID in `ncid`. `mode` describes the type of access to the file and can have any of the following values.

Value	Description
`'NC_WRITE'`	Read-write access
`'NC_SHARE'`	Synchronous file updates
`'NC_NOWRITE'`	Read-only access

You can also specify `mode` as a numeric value that can be retrieved using`netcdf.getConstant`. Use these numeric values when you want to specify a bitwise-OR of several mode.

`[chosen_chunksize, ncid] = netcdf.open(filename, mode, chunksize)` opens an existing netCDF file, specifying an additional I/O performance tuning parameter, `chunksize`. The actual value used by the netCDF library might not correspond to the input value you specify.

This function corresponds to the `nc_open` and `nc__open` functions in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');
```

See Also

`netcdf.create`,`netcdf.getConstant`

netcdf.putAtt

Purpose

Write netCDF attribute

Syntax

`netcdf.putAtt(ncid,varid,attrname,attrvalue)`

Description

`netcdf.putAtt(ncid,varid,attrname,attrvalue)` writes the attribute named `attrname` with value `attrvalue` to the netCDF variable specified by `varid`. To specify a global attribute, use `netcdf.getConstant('NC_GLOBAL')` for `varid`

`ncid` is a netCDF file identifier returned by `netCDF.create` or `netCDF.open`.

This function corresponds to several attribute I/O functions in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example creates a new netCDF file, defines a dimension and a variable, adds data to the variable, and then creates an attribute associated with the variable. To run this example, you must have writer permission in your current directory.

```
% Create a variable in the workspace.
my_vardata = linspace(0,50,50);

% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE');

% Define a dimension in the file.
dimid = netcdf.defDim(ncid,'my_dim',50);

% Define a new variable in the file.
varid = netcdf.defVar(ncid,'my_var','double',dimid);

% Leave define mode and enter data mode to write data.
netcdf.endDef(ncid);

% Write data to variable.
```

```
netcdf.putVar(ncid,varid,my_vardata);

% Re-enter define mode.
netcdf.reDef(ncid);

% Create an attribute associated with the variable.
netcdf.putAtt(ncid,0,'my_att',10);

% Verify that the attribute was created.
[xtype xlen] = netcdf.inqAtt(ncid,0,'my_att')

xtype =

     6


xlen =

     1
```

See Also `netcdf.getatt`

netcdf.putVar

Purpose

Write data to netCDF variable

Syntax

```
netcdf.putVar(ncid,varid,data)
netcdf.putVar(ncid,varid,start,data)
netcdf.putVar(ncid,varid,start,count,data)
netcdf.putVar(ncid,varid,start,count,stride,data)
```

Description

`netcdf.putVar(ncid,varid,data)` writes `data` to a netCDF variable identified by `varid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`netcdf.putVar(ncid,varid,start,data)` writes a single data value into the variable at the index specified by `start`.

`netcdf.putVar(ncid,varid,start,count,data)` writes a section of values into the netCDF variable at the index specified by the vector `start` to the extent specified by the vector `count`, along each dimension of the specified variable.

`netcdf.putVar(ncid,varid,start,count,stride,data)` writes the subsection specified by sampling interval,`stride`, of the values in the section of the variable beginning at the index `start` and to the extent specified by `count`.

This function corresponds to several variable I/O functions in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example creates a new netCDF file and writes a variable to the file.

```
% Create a 50 element vector for a variable.
my_vardata = linspace(0,50,50);

% Open netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE')
```

```
% Define the dimensions of the variable.
dimid = netcdf.defDim(ncid,'my_dim',50);

% Define a new variable in the file.
my_varID = netcdf.defVar(ncid,'my_var','double',dimid)

% Leave define mode and enter data mode to write data.
netcdf.endDef(ncid)

% Write data to variable.
netcdf.putVar(ncid,my_varID,my_vardata);

% Verify that the variable was created.
[varname xtype dimid natts ] = netcdf.inqVar(ncid,0)

varname =

my_var


xtype =

     6


dimid =

     0


natts =

     0
```

See Also

`netcdf.getVar`

netcdf.reDef

Purpose

Put open netCDF file into define mode

Syntax

`netcdf.reDef(ncid)`

Description

`netcdf.reDef(ncid)` puts an open netCDF file into define mode so that dimensions, variables, and attributes can be added or renamed. Attributes can also be deleted in define mode. `ncid` is a valid NetCDF file ID, returned from a previous call to `netcdf.open` or `netcdf.create`.

This function corresponds to the `nc_redef` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example opens a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open a netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Try to define a dimension.
dimid = netcdf.defdim(ncid, 'lat', 50); % should fail.
??? Error using ==> netcdflib
NetCDF: Operation not allowed in data mode

Error in ==> defDim at 22
dimid = netcdflib('def_dim', ncid,dimname,dimlen);

% Put file in define mode.
netcdf.reDef(ncid);

% Try to define a dimension again. Should succeed.
dimid = netcdf.defDim(ncid, 'lat', 50);
```

See Also

`netcdf.create`, `netcdf.endDef`,`netcdf.open`

Purpose Change name of attribute

Syntax `netcdf.renameAtt(ncid,varid,oldName,newName)`

Description `netcdf.renameAtt(ncid,varid,oldName,newName)` changes the name of the attribute specified by the character string `oldName`.

`newName` is a character string that specifies the new name.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` identifies the variable to which the attribute is associated. To specify a global attribute, use `netcdf.getConstant('NC_GLOBAL')` for `varid`.

This function corresponds to the `nc_rename_att` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Get the ID of a variable the attribute is associated with.
varID = netcdf.inqVarID(ncid,'avagadros_number')

% Rename the attribute.
netcdf.renameAtt(ncid,varID,'description','Description');

% Verify that the name changed.
attname = netcdf.inqAttName(ncid,varID,0)

attname =

Description
```

See Also `netcdf.inqAttName`

Purpose

Change name of netCDF dimension

Syntax

`netcdf.renameDim(ncid,dimid,newName)`

Description

`netcdf.renameDim(ncid,dimid,newName)` renames the dimension identified by the dimension identifier, `dimid`.

`newName` is a character string specifying the new name. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`

This function corresponds to the `nc_rename_dim` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This examples modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Put file is define mode.
netcdf.reDef(ncid)

% Get the identifer of a dimension to rename.
dimid = netcdf.inqDimID(ncid,'x');

% Rename the dimension.
netcdf.renameDim(ncid,dimid,'Xdim')

% Verify that the name changed.
data = netcdf.inqDim(ncid,dimid)

data =

Xdim
```

See Also `netcdf.defDim`

Purpose

Change name of netCDF variable

Syntax

`netcdf.renameVar(ncid,varid,newName)`

Description

`netcdf.renameVar(ncid,varid,newName)` renames the variable identified by `varid` in the netCDF file identified by `ncid.newName` is a character string specifying the new name.

This function corresponds to the `nc_rename_var` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Put file in define mode.
netcdf.redef(ncid)

% Get name of first variable
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

varname

varname =

avagadros_number

% Rename the variable, using a capital letter to start the name.
netcdf.renameVar(ncid,0,'Avagadros_number')

% Verify that the name of the variable changed.
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);
```

```
varname

varname =

Avagadros_number
```

See Also `netCDF.defVar`, `netCDF.inqVar`, `netCDF.putVar`

Purpose Change default netCDF file format

Syntax

```
oldFormat = netcdf.setDefaultFormat(newFormat)
```

Description `oldFormat = netcdf.setDefaultFormat(newFormat)` changes the default format used by `netCDF.create` when creating new netCDF files, and returns the value of the old format. You can use this function to change the format used by a netCDF file without having to change the creation mode flag used in each call to `netCDF.create`.

`newFormat` can be either of the following values.

Value	Description
`'NC_FORMAT_CLASSIC'`	Original netCDF file format
`'NC_FORMAT_64BIT'`	64-bit offset format; relaxes limitations on creating very large files

You can also specify the numeric equivalent of these values, as retrieved by `netcdf.getConstant`.

This function corresponds to the `nc_set_default_format` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

```
oldFormat = netcdf.setDefaultFormat('NC_FORMAT_64BIT');
```

See Also `netcdf.create`

netcdf.setFill

Purpose

Set netCDF fill mode

Syntax

```
old_mode = netcdf.setFill(ncid,new_mode)
```

Description

`old_mode = netcdf.setFill(ncid,new_mode)` sets the fill mode for a netCDF file identified by `ncid`.

`new_mode` can be either `'FILL'` or `'NOFILL'` or their numeric equivalents, as retrieved by `netcdf.getConstant`. The default mode is `'FILL'`. netCDF pre-fills data with fill values. Specifying `'NOFILL'` can be used to enhance performance, because it avoids the duplicate writes that occur when the netCDF writes fill values that are later overwritten with data.

This function corresponds to the `nc_set_fill` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples

This example creates a new file and specifies the fill mode used by netCDF with the file.

```
ncid = netcdf.open('foo.nc','NC_WRITE');

% Set filling behavior
old_mode = netcdf.setFill(ncid,'NC_NOFILL');
```

See Also

`netcdf.getConstant`

Purpose Synchronize netCDF file to disk

Syntax `netcdf.sync(ncid)`

Description `netcdf.sync(ncid)` synchronizes the state of a netCDF file to disk. The netCDF library normally buffers accesses to the underlying netCDF file, unless you specify the `NC_SHARE` mode when you opened the file with `netcdf.open` or `netcdf.create`. To call `netcdf.sync`, the netCDF file must be in data mode.

This function corresponds to the `nc_sync` function in the netCDF library C API. To use this function, you should be familiar with the information about netCDF contained in the `NetCDF C Interface Guide` for version 3.6.2.

Examples This example creates a new netCDF file for write access, performs an operation on the file, takes the file out of define mode, and then synchronizes the file to disk.

```
% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE');

% Perform an operation.
dimid = netcdf.defDim(ncid,'Xdim',50);

% Take file out of define mode.
netcdf.endDef(ncid);

% Synchronize the file to disk.
netcdf.sync(ncid)
```

See Also `netcdf.close`, `netcdf.create`, `netcdf.open`, `netcdf.endDef`

newplot

Purpose

Determine where to draw graphics objects

Syntax

```
newplot
h = newplot
h = newplot(hsave)
```

Description

`newplot` prepares a figure and axes for subsequent graphics commands.

`h = newplot` prepares a figure and axes for subsequent graphics commands and returns a handle to the current axes.

`h = newplot(hsave)` prepares and returns an axes, but does not delete any objects whose handles you have assigned to the `hsave` argument, which can be a vector of handles. If `hsave` is not empty, the figure and axes containing `hsave` are prepared for plotting instead of the current axes of the current figure. If `hsave` is empty, `newplot` behaves as if it were called without any inputs.

Remarks

Use `newplot` at the beginning of high-level graphics M-files to determine which figure and axes to target for graphics output. Calling `newplot` can change the current figure and current axes. Basically, there are three options when you are drawing graphics in existing figures and axes:

- Add the new graphics without changing any properties or deleting any objects.
- Delete all existing objects whose handles are not hidden before drawing the new objects.
- Delete all existing objects regardless of whether or not their handles are hidden, and reset most properties to their defaults before drawing the new objects (refer to the following table for specific information).

The figure and axes `NextPlot` properties determine how `newplot` behaves. The following two tables describe this behavior with various property values.

First, `newplot` reads the current figure's `NextPlot` property and acts accordingly.

NextPlot	What Happens
`new`	Create a new figure and use it as the current figure.
`add`	Draw to the current figure without clearing any graphics objects already present.
`replacechildren`	Remove all child objects whose `HandleVisibility` property is set to `on` and reset figure `NextPlot` property to `add`. This clears the current figure and is equivalent to issuing the `clf` command.
`replace`	Remove all child objects (regardless of the setting of the `HandleVisibility` property) and reset figure properties to their defaults, except `NextPlot` is reset to `add` regardless of user-defined defaults. • `Position`, `Units`, `PaperPosition`, and `PaperUnits` are not reset. This clears and resets the current figure and is equivalent to issuing the `clf reset` command.

After `newplot` establishes which figure to draw in, it reads the current axes' `NextPlot` property and acts accordingly.

NextPlot	Description
`add`	Draw into the current axes, retaining all graphics objects already present.

NextPlot	Description
`replacechildren`	Remove all child objects whose `HandleVisibility` property is set to `on`, but do not reset axes properties. This clears the current axes like the `cla` command.
`replace`	Remove all child objects (regardless of the setting of the `HandleVisibility` property) and reset axes properties to their defaults, except `Position` and `Units`. This clears and resets the current axes like the `cla reset` command.

See Also

`axes`, `cla`, `clf`, `figure`, `hold`, `ishold`, `reset`

The `NextPlot` property for figure and axes graphics objects

“Figure Windows” on page 1-100 for related functions

Controlling Graphics Output for more examples.

Purpose

Make next IFD current IFD

Syntax

`tiffobj.nextDirectory()`

Description

`tiffobj.nextDirectory()` makes the next image file directory (IFD) in the file the current IFD. `Tiff` object methods operate on the current IFD. Use this method to navigate among IFDs in a TIFF file containing multiple images.

Examples

Open a `Tiff` object and change the current IFD to the next IFD in the file. 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.nextDirectory();
```

References

This method corresponds to the `TIFFReadDirectory` 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

-
-

nextpow2

Purpose

Next higher power of 2

Syntax

`p = nextpow2(A)`

Description

`p = nextpow2(A)` returns the smallest power of two that is greater than or equal to the absolute value of `A`. (That is, `p` that satisfies `2^p >= abs(A)`).

This function is useful for optimizing FFT operations, which are most efficient when sequence length is an exact power of two.

If `A` is non-scalar, `nextpow2` returns the smallest power of two greater than or equal to `length(A)`.

Examples

For any integer `n` in the range from `513` to `1024`, `nextpow2(n)` is `10`.

For a 1-by-30 vector `A`, `length(A)` is `30` and `nextpow2(A)` is `5`.

See Also

`fft`, `log2`, `pow2`

Purpose

Number of nonzero matrix elements

Syntax

`n = nnz(X)`

Description

`n = nnz(X)` returns the number of nonzero elements in matrix `X`.

The density of a sparse matrix is `nnz(X)/prod(size(X))`.

Examples

The matrix

```
w = sparse(wilkinson(21));
```

is a tridiagonal matrix with `20` nonzeros on each of three diagonals, so `nnz(w) = 60`.

See Also

`find`, `isa`, `nonzeros`, `nzmax`, `size`, `whos`

noanimate

Purpose

Change `EraseMode` of all objects to `normal`

Note `noanimate` will be removed in a future release.

Syntax

```
noanimate(state,fig_handle)
noanimate(state)
```

Description

`noanimate(state,fig_handle)` sets the `EraseMode` of all image, line, patch, surface, and text graphics objects in the specified figure to `normal`. `state` can be the following strings:

- `'save'` — Set the values of the `EraseMode` properties to `normal` for all the appropriate objects in the designated figure.
- `'restore'` — Restore the `EraseMode` properties to the previous values (i.e., the values before calling `noanimate` with the `'save'` argument).

`noanimate(state)` operates on the current figure.

`noanimate` is useful if you want to print the figure to a TIFF or JPEG format.

See Also

`print`

Purpose Nonzero matrix elements

Syntax `s = nonzeros(A)`

Description `s = nonzeros(A)` returns a full column vector of the nonzero elements in `A`, ordered by columns.

This gives the `s`, but not the `i` and `j`, from `[i,j,s] = find(A)`. Generally,

```
length(s) = nnz(A) <= nzmax(A) <= prod(size(A))
```

See Also `find`, `isa`, `nnz`, `nzmax`, `size`, `whos`

Purpose

Vector and matrix norms

Syntax

```
n = norm(A)
n = norm(A,p)
```

Description

The *norm* of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. The `norm` function calculates several different types of matrix norms:

`n = norm(A)` returns the largest singular value of `A`, `max(svd(A))`.

`n = norm(A,p)` returns a different kind of norm, depending on the value of `p`.

If `p` is...	**Then `norm` returns...**
`1`	The 1-norm, or largest column sum of `A`, `max(sum(abs(A)))`.
`2`	The largest singular value (same as `norm(A)`).
`inf`	The infinity norm, or largest row sum of `A`, `max(sum(abs(A')))`.
`'fro'`	The Frobenius-norm of matrix `A`, `sqrt(sum(diag(A'*A)))`.

When `A` is a vector:

`norm(A,p)`	Returns `sum(abs(A).^p)^(1/p)`, for any 1 <= `p` <= ∞.
`norm(A)`	Returns `norm(A,2)`.
`norm(A,inf)`	Returns `max(abs(A))`.
`norm(A,-inf)`	Returns `min(abs(A))`.

Remarks

Note that `norm(x)` is the Euclidean length of a vector `x`. On the other hand, MATLAB software uses “length” to denote the number of elements `n` in a vector. This example uses `norm(x)/sqrt(n)` to obtain the root-mean-square (RMS) value of an `n`-element vector `x`.

```
x = [0 1 2 3]
x =
     0     1     2     3

sqrt(0+1+4+9)   % Euclidean length
ans =
    3.7417

norm(x)
ans =
    3.7417

n = length(x)   % Number of elements
n =
     4

rms = 3.7417/2  % rms = norm(x)/sqrt(n)
rms =
    1.8708
```

See Also

`cond`, `condest`, `hypot`, `normest`, `rcond`

normest

Purpose

2-norm estimate

Syntax

```
nrm = normest(S)
nrm = normest(S,tol)
[nrm,count] = normest(...)
```

Description

This function is intended primarily for sparse matrices, although it works correctly and may be useful for large, full matrices as well.

`nrm = normest(S)` returns an estimate of the 2-norm of the matrix `S`.

`nrm = normest(S,tol)` uses relative error `tol` instead of the default tolerance `1.e-6`. The value of `tol` determines when the estimate is considered acceptable.

`[nrm,count] = normest(...)` returns an estimate of the 2-norm and also gives the number of power iterations used.

Examples

The matrix `W = gallery('wilkinson',101)` is a tridiagonal matrix. Its order, 101, is small enough that `norm(full(W))`, which involves `svd(full(W))`, is feasible. The computation takes 4.13 seconds (on one computer) and produces the exact norm, 50.7462. On the other hand, `normest(sparse(W))` requires only 1.56 seconds and produces the estimated norm, 50.7458.

Algorithm

The power iteration involves repeated multiplication by the matrix `S` and its transpose, `S'`. The iteration is carried out until two successive estimates agree to within the specified relative tolerance.

See Also

`cond`, `condest`, `norm`, `rcond`, `svd`

Purpose

Find logical NOT of array or scalar input

Syntax

```
~A
not(A)
```

Description

`~A` performs a logical NOT of input array `A`, and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if the input array contains a zero value element at that same array location. Otherwise, that element is set to 0.

The input of the expression can be an array or can be a scalar value. If the input is an array, then the output is an array of the same dimensions. If the input is scalar, then the output is scalar.

`not(A)` is called for the syntax `~A` when `A` is an object.

Example

If matrix `A` is

```
     0    29     0    36     0
    23    34    35     0    39
     0    24    31    27     0
     0    29     0     0    34
```

then

```
~A
ans =
     1     0     1     0     1
     0     0     0     1     0
     1     0     0     0     1
     1     0     1     1     0
```

See Also

`bitcmp`, `and`, `or`, `xor`, `any`, `all`, , ,

notebook

Purpose

Open M-book in Microsoft Word software (on Microsoft Windows platforms)

Syntax

```
notebook
notebook('filename')
notebook('-setup')
```

Description

`notebook` starts Microsoft Word software and creates a new M-book titled `Document 1`.

`notebook('filename')` starts Microsoft Word and opens the M-book `filename`, where `filename` is either in the MATLAB current folder or is a full path. If `filename` does not exist, MATLAB creates a new M-book titled `filename`. If the file name extension is not specified, MATLAB assumes `.doc`.

`notebook('-setup')` runs an interactive setup function for Notebook. It copies the Notebook template, `m-book.dot`, to the Microsoft Word template folder, whose location MATLAB automatically determines from the Windows system registry. Upon completion, MATLAB displays a message indicating whether or not the setup was successful.

See Also

MATLAB Desktop Tools and Development Environment documentation

- Notebook for Publishing to Word
-

notify (handle)

Purpose Notify listeners that event is occurring

Syntax

```
notify(Hobj,'EventName')
notify(Hobj,'EventName',data)
```

Description `notify(Hobj,'EventName')` notifies listeners that the specified event is taking place on the specified handle objects.

`notify(Hobj,'EventName',data)` includes user-defined event data.

Arguments

`Hobj`
Array of handle objects triggering the specified event.

`EventName`
Name of the event.

`data`
An `event.EventData` object encapsulating information about the event. You can define custom event data by subclassing `event.EventData` and passing an instance of your subclass as the `data` argument. See for information on defining event data.

See Also

See

`handle`, `addlistener`

now

Purpose Current date and time

Syntax `t = now`

Description `t = now` returns the current date and time as a serial date number. To return the time only, use `rem(now,1)`. To return the date only, use `floor(now)`.

Examples

```
t1 = now, t2 = rem(now,1)

t1 =

   7.2908e+05


t2 =

    0.4013
```

See Also `clock`, `date`, `datenum`

Purpose

Real `n`th root of real numbers

Syntax

```
y = nthroot(X, n)
```

Description

`y = nthroot(X, n)` returns the real `n`th root of the elements of `X`. Both `X` and `n` must be real and `n` must be a scalar. If `X` has negative entries, `n` must be an odd integer.

Example

```
nthroot(-2, 3)
```

returns the real cube root of `-2`.

```
ans =

   -1.2599
```

By comparison,

```
(-2)^(1/3)
```

returns a complex cube root of `-2`.

```
ans =

   0.6300 + 1.0911i
```

See Also

`power`

null

Purpose

Null space

Syntax

```
Z = null(A)
Z = null(A,'r')
```

Description

`Z = null(A)` is an orthonormal basis for the null space of `A` obtained from the singular value decomposition. That is, `A*Z` has negligible elements, `size(Z,2)` is the nullity of `A`, and `Z'*Z = I`.

`Z = null(A,'r')` is a "rational" basis for the null space obtained from the reduced row echelon form. `A*Z` is zero, `size(Z,2)` is an estimate for the nullity of `A`, and, if `A` is a small matrix with integer elements, the elements of the reduced row echelon form (as computed using `rref`) are ratios of small integers.

The orthonormal basis is preferable numerically, while the rational basis may be preferable pedagogically.

Example

Example 1

Compute the orthonormal basis for the null space of a matrix A.

```
A = [1     2     3
     1     2     3
     1     2     3];

Z = null(A);
A*Z

ans =
  1.0e-015 *
     0.2220    0.2220
     0.2220    0.2220
     0.2220    0.2220

Z'*Z

ans =
```

```
     1.0000   -0.0000
    -0.0000    1.0000
```

Example 2

Compute the 1-norm of the matrix A*Z and determine that it is within a small tolerance.

```
norm(A*Z,1) < 1e-12
ans =
     1
```

Example 3

Compute the rational basis for the null space of the same matrix `A`.

```
ZR = null(A,'r')

ZR =
    -2    -3
     1     0
     0     1

A*ZR

ans =

     0     0
     0     0
     0     0
```

See Also

`orth`, `rank`, `rref`, `svd`

num2cell

Purpose Convert numeric array to cell array

Syntax

```
C = num2cell(A)
C = num2cell(A, dim)
C = num2cell(A, [dim1, dim2, ...])
```

Description `C = num2cell(A)` converts numeric array `A` into cell array `C` by placing each element of `A` into a separate cell in `C`. The output array has the same size and dimensions as the input array. Each cell in `C` contains the same numeric value as its respective element in `A`.

`C = num2cell(A, dim)` converts numeric array `A` into a cell array of numeric vectors, the dimensions of which depend on the value of the `dim` argument. Return value `C` contains `numel(A)/size(A,dim)` vectors, each of length `size(A, dim)`. (See Example 2, below). The `dim` input must be an integer with a value from `ndims(A)` to 1.

`C = num2cell(A, [dim1, dim2, ...])` converts numeric array `A` into a cell array of numeric arrays, the dimensions of which depend on the values of arguments `[dim1, dim2, ...]`. Given the variables `X` and `Y`, where `X=size(A,dim1)` and `Y=size(A,dim2)`, return value `C` contains `numel(A)/prod(X,Y,...)` arrays, each of size `X`-by-`Y`-by-`...`. All `dimN` inputs must be an integer with a value from `ndims(A)` to 1.

Remarks `num2cell` works for all array types.

To convert the cell array back to a numeric array, use `cell2mat` or `cat(dim, c{:})`. See "Example 2 — Converting to Cell Array and Back to Numeric" on page 2-2587.

Examples

Example 1 — Converting to a Cell Array of the Same Dimensions

Create a 4x7x3 array of random numbers:

```
A = rand(4,7,3);
```

Convert the numeric array to a cell array of the same dimensions:

```
C = num2cell(A);
whos C
  Name      Size             Bytes  Class    Attributes

  C         4x7x3             5712  cell
```

Example 2 — Converting to Cell Array and Back to Numeric

Create a 3-by-4 numeric array `A`:

```
A = [5:3:14; -9:4:3; 20:-4:8]
A =
     5     8    11    14
    -9    -5    -1     3
    20    16    12     8
```

Convert `A` to a cell array `C1` of the same size. Convert it back to a numeric array using the `cell2mat` function:

```
C1 = num2cell(A)
C1 =
   [5]   [ 8]   [11]   [14]
  [-9]   [-5]   [-1]   [ 3]
  [20]   [16]   [12]   [ 8]
```

```
N1 = cell2mat(C1)
N1 =
     5    8   11   14
    -9   -5   -1    3
    20   16   12    8
```

Convert `A` to a cell array `C2` of 1-by-4 arrays. Convert it back to a numeric array using the `cat` function:

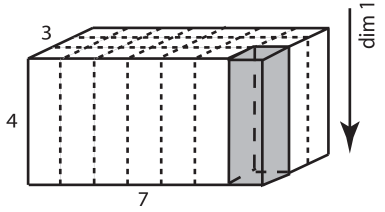
```
C2 = num2cell(A, 2)
C2 =
    [1x4 double]
    [1x4 double]
    [1x4 double]
```

```
N2 = cat(1, C2{:})
N2 =
     5    8   11   14
    -9   -5   -1    3
    20   16   12    8
```

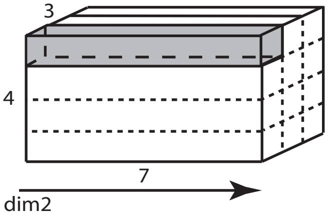
Example 3 — Converting to a Cell Array of Different Dimensions

The following diagrams show the output of `C=num2cell(A,dim)`, where `A` is a 4x7x3 numeric array, and `dim` is one dimension of the input array:

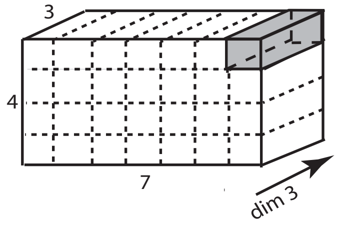
either 1, 2, or 3 in this case. The shaded segment of each diagram represents one cell of the output cell array. Each of these cells contains those elements of `A` that are positioned along the dimension specified by `dim`:



C = num2cell(A,1) yields a 21-element (1x7x3) cell array C, each cell containing a numeric vector of 4 elements along the 1st dimension of A.



C = num2cell(A,2) yields a 12-element (4x1x3) cell array C, each cell containing a numeric vector of 7 elements along the 2nd dimension of A.



C = num2cell(A,3) yields a 28-element (4x7x1) cell array C, each cell containing a numeric vector of 3 elements along the 3rd dimension of A.

Example 4 — Specifying More Than One Dimension for the Output

Given a 4x7x3 numeric array `A`

```
A = rand(4,7,3)
```

and the following two values X and Y

```
X = size(A,dim1)
Y = size(A,dim2)
```

then the statement

```
C = num2cell(A, [1 3])
```

returns in C a cell array of `numel(A)/prod(X,Y)` numeric arrays, each having the dimensions `X-by-Y`.

See Also

`cat`, `mat2cell`, `cell2mat`

Purpose

Convert singles and doubles to IEEE hexadecimal strings

Syntax

`num2hex(X)`

Description

If `X` is a single or double precision array with n elements, `num2hex(X)` is an n-by-8 or n-by-16 char array of the hexadecimal floating-point representation. The same representation is printed with format `hex`.

Examples

```
num2hex([1 0 0.1 -pi Inf NaN])
```

returns

```
ans =

3ff0000000000000
0000000000000000
3fb999999999999a
c00921fb54442d18
7ff0000000000000
fff8000000000000
num2hex(single([1 0 0.1 -pi Inf NaN]))
```

returns

```
ans =

3f800000
00000000
3dcccccd
c0490fdb
7f800000
ffc00000
```

See Also

`hex2num`, `dec2hex`, `format`

num2str

Purpose

Convert number to string

Syntax

```
str = num2str(A)
str = num2str(A, precision)
str = num2str(A, format)
```

Description

The `num2str` function converts numbers to their string representations. This function is useful for labeling and titling plots with numeric values.

`str = num2str(A)` converts array `A` into a string representation `str` with roughly four digits of precision and an exponent if required.

`str = num2str(A, precision)` converts the array `A` into a string representation `str` with maximum precision specified by *`precision`*. Argument *`precision`* specifies the number of digits the output string is to contain. The default is four.

`str = num2str(A, format)` converts array `A` using the supplied *`format`*. (See `fprintf` for format string details.) By default, `num2str` displays floating point values using `'%11.4g'` format (four significant digits in exponential or fixed-point notation, whichever is shorter).

If the input array is integer-valued, `num2str` returns the exact string representation of that integer. The term integer-valued includes large floating-point numbers that lose precision due to limitations of the hardware.

`num2str` removes any leading spaces from the output string. Thus, `num2str(42.67, '%10.2f')` returns a 1-by-5 character array `'42.67'`.

Examples

`num2str(pi)` is `3.142`.

`num2str(eps)` is `2.22e-16`.

`num2str(randn(2,2),3)` produces the following string matrix:

```
num2str(randn(2,2),3)
ans =
  0.538     -2.26
   1.83     0.862
```

`num2str` with a format of `%10.5e\n` returns a matrix of strings in exponential format, having 5 decimal places, with each element separated by a `newline` character:

```
x = rand(2,3) * 9999;           % Create a 2-by-3 matrix.

A = num2str(x, '%10.5e\n')      % Convert to string array.
A =
6.87255e+003
1.55597e+003
8.55890e+003
3.46077e+003
1.91097e+003
4.90201e+003
```

See Also

`mat2str`, `int2str`, `str2num`, `sprintf`, `fprintf`

Tiff.numberOfStrips

Purpose Total number of strips in image

Syntax `numStrips = tiffobj.numberOfStrips()`

Description `numStrips = tiffobj.numberOfStrips()` returns the total number of strips in the image.

Examples Open a `Tiff` object and determine the number of strips in the image. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path:

```
t = Tiff('myfile.tif', 'r');
%
% Check if image has a stripped organization
if ~t.isTiled()
  nStrips = t.numberOfStrips();
end
```

References This method corresponds to the `TIFFNumberOfStrips` 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.numberOfTiles` | `Tiff.isTiled`

Tutorials

-
-

Purpose

Total number of tiles in image

Syntax

`numTiles = tiffobj.numberOfTiles()`

Description

`numTiles = tiffobj.numberOfTiles()` returns the total number of tiles in the image.

Examples

Open a `Tiff` object and determine the number of tiles in the image. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path:

```
t = Tiff('myfile.tif', 'r');
%
% Check if image has a tiled organization
if t.isTiled()
  nTiles = t.numberOfTiles();
end
```

References

This method corresponds to the `TIFFNumberOfTiles` 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.numberOfStrips` | `Tiff.isTiled`

Tutorials

-
-

numel

Purpose Number of elements in array or subscripted array expression

Syntax

```
n = numel(A)
n = numel(A, index1, index2, ... indexn)
```

Description `n = numel(A)` returns the number of elements, `n`, in array `A`.

`n = numel(A, index1, index2, ...  indexn)` returns the number of subscripted elements, `n`, in `A(index1, index2, ..., indexn)`. To handle the variable number of arguments, `numel` is typically written with the header `function n = numel(A, varargin)`, where `varargin` is a cell array with elements `index1`, `index2`, `... indexn`.

The MATLAB software implicitly calls the `numel` built-in function whenever an expression generates a comma-separated list. This includes brace indexing (i.e., `A{index1,index2,...,indexN}`), and dot indexing (i.e., `A.fieldname`).

Remarks It is important to note the significance of `numel` with regards to the overloaded `subsref` and `subsasgn` functions. In the case of the overloaded `subsref` function for brace and dot indexing (as described in the last paragraph), `numel` is used to compute the number of expected outputs (`nargout`) returned from `subsref`. For the overloaded `subsasgn` function, `numel` is used to compute the number of expected inputs (`nargin`) to be assigned using `subsasgn`. The `nargin` value for the overloaded `subsasgn` function is the value returned by `numel` plus 2 (one for the variable being assigned to, and one for the structure array of subscripts).

As a class designer, you must ensure that the value of `n` returned by the built-in `numel` function is consistent with the class design for that object. If `n` is different from either the `nargout` for the overloaded `subsref` function or the `nargin` for the overloaded `subsasgn` function, then you need to overload `numel` to return a value of `n` that is consistent with the class' `subsref` and `subsasgn` functions. Otherwise, MATLAB produces errors when calling these functions.

Examples

Create a 4-by-4-by-2 matrix. `numel` counts 32 elements in the matrix.

```
a = magic(4);
a(:,:,2) = a'

a(:,:,1) =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

a(:,:,2) =
    16     5     9     4
     2    11     7    14
     3    10     6    15
    13     8    12     1

numel(a)
ans =
    32
```

See Also

`nargin`, `nargout`, `prod`, `size`, `subsasgn`, `subsref`

nzmax

Purpose

Amount of storage allocated for nonzero matrix elements

Syntax

```
n = nzmax(S)
```

Description

`n = nzmax(S)` returns the amount of storage allocated for nonzero elements.

If `S` is a sparse matrix...	`nzmax(S)` is the number of storage locations allocated for the nonzero elements in `S`.
If `S` is a full matrix...	`nzmax(S) = prod(size(S))`.

Often, `nnz(S)` and `nzmax(S)` are the same. But if `S` is created by an operation which produces fill-in matrix elements, such as sparse matrix multiplication or sparse LU factorization, more storage may be allocated than is actually required, and `nzmax(S)` reflects this. Alternatively, `sparse(i,j,s,m,n,nzmax)` or its simpler form, `spalloc(m,n,nzmax)`, can set `nzmax` in anticipation of later fill-in.

See Also

`find`, `isa`, `nnz`, `nonzeros`, `size`, `whos`

Purpose

Solve fully implicit differential equations, variable order method

Syntax

```
[T,Y] = ode15i(odefun,tspan,y0,yp0)
[T,Y] = ode15i(odefun,tspan,y0,yp0,options)
[T,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)
sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)
```

Arguments

The following table lists the input arguments for `ode15i`.

`odefun`	A function handle that evaluates the left side of the differential equations, which are of the form $f(t, y, y') = 0$. See in the MATLAB Programming documentation for more information.
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.
`y0, yp0`	Vectors of initial conditions for y and y' respectively.
`options`	Optional integration argument created using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for `ode15i`.

`T`	Column vector of time points
`Y`	Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

Description

`[T,Y] = ode15i(odefun,tspan,y0,yp0)` with `tspan = [t0 tf]` integrates the system of differential equations $f(t, y, y') = 0$ from time `t0` to `tf` with initial conditions `y0` and `yp0`. `odefun` is a function handle. Function `ode15i` solves ODEs and DAEs of index 1. The initial conditions must be consistent, meaning that $f(\mathrm{t0}, \mathrm{y0}, \mathrm{yp0}) = 0$. You can use the function `decic` to compute consistent initial conditions close to guessed values. Function `odefun(t,y,yp)`, for a scalar `t` and column vectors `y` and `yp`, must return a column vector corresponding to $f(t, y, y')$. Each row in the solution array `Y` corresponds to a time returned in the column vector `T`. To obtain solutions at specific times `t0,t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `odefun`, if necessary.

`[T,Y] = ode15i(odefun,tspan,y0,yp0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used options include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components `1e-6` by default). See `odeset` for details.

`[T,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)` with the `'Events'` property in `options` set to a function `events`, solves as above while also finding where functions of (t, y, y'), called event functions, are zero. The function `events` is of the form `[value,isterminal,direction] = events(t,y,yp) and` includes the necessary event functions. Code the function `events` so that the `i`th element of each output vector corresponds to the `i`th event. For the `i`th event function in `events`:

- `value(i)` is the value of the function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Output `TE` is a column vector of times at which events occur. Rows of `YE` are the corresponding solutions, and indices in vector `IE` specify which event occurred. See in the MATLAB Mathematics documentation for more information.

`sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)` returns a structure that can be used with `deval` to evaluate the solution at any point between `t0` and `tfinal`. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver. If you specify the `Events` option and a terminal event is detected, `sol.x(end)` contains the end of the step at which the event occurred.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Options

`ode15i` accepts the following parameters in `options`. For more information, see `odeset` and Changing ODE Integration Properties in the MATLAB Mathematics documentation.

Error control	`RelTol`, `AbsTol`, `NormControl`
Solver output	`OutputFcn`, `OutputSel`, `Refine`, `Stats`
Event location	`Events`

Step size	`MaxStep`, `InitialStep`
Jacobian matrix	`Jacobian`, `JPattern`, `Vectorized`

Solver Output

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

Jacobian Matrices

The Jacobian matrices $\partial f / \partial y$ and $\partial f / \partial y'$ are critical to reliability and efficiency. You can provide these matrices as one of the following:

- Function of the form `[dfdy,dfdyp] = FJAC(t,y,yp)` that computes the Jacobian matrices. If `FJAC` returns an empty matrix `[]` for either `dfdy` or `dfdyp`, then `ode15i` approximates that matrix by finite differences.
- Cell array of two constant matrices `{dfdy,dfdyp}`, either of which could be empty.

Use `odeset` to set the `Jacobian` option to the function or cell array. If you do not set the `Jacobian` option, `ode15i` approximates both Jacobian matrices by finite differences.

For `ode15i`, `Vectorized` is a two-element cell array. Set the first element to `'on'` if `odefun(t,[y1,y2,...],yp)` returns `[odefun(t,y1,yp),odefun(t,y2,yp),...]`. Set the second element to `'on'` if `odefun(t,y,[yp1,yp2,...])` returns `[odefun(t,y,yp1),odefun(t,y,yp2),...]`. The default value of `Vectorized` is `{'off','off'}`.

For `ode15i`, `JPattern` is also a two-element sparse matrix cell array. If $\partial f/\partial y$ or $\partial f/\partial y'$ is a sparse matrix, set `JPattern` to the sparsity patterns, `{SPDY,SPDYP}`. A sparsity pattern of $\partial f/\partial y$ is a sparse matrix `SPDY` with `SPDY(i,j) = 1` if component `i` of `f(t,y,yp)` depends on component `j` of `y`, and `0` otherwise. Use `SPDY = []` to indicate that $\partial f/\partial y$ is a full matrix. Similarly for $\partial f/\partial y'$ and `SPDYP`. The default value of `JPattern` is `{[],[]}`.

Examples

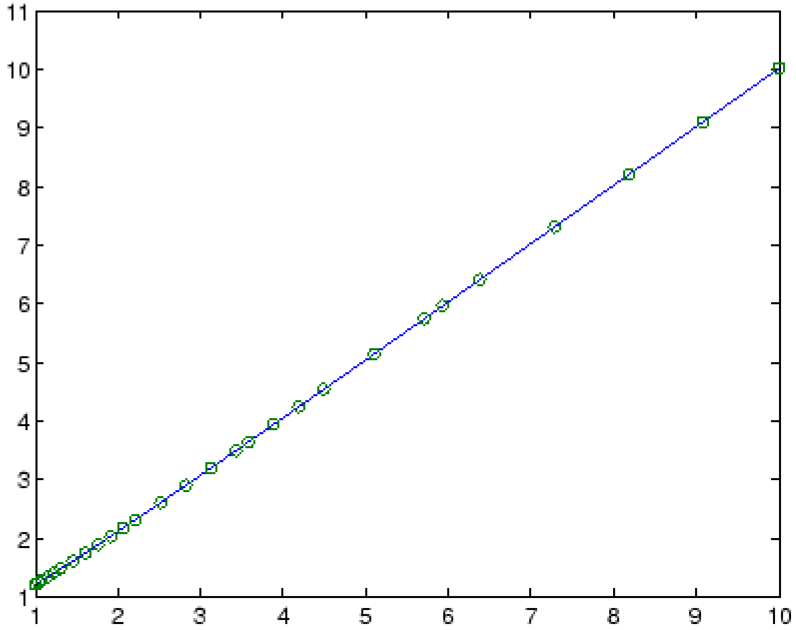
Example 1

This example uses a helper function `decic` to hold fixed the initial value for $y(t_0)$ and compute a consistent initial value for $y'(t_0)$ for the Weissinger implicit ODE. The Weissinger function evaluates the residual of the implicit ODE.

```
t0 = 1;
y0 = sqrt(3/2);
yp0 = 0;
[y0,yp0] = decic(@weissinger,t0,y0,1,yp0,0);
```

The example uses `ode15i` to solve the ODE, and then plots the numerical solution against the analytical solution.

```
[t,y] = ode15i(@weissinger,[1 10],y0,yp0);
ytrue = sqrt(t.^2 + 0.5);
plot(t,y,t,ytrue,'o');
```



Other Examples

These demos provide examples of implicit ODEs: `ihb1dae`, `iburgersode`.

See Also

`decic`, `deval`, `odeget`, `odeset`, `function_handle` (@)

Other ODE initial value problem solvers: `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`

ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

Purpose

Solve initial value problems for ordinary differential equations

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

where *`solver`* is one of `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. See in the MATLAB Programming documentation for more information. All solvers solve systems of equations in the form $y' = f(t, y)$ or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.

`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = `*`solver`*`(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t, y)$ from time `t0` to `tf` with initial conditions `y0`. `odefun` is a function handle. See Function Handles in the MATLAB Programming documentation for more information. Function `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t, y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = `*`solver`*`(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = `*`solver`*`(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.

- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = `*`solver`*`(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property '`on`' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t, y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t, y)y' = f(t, y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to '`none`'.

- If the mass matrix depends weakly on y, set `MStateDependence` to ’`weak`’ (the default); otherwise, set it to ’`strong`’. In either case, the function `MASS` is called with the two arguments (`t`,`y`).

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t, y)$.
- Supply the sparsity pattern of $\partial f / \partial y$ using the `JPattern` property or a sparse $\partial f / \partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t, y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the `(i,k)` component of $M(t, y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t, y)y' = f(t, y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0, y_0)yp_0 = f(t_0, y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.

Solver	Problem Type	Order of Accuracy	When to Use
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 2-2616 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`RelTol, AbsTol, NormControl`	√	√	√	√	√	√	√
`OutputFcn, OutputSel, Refine, Stats`	√	√	√	√	√	√	√
`NonNegative`	√	√	√	√ *	—	√ *	√ *
`Events`	√	√	√	√	√	√	√
`MaxStep, InitialStep`	√	√	√	√	√	√	√
`Jacobian, JPattern, Vectorized`	—	—	—	√	√	√	√

`Mass`
`MStateDependence`
`MvPattern`
`MassSingular`
	√
	√
	—
	—
		√
		√
		—
		—
			√
			√
			—
			—
				√
				√
				√
				√
					√
					—
					—
					—
						√
						√
						√
						√
							√
							√
							√
							—

`InitialSlope`	—	—	—	√	—	√	—
`MaxOrder, BDF`	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned} y'_1 &= y_2\, y_3 & y_1(0) &= 0 \\ y'_2 &= -y_1\, y_3 & y_2(0) &= 1 \\ y'_3 &= -0.51\, y_1\, y_2 & y_3(0) &= 1 \end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

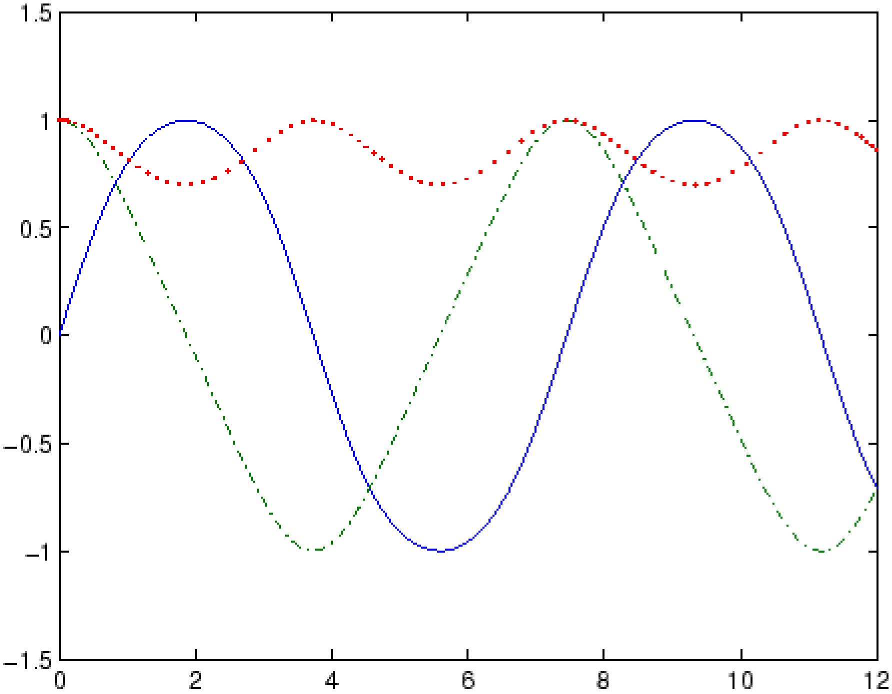
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$\begin{array}{ll} y'_1 = y_2 & y_1(0) = 2 \\ y'_2 = 1000(1 - y_1^2)y_2 - y_1 & y_2(0) = 0 \end{array}$$

To simulate this system, create a function `vdp1000` containing the equations

```
function dy = vdp1000(t,y)
```

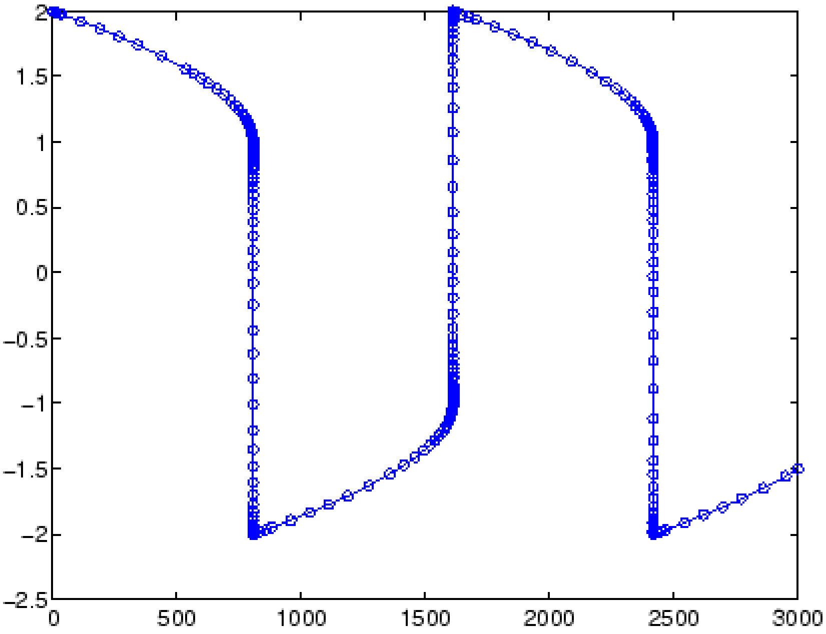
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

```
y'(t) + f(t)y(t) = g(t)
```

The initial condition is `y(0) = 0`, where the function `f(t)` is defined through the `n`-by-`1` vectors `tf` and `f`, and the function `g(t)` is defined through the `m`-by-`1` vectors `tg` and `g`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write an M-file function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evalute ODE at time t
```

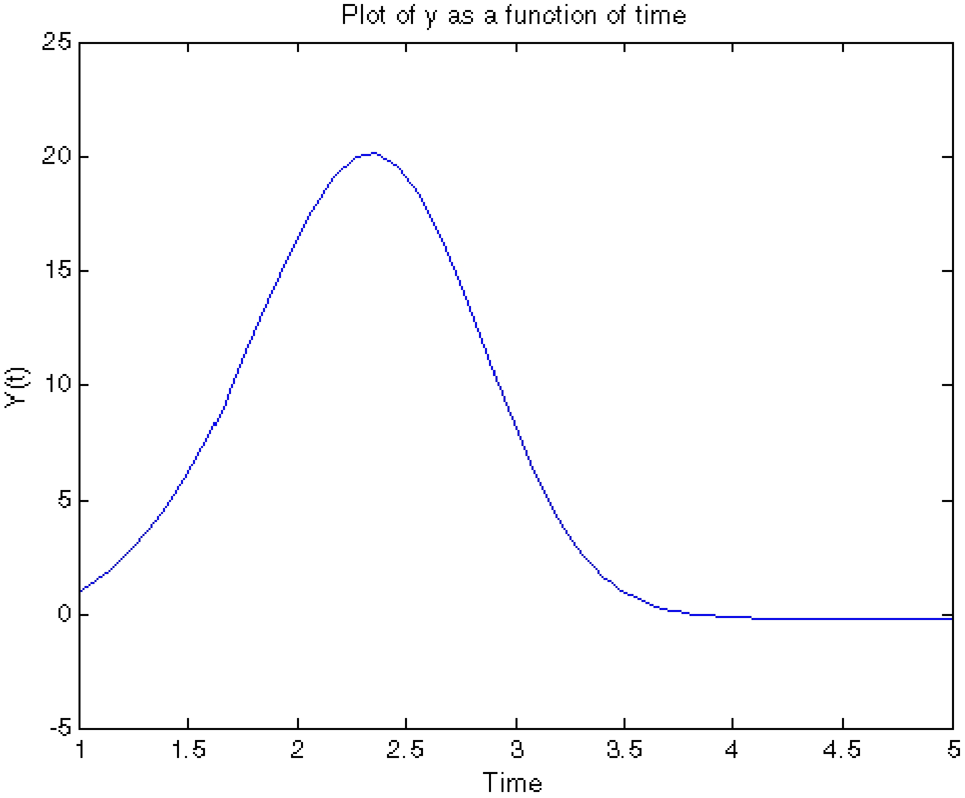
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
```

```
title('Plot of y as a function of time');
xlabel('Time'); ylabel('Y(t)');
```



Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing `y(t`$_n$`)`, it needs only the solution at the immediately preceding time point, `y(t`$_{n-1}$`)`. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

See Also

`deval`, `ode15i`, `odeget`, `odeset`, `function_handle` (`@`)

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp 436-451.

[2] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp 1-9.

[3] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp 19-26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp 1-22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp 538-552.

Purpose

Define differential equation problem for ordinary differential equation solvers

---

Note This reference page describes the `odefile` and the syntax of the ODE solvers used in MATLAB, Version 5. MATLAB, Version 6, supports the `odefile` for backward compatibility, however the new solver syntax does not use an ODE file. New functionality is available only with the new syntax. For information about the new syntax, see `odeset` or any of the ODE solvers.

Description

`odefile` is not a command or function. It is a help entry that describes how to create an M-file defining the system of equations to be solved. This definition is the first step in using any of the MATLAB ODE solvers. In MATLAB documentation, this M-file is referred to as an `odefile`, although you can give your M-file any name you like.

You can use the `odefile` M-file to define a system of differential equations in one of these forms

$$y' = f(t, y)$$

or

$$M(t, y)y' = f(t, y)v$$

where:

- t is a scalar independent variable, typically representing time.
- y is a vector of dependent variables.
- f is a function of t and y returning a column vector the same length as y.
- $M(t, y)$ is a time-and-state-dependent mass matrix.

The ODE file must accept the arguments `t` and `y`, although it does not have to use them. By default, the ODE file must return a column vector the same length as `y`.

All of the solvers of the ODE suite can solve $M(t, y)y' = f(t, y)$, except `ode23s`, which can only solve problems with constant mass matrices. The `ode15s` and `ode23t` solvers can solve some differential-algebraic equations (DAEs) of the form $M(t)y' = f(t, y)$.

Beyond defining a system of differential equations, you can specify an entire initial value problem (IVP) within the ODE M-file, eliminating the need to supply time and initial value vectors at the command line (see "Examples" on page 2-2622).

To Use the ODE File Template

- Enter the command `help odefile` to display the help entry.
- Cut and paste the ODE file text into a separate file.
- Edit the file to eliminate any cases not applicable to your IVP.
- Insert the appropriate information where indicated. The definition of the ODE system is required information.

```
switch flag
 case ''  % Return dy/dt = f(t,y).
   varargout{1} = f(t,y,p1,p2);
 case 'init'  % Return default [tspan,y0,options].
   [varargout{1:3}] = init(p1,p2);
 case 'jacobian'  % Return Jacobian matrix df/dy.
   varargout{1} = jacobian(t,y,p1,p2);
 case 'jpattern'  % Return sparsity pattern matrix S.
   varargout{1} = jpattern(t,y,p1,p2);
 case 'mass'  % Return mass matrix.
   varargout{1} = mass(t,y,p1,p2);
 case 'events'  % Return [value,isterminal,direction].
   [varargout{1:3}] = events(t,y,p1,p2);
 otherwise
   error(['Unknown flag ''' flag '''.']);
```

```
 end
% ----------------------------------------------------------
function dydt = f(t,y,p1,p2)
 dydt = Insert a function of t and/or y, p1, and p2 here.>
% ----------------------------------------------------------
function [tspan,y0,options] = init(p1,p2)
 tspan = <Insert tspan here.>;
 y0 = <Insert y0 here.>;
 options = <Insert options = odeset(...) or [] here.>;
% ----------------------------------------------------------
function dfdy = jacobian(t,y,p1,p2)
 dfdy = <Insert Jacobian matrix here.>;
% ----------------------------------------------------------
function S = jpattern(t,y,p1,p2)
 S = <Insert Jacobian matrix sparsity pattern here.>;
% ----------------------------------------------------------
function M = mass(t,y,p1,p2)
 M = <Insert mass matrix here.>;
% ----------------------------------------------------------
function [value,isterminal,direction] = events(t,y,p1,p2)
 value = <Insert event function vector here.>
 isterminal = <Insert logical ISTERMINAL vector here.>;
 direction = <Insert DIRECTION vector here.>;
```

Notes

1 The ODE file must accept `t` and `y` vectors from the ODE solvers and must return a column vector the same length as `y`. The optional input argument `flag` determines the type of output (mass matrix, Jacobian, etc.) returned by the ODE file.

2 The solvers repeatedly call the ODE file to evaluate the system of differential equations at various times. *This is required information* – you must define the ODE system to be solved.

3 The `switch` statement determines the type of output required, so that the ODE file can pass the appropriate information to the solver. (See notes 4 - 9.)

4. In the default *initial conditions (*`'init'`*)* case, the ODE file returns basic information (time span, initial conditions, options) to the solver. If you omit this case, you must supply all the basic information on the command line.

5. In the `'jacobian'` case, the ODE file returns a Jacobian matrix to the solver. You need only provide this case when you want to improve the performance of the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`.

6. In the `'jpattern'` case, the ODE file returns the Jacobian sparsity pattern matrix to the solver. You need to provide this case only when you want to generate sparse Jacobian matrices numerically for a stiff solver.

7. In the `'mass'` case, the ODE file returns a mass matrix to the solver. You need to provide this case only when you want to solve a system in the form $M(t, y)y' = f(t, y)$.

8. In the `'events'` case, the ODE file returns to the solver the values that it needs to perform event location. When the `Events` property is set to `on`, the ODE solvers examine any elements of the `event` vector for transitions to, from, or through zero. If the corresponding element of the logical `isterminal` vector is set to `1`, integration will halt when a zero-crossing is detected. The elements of the `direction` vector are `-1`, `1`, or `0`, specifying that the corresponding event must be decreasing, increasing, or that any crossing is to be detected.

9. An unrecognized `flag` generates an error.

Examples

The van der Pol equation, $y''_1 - \mu(1 - y_1^2)y' + y_1 = 0$, is equivalent to a system of coupled first-order differential equations.

$$y'_1 = y_2$$

$$y'_2 = \mu(1 - y_1^2)y_2 - y_1$$

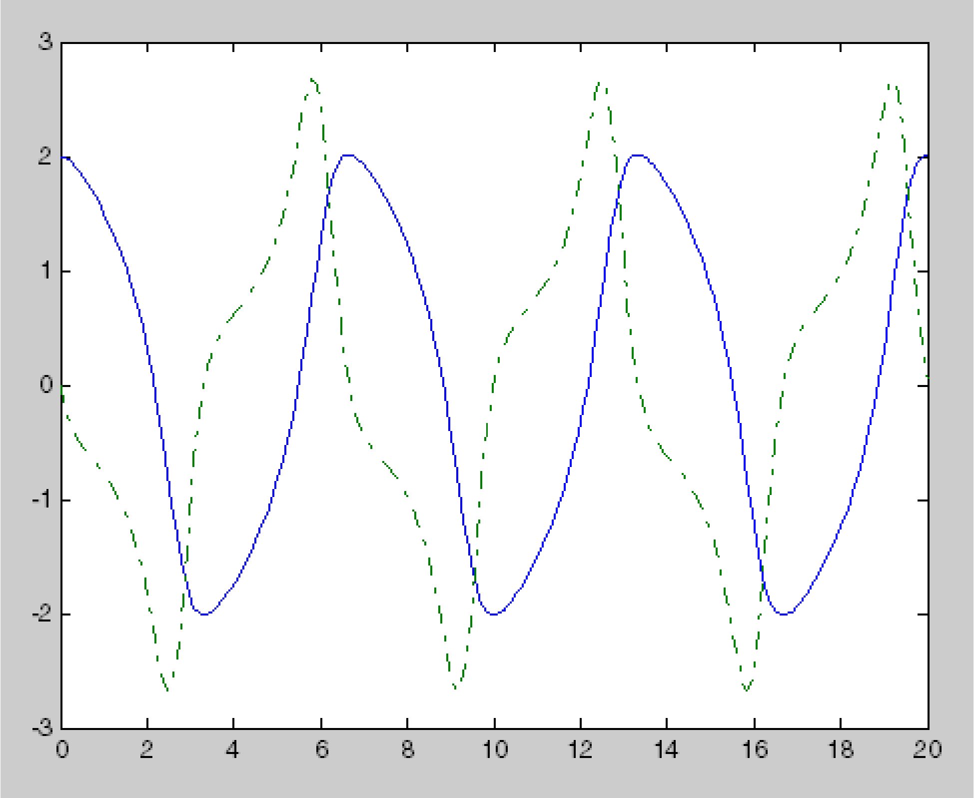
The M-file

```
function out1 = vdp1(t,y)
out1 = [y(2); (1-y(1)^2)*y(2) - y(1)];
```

defines this system of equations (with $\mu = 1$).

To solve the van der Pol system on the time interval `[0 20]` with initial values (at time `0`) of `y(1) = 2` and `y(2) = 0`, use

```
[t,y] = ode45('vdp1',[0 20],[2; 0]);
plot(t,y(:,1),'-',t,y(:,2),'-.')
```



To specify the entire initial value problem (IVP) within the M-file, rewrite `vdp1` as follows.

```
function [out1,out2,out3] = vdp1(t,y,flag)
if nargin < 3 | isempty(flag)
  out1 = [y(1).*(1-y(2).^2)-y(2); y(1)];
else
  switch(flag)
    case 'init'                 % Return tspan, y0, and options.
      out1 = [0 20];
      out2 = [2; 0];
      out3 = [];
  otherwise
    error(['Unknown request ''' flag '''.']);
  end
end
```

You can now solve the IVP without entering any arguments from the command line.

```
[t,Y] = ode23('vdp1')
```

In this example the `ode23` function looks to the `vdp1` M-file to supply the missing arguments. Note that, once you've called `odeset` to define `options`, the calling syntax

```
[t,Y] = ode23('vdp1',[],[],options)
```

also works, and that any options supplied via the command line override corresponding options specified in the M-file (see `odeset`).

See Also

The MATLAB Version 5 help entries for the ODE solvers and their associated functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `odeget`, `odeset`

Type at the MATLAB command line:
`more on, type function, more off`. The Version 5 help follows the Version 6 help.

Purpose

Ordinary differential equation options parameters

Syntax

```
o = odeget(options,'name')
o = odeget(options,'name',default)
```

Description

`o = odeget(options,'name')` extracts the value of the property specified by string `'name'` from integrator options structure `options`, returning an empty matrix if the property value is not specified in `options`. It is only necessary to type the leading characters that uniquely identify the property name. Case is ignored for property names. The empty matrix `[]` is a valid `options` argument.

`o = odeget(options,'name',default)` returns `o = default` if the named property is not specified in `options`.

Example

Having constructed an ODE options structure,

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-3 2e-3 3e-3]);
```

you can view these property settings with `odeget`.

```
odeget(options,'RelTol')
ans =

       1.0000e-04

odeget(options,'AbsTol')
ans =

    0.0010    0.0020    0.0030
```

See Also

`odeset`

odeset

Purpose Create or alter options structure for ordinary differential equation solvers

Syntax

```
options = odeset('name1',value1,'name2',value2,...)
options = odeset(oldopts,'name1',value1,...)
options = odeset(oldopts,newopts)
odeset
```

Description The `odeset` function lets you adjust the integration parameters of the following ODE solvers.

For solving fully implicit differential equations:

```
ode15i
```

For solving initial value problems:

```
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
```

See below for information about the integration parameters.

`options = odeset('name1',value1,'name2',value2,...)` creates an options structure that you can pass as an argument to any of the ODE solvers. In the resulting structure, `options`, the named properties have the specified values. For example, `'name1'` has the value `value1`. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify a property name. Case is ignored for property names.

`options = odeset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This sets `options` equal to the existing structure `oldopts`, overwrites any values in `oldopts` that are respecified using name/value pairs, and adds any new pairs to the structure. The modified structure is returned as an output argument.

`options = odeset(oldopts,newopts)` alters an existing options structure `oldopts` by combining it with a new options structure `newopts`. Any new options not equal to the empty matrix overwrite corresponding options in `oldopts`.

`odeset` with no input arguments displays all property names as well as their possible and default values.

ODE Properties

The following sections describe the properties that you can set using `odeset`. The available properties depend on the ODE solver you are using. There are several categories of properties:

- "Error Control Properties" on page 2-2627
- "Solver Output Properties" on page 2-2629
- "Step-Size Properties" on page 2-2633
- "Event Location Property" on page 2-2634
- "Jacobian Matrix Properties" on page 2-2636
- "Mass Matrix and DAE Properties" on page 2-2640
- "ode15s and ode15i-Specific Properties" on page 2-2642

Note This reference page describes the ODE properties for MATLAB, Version 7. The Version 5 properties are supported only for backward compatibility. For information on the Version 5 properties, type at the MATLAB command line: `more on, type odeset, more off`.

Error Control Properties

At each step, the solver estimates the local error `e` in the `i`th component of the solution. This error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, `RelTol`, and the specified absolute tolerance, `AbsTol`.

```
|e(i)| ≤ max(RelTol*abs(y(i)),AbsTol(i))
```

For routine problems, the ODE solvers deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over "long" intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust `RelTol`. For the absolute error tolerance, the scaling of the solution components is important: if `|y|` is somewhat smaller than `AbsTol`, the solver is not constrained to obtain any correct digits in `y`. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components.

The following table describes the error control properties. Further information on each property is given following the table.

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	Relative error tolerance that applies to all components of the solution vector `y`.
`AbsTol`	Positive scalar or vector {`1e-6`}	Absolute error tolerances that apply to the individual components of the solution vector.
`NormControl`	`on` | {`off`}	Control error relative to norm of solution.

Description of Error Control Properties

`RelTol` — This tolerance is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components, except those smaller than thresholds `AbsTol(i)`.

The default, `1e-3`, corresponds to 0.1% accuracy.

`AbsTol` — `AbsTol(i)` is a threshold below which the value of the `i`th solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero.

If `AbsTol` is a vector, the length of `AbsTol` must be the same as the length of the solution vector `y`. If `AbsTol` is a scalar, the value applies to all components of `y`.

`NormControl` — Set this property `on` to request that the solvers control the error in each integration step with `norm(e) <= max(RelTol*norm(y),AbsTol)`. By default the solvers use a more stringent componentwise error control.

Solver Output Properties

The following table lists the solver output properties that control the output that the solvers generate. Further information on each property is given following the table.

Property	Value	Description
`NonNegative`	Vector of integers	Specifies which components of the solution vector must be nonnegative. The default value is `[]`.
`OutputFcn`	Function handle	A function for the solver to call after every successful integration step.
`OutputSel`	Vector of indices	Specifies which components of the solution vector are to be passed to the output function.
`Refine`	Positive integer	Increases the number of output points by a factor of `Refine`.
`Stats`	`on` | {`off`}	Determines whether the solver should display statistics about its computations. By default, `Stats` is `off`.

Description of Solver Output Properties

`NonNegative` — The `NonNegative` property is not available in `ode23s`, `ode15i`. In `ode15s`, `ode23t`, and `ode23tb`, `NonNegative` is not available for problems where there is a mass matrix.

`OutputFcn` — To specify an output function, set `'OutputFcn'` to a function handle. For example,

```
options = odeset('OutputFcn',@myfun)
```

sets `'OutputFcn'` to `@myfun`, a handle to the function `myfun`. See in the MATLAB Programming documentation for more information.

The output function must be of the form

```
status = myfun(t,y,flag)
```

, in the MATLAB Mathematics documentation, explains how to provide additional parameters to `myfun`, if necessary.

The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:

Flag	Description
`init`	The solver calls `myfun(tspan,y0,'init')` before beginning the integration to allow the output function to initialize. `tspan` and `y0` are the input arguments to the ODE solver.

Flag	Description
`{[]}`	The solver calls `status = myfun(t,y,[])` after each integration step on which output is requested. `t` contains points where output was generated during the step, and `y` is the numerical solution at the points in `t`. If `t` is a vector, the `i`th column of `y` corresponds to the `i`th element of `t`. When `length(tspan) > 2` the output is produced at every point in `tspan`. When `length(tspan) = 2` the output is produced according to the `Refine` option. `myfun` must return a `status` output value of `0` or `1`. If `status = 1`, the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button.
`done`	The solver calls `myfun([],[],'done')` when integration is complete to allow the output function to perform any cleanup chores.

You can use these general purpose output functions or you can edit them to create your own. Type `help function` at the command line for more information.

- `odeplot` — Time series plotting (default when you call the solver with no output arguments and you have not specified an output function)
- `odephas2` — Two-dimensional phase plane plotting
- `odephas3` — Three-dimensional phase plane plotting
- `odeprint` — Print solution as it is computed

Note If you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history.

`OutputSel` — Use `OutputSel` to specify which components of the solution vector you want passed to the output function. For example, if you want to use the `odeplot` output function, but you want to plot only the first and third components of the solution, you can do this using

```
options = ...
odeset('OutputFcn',@odeplot,'OutputSel',[1 3]);
```

By default, the solver passes all components of the solution to the output function.

`Refine` — If `Refine` is `1`, the solver returns solutions only at the end of each time step. If `Refine` is `n >1`, the solver subdivides each time step into `n` smaller intervals and returns solutions at each time point. `Refine` does not apply when `length(tspan)>2`.

Note In all the solvers, the default value of `Refine` is `1`. Within `ode45`, however, the default is `4` to compensate for the solver's large step sizes. To override this and see only the time steps chosen by `ode45`, set `Refine` to `1`.

The extra values produced for `Refine` are computed by means of continuous extension formulas. These are specialized formulas used by the ODE solvers to obtain accurate solutions between computed time steps without significant increase in computation time.

`Stats` — By default, `Stats` is `off`. If it is `on`, after solving the problem the solver displays

- Number of successful steps
- Number of failed attempts
- Number of times the ODE function was called to evaluate $f(t, y)$

Solvers based on implicit methods, including `ode23s`, `ode23t`, `ode23t`, `ode15s`, and `ode15i`, also display

- Number of times that the partial derivatives matrix $\partial f / \partial x$ was formed
- Number of LU decompositions
- Number of solutions of linear systems

Step-Size Properties

The step-size properties specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step-size properties. Further information on each property is given following the table.

Property	Value	Description
`InitialStep`	Positive scalar	Suggested initial step size.
`MaxStep`	Positive scalar {`0.1*abs(t0-tf)`}	Upper bound on solver step size.

Description of Step-Size Properties

`InitialStep` — `InitialStep` sets an upper bound on the magnitude of the first step size the solver tries. If you do not set `InitialStep`, the initial step size is based on the slope of the solution at the initial time `tspan(1)`, and if the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable `InitialStep`.

`MaxStep` — If the differential equation has periodic coefficients or solutions, it might be a good idea to set `MaxStep` to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce `MaxStep` for any of the following purposes:

- To produce more output points. This can significantly slow down solution time. Instead, use `Refine` to compute additional outputs by continuous extension at very low cost.
- When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance `RelTol`, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector `AbsTol`. See "Error Control Properties" on page 2-2627 for a description of the error tolerance properties.
- To make sure that the solver doesn't step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solver twice. If you do not know the time at which the change occurs, try reducing the error tolerances `RelTol` and `AbsTol`. Use `MaxStep` as a last resort.

Event Location Property

In some ODE problems the times of specific events are important, such as the time at which a ball hits the ground, or the time at which a spaceship returns to the earth. While solving a problem, the ODE solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the `Events` property. Further information on each property is given following the table.

ODE Events Property

String	Value	Description
`Events`	Function handle	Handle to a function that includes one or more event functions.

Description of Event Location Properties

`Events` — The function is of the form

```
[value,isterminal,direction] = events(t,y)
```

`value`, `isterminal`, and `direction` are vectors for which the `i`th element corresponds to the `i`th event function:

- `value(i)` is the value of the `i`th event function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function, otherwise, `0`.
- `direction(i) = 0` if all zeros are to be located (the default), `+1` if only zeros where the event function is increasing, and `-1` if only zeros where the event function is decreasing.

If you specify an events function and events are detected, the solver returns three additional outputs:

- `A` column vector of times at which events occur
- Solution values corresponding to these times
- Indices into the vector returned by the events function. The values indicate which event the solver detected.

If you call the solver as

```
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
```

the solver returns these outputs as `TE`, `YE`, and `IE` respectively. If you call the solver as

```
sol = solver(odefun,tspan,y0,options)
```

the solver returns these outputs as `sol.xe`, `sol.ye`, and `sol.ie`, respectively.

For examples that use an event function, see and in the MATLAB Mathematics documentation.

Jacobian Matrix Properties

The stiff ODE solvers often execute faster if you provide additional information about the Jacobian matrix $\partial f / \partial y$, a matrix of partial derivatives of the function that defines the differential equations.

$$\frac{\partial f}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots \\ \vdots & \vdots & \end{bmatrix}$$

The Jacobian matrix properties pertain only to those solvers for stiff problems (`ode15s`, `ode23s`, `ode23t`, `ode23tb`, and `ode15i`) for which the Jacobian matrix $\partial f / \partial y$ can be critical to reliability and efficiency. If you do not provide a function to calculate the Jacobian, these solvers approximate the Jacobian numerically using finite differences. In this case, you might want to use the `Vectorized` or `JPattern` properties.

The following table describes the Jacobian matrix properties for all implicit solvers except `ode15i`. Further information on each property is given following the table. See Jacobian Properties for ode15i on page 2-2639 for `ode15i`-specific information.

Jacobian Properties for All Implicit Solvers Except ode15i

Property	Value	Description
`Jacobian`	Function | handle constant matrix	Matrix or function that evaluates the Jacobian.

Jacobian Properties for All Implicit Solvers Except ode15i (Continued)

Property	Value	Description
`JPattern`	Sparse matrix of {0,1}	Generates a sparse Jacobian matrix numerically.
`Vectorized`	`on` | {`off`}	Allows the solver to reduce the number of function evaluations required.

Description of Jacobian Properties

`Jacobian` — Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function `FJac`, where `FJac(t,y)` computes $\partial f / \partial y$, or to the constant value of $\partial f / \partial y$.

The Jacobian for the , described in the MATLAB Mathematics documentation, can be coded as

```
function J = vdp1000jac(t,y)
J = [  0                  1
     (-2000*y(1)*y(2)-1)  (1000*(1-y(1)^2)) ];
```

`JPattern` — `JPattern` is a sparsity pattern with `1`s where there might be nonzero entries in the Jacobian.

Note If you specify `Jacobian`, the solver ignores any setting for `JPattern`.

Set this property to a sparse matrix S with $S(i,j) = 1$ if component i of $f(t,y)$ depends on component j of y, and 0 otherwise. The solver uses this sparsity pattern to generate a sparse Jacobian matrix numerically. If the Jacobian matrix is large and sparse, this can greatly accelerate execution. For an example using the `JPattern` property, see Example: Large, Stiff, Sparse Problem in the MATLAB Mathematics documentation.

`Vectorized` — The `Vectorized` property allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time.

Set `on` to inform the solver that you have coded the ODE function `F` so that `F(t,[y1 y2 ...])` returns `[F(t,y1) F(t,y2) ...]`. This allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time.

Note If you specify `Jacobian`, the solver ignores a setting of `'on'` for `'Vectorized'`.

With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. For example, you can vectorize the , described in the MATLAB Mathematics documentation, by introducing colon notation into the subscripts and by using the array power (`.^`) and array multiplication (`.*`) operators.

```
function dydt = vdp1000(t,y)
dydt = [y(2,:); 1000*(1-y(1,:).^2).*y(2,:)-y(1,:)];
```

Note Vectorization of the ODE function used by the ODE solvers differs from the vectorization used by the boundary value problem (BVP) solver, `bvp4c`. For the ODE solvers, the ODE function is vectorized only with respect to the second argument, while `bvp4c` requires vectorization with respect to the first and second arguments.

The following table describes the Jacobian matrix properties for `ode15i`.

Jacobian Properties for ode15i

Property	**Value**	**Description**
`Jacobian`	Function handle | Cell array of constant values	Function that evaluates the Jacobian or a cell array of constant values.
`JPattern`	Sparse matrices of {0,1}	Generates a sparse Jacobian matrix numerically.
`Vectorized`	`on` | {`off`}	Vectorized ODE function

Description of Jacobian Properties for ode15i

`Jacobian` — Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function

```
[dFdy, dFdp] = Fjac(t,y,yp)
```

or to a cell array of constant values $\{\partial F / \partial y, (\partial F / \partial y)'\}$.

`JPattern` — `JPattern` is a sparsity pattern with 1's where there might be nonzero entries in the Jacobian.

Set this property to `{dFdyPattern, dFdypPattern}`, the sparsity patterns of $\partial F / \partial y$ and $\partial F / \partial y'$, respectively.

`Vectorized` —

Set this property to `{yVect, ypVect}`. Setting `yVect` to `'on'` indicates that

```
F(t, [y1 y2 ...], yp)
```

returns

```
[F(t,y1,yp), F(t,y2,yp) ...]
```

Setting `ypVect` to `'on'` indicates that

```
F(t,y,[yp1 yp2 ...])
```

returns

```
[F(t,y,yp1) F(t,y,yp2) ...]
```

Mass Matrix and DAE Properties

This section describes mass matrix and differential-algebraic equation (DAE) properties, which apply to all the solvers except `ode15i`. These properties are not applicable to `ode15i` and their settings do not affect its behavior.

The solvers of the ODE suite can solve ODEs of the form

$$M(t,y)y' = f(t,y) \tag{2-1}$$

with a mass matrix $M(t,y)$ that can be sparse.

When $M(t,y)$ is nonsingular, the equation above is equivalent to $y' = M^{-1}f(t,y)$ and the ODE has a solution for any initial values y_0 at t_0. The more general form (Equation 2-1) is convenient when you express a model naturally in terms of a mass matrix. For large, sparse $M(t,y)$, solving Equation 2-1 directly reduces the storage and run-time needed to solve the problem.

When $M(t,y)$ is singular, then $M(t,y)$ times $M(t,y)y' = f(t,y)$ is a DAE. A DAE has a solution only when y_0 is consistent; that is, there exists an initial slope yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. If y_0 and yp_0 are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. For DAEs of index 1, solving an initial value problem with consistent initial conditions is much like solving an ODE.

The `ode15s` and `ode23t` solvers can solve DAEs of index 1. For examples of DAE problems, see Example: Differential-Algebraic Problem, in the MATLAB Mathematics documentation, and the examples `amp1dae` and `hb1dae`.

The following table describes the mass matrix and DAE properties. Further information on each property is given following the table.

Mass Matrix and DAE Properties (Solvers Other Than ode15i)

Property	Value	Description
`Mass`	Matrix | function handle	Mass matrix or a function that evaluates the mass matrix $M(t, y)$.

`MStateDependence`	`none |`
	`{weak} |`
	`strong`
		Dependence of the mass matrix on y.

`MvPattern`	Sparse matrix	$\partial(M(t, y)v)/\partial y$ sparsity pattern.
`MassSingular`	`yes | no |` `{maybe}`	Indicates whether the mass matrix is singular.
`InitialSlope`	Vector {zero vector}	Vector representing the consistent initial slope yp_0.

Description of Mass Matrix and DAE Properties

`Mass` — For problems of the form $M(t)y' = f(t, y)$, set `'Mass'` to a mass matrix M. For problems of the form $M(t)y' = f(t, y)$, set `'Mass'` to a function handle `@Mfun`, where `Mfun(t,y)` evaluates the mass matrix $M(t, y)$. The `ode23s` solver can only solve problems with a constant mass matrix M. When solving DAEs, using `ode15s` or `ode23t`, it is advantageous to formulate the problem so that$M$ is a diagonal matrix (a semiexplicit DAE).

For example problems, see in the MATLAB Mathematics documentation, or the examples `fem2ode` or `batonode`.

`MStateDependence` — Set this property to `none` for problems $M(t)y' = f(t, y)$. Both `weak` and `strong` indicate $M(t, y)$, but `weak` results in implicit solvers using approximations when solving algebraic equations.

`MvPattern` — Set this property to a sparse matrix *S* with *S(i,j) = 1* if, for any *k*, the *(i,k)* component of $M(t,y)$ depends on component *j* of *y*, and 0 otherwise. For use with the `ode15s`, `ode23t`, and `ode23tb` solvers when `MStateDependence` is `strong`. See `burgersode` as an example.

`MassSingular` — Set this property to `no` if the mass matrix is not singular and you are using either the `ode15s` or `ode23t` solver. The default value of `maybe` causes the solver to test whether the problem is a DAE, by testing whether $M(t_0, y_0)$ is singular.

`InitialSlope` — Vector representing the consistent initial slope yp_0, where yp_0 satisfies $M(t_0, y_0) \cdot y'_0 = f(t_0, y_0)$. The default is the zero vector.

This property is for use with the `ode15s` and `ode23t` solvers when solving DAEs.

ode15s and ode15i-Specific Properties

`ode15s` is a variable-order solver for stiff problems. It is based on the numerical differentiation formulas (NDFs). The NDFs are generally more efficient than the closely related family of backward differentiation formulas (BDFs), also known as Gear's methods. The `ode15s` properties let you choose among these formulas, as well as specifying the maximum order for the formula used.

`ode15i` solves fully implicit differential equations of the form

$$f(t, y, y') = 0$$

using the variable order BDF method.

The following table describes the `ode15s` and `ode15i`-specific properties. Further information on each property is given following the table. Use `odeset` to set these properties.

ode15s and ode15i-Specific Properties

Property	Value	Description
`MaxOrder`	`1 | 2 | 3 | 4 | {5}`	Maximum order formula used to compute the solution.
`BDF` (`ode15s` only)	`on | {off}`	Specifies whether you want to use the BDFs instead of the default NDFs.

Description of ode15s and ode15i-Specific Properties

`MaxOrder` — Maximum order formula used to compute the solution.

`BDF` (`ode15s` only) — Set `BDF` `on` to have `ode15s` use the BDFs.

For both the NDFs and BDFs, the formulas of orders 1 and 2 are A-stable (the stability region includes the entire left half complex plane). The higher order formulas are not as stable, and the higher the order the worse the stability. There is a class of stiff problems (stiff oscillatory) that is solved more efficiently if `MaxOrder` is reduced (for example to `2`) so that only the most stable formulas are used.

See Also

`deval`, `odeget`, `ode45`, `ode23`, `ode23t`, `ode23tb`, `ode113`, `ode15s`, `ode23s`, `function_handle` (`@`)

odextend

Purpose

Extend solution of initial value problem for ordinary differential equation

Syntax

```
solext = odextend(sol, odefun, tfinal)
solext = odextend(sol, [], tfinal)
solext = odextend(sol, odefun, tfinal, yinit)
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
solext = odextend(sol, odefun, tfinal, yinit, options)
```

Description

`solext = odextend(sol, odefun, tfinal)` extends the solution stored in `sol` to an interval with upper bound `tfinal` for the independent variable. `odefun` is a function handle. See in the MATLAB Programming documentation for more information. `sol` is an ODE solution structure created using an ODE solver. The lower bound for the independent variable in `solext` is the same as in `sol`. If you created `sol` with an ODE solver other than `ode15i`, the function `odefun` computes the right-hand side of the ODE equation, which is of the form $y' = f(t, y)$. If you created `sol` using `ode15i`, the function `odefun` computes the left-hand side of the ODE equation, which is of the form $f(t, y, y') = 0$.

, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `odefun`, if necessary.

`odextend` extends the solution by integrating `odefun` from the upper bound for the independent variable in `sol` to `tfinal`, using the same ODE solver that created `sol`. By default, `odextend` uses

- The initial conditions `y = sol.y(:, end)` for the subsequent integration
- The same integration properties and additional input arguments the ODE solver originally used to compute `sol`. This information is stored as part of the solution structure `sol` and is subsequently passed to `solext`. Unless you want to change these values, you do not need to pass them to `odextend`.

`solext = odextend(sol, [], tfinal)` uses the same ODE function that the ODE solver uses to compute `sol` to extend the solution. It is not necessary to pass in `odefun` explicitly unless it differs from the original ODE function.

`solext = odextend(sol, odefun, tfinal, yinit)` uses the column vector `yinit` as new initial conditions for the subsequent integration, instead of the vector `sol.y(end)`.

Note To extend solutions obtained with `ode15i`, use the following syntax, in which the column vector `ypinit` is the initial derivative of the solution:

```
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
```

`solext = odextend(sol, odefun, tfinal, yinit, options)` uses the integration properties specified in `options` instead of the options the ODE solver originally used to compute `sol`. The new options are then stored within the structure `solext`. See `odeset` for details on setting `options` properties. Set `yinit = []` as a placeholder to specify the default initial conditions.

Example

The following command

```
sol=ode45(@vdp1,[0 10],[2 0]);
```

uses `ode45` to solve the system `y' = vdp1(t,y)`, where `vdp1` is an example of an ODE function provided with MATLAB software, on the interval `[0 10]`. Then, the commands

```
sol=odextend(sol,@vdp1,20);
plot(sol.x,sol.y(1,:));
```

extend the solution to the interval `[0 20]` and plot the first component of the solution on `[0 20]`.

See Also

`deval`, `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `ode15i`, `odeset`, `odeget`, `deval`, `function_handle` (`@`)

Purpose

Cleanup tasks at function completion

Syntax

`C = onCleanup(S)`

Description

`C = onCleanup(S)` when called in function `F`, specifies any cleanup tasks that need to be performed when `F` completes. `S` is a handle to a function that performs necessary cleanup work when `F` exits. (For example, closing files that have been opened by `F`). `S` is called whether `F` exits normally or because of an error.

`onCleanup` is a MATLAB class and `C = onCleanup(S)` constructs an instance `C` of that class. Whenever an object of this class is explicitly or implicitly cleared from the workspace, it runs the cleanup function, `S`. Objects that are local variables in a function are implicitly cleared at the termination of that function.

Examples

Use `onCleanup` to close a file in the first example, and to restore the current directory in the second:

```
function fileOpenSafely(fileName)
   fid = fopen(fileName, 'w');
   c = onCleanup(@()fclose(fid));

   functionThatMayError(fid);
end   % c executes fclose(fid) here
```

MATLAB closes `fid` whether `functionThatMayError` returns an error or not.

```
function changeDirectorySafely(fileName)
   currentDir = pwd;
   c = onCleanup(@()cd(currentDir));

   functionThatMayError;
end   % c executes cd(currentDir) here
```

The current directory is preserved whether `functionThatMayError` returns an error or not.

See Also

`clear`, `clearvars`

Purpose

Create array of all ones

Syntax

```
Y = ones(n)
Y = ones(m,n)
Y = ones([m n])
Y = ones(m,n,p,...)
Y = ones([m n p ...])
Y = ones(size(A))
ones(m, n,...,classname)
ones([m,n,...],classname)
```

Description

`Y = ones(n)` returns an `n`-by-`n` matrix of `1`s. An error message appears if `n` is not a scalar.

`Y = ones(m,n)` or `Y = ones([m n])` returns an `m`-by-`n` matrix of ones.

`Y = ones(m,n,p,...)` or `Y = ones([m n p ...])` returns an `m`-by-`n`-by-`p`-by-`...` array of `1`s.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`Y = ones(size(A))` returns an array of `1`s that is the same size as `A`.

`ones(m, n,...,classname)` or `ones([m,n,...],classname)` is an `m`-by-`n`-by-... array of ones of data type `classname`. `classname` is a string specifying the data type of the output. `classname` can have the following values: `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Example

```
x = ones(2,3,'int8');
```

See Also

`eye`, `zeros`, `complex`

Purpose Open file in appropriate application

Syntax `open(name)`

Description `open(name)` opens the specified file or variable in the appropriate application.

Inputs *`name`*

Name of file or variable to open. If *`name`* does not include an extension, the `open` function:

1. Searches for a variable named *`name`*. If the variable exists, `open` opens it in the MATLAB Variable Editor.
2. Searches the MATLAB path for *`name`*`.m` or *`name`*`.mdl`. If only *`name`*`.m` exists, `open` opens the file in the M-file Editor. If only *`name`*`.mdl` exists, then `open` opens the model in Simulink.

If more than one file named *`name`* exists on the MATLAB path, the `open` function opens the file returned by `which(name)`.

Extension	Action
`.m`	Open in M-file Editor.
`.mat`	Return variables in structure *`st`* when called with the syntax: `st = open(name)`
`.fig`	Open figure in Handle Graphics, with handle *`hd`* when called with the syntax: `hd = open(name)`
`.mdl`	Open model in Simulink.

(Continued)

Extension	Action
`.prj`	Open project in the MATLAB Compiler Deployment Tool.
`.doc*`	Open document in Microsoft Word.
`.exe`	Run executable file (only on Windows systems).
`.pdf`	Open document in Adobe® Acrobat®.
`.ppt*`	Open document in Microsoft PowerPoint.
`.xls*`	Start MATLAB Import Wizard.
`.htm` or `.html`	Open document in MATLAB browser.
`.url`	Open file in your default Web browser.

Extend the functionality of `open` by defining your own function of the form `open`*`xxx`*, where *`xxx`* is a file extension. For example, if you create a function `openlog`, the `open` function calls `openlog` to process any files with the `.log` extension.

Examples

Open `Contents.m` in the M-file Editor by typing:

```
open Contents.m
```

Generally, MATLAB opens *`matlabroot`*`\toolbox\matlab\general\Contents.m`. However, if you have a file called `Contents.m` in a directory that is before `toolbox\matlab\general` on the MATLAB path, then `open` opens that file instead. To change the path, select **File > Set Path**.

Open a file not on the MATLAB path by including the complete file specification:

```
open('D:\temp\data.mat')
```

If the file does not exist, MATLAB displays an error message.

Create an M-file function called `opentxt` to handle files with extension `.txt`:

```
function opentxt(filename)

   fprintf('You have requested file: %s\n', filename);

   wh = which(filename);
   if ~isempty(wh)
     fprintf('Opening in M-file Editor: %s\n', wh);
     edit(wh);
   elseif exist(filename, 'file') == 2
     fprintf('Opening in M-file Editor: %s\n', filename);
     edit(filename);
   else
     warning('MATLAB:fileNotFound', ...
             'File was not found: %s', filename);
   end

end
```

Open the file `ngc6543a.txt` (a description of `ngc6543a.jpg`, located in *`matlabroot`*`\toolbox\matlab\demos`):

```
photo_text = 'ngc6543a.txt';
open(photo_text)
```

`open` calls your function with the following syntax:

```
opentxt(photo_text)
```

See Also

`edit` | `fileformats` | `load` | `openfig` | `openvar` | `path` | `uiopen` | `which` | `winopen`

Purpose

Open new copy or raise existing copy of saved figure

Syntax

```
openfig('filename.fig')
openfig('filename.fig','new')
openfig('filename.fig','reuse')
openfig('filename.fig','new','invisible')
openfig('filename.fig','reuse','invisible')
openfig('filename.fig','new','visible')
openfig('filename.fig','reuse','visible')
figure_handle = openfig(...)
```

Description

`openfig('filename.fig')` and `openfig('filename.fig','new')` opens the figure contained in the FIG-file, `filename.fig`, and ensures it is visible and positioned completely on screen. You do not have to specify the full path to the FIG-file as long as it is on your MATLAB path. The `.fig` extension is optional.

`openfig('filename.fig','reuse')` opens the figure contained in the FIG-file only if a copy of the figure is not currently open. Otherwise, `openfig` brings the existing copy forward, making sure it is still visible and completely on screen.

`openfig('filename.fig','new','invisible')` or `openfig('filename.fig','reuse','invisible')` opens the figure as in the preceding example, while forcing the figure to be invisible.

`openfig('filename.fig','new','visible')` or `openfig('filename.fig','reuse','visible')` opens the figure, while forcing the figure to be visible.

`figure_handle = openfig(...)` returns the handle to the figure.

Remarks

`openfig` is designed for use with GUI figures. Use this function to:

- Open the FIG-file creating the GUI and ensure it is displayed on screen. This provides compatibility with different screen sizes and resolutions.

- Control whether the MATLAB software displays one or multiple instances of the GUI at any given time.
- Return the handle of the figure created, which is typically hidden for GUI figures.

If the FIG-file contains an invisible figure, `openfig` returns its handle and leaves it invisible. The caller should make the figure visible when appropriate.

Do not use `openfig` or double-click a FIG-file to open a GUI created with GUIDE. Instead open the GUI code file by typing its name in the command window or by right-clicking its name in the Current Folder Browser and selecting **Run File**. To open a GUIDE GUI, for example one called `guifile.m`, in an invisible state, specify the `Visible` property in your command:

```
guifile('Visible','off')
```

Your code should then make the figure visible at an appropriate time.

See Also

`guide`, `guihandles`, `movegui`, `open`, `hgload`, `save`

See Deploying User Interfaces in the MATLAB documentation for related functions

Purpose

Control OpenGL rendering

Syntax

```
opengl info
s = opengl('data')
opengl software
opengl hardware
opengl verbose
opengl quiet
opengl DriverBugWorkaround
opengl('DriverBugWorkaround',WorkaroundState)
```

Description

The OpenGL autoselection mode applies when the `RendererMode` of the figure is `auto`. Possible values for *`selection_mode`* are

- `autoselect` – allows OpenGL to be automatically selected if OpenGL is available and if there is graphics hardware on the host machine.
- `neverselect` – disables autoselection of OpenGL.
- `advise` – prints a message to the command window if OpenGL rendering is advised, but `RenderMode` is set to `manual`.

`opengl`, by itself, returns the current autoselection state.

Note that the autoselection state only specifies whether OpenGL should or should not be considered for rendering; it does not explicitly set the rendering to OpenGL. You can do this by setting the `Renderer` property of the figure to `OpenGL`. For example,

```
set(figure_handle,'Renderer','OpenGL')
```

`opengl info` prints information with the version and vendor of the OpenGL on your system. Also indicates wether your system is currently using hardware of software OpenGL and the state of various driver bug workarounds. Note that calling `opengl info` loads the OpenGL Library.

For example, the following output is generated on a Windows XP computer that uses ATI Technologies graphics hardware:

```
>> opengl info
Version         = 1.3.4010 WinXP Release
Vendor          = ATI Technologies Inc.
Renderer        = RADEON 9600SE x86/SSE2
MaxTextureSize  = 2048
Visual          = 05 (RGB 16 bits(05 06 05 00) zdepth 16, Hardware
Accelerated, Opengl, Double Buffered, Window)
Software        = false
# of Extensions = 85
Driver Bug Workarounds:
OpenGLBitmapZbufferBug    = 0
OpenGLWobbleTesselatorBug = 0
OpenGLLineSmoothingBug    = 0
OpenGLDockingBug          = 0
OpenGLClippedImageBug     = 0
```

Note that different computer systems may not list all OpenGL bugs.

`s = opengl('data')` returns a structure containing the same data that is displayed when you call `opengl info`, with the exception of the driver bug workaround state.

`opengl software` forces the MATLAB software to use software OpenGL rendering instead of hardware OpenGL. Note that Macintosh systems do not support software OpenGL.

`opengl hardware` reverses the `opengl software` command and enables MATLAB to use hardware OpenGL rendering if it is available. If your computer does not have OpenGL hardware acceleration, MATLAB automatically switches to software OpenGL rendering (except on Macintosh systems, which do not support software OpenGL).

Note that on UNIX systems, the `software` or `hardware` options with the `opengl` command works only if MATLAB has not yet used the OpenGL renderer or you have not issued the `opengl info` command (which attempts to load the OpenGL Library).

`opengl verbose` displays verbose messages about OpenGL initialization (if OpenGL is not already loaded) and other runtime messages.

`opengl quiet` disables verbose message setting.

`opengl` *`DriverBugWorkaround`* queries the state of the specified driver bug workaround. Use the command `opengl info` to see a list of all driver bug workarounds. See "Driver Bug Workarounds" on page 2-2657 for more information.

`opengl('`*`DriverBugWorkaround`*`',WorkaroundState)` sets the state of the specified driver bug workaround. You can set WorkaroundState to one of three values:

- `0` – Disable the specified *`DriverBugWorkaround`* (if enabled) and do not allow MATLAB to autoselect this workaround.
- `1` – Enable the specified *`DriverBugWorkaround`*.
- `-1` – Set the specified *`DriverBugWorkaround`* to autoselection mode, which allows MATLAB to enable this workaround if the requisite conditions exist.

Driver Bug Workarounds

The MATLAB software enables various OpenGL driver bug workarounds when it detects certain known problems with installed hardware. However, because there are many versions of graphics drivers, you might encounter situations when MATLAB does not enable a workaround that would solve a problem you are having with OpenGL rendering.

This section describes the symptoms that each workaround is designed to correct so you can decide if you want to try using one to fix an OpenGL rendering problem.

Use the `opengl info` command to see what driver bug workarounds are available on your computer.

Note These workarounds have not been tested under all driver combinations and therefore might produce undesirable results under certain conditions.

OpenGLBitmapZbufferBug

Symptom: text with background color (including data tips) and text displayed on image, patch, or surface objects is not visible when using OpenGL renderer.

Possible side effect: text is always on top of other objects.

Command to enable:

```
opengl('OpenGLBitmapZbufferBug',1)
```

OpenGLWobbleTesselatorBug

Symptom: Rendering complex patch object causes segmentation violation and returns a tesselator error message in the stack trace.

Command to enable:

```
opengl('OpenGLWobbleTesselatorBug',1)
```

OpenGLLineSmoothingBug

Symptom: Lines with a `LineWidth` greater than 3 look bad.

Command to enable:

```
opengl('OpenGLLineSmoothingBug',1)
```

OpenGLDockingBug

Symptom: MATLAB crashes when you dock a figure that has its `Renderer` property set to `opengl`.

Command to enable:

```
opengl('OpenGLDockingBug',1)
```

OpenGLClippedImageBug

Symptom: Images (as well as `colorbar` displays) do not display when the `Renderer` property set to `opengl`.

Command to enable:

```
opengl('OpenGLClippedImageBug',1)
```

OpenGLEraseModeBug

Symptom: Graphics objects with `EraseMode` property set to non-normal erase modes (`xor`, `none`, or `background`) do not draw when the figure `Renderer` property is set to `opengl`.

Command to enable:

```
opengl('OpenGLEraseModeBug',1)
```

See Also

Figure `Renderer` property for information on autoselection.

openvar

Purpose

Open workspace variable in Variable Editor or other graphical editing tool

GUI Alternatives

As an alternative to the `openvar` function, double-click a variable in the Workspace browser.

Syntax

```
openvar('varname')
```

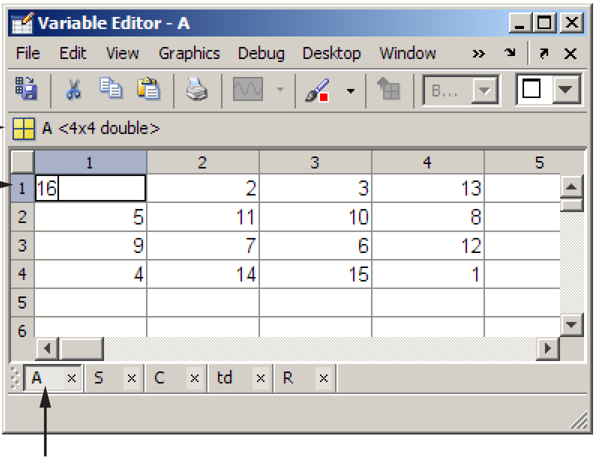
Description

`openvar('varname')` opens the workspace variable `varname` in the Variable Editor for graphical editing, where `name` is a one- or two-dimensional array, character string, cell array, structure, or an object and its properties. You also can view the contents of a multidimensional array. Changes that you make to variables in the Variable Editor occur in the workspace as soon as you enter them.

You need to enclose the variable's name in single quotation marks because the Variable Editor needs to know the name of the variable to be notified if the variable changes value is deleted or goes out of scope. Typing `openvar(varname)` instead of `openvar('varname')`, passes the Variable Editor the value of `varname` instead of its name, and generally results in an error. However, `openvar varname` and `openvar 'varname'` both work, because string arguments are assumed when using *command syntax*. See in the MATLAB Programming Fundamentals documentation for more information.

The MATLAB software does not impose any limitation on the size of a variable that you can open in the Variable Editor. Your operating system or the amount of physical memory installed on your computer can impose such limits, however.

In some toolboxes, `openvar` opens a tool appropriate for viewing or editing objects they define instead of opening the Variable Editor.



A <4x4 double>

	1	2	3	4	5
1	16	2	3	13	
2	5	11	10	8	
3	9	7	6	12	
4	4	14	15	1	
5					
6					

Size and class of the variable.

Click an element and then edit its value.

Select a tab to view a variable that you have open in the Variable Editor.

Data Brushing in the Variable Editor

The Data Brushing tool and the `brush` function let you manually highlight portions of graphs in the figure. You can also connect the data within graphs of numeric variables to their data sources (using the Linked Plot tool in the figure window or the `linkdata` function).

When you link graphs to source data and view the source data in the Variable Editor, observations that you highlight on graphs in Data Brushing mode also appear highlighted in the Variable Editor. Likewise, cells that you select in the Variable editor with its Data Brushing Tool appear highlighted in all linked figures which graph the variable.

Example — Identifying Outliers in a Linked Graph

Data Brushing helps to identify unusual observations in a data set that might warrant further analysis, for example extreme values. To explore this capability, follow these steps:

1. Make a scatter plot of data in MAT-file `count.dat`, and open the variable `count` in the Variable Editor. For example:

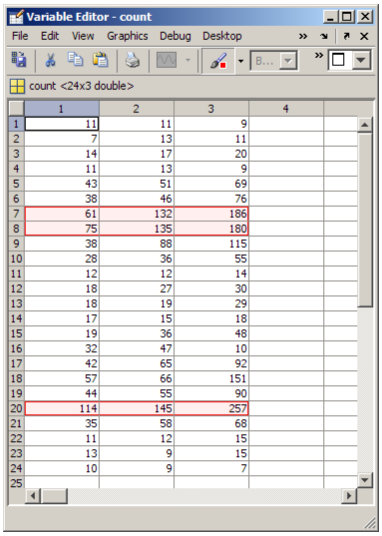
```
load count.dat
scatter(count(:,1),count(:,2))
openvar('count')
```

2 Open the Variable Editor, turn on its Data Brushing mode, and select the three highest values (rows 7, 8, and 20). (You select noncontiguous rows by holding down the **Ctrl** key and clicking them.)

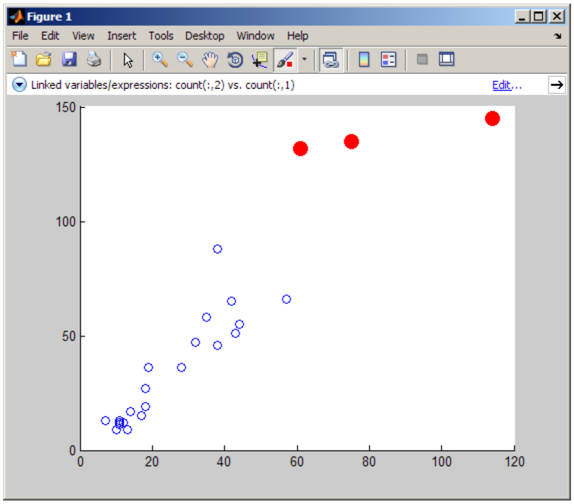
3 Turn on Data Brushing mode  and Data Linking mode  for the figure with the scatter plot, or type the following commands:

```
brush on
linkdata on
```

The data observations you brushed in the Variable Editor appear highlighted in the scatter plot, as the following figure shows.



	1	2	3	4
1	11	11	9	
2	7	13	11	
3	14	17	20	
4	11	13	9	
5	43	51	69	
6	38	46	76	
7	61	132	186	
8	75	135	180	
9	38	88	115	
10	28	36	55	
11	12	12	14	
12	18	27	30	
13	18	19	29	
14	17	15	18	
15	19	36	48	
16	32	47	10	
17	42	65	92	
18	57	66	151	
19	44	55	90	
20	114	145	257	
21	35	58	68	
22	11	12	15	
23	13	9	15	
24	10	9	7	
25				



Now brush other observations in the scatter plot and notice how the Variable Editor highlights these values, as long as the figure is in data linking mode. When a figure is not linked to its data sources, you can still brush its graphs and you can brush the same data in the Variable Editor, but only the display that you brush responds by highlighting.

You can turn data brushing on and off and perform a number of operations on brushed data from the **Brushing** item on the **Edit** menu. The operations include removing and replacing brushed observations, copying them to the clipboard or Command Window, and creating a new variable containing them. A **Brushing** context menu item also provides these options.

See Also

`brush`, `linkdata`, `load`, `save`, `workspace`

in the MATLAB Desktop Tools and Development Environment Documentation.

in the MATLAB Data Analysis documentation.

Purpose

Optimization options values

Syntax

```
val = optimget(options,'param')
val = optimget(options,'param',default)
```

Description

`val = optimget(options,'param')` returns the value of the specified parameter in the optimization options structure `options`. You need to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`val = optimget(options,'param',default)` returns `default` if the specified parameter is not defined in the optimization options structure `options`. Note that this form of the function is used primarily by other optimization functions.

Examples

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options.`

```
val = optimget(my_options,'Display')
```

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options` (as in the previous example) except that if the `Display` parameter is not defined, it returns the value `'final'.`

```
optnew = optimget(my_options,'Display','final');
```

See Also

`optimset`, `fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`

Purpose

Create or edit optimization options structure

Syntax

```
options = optimset('param1',value1,'param2',value2,...)
optimset
options = optimset
options = optimset(optimfun)
options = optimset(oldopts,'param1',value1,...)
options = optimset(oldopts,newopts)
```

Description

The function `optimset` creates an `options` structure that you can pass as an input argument to the following four MATLAB optimization functions:

- `fminbnd`
- `fminsearch`
- `fzero`
- `lsqnonneg`

You can use the `options` structure to change the default parameters for these functions.

Note If you have purchased the Optimization Toolbox, you can also use `optimset` to create an expanded `options` structure containing additional options specifically designed for the functions provided in that toolbox. See the reference page for the enhanced `optimset` function in the Optimization Toolbox for more information about these additional options.

`options = optimset('param1',value1,'param2',value2,...)` creates an optimization options structure called `options`, in which the specified parameters (`param`) have specified values. Any unspecified parameters are set to `[]` (parameters with value `[]` indicate to use the default value for that parameter when `options` is passed to the optimization function). It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`optimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = optimset` (with no input arguments) creates an options structure `options` where all fields are set to `[]`.

`options = optimset(optimfun)` creates an options structure `options` with all parameter names and default values relevant to the optimization function `optimfun`.

`options = optimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = optimset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

Options

The following table lists the available options for the MATLAB optimization functions.

Option	Value	Description
`Display`	`'off' | ’iter' | {'final'} | 'notify'`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` displays output only if the function does not converge.

Option	Value	Description
`FunValCheck`	{`'off'`} | `'on'`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex` or `NaN`. `'off'` displays no error.
`MaxFunEvals`	positive integer	Maximum number of function evaluations allowed.
`MaxIter`	positive integer	Maximum number of iterations allowed.
`OutputFcn`	function | {[ ]}	User-defined function that an optimization function calls at each iteration.
`PlotFcns`	function | {[ ]}	User-defined plot function that an optimization function calls at each iteration.
`TolFun`	positive scalar	Termination tolerance on the function value.
`TolX`	positive scalar	Termination tolerance on x.

Examples

This statement creates an optimization options structure called `options` in which the `Display` parameter is set to `'iter'` and the `TolFun` parameter is set to `1e-8`.

```
options = optimset('Display','iter','TolFun',1e-8)
```

This statement makes a copy of the options structure called `options`, changing the value of the `TolX` parameter and storing new values in `optnew`.

```
optnew = optimset(options,'TolX',1e-4);
```

This statement returns an optimization options structure that contains all the parameter names and default values relevant to the function `fminbnd`.

```
optimset('fminbnd')
```

See Also

`optimset` (Optimization Toolbox version), `optimget`, `fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`

Purpose

Find logical OR of array or scalar inputs

Syntax

```
A | B | ...
or(A, B)
```

Description

`A | B | ...` performs a logical OR of all input arrays `A`, `B`, etc., and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if any input arrays contain a nonzero element at that same array location. Otherwise, that element is set to 0.

Each input of the expression can be an array or can be a scalar value. All nonscalar input arrays must have equal dimensions. If one or more inputs are an array, then the output is an array of the same dimensions. If all inputs are scalar, then the output is scalar.

If the expression contains both scalar and nonscalar inputs, then each scalar input is treated as if it were an array having the same dimensions as the other input arrays. In other words, if input `A` is a 3-by-5 matrix and input `B` is the number 1, then `B` is treated as if it were a 3-by-5 matrix of ones.

`or(A, B)` is called for the syntax `A | B` when either `A` or `B` is an object.

Note The symbols `|` and `||` perform different operations in a MATLAB application. The element-wise OR operator described here is `|`. The short-circuit OR operator is `||`.

Example

If matrix `A` is

```
0.4235    0.5798         0    0.7942         0
0.5155         0         0         0    0.8744
     0         0         0    0.4451    0.0150
0.4329    0.6405    0.6808         0         0
```

and matrix `B` is

```
     0     1     0     1     0
     1     1     0     0     1
     0     0     0     1     0
     0     1     0     0     1
```

then

```
A | B
ans =
     1     1     0     1     0
     1     1     0     0     1
     0     0     0     1     1
     1     1     1     0     1
```

See Also

`bitor`, `and`, `xor`, `not`, `any`, `all`, logical operators, logical types, bitwise functions

Purpose

Eigenvalues of quasitriangular matrices

Syntax

```
E = ordeig(T)
E = ordeig(AA,BB)
```

Description

`E = ordeig(T)` takes a quasitriangular Schur matrix `T`, typically produced by `schur`, and returns the vector `E` of eigenvalues in their order of appearance down the diagonal of T.

`E = ordeig(AA,BB)` takes a quasitriangular matrix pair `AA` and `BB`, typically produced by `qz`, and returns the generalized eigenvalues in their order of appearance down the diagonal of `AA-λ*BB`.

`ordeig` is an order-preserving version of `eig` for use with `ordschur` and `ordqz`. It is also faster than `eig` for quasitriangular matrices.

Examples

Example 1

```
T=diag([1 -1 3 -5 2]);
```

`ordeig(T)` returns the eigenvalues of `T` in the same order they appear on the diagonal.

```
ordeig(T)

ans =

      1
     -1
      3
     -5
      2
```

`eig(T)`, on the other hand, returns the eigenvalues in order of increasing magnitude.

```
eig(T)

ans =
```

```
    -5
    -1
     1
     2
     3
```

Example 2

```
A = rand(10);
[U, T] = schur(A);
abs(ordeig(T))

ans =

    5.3786
    0.7564
    0.7564
    0.7802
    0.7080
    0.7080
    0.5855
    0.5855
    0.1445
    0.0812
% Move eigenvalues with magnitude < 0.5 to the
% upper-left corner of T.
[U,T] = ordschur(U,T,abs(E)<0.5);
abs(ordeig(T))

ans =

    0.1445
    0.0812
    5.3786
    0.7564
    0.7564
    0.7802
```

```
    0.7080
    0.7080
    0.5855
    0.5855
```

See Also `schur`, `qz`, `ordschur`, `ordqz`, `eig`

orderfields

Purpose

Order fields of structure array

Syntax

```
s = orderfields(s1)
s = orderfields(s1, s2)
s = orderfields(s1, c)
s = orderfields(s1, perm)
[s, perm] = orderfields(...)
```

Description

`s = orderfields(s1)` orders the fields in `s1` so that the new structure array `s` has field names in ASCII dictionary order.

`s = orderfields(s1, s2)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in `s2`. Structures `sl` and `s2` must have the same fields.

`s = orderfields(s1, c)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in the cell array of field name strings `c`. Structure `s1` and cell array `c` must contain the same field names.

`s = orderfields(s1, perm)` orders the fields in `s1` so that the new structure array `s` has fieldnames in the order specified by the indices in permutation vector `perm`.

If `s1` has `N` fieldnames, the elements of `perm` must be an arrangement of the numbers from `1` to `N`. This is particularly useful if you have more than one structure array that you would like to reorder in the same way.

`[s, perm] = orderfields(...)` returns a permutation vector representing the change in order performed on the fields of the structure array that results in `s`.

Remarks

`orderfields` only orders top-level fields. It is not recursive.

Examples

Create a structure `s`. Then create a new structure from `s`, but with the fields ordered alphabetically:

```
s = struct('b', 2, 'c', 3, 'a', 1)
s =
```

```
    b: 2
    c: 3
    a: 1

snew = orderfields(s)
snew =
    a: 1
    b: 2
    c: 3
```

Arrange the fields of `s` in the order specified by the second (cell array) argument of `orderfields`. Return the new structure in `snew` and the permutation vector used to create it in `perm`:

```
[snew, perm] = orderfields(s, {'b', 'a', 'c'})
snew =
    b: 2
    a: 1
    c: 3
perm =
     1
     3
     2
```

Now create a new structure, `s2`, having the same fieldnames as `s`. Reorder the fields using the permutation vector returned in the previous operation:

```
s2 = struct('b', 3, 'c', 7, 'a', 4)
s2 =
    b: 3
    c: 7
    a: 4

snew = orderfields(s2, perm)
snew =
    b: 3
    a: 4
```

```
c: 7
```

See Also `struct`, `fieldnames`, `setfield`, `getfield`, `isfield`, `rmfield`, dynamic field names

Purpose

Reorder eigenvalues in QZ factorization

Syntax

```
[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)
[...] = ordqz(AA,BB,Q,Z,keyword)
[...] = ordqz(AA,BB,Q,Z,clusters)
```

Description

`[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)` reorders the QZ factorizations `Q*A*Z = AA` and `Q*B*Z = BB` produced by the `qz` function for a matrix pair `(A,B)`. It returns the reordered pair `(AAS,BBS)` and the cumulative orthogonal transformations `QS` and `ZS` such that `QS*A*ZS = AAS` and `QS*B*ZS = BBS`. In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular pair `(AAS,BBS)`, and the corresponding invariant subspace is spanned by the leading columns of `ZS`. The logical vector `select` specifies the selected cluster as `E(select)` where `E` is the vector of eigenvalues as they appear along the diagonal of `AA-λ*BB`.

Note To extract `E` from `AA` and `BB`, use `ordeig(BB)`, instead of `eig`. This ensures that the eigenvalues in `E` occur in the same order as they appear on the diagonal of `AA-λ*BB`.

`[...] = ordqz(AA,BB,Q,Z,keyword)` sets the selected cluster to include all eigenvalues in the region specified by `keyword`:

keyword	Selected Region
`'lhp'`	Left-half plane (`real(E) < 0`)
`'rhp'`	Right-half plane (`real(E) > 0`)
`'udi'`	Interior of unit disk (`abs(E) < 1`)
`'udo'`	Exterior of unit disk (`abs(E) > 1`)

`[...] = ordqz(AA,BB,Q,Z,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices commensurate with `E = ordeig(AA,BB)`, such that all eigenvalues with the same `clusters` value form one cluster, `ordqz` sorts the specified clusters in descending order along the diagonal of `(AAS,BBS)`. The cluster with highest index appears in the upper left corner.

Algorithm

For full matrices `AA` and `BB`, `qz` uses the LAPACK routines listed in the following table.

	AA and BB Real	**AA or BB Complex**
`A` and `B` `double`	`DTGSEN`	`ZTGSEN`
`A` or `B` `single`	`STGSEN`	`CTGSEN`

See Also

`ordeig`, `ordschur`, `qz`

Purpose

Reorder eigenvalues in Schur factorization

Syntax

```
[US,TS] = ordschur(U,T,select)
[US,TS] = ordschur(U,T,keyword)
[US,TS] = ordschur(U,T,clusters)
```

Description

`[US,TS] = ordschur(U,T,select)` reorders the Schur factorization `X = U*T*U'` produced by the `schur` function and returns the reordered Schur matrix `TS` and the cumulative orthogonal transformation `US` such that `X = US*TS*US'`. In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular Schur matrix `TS`, and the corresponding invariant subspace is spanned by the leading columns of `US`. The logical vector `select` specifies the selected cluster as `E(select)` where `E` is the vector of eigenvalues as they appear along `T`'s diagonal.

Note To extract `E` from `T`, use `E = ordeig(T)`, instead of `eig`. This ensures that the eigenvalues in `E` occur in the same order as they appear on the diagonal of `TS`.

`[US,TS] = ordschur(U,T,keyword)` sets the selected cluster to include all eigenvalues in one of the following regions:

keyword	Selected Region
`'lhp'`	Left-half plane (`real(E) < 0`)
`'rhp'`	Right-half plane (`real(E) > 0`)
`'udi'`	Interior of unit disk (`abs(E) < 1`)
`'udo'`	Exterior of unit disk (`abs(E) > 1`)

`[US,TS] = ordschur(U,T,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices, commensurate with `E = ordeig(T)`, and such that all eigenvalues with the same `clusters` value form one cluster, `ordschur` sorts the specified clusters in descending order along the diagonal of `TS`, the cluster with highest index appearing in the upper left corner.

Algorithm

Input of Type Double

If `U` and `T` have type `double`, `ordschur` uses the LAPACK routines listed in the following table to compute the Schur form of a matrix:

Matrix Type	Routine
Real	`DTRSEN`
Complex	`ZTRSEN`

Input of Type Single

If `U` and `T` have type `single`, `ordschur` uses the LAPACK routines listed in the following table to reorder the Schur form of a matrix:

Matrix Type	Routine
Real	`STRSEN`
Complex	`CTRSEN`

See Also

`ordeig`, `ordqz`, `schur`

Purpose

Hardcopy paper orientation

GUI Alternative

Use **File —> Print Preview** on the figure window menu to directly manipulate print layout, paper size, headers, fonts and other properties when printing figures. For details, see Using Print Preview in the MATLAB Graphics documentation.

Syntax

```
orient
orient landscape
orient portrait
orient tall
orient(fig_handle), orient(simulink_model)
orient(fig_handle,orientation), orient(simulink_model,
   orientation)
```

Description

`orient` returns a string with the current paper orientation: `portrait`, `landscape`, or `tall`.

`orient landscape` sets the paper orientation of the current figure to full-page landscape, orienting the longest page dimension horizontally. The figure is centered on the page and scaled to fit the page with a 0.25 inch border.

`orient portrait` sets the paper orientation of the current figure to portrait, orienting the longest page dimension vertically. The `portrait` option returns the page orientation to the MATLAB default. (Note that the result of using the `portrait` option is affected by changes you make to figure properties. See the "Algorithm" section for more specific information.)

`orient tall` maps the current figure to the entire page in portrait orientation, leaving a 0.25 inch border.

`orient(fig_handle), orient(simulink_model)` returns the current orientation of the specified figure or Simulink model.

`orient(fig_handle,`*`orientation`*`), orient(simulink_model,`*`orientation`*`)` sets the orientation for the specified figure or Simulink model to the specified orientation (`landscape`, `portrait`, or `tall`).

Algorithm

`orient` sets the `PaperOrientation`, `PaperPosition`, and `PaperUnits` properties of the current figure. Subsequent print operations use these properties. The result of using the `portrait` option can be affected by default property values as follows:

- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the current values of `PaperOrientation` and `PaperPosition` to place the figure on the page.
- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the default figure `PaperPosition` with the x, y and width, height values reversed (i.e., [y,x,height,width]) to position the figure on the page.
- If the current figure `PaperType` is different from the default figure `PaperType`, then the `orient portrait` command uses the current figure `PaperPosition` with the x, y and width, height values reversed (i.e., [y,x,height,width]) to position the figure on the page.

See Also

`print`, `printpreview`, `set`

`PaperOrientation`, `PaperPosition`, `PaperSize`, `PaperType`, and `PaperUnits` properties of figure graphics objects

“Printing” on page 1-97 for related functions

Purpose Range space of matrix

Syntax `B = orth(A)`

Description `B = orth(A)` returns an orthonormal basis for the range of `A`. The columns of `B` span the same space as the columns of `A`, and the columns of `B` are orthogonal, so that `B'*B = eye(rank(A))`. The number of columns of `B` is the rank of `A`.

See Also `null`, `svd`, `rank`

otherwise

Purpose

Default part of switch statement

Syntax

```
switch switch_expr
 case case_expr
    statement, ..., statement
  case {case_expr1, case_expr2, case_expr3, ...}
    statement, ..., statement
  otherwise
    statement, ..., statement
end
```

Description

`otherwise` is part of the `switch` statement syntax, which allows for conditional execution. The statements following `otherwise` are executed only if none of the preceding case expressions (`case_expr`) matches the switch expression (`sw_expr`).

Examples

The general form of the `switch` statement is

```
switch sw_expr
    case case_expr
        statement
        statement
    case {case_expr1,case_expr2,case_expr3}
        statement
        statement
    otherwise
        statement
        statement
end
```

See `switch` for more details.

See Also

`switch`, `case`, `end`, `if`, `else`, `elseif`, `while`

Index