MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

The MATH WORKS Inc.

MATLAB Function Reference

Volume 2: F - O

Version 6

How to Contact The MathWorks:

508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail
`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

MATLAB Function Reference Volume 2 F- O

Printing History:			
	December 1996	First printing	New for MATLAB 5.0 (Release 10)
	June 1997	Revised for 5.1	Online version, MATLAB 5.1
	October 1997	Revised for 5.2	Online version, MATLAB 5.2
	January 1999	Revised for 5.3	Online version (Release 11)
	June 1999	Second printing	MATLAB 5.3 (Release 11)
	November 2000	Revised for 6.0	Online version (Release 12)

1 Functions by Category

1

Functions by Category

This section lists MATLAB functions grouped by functional area.

General Purpose Commands

Managing Commands and Functions

`addpath`	Add directories to MATLAB's search path
`doc`	Display HTML documentation in Help browser
`docopt`	Display location of help file directory for UNIX platforms
`genpath`	Generate a path string
`help`	Display M-file help for MATLAB functions in the Command Window
`helpbrowser`	Display Help browser for access to all MathWorks online help
`helpdesk`	Display the Help browser
`helpwin`	Display M-file help and provide access to M-file help for all functions
`lasterr`	Last error message
`lastwarn`	Last warning message
`license`	Show MATLAB license number
`lookfor`	Search for specified keyword in all help entries
`partialpath`	Partial pathname
`path`	Control MATLAB's directory search path
`pathtool`	Open the GUI for viewing and modifying MATLAB's path
`profile`	Start the M-file profiler, a utility for debugging and optimizing code
`profreport`	Generate a profile report
`rehash`	Refresh function and file system caches
`rmpath`	Remove directories from MATLAB's search path
`support`	Open MathWorks Technical Support Web Page
`type`	List file
`ver`	Display version information for MATLAB, Simulink, and toolboxes
`version`	Get MATLAB version number
`web`	Point Help browser or Web browser at file or Web site
`what`	List MATLAB-specific files in current directory
`whatsnew`	Display README files for MATLAB and toolboxes
`which`	Locate functions and files

Managing Variables and the Workspace

`clear`	Remove items from the workspace
`disp`	Display text or array
`length`	Length of vector
`load`	Retrieve variables from disk
`memory`	Help for memory limitations
`mlock`	Prevent M-file clearing
`munlock`	Allow M-file clearing
`openvar`	Open workspace variable in Array Editor, for graphical editing
`pack`	Consolidate workspace memory
`save`	Save workspace variables on disk
`saveas`	Save figure or model using specified format
`size`	Array dimensions
`who, whos`	List the variables in the workspace
`workspace`	Display the Workspace Browser, a GUI for managing the workspace

Controlling the Command Window

`clc`	Clear Command Window
`echo`	Echo M-files during execution
`format`	Control the display format for output
`home`	Move cursor to upper left corner of Command Window
`more`	Control paged output for the Command Window

Working with Files and the Operating Environment

`beep`	Produce a beep sound
`cd`	Change working directory
`checkin`	Check file into source control system
`checkout`	Check file out of source control system
`cmopts`	Get name of source control system, and PVCS project filename
`copyfile`	Copy file
`customverctrl`	Allow custom source control system
`delete`	Delete files or graphics objects
`diary`	Save session to a disk file
`dir`	Display a directory listing
`dos`	Execute a DOS command and return the result
`edit`	Edit an M-file
`fileparts`	Get filename parts
`filebrowser`	Display Current Directory browser, for viewing files
`fullfile`	Build full filename from parts
`info`	Display contact information or toolbox Readme files
`inmem`	Functions in memory

`ls`	List directory on UNIX
`matlabroot`	Get root directory of MATLAB installation
`mkdir`	Make new directory
`open`	Open files based on extension
`pwd`	Display current directory
`tempdir`	Return the name of the system's temporary directory
`tempname`	Unique name for temporary file
`undocheckout`	Undo previous checkout from source control system
`unix`	Execute a UNIX command and return the result
`!`	Execute operating system command

Starting and Quitting MATLAB

`finish`	MATLAB termination M-file
`exit`	Terminate MATLAB
`matlab`	Start MATLAB (UNIX systems only)
`matlabrc`	MATLAB startup M-file
`quit`	Terminate MATLAB
`startup`	MATLAB startup M-file

Operators and Special Characters

`+`	Plus
`-`	Minus
`*`	Matrix multiplication
`.*`	Array multiplication
`^`	Matrix power
`.^`	Array power
`kron`	Kronecker tensor product
`\`	Backslash or left division
`/`	Slash or right division
`./` and `.\`	Array division, right and left
`:`	Colon
`( )`	Parentheses
`[ ]`	Brackets
`{}`	Curly braces
`.`	Decimal point
`...`	Continuation
`,`	Comma
`;`	Semicolon
`%`	Comment
`!`	Exclamation point

`'`	Transpose and quote
`.'`	Nonconjugated transpose
`=`	Assignment
`==`	Equality
`< >`	Relational operators
`&`	Logical AND
`|`	Logical OR
`~`	Logical NOT
`xor`	Logical EXCLUSIVE OR

Logical Functions

`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzeros
`exist`	Check if a variable or file exists
`find`	Find indices and values of nonzero elements
`is*`	Detect state
`isa`	Detect an object of a given class
`iskeyword`	Test if string is a MATLAB keyword
`isvarname`	Test if string is a valid variable name
`logical`	Convert numeric values to logical
`mislocked`	True if M-file cannot be cleared

Language Constructs and Debugging

MATLAB as a Programming Language

`builtin`	Execute builtin function from overloaded method
`eval`	Interpret strings containing MATLAB expressions
`evalc`	Evaluate MATLAB expression with capture
`evalin`	Evaluate expression in workspace
`feval`	Function evaluation
`function`	Function M-files
`global`	Define global variables
`nargchk`	Check number of input arguments
`persistent`	Define persistent variable
`script`	Script M-files

Control Flow

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

`case`	Case switch
`catch`	Begin catch block
`continue`	Pass control to the next iteration of `for` or `while` loop
`else`	Conditionally execute statements
`elseif`	Conditionally execute statements
`end`	Terminate `for`, `while`, `switch`, `try`, and `if` statements or indicate last index
`error`	Display error messages
`for`	Repeat statements a specific number of times
`if`	Conditionally execute statements
`otherwise`	Default part of `switch` statement
`return`	Return to the invoking function
`switch`	Switch among several cases based on expression
`try`	Begin `try` block
`warning`	Display warning message
`while`	Repeat statements an indefinite number of times

Interactive Input

`input`	Request user input
`keyboard`	Invoke the keyboard in an M-file
`menu`	Generate a menu of choices for user input
`pause`	Halt execution temporarily

Object-Oriented Programming

`class`	Create object or return class of object
`double`	Convert to double precision
`inferiorto`	Inferior class relationship
`inline`	Construct an inline object
`int8`, `int16`, `int32`	Convert to signed integer
`isa`	Detect an object of a given class
`loadobj`	Extends the `load` function for user objects
`saveobj`	Save filter for objects
`single`	Convert to single precision
`superiorto`	Superior class relationship
`uint8`, `uint16`, `uint32`	Convert to unsigned integer

Debugging

`dbclear`	Clear breakpoints

`dbcont`	Resume execution
`dbdown`	Change local workspace context
`dbmex`	Enable MEX-file debugging
`dbquit`	Quit debug mode
`dbstack`	Display function call stack
`dbstatus`	List all breakpoints
`dbstep`	Execute one or more lines from a breakpoint
`dbstop`	Set breakpoints in an M-file function
`dbtype`	List M-file with line numbers
`dbup`	Change local workspace context

Function Handles

`function_handle`	MATLAB data type that is a handle to a function
`functions`	Return information about a function handle
`func2str`	Constructs a function name string from a function handle
`str2func`	Constructs a function handle from a function name string

Elementary Matrices and Matrix Manipulation

Elementary Matrices and Arrays

`blkdiag`	Construct a block diagonal matrix from input arguments
`eye`	Identity matrix
`linspace`	Generate linearly spaced vectors
`logspace`	Generate logarithmically spaced vectors
`numel`	Number of elements in a matrix or cell array
`ones`	Create an array of all ones
`rand`	Uniformly distributed random numbers and arrays
`randn`	Normally distributed random numbers and arrays
`zeros`	Create an array of all zeros
`:` (colon)	Regularly spaced vector

Special Variables and Constants

`ans`	The most recent answer
`computer`	Identify the computer on which MATLAB is running
`eps`	Floating-point relative accuracy
`i`	Imaginary unit
`Inf`	Infinity
`inputname`	Input argument name

`j`	Imaginary unit
`NaN`	Not-a-Number
`nargin`, `nargout`	Number of function arguments
`nargoutchk`	Validate number of output arguments
`pi`	Ratio of a circle's circumference to its diameter, π
`realmax`	Largest positive floating-point number
`realmin`	Smallest positive floating-point number
`varargin`, `varargout`	Pass or return variable numbers of arguments

Time and Dates

`calendar`	Calendar
`clock`	Current time as a date vector
`cputime`	Elapsed CPU time
`date`	Current date string
`datenum`	Serial date number
`datestr`	Date string format
`datevec`	Date components
`eomday`	End of month
`etime`	Elapsed time
`now`	Current date and time
`tic, toc`	Stopwatch timer
`weekday`	Day of the week

Matrix Manipulation

`cat`	Concatenate arrays
`diag`	Diagonal matrices and diagonals of a matrix
`fliplr`	Flip matrices left-right
`flipud`	Flip matrices up-down
`repmat`	Replicate and tile an array
`reshape`	Reshape array
`rot90`	Rotate matrix 90 degrees
`tril`	Lower triangular part of a matrix
`triu`	Upper triangular part of a matrix
`:` (colon)	Index into array, rearrange array

Vector Functions

`cross`	Vector cross product
`dot`	Vector dot product

`intersect`	Set intersection of two vectors
`ismember`	Detect members of a set
`setdiff`	Return the set difference of two vector
`setxor`	Set exclusive or of two vectors
`union`	Set union of two vectors
`unique`	Unique elements of a vector

Specialized Matrices

`compan`	Companion matrix
`gallery`	Test matrices
`hadamard`	Hadamard matrix
`hankel`	Hankel matrix
`hilb`	Hilbert matrix
`invhilb`	Inverse of the Hilbert matrix
`magic`	Magic square
`pascal`	Pascal matrix
`toeplitz`	Toeplitz matrix
`wilkinson`	Wilkinson's eigenvalue test matrix

Elementary Math Functions

`abs`	Absolute value and complex magnitude
`acos, acosh`	Inverse cosine and inverse hyperbolic cosine
`acot, acoth`	Inverse cotangent and inverse hyperbolic cotangent
`acsc, acsch`	Inverse cosecant and inverse hyperbolic cosecant
`angle`	Phase angle
`asec, asech`	Inverse secant and inverse hyperbolic secant
`asin, asinh`	Inverse sine and inverse hyperbolic sine
`atan, atanh`	Inverse tangent and inverse hyperbolic tangent
`atan2`	Four-quadrant inverse tangent
`ceil`	Round toward infinity
`complex`	Construct complex data from real and imaginary components
`conj`	Complex conjugate
`cos, cosh`	Cosine and hyperbolic cosine
`cot, coth`	Cotangent and hyperbolic cotangent
`csc, csch`	Cosecant and hyperbolic cosecant
`exp`	Exponential
`fix`	Round towards zero
`floor`	Round towards minus infinity
`gcd`	Greatest common divisor

`imag`	Imaginary part of a complex number
`lcm`	Least common multiple
`log`	Natural logarithm
`log2`	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
`log10`	Common (base 10) logarithm
`mod`	Modulus (signed remainder after division)
`nchoosek`	Binomial coefficient or all combinations
`real`	Real part of complex number
`rem`	Remainder after division
`round`	Round to nearest integer
`sec, sech`	Secant and hyperbolic secant
`sign`	Signum function
`sin, sinh`	Sine and hyperbolic sine
`sqrt`	Square root
`tan, tanh`	Tangent and hyperbolic tangent

Specialized Math Functions

`airy`	Airy functions
`besselh`	Bessel functions of the third kind (Hankel functions)
`besseli`, `besselk`	Modified Bessel functions
`besselj`, `bessely`	Bessel functions
`beta`, `betainc`, `betaln`	Beta functions
`ellipj`	Jacobi elliptic functions
`ellipke`	Complete elliptic integrals of the first and second kind
`erf`, `erfc`, `erfcx`, `erfinv`	Error functions
`expint`	Exponential integral
`factorial`	Factorial function
`gamma`, `gammainc`, `gammaln`	Gamma functions
`legendre`	Associated Legendre functions
`pow2`	Base 2 power and scale floating-point numbers
`rat, rats`	Rational fraction approximation

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

Matrix Functions - Numerical Linear Algebra

Matrix Analysis

`cond`	Condition number with respect to inversion
`condeig`	Condition number with respect to eigenvalues
`det`	Matrix determinant
`norm`	Vector and matrix norms
`null`	Null space of a matrix
`orth`	Range space of a matrix
`rank`	Rank of a matrix7
`rcond`	Matrix reciprocal condition number estimate
`rref`, `rrefmovie`	Reduced row echelon form
`subspace`	Angle between two subspaces
`trace`	Sum of diagonal elements

Linear Equations

`chol`	Cholesky factorization
`inv`	Matrix inverse
`lscov`	Least squares solution in the presence of known covariance
`lu`	LU matrix factorization
`lsqnonneg`	Nonnegative least squares
`minres`	Minimum Residual Method
`pinv`	Moore-Penrose pseudoinverse of a matrix
`qr`	Orthogonal-triangular decomposition
`symmlq`	Symmetric LQ method

Eigenvalues and Singular Values

`balance`	Improve accuracy of computed eigenvalues
`cdf2rdf`	Convert complex diagonal form to real block diagonal form
`eig`	Eigenvalues and eigenvectors
`gsvd`	Generalized singular value decomposition

hess	Hessenberg form of a matrix
poly	Polynomial with specified roots
qz	QZ factorization for generalized eigenvalues
rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
svd	Singular value decomposition

Matrix Functions

expm	Matrix exponential
funm	Evaluate general matrix function
logm	Matrix logarithm
sqrtm	Matrix square root

Low Level Functions

qrdelete	Delete column from QR factorization
qrinsert	Insert column in QR factorization

Data Analysis and Fourier Transform Functions

Basic Operations

cumprod	Cumulative product
cumsum	Cumulative sum
cumtrapz	Cumulative trapezoidal numerical integration
factor	Prime factors
inpolygon	Detect points inside a polygonal region
max	Maximum elements of an array
mean	Average or mean value of arrays
median	Median value of arrays
min	Minimum elements of an array
perms	All possible permutations
polyarea	Area of polygon
primes	Generate list of prime numbers
prod	Product of array elements
rectint	Rectangle intersection Area
sort	Sort elements in ascending order
sortrows	Sort rows in ascending order
std	Standard deviation
sum	Sum of array elements
trapz	Trapezoidal numerical integration

`var`	Variance

Finite Differences

`del2`	Discrete Laplacian
`diff`	Differences and approximate derivatives
`gradient`	Numerical gradient

Correlation

`corrcoef`	Correlation coefficients
`cov`	Covariance matrix

Filtering and Convolution

`conv`	Convolution and polynomial multiplication
`conv2`	Two-dimensional convolution
`deconv`	Deconvolution and polynomial division
`filter`	Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter
`filter2`	Two-dimensional digital filtering

Fourier Transforms

`abs`	Absolute value and complex magnitude
`angle`	Phase angle
`cplxpair`	Sort complex numbers into complex conjugate pairs
`fft`	One-dimensional fast Fourier transform
`fft2`	Two-dimensional fast Fourier transform
`fftshift`	Shift DC component of fast Fourier transform to center of spectrum
`ifft`	Inverse one-dimensional fast Fourier transform
`ifft2`	Inverse two-dimensional fast Fourier transform
`ifftn`	Inverse multidimensional fast Fourier transform
`ifftshift`	Inverse FFT shift
`nextpow2`	Next power of two
`unwrap`	Correct phase angles

Polynomial and Interpolation Functions

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	Analytic polynomial integration
polyval	Polynomial evaluation
polyvalm	Matrix polynomial evaluation
residue	Convert between partial fraction expansion and polynomial coefficients
roots	Polynomial roots

Data Interpolation

convhull	Convex hull
convhulln	Multidimensional convex hull
delaunay	Delaunay triangulation
delaunay3	Three-dimensionalDelaunay tessellation
delaunayn	Multidimensional Delaunay tessellation
dsearch	Search for nearest point
dsearchn	Multidimensional closest point search
griddata	Data gridding
griddata3	Data gridding and hypersurface fitting for three-dimensional data
griddatan	Data gridding and hypersurface fitting (dimension >= 2)
interp1	One-dimensional data interpolation (table lookup)
interp2	Two-dimensional data interpolation (table lookup)
interp3	Three-dimensional data interpolation (table lookup)
interpft	One-dimensional interpolation using the FFT method
interpn	Multidimensional data interpolation (table lookup)
meshgrid	Generate X and Y matrices for three-dimensional plots
ndgrid	Generate arrays for multidimensional functions and interpolation
pchip	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)
ppval	Piecewise polynomial evaluation
spline	Cubic spline data interpolation
tsearch	Search for enclosing Delaunay triangle
tsearchn	Multidimensional closest simplex search
voronoi	Voronoi diagram
voronoin	Multidimensional Voronoi diagrams

Function Functions – Nonlinear Numerical Methods

bvp4c	Solve two-point boundry value problems (BVPs) for

	ordinary differential equations (ODEs)
`bvpget`	Extract parameters from BVP options structure
`bvpinit`	Form the initial guess for `bvp4c`
`bvpset`	Create/alter BVP options structure
`bvpval`	Evaluate the solution computed by `bvp4c`
`dblquad`	Numerical evaluation of double integrals
`fminbnd`	Minimize a function of one variable
`fminsearch`	Minimize a function of several variables
`fzero`	Find zero of a function of one variable
`ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`	
	Solve initial value problems for ODEs
`odeget`	Extract parameters from ODE options structure
`odeset`	Create/alter ODE options structure
`optimget`	Get optimization options structure parameter values
`optimset`	Create or edit optimization options parameter structure
`pdepe`	Solve initial-boundary value problems
`pdeval`	Evaluate the solution computed by pdepe
`quad`	Numerical evaluation of integrals, adaptive Simpson quadrature
`quadl`	Numerical evaluation of integrals, adaptive Lobatto quadrature
`vectorize`	Vectorize expression

Sparse Matrix Functions

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

Working with Nonzero Entries of Sparse Matrices

`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

Visualizing Sparse Matrices

`spy`	Visualize sparsity pattern

Reordering Algorithms

`colamd`	Column approximate minimum degree permutation
`colmmd`	Sparse column minimum degree permutation
`colperm`	Sparse column permutation based on nonzero count
`dmperm`	Dulmage-Mendelsohn decomposition
`randperm`	Random permutation
`symamd`	Symmetric approximate minimum degree permutation
`symmmd`	Sparse symmetric minimum degree ordering
`symrcm`	Sparse reverse Cuthill-McKee ordering

Norm, Condition Number, and Rank

`condest`	1-norm matrix condition number estimate
`normest`	2-norm estimate

Sparse Systems of Linear Equations

`bicg`	BiConjugate Gradients method
`bicgstab`	BiConjugate Gradients Stabilized method
`cgs`	Conjugate Gradients Squared method
`cholinc`	Sparse Incomplete Cholesky and Cholesky-Infinity factorizations
`cholupdate`	Rank 1 update to Cholesky factorization
`gmres`	Generalized Minimum Residual method (with restarts)
`lsqr`	LSQR implementation of Conjugate Gradients on the normal equations
`luinc`	Incomplete LU matrix factorizations
`pcg`	Preconditioned Conjugate Gradients method
`qmr`	Quasi-Minimal Residual method
`qr`	Orthogonal-triangular decomposition
`qrdelete`	Delete column from QR factorization
`qrinsert`	Insert column in QR factorization
`qrupdate`	Rank 1 update to QR factorization

Sparse Eigenvalues and Singular Values

`eigs`	Find eigenvalues and eigenvectors
`svds`	Find singular values

Miscellaneous

`spparms`	Set parameters for sparse matrix routines

Sound Processing Functions

General Sound Functions

`lin2mu`	Convert linear audio signal to mu-law
`mu2lin`	Convert mu-law audio signal to linear
`sound`	Convert vector into sound
`soundsc`	Scale data and play as sound

SPARCstation-Specific Sound Functions

`auread`	Read NeXT/SUN (.au) sound file
`auwrite`	Write NeXT/SUN (.au) sound file

.WAV Sound Functions

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

Character String Functions

General

`abs`	Absolute value and complex magnitude
`eval`	Interpret strings containing MATLAB expressions
`real`	Real part of complex number
`strings`	MATLAB string handling

String to Function Handle Conversion

`func2str`	Constructs a function name string from a function handle

`str2func`	Constructs a function handle from a function name string

String Manipulation

`deblank`	Strip trailing blanks from the end of a string
`findstr`	Find one string within another
`lower`	Convert string to lower case
`strcat`	String concatenation
`strcmp`	Compare strings
`strcmpi`	Compare strings, ignoring case
`strjust`	Justify a character array
`strmatch`	Find possible matches for a string
`strncmp`	Compare the first `n` characters of strings
`strncmpi`	Compare the first `n` characters of strings, ignoring case
`strrep`	String search and replace
`strtok`	First token in string
`strvcat`	Vertical concatenation of strings
`symvar`	Determine symbolic variables in an expression
`texlabel`	Produce the TeX format from a character string
`upper`	Convert string to upper case

String to Number Conversion

`char`	Create character array (string)
`int2str`	Integer to string conversion
`mat2str`	Convert a matrix into a string
`num2str`	Number to string conversion
`sprintf`	Write formatted data to a string
`sscanf`	Read string under format control
`str2double`	Convert string to double-precision value
`str2mat`	String to matrix conversion
`str2num`	String to number conversion

Radix Conversion

`bin2dec`	Binary to decimal number conversion
`dec2bin`	Decimal to binary number conversion
`dec2hex`	Decimal to hexadecimal number conversion
`hex2dec`	Hexadecimal to decimal number conversion
`hex2num`	Hexadecimal to double number conversion

File I/O Functions

File Opening and Closing

`fclose`	Close one or more open files
`fopen`	Open a file or obtain information about open files

Unformatted I/O

`fread`	Read binary data from file
`fwrite`	Write binary data to a file

Formatted I/O

`fgetl`	Return the next line of a file as a string without line terminator(s)
`fgets`	Return the next line of a file as a string with line terminator(s)
`fprintf`	Write formatted data to file
`fscanf`	Read formatted data from file

File Positioning

`feof`	Test for end-of-file
`ferror`	Query MATLAB about errors in file input or output
`frewind`	Rewind an open file
`fseek`	Set file position indicator
`ftell`	Get file position indicator

String Conversion

`sprintf`	Write formatted data to a string
`sscanf`	Read string under format control

Specialized File I/O

`dlmread`	Read an ASCII delimited file into a matrix
`dlmwrite`	Write a matrix to an ASCII delimited file
`hdf`	HDF interface
`imfinfo`	Return information about a graphics file
`imread`	Read image from graphics file
`imwrite`	Write an image to a graphics file
`strread`	Read formatted data from a string
`textread`	Read formatted data from text file
`wk1read`	Read a Lotus123 WK1 spreadsheet file into a matrix

`wk1write`	Write a matrix to a Lotus123 WK1 spreadsheet file

Bitwise Functions

`bitand`	Bit-wise AND
`bitcmp`	Complement bits
`bitor`	Bit-wise OR
`bitmax`	Maximum floating-point integer
`bitset`	Set bit
`bitshift`	Bit-wise shift
`bitget`	Get bit
`bitxor`	Bit-wise XOR

Structure Functions

`fieldnames`	Field names of a structure
`getfield`	Get field of structure array
`rmfield`	Remove structure fields
`setfield`	Set field of structure array
`struct`	Create structure array
`struct2cell`	Structure to cell array conversion

MATLAB Object Functions

`class`	Create object or return class of object
`isa`	Detect an object of a given class
`methods`	Display method names
`methodsview`	Displays information on all methods implemented by a class
`subsasgn`	Overloaded method for A(I)=B, A{I}=B, and A.field=B
`subsindex`	Overloaded method for X(A)
`subsref`	Overloaded method for A(I), A{I} and A.field

MATLAB Interface to Java

`class`	Create object or return class of object
`import`	Add a package or class to the current Java import list
`isa`	Detect an object of a given class
`isjava`	Test whether an object is a Java object
`javaArray`	Constructs a Java array

`javaMethod`	Invokes a Java method
`javaObject`	Constructs a Java object
`methods`	Display method names
`methodsview`	Displays information on all methods implemented by a class

Cell Array Functions

`cell`	Create cell array
`cellfun`	Apply a function to each element in a cell array
`cellstr`	Create cell array of strings from character array
`cell2struct`	Cell array to structure array conversion
`celldisp`	Display cell array contents
`cellplot`	Graphically display the structure of cell arrays
`num2cell`	Convert a numeric array into a cell array

Multidimensional Array Functions

`cat`	Concatenate arrays
`flipdim`	Flip array along a specified dimension
`ind2sub`	Subscripts from linear index
`ipermute`	Inverse permute the dimensions of a multidimensional array
`ndgrid`	Generate arrays for multidimensional functions and interpolation
`ndims`	Number of array dimensions
`permute`	Rearrange the dimensions of a multidimensional array
`reshape`	Reshape array
`shiftdim`	Shift dimensions
`squeeze`	Remove singleton dimensions
`sub2ind`	Single index from subscripts

Plotting and Data Visualization

Basic Plots and Graphs

`bar`	Vertical bar chart
`barh`	Horizontal bar chart
`hist`	Plot histograms
`histc`	Histogram count
`hold`	Hold current graph
`loglog`	Plot using log-log scales
`pie`	Pie plot

`plot`	Plot vectors or matrices.
`polar`	Polar coordinate plot
`semilogx`	Semi-log scale plot
`semilogy`	Semi-log scale plot
`subplot`	Create axes in tiled positions

Three-Dimensional Plotting

`bar3`	Vertical 3-D bar chart
`bar3h`	Horizontal 3-D bar chart
`comet3`	3-D comet plot
`cylinder`	Generate cylinder
`fill3`	Draw filled 3-D polygons in 3-space
`plot3`	Plot lines and points in 3-D space
`quiver3`	3-D quiver (or velocity) plot
`slice`	Volumetric slice plot
`sphere`	Generate sphere
`stem3`	Plot discrete surface data
`waterfall`	Waterfall plot

Plot Annotation and Grids

`clabel`	Add contour labels to a contour plot
`datetick`	Date formatted tick labels
`grid`	Grid lines for 2-D and 3-D plots
`gtext`	Place text on a 2-D graph using a mouse
`legend`	Graph legend for lines and patches
`plotyy`	Plot graphs with Y tick labels on the left and right
`title`	Titles for 2-D and 3-D plots
`xlabel`	X-axis labels for 2-D and 3-D plots
`ylabel`	Y-axis labels for 2-D and 3-D plots
`zlabel`	Z-axis labels for 3-D plots

Surface, Mesh, and Contour Plots

`contour`	Contour (level curves) plot
`contourc`	Contour computation
`contourf`	Filled contour plot
`hidden`	Mesh hidden line removal mode
`meshc`	Combination mesh/contourplot
`mesh`	3-D mesh with reference plane
`peaks`	A sample function of two variables
`surf`	3-D shaded surface graph

`surface`	Create surface low-level objects
`surfc`	Combination surf/contourplot
`surfl`	3-D shaded surface with lighting
`trimesh`	Triangular mesh plot
`trisurf`	Triangular surface plot

Volume Visualization

`coneplot`	Plot velocity vectors as cones in 3-D vector field
`contourslice`	Draw contours in volume slice plane
`curl`	Compute the curl and angular velocity of a vector field
`divergence`	Compute the divergence of a vector field
`flow`	Generate scalar volume data
`interpstreamspeed`	Interpolate streamline vertices from vector-field magnitudes
`isocaps`	Compute isosurface end-cap geometry
`isocolors`	Compute the colors of isosurface vertices
`isonormals`	Compute normals of isosurface vertices
`isosurface`	Extract isosurface data from volume data
`reducepatch`	Reduce the number of patch faces
`reducevolume`	Reduce number of elements in volume data set
`shrinkfaces`	Reduce the size of patch faces
`slice`	Draw slice planes in volume
`smooth3`	Smooth 3-D data
`stream2`	Compute 2-D stream line data
`stream3`	Compute 3-D stream line data
`streamline`	Draw stream lines from 2- or 3-D vector data
`streamparticles`	Draws stream particles from vector volume data
`streamribbon`	Draws stream ribbons from vector volume data
`streamslice`	Draws well-spaced stream lines from vector volume data
`streamtube`	Draws stream tubes from vector volume data
`surf2patch`	Convert srface data to patch data
`subvolume`	Extract subset of volume data set
`volumebounds`	Return coordinate and color limits for volume (scalar and vector)

Domain Generation

`griddata`	Data gridding and surface fitting
`meshgrid`	Generation of X and Y arrays for 3-D plots

Specialized Plotting

`area`	Area plot
`box`	Axis box for 2-D and 3-D plots

`comet`	Comet plot
`compass`	Compass plot
`errorbar`	Plot graph with error bars
`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
`feather`	Feather plot
`fill`	Draw filled 2-D polygons
`fplot`	Plot a function
`pareto`	Pareto char
`pie3`	3-D pie plot
`plotmatrix`	Scatter plot matrix
`pcolor`	Pseudocolor (checkerboard) plot
`rose`	Plot rose or angle histogram
`quiver`	Quiver (or velocity) plot
`ribbon`	Ribbon plot
`stairs`	Stairstep graph
`scatter`	Scatter plot
`scatter3`	3-D scatter plot
`stem`	Plot discrete sequence data
`convhull`	Convex hull
`delaunay`	Delaunay triangulation
`dsearch`	Search Delaunay triangulation for nearest point
`inpolygon`	True for points inside a polygonal region
`polyarea`	Area of polygon
`tsearch`	Search for enclosing Delaunay triangle
`voronoi`	Voronoi diagram

View Control

`camdolly`	Move camera position and target
`camlookat`	View specific objects
`camorbit`	Orbit about camera target
`campan`	Rotate camera target about camera position
`campos`	Set or get camera position
`camproj`	Set or get projection type
`camroll`	Rotate camera about viewing axis

`camtarget`	Set or get camera target
`camup`	Set or get camera up-vector
`camva`	Set or get camera view angle
`camzoom`	Zoom camera in or out
`daspect`	Set or get data aspect ratio
`pbaspect`	Set or get plot box aspect ratio
`view`	3-D graph viewpoint specification.
`viewmtx`	Generate view transformation matrices
`xlim`	Set or get the current *x*-axis limits
`ylim`	Set or get the current *y*-axis limits
`zlim`	Set or get the current *z*-axis limits

Lighting

`camlight`	Cerate or position Light
`light`	Light object creation function
`lighting`	Lighting mode
`lightangle`	Position light in sphereical coordinates
`material`	Material reflectance mode

Transparency

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

Color Operations

`brighten`	Brighten or darken color map
`caxis`	Pseudocolor axis scaling
`colorbar`	Display color bar (color scale)
`colordef`	Set up color defaults
`colormap`	Set the color look-up table (list of colormaps)
`graymon`	Graphics figure defaults set for grayscale monitor
`hsv2rgb`	Hue-saturation-value to red-green-blue conversion
`rgb2hsv`	RGB to HSVconversion
`rgbplot`	Plot color map
`shading`	Color shading mode
`spinmap`	Spin the colormap
`surfnorm`	3-D surface normals
`whitebg`	Change axes background color for plots

Colormaps

`autumn`	Shades of red and yellow color map
`bone`	Gray-scale with a tinge of blue color map
`contrast`	Gray color map to enhance image contrast
`cool`	Shades of cyan and magenta color map
`copper`	Linear copper-tone color map
`flag`	Alternating red, white, blue, and black color map
`gray`	Linear gray-scale color map
`hot`	Black-red-yellow-white color map
`hsv`	Hue-saturation-value (HSV) color map
`jet`	Variant of HSV
`lines`	Line color colormap
`prism`	Colormap of prism colors
`spring`	Shades of magenta and yellow color map
`summer`	Shades of green and yellow colormap
`winter`	Shades of blue and green color map

Printing

`orient`	Hardcopy paper orientation
`pagesetupdlg`	Page position dialog box
`print`	Print graph or save graph to file
`printdlg`	Print dialog box
`printopt`	Configure local printer defaults
`saveas`	Save figure to graphic file

Handle Graphics, General

`allchild`	Find all children of specified objects
`copyobj`	Make a copy of a graphics object and its children
`findall`	Find all graphics objects (including hidden handles)
`findobj`	Find objects with specified property values
`gcbo`	Return object whose callback is currently executing
`gco`	Return handle of current object
`get`	Get object properties
`rotate`	Rotate objects about specified origin and direction
`ishandle`	True for graphics objects
`set`	Set object properties

Working with Application Data

`getappdata`	Get value of application data
`isappdata`	True if application data exists

`rmappdata`	Remove application data
`setappdata`	Specify application data

Handle Graphics, Object Creation

`axes`	Create Axes object
`figure`	Create Figure (graph) windows
`image`	Create Image (2-D matrix)
`light`	Create Light object (illuminates Patch and Surface)
`line`	Create Line object (3-D polylines)
`patch`	Create Patch object (polygons)
`rectangle`	Create Rectangle object (2-D rectangle)
`surface`	Create Surface (quadrilaterals)
`text`	Create Text object (character strings)
`uicontextmenu`	Create context menu (popup associated with object)

Handle Graphics, Figure Windows

`capture`	Screen capture of the current figure
`clc`	Clear figure window
`clf`	Clear figure
`close`	Close specified window
`closereq`	Default close request function
`gcf`	Get current figure handle
`newplot`	Graphics M-file preamble for `NextPlot` property
`refresh`	Refresh figure
`saveas`	Save figure or model to desired output format

Handle Graphics, Axes

`axis`	Plot axis scaling and appearance
`cla`	Clear Axes
`gca`	Get current Axes handle

Object Manipulation

`reset`	Reset axis or figure
`rotate3d`	Interactively rotate the view of a 3-D plot
`selectmoveresize`	Interactively select, move, or resize objects

Interactive User Input

`ginput`	Graphical input from a mouse or cursor

`zoom`	Zoom in and out on a 2-D plot

Region of Interest

`dragrect`	Drag XOR rectangles with mouse
`drawnow`	Complete any pending drawing
`rbbox`	Rubberband box

Graphical User Interfaces

Dialog Boxes

`dialog`	Create a dialog box
`errordlg`	Create error dialog box
`helpdlg`	Display help dialog box
`inputdlg`	Create input dialog box
`listdlg`	Create list selection dialog box
`msgbox`	Create message dialog box
`pagedlg`	Display page layout dialog box
`printdlg`	Display print dialog box
`questdlg`	Create question dialog box
`uigetfile`	Display dialog box to retrieve name of file for reading
`uiputfile`	Display dialog box to retrieve name of file for writing
`uisetcolor`	Interactively set a `ColorSpec` using a dialog box
`uisetfont`	Interactively set a font using a dialog box
`warndlg`	Create warning dialog box

User Interface Deployment

`guidata`	Store or retrieve application data
`guihandles`	Create a structure of handles
`movegui`	Move GUI figure onscreen
`openfig`	Open or raise GUI figure

User Interface Development

`guide`	Open the GUI Layout Editor
`inspect`	Display Property Inspector

User Interface Objects

`menu`	Generate a menu of choices for user input

`uicontextmenu`	Create context menu
`uicontrol`	Create user interface control
`uimenu`	Create user interface menu

Other Functions

`dragrect`	Drag rectangles with mouse
`findfigs`	Display off-screen visible figure windows
`gcbf`	Return handle of figure containing callback object
`gcbo`	Return handle of object whose callback is executing
`rbbox`	Create rubberband box for area selection
`selectmoveresize`	Select, move, resize, or copy Axes and Uicontrol graphics objects
`textwrap`	Return wrapped string matrix for given Uicontrol
`uiresume`	Used with `uiwait`, controls program execution
`uiwait`	Used with `uiresume`, controls program execution
`waitbar`	Display wait bar
`waitforbuttonpress`	Wait for key/buttonpress over figure

Serial Port I/O

Creating a Serial Port Object

`serial`	Create a serial port object

Writing and Reading Data

`fgetl`	Read one line of text from the device and discard the terminator
`fgets`	Read one line of text from the device and include the terminator
`fprintf`	Write text to the device
`fread`	Read binary data from the device
`fscanf`	Read data from the device, and format as text
`fwrite`	Write binary data to the device
`readasync`	Read data asynchronously from the device
`stopasync`	Stop asynchronous read and write operations

Configuring and Returning Properties

`get`	Return serial port object properties
`set`	Configure or display serial port object properties

State Change

`fclose`	Disconnect a serial port object from the device
`fopen`	Connect a serial port object to the device
`record`	Record data and event information to a file

General Purpose

`clear`	Remove a serial port object from the MATLAB workspace
`delete`	Remove a serial port object from memory
`disp`	Display serial port object summary information
`instraction`	Display event information when an event occurs
`instrfind`	Return serial port objects from memory to the MATLAB workspace
`isvalid`	Determine if serial port objects are valid
`length`	Length of serial port object array
`load`	Load serial port objects and variables into the MATLAB workspace
`save`	Save serial port objects and variables to a MAT-file
`serialbreak`	Send a break to the device connected to the serial port
`size`	Size of serial port object array

Volume 2 Reference

This volume describes the MATLAB operators, special characters, commands, and functions listed alphabetically from F through O.

Please note that in the three volumes of the *MATLAB Function Reference*, operators and special characters are listed alphabetically according to these categories:

- Arithmetic Operators
- Colon
- Logical Operators
- Special Characters
- Relational Operators

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)` is the product of all the integers from `1` to `n`, i.e. `prod(1:n)`. Since double pricision numbers only have about 15 digits, the answer is only accurate for `n <= 21`. For larger `n`, the answer will have the right magnitute, and is accurate for the first 15 digits.

See Also `prod`

Purpose

Close one or more open files

Syntax

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

Description

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

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

See Also

`ferror`, `fopen`, `fprintf`, `fread`, `frewind`, `fscanf`, `fseek`, `ftell`, `fwrite`

fclose (serial)

Purpose Disconnect a serial port object from the device

Syntax

```
fclose(obj)
```

Arguments

`obj` A serial port object or an array of serial port objects.

Description

`fclose(obj)` disconnects `obj` from the device.

Remarks

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

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

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

```
help serial/fclose
```

Example

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

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

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

See Also

Functions

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

Properties

`RecordStatus`, `Status`

Purpose Plot velocity vectors

Syntax

```
feather(U,V)
feather(Z)
feather(...,LineSpec)
```

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.

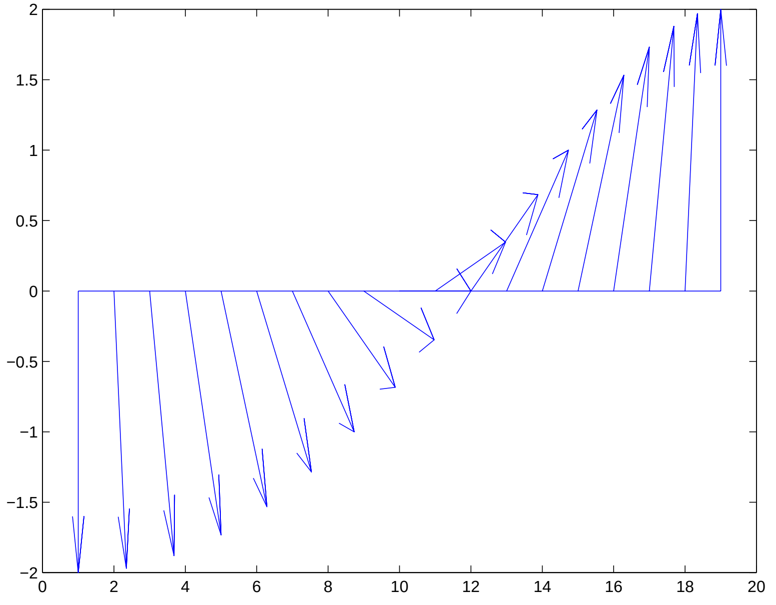
`feather(U,V)` displays the vectors specified by `U` and `V`, where `U` contains the *x* components as relative coordinates, and `V` contains the *y* components as relative coordinates.

`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`.

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`

Purpose Test for end-of-file

Syntax `eofstat = feof(fid)`

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

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

See Also `fopen`

ferror

Purpose Query MATLAB about errors in file input or output

Syntax

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

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

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

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

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

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

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

Purpose Function evaluation

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.

`[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.

Note The preferred means of evaluating a function by reference is to use a function handle. To support backward compatibility, `feval` also accepts a function name string as a first argument. However, function handles offer the additional performance, reliability, and source file control benefits listed in the section "An Overview of Function Handles".

Remarks

The following two statements are equivalent.

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

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{:});
```

In the next example, `@deblank` returns a function handle to variable, `fhandle`. Examining the handle using `functions(fhandle)` reveals that it is bound to two M-files that implement the `deblank` function. The default, `strfun\deblank.m`, handles most argument types. However, the function is overloaded by a second M-file (in the `@cell` subdirectory) to handle cell array arguments as well.

```
fhandle = @deblank;

ff = functions(fhandle);
ff.default
ans =
    matlabroot\toolbox\matlab\strfun\deblank.m
ff.methods
ans =
    cell: 'matlabroot\toolbox\matlab\strfun\@cell\deblank.m'
```

When the function handle is evaluated on a cell array, `feval` determines from the argument type that the appropriate function to dispatch to is the one that resides in `strfun\@cell`.

```
feval(fhandle, {'string   ','with  ','blanks        '})
ans =
    'string'    'with'    'blanks'
```

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

Purpose One-dimensional fast Fourier transform

Syntax

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

Definition The functions `X = fft(x)` and `x = 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.

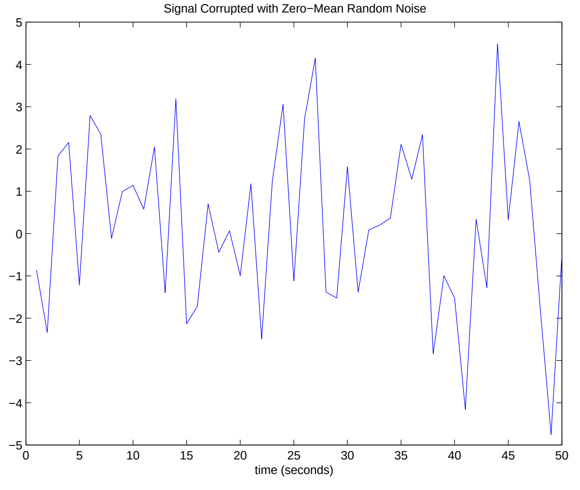
`Y = fft(X,n)` returns the `n`-point DFT. If the length of `X` is less than `n`, `X` is padded with trailing zeros to length `n`. If the length of `X` is greater than `n`, the sequence `X` is truncated. When `X` is a matrix, the length of the columns are adjusted in the same manner.

`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 50 Hz and 120 Hz and corrupt it with some zero-mean random noise:

```
t = 0:0.001:0.6;
x = sin(2*pi*50*t)+sin(2*pi*120*t);
y = x + 2*randn(size(t));
plot(y(1:50))
title('Signal Corrupted with Zero-Mean Random Noise')
xlabel('time (seconds)')
```



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 512-point fast Fourier transform (FFT):

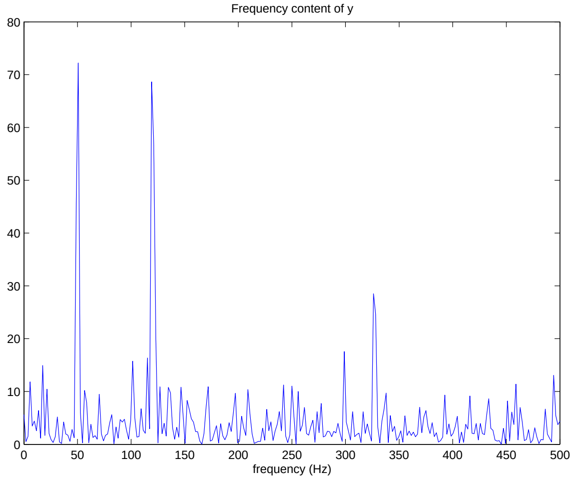
```
Y = fft(y,512);
```

The power spectrum, a measurement of the power at various frequencies, is

```
Pyy = Y.* conj(Y) / 512;
```

Graph the first 257 points (the other 255 points are redundant) on a meaningful frequency axis.

```
f = 1000*(0:256)/512;
plot(f,Pyy(1:257))
title('Frequency content of y')
xlabel('frequency (Hz)')
```



This represents the frequency content of `y` in the range from DC up to and including the Nyquist frequency. (The signal produces the strong peaks.)

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.

See Also

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

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

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 Two-dimensional fast 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.

See Also `fft`, `fftn`, `fftshift`, `ifft2`

fftn

Purpose

Multidimensional fast 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.

See Also

`fft`, `fft2`, `fftn`, `ifftn`

Purpose

Shift zero-frequency component of fast Fourier transform 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 quadrants one and three of `X` with quadrants two and four. 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`.

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.

See Also

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

fgetl

Purpose Read line from file, discard newline character

Syntax

```
tline = fgetl(fid)
```

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

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

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

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

See Also `fgets`

Purpose Read one line of text from the device and discard the terminator

Syntax

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

Arguments

`obj`	A serial port object.
`tline`	Text read from the instrument, excluding the terminator.
`count`	The number of values read, including the terminator.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`tline = fgetl(obj)` reads one line of text from the device connected to `obj`, and returns the data to `tline`. The 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`.

`[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

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?')
```

Since 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, keep newline character

Syntax

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

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

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

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

See Also `fgetl`

fgets (serial)

Purpose

Read one line of text from the device and include the terminator

Syntax

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

Arguments

`obj`	A serial port object.
`tline`	Text read from the instrument, including the terminator.
`count`	The number of bytes read, including the terminator.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`tline = fgets(obj)` reads one line of text from the device connected to `obj`, and returns the data to `tline`. The 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`.

`[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?')
```

Since 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, BytesAvailableAction, InputBufferSize, Status, Terminator, Timeout, ValuesReceived

fieldnames

Purpose Return field names of a structure, or property names of a MATLAB object or Java 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 the names of the public data fields associated with `obj`, which is either a MATLAB or a Java object.

`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 either a MATLAB or a Java object.

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 `x` is an object of Java class `java.awt.Frame`, the command `fieldnames(x)` results in the display

```
ans =
'width'
'height'
```

See Also `getfield` `setfield` `rmfield`

Purpose

Test if figure is on screen

Syntax

```
[flag] = figflag('figurename')
[flag,fig] = figflag('figurename')
[...] = figflag('figurename',silent)
```

Description

Use `figflag` to determine if a particular figure exists, bring a figure to the foreground, or set the window focus to a figure.

`[flag] = figflag('`*`figurename`*`')` returns a `1` if the figure named `'`*`figurename`*`'` exists and pops the figure to the foreground; otherwise this function returns `0`.

`[flag,fig] = figflag('`*`figurename`*`')` returns a `1` in `flag`, returns the figure's handle in `fig`, and pops the figure to the foreground, if the figure named `'`*`figurename`*`'` exists. Otherwise this function returns `0`.

`[...] = figflag('`*`figurename`*`',silent)` pops the figure window to the foreground if `silent` is `0`, and leaves the figure in its current position if `silent` is `1`.

Examples

To determine if a figure window named `'Fluid Jet Simulation'` exists, type

```
[flag,fig] = figflag('Fluid Jet Simulation')
```

MATLAB returns:

```
flag =
   1
fig =
   1
```

If two figures with handles 1 and 3 have the name `'Fluid Jet Simulation'`, MATLAB returns:

```
flag =
   1
fig =
   1 3
```

See Also

`figure`

figure

Purpose Create a 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 MATLAB displays graphical output.

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

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

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

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

Remarks To create a figure object, MATLAB creates a new window whose characteristics are controlled by default figure properties (both factory installed and user defined) and properties specified as arguments. See the 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.

Example

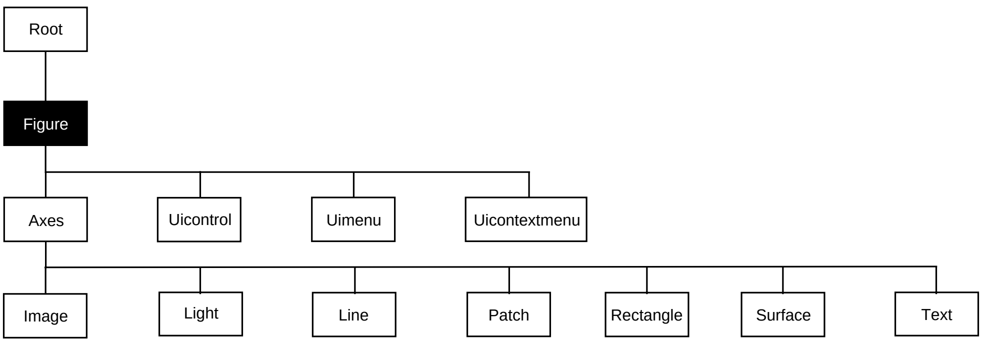
To create a figure window that is one quarter the size of your screen and is positioned in the upper-left corner, use the root object's `ScreenSize` property to determine the size. `ScreenSize` is a four-element vector: [`left`, `bottom`, `width`, `height`]:

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

See Also

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

Object Hierarchy



Setting Default Properties

You can set default figure properties only on the root 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.

Property List

The following table lists all figure properties and provides a brief description of each. The property name links bring you an expanded description of the properties.

Property Name	Property Description	Property Value
Positioning the Figure		
`Position`	Location and size of figure	Value: a 4-element vector [left, bottom, width, height] Default: depends on display
`Units`	Units used to interpret the `Position` property	Values: `inches`, `centimeters`, `normalized`, `points`, `pixels`, `characters` Default: `pixels`
Specifying Style and Appearance		
`Color`	Color of the figure background	Values: `ColorSpec` Default: depends on color scheme (see `colordef`)
`MenuBar`	Toggle the figure menu bar on and off	Values: `none`, `figure` Default: `figure`
`Name`	Figure window title	Values: string Default: `''` (empty string)
`NumberTitle`	Display “Figure No. n”, where n is the figure number	Values: `on`, `off` Default: `on`
`Resize`	Specify whether the figure window can be resized using the mouse	Values: `on`, `off` Default: `on`
`SelectionHighlight`	Highlight figure when selected (`Selected` property set to `on`)	Values: `on`, `off` Default: `on`
`Visible`	Make the figure visible or invisible	Values: `on`, `off` Default: `on`

Property Name	Property Description	Property Value
`WindowStyle`	Select normal or modal window	Values: `normal`, `modal` Default: `normal`
Controlling the Colormap		
`Colormap`	The figure colormap	Values: m-by-3 matrix of RGB values Default: the `jet` colormap
`Dithermap`	Colormap used for truecolor data on pseudocolor displays	Values: m-by-3 matrix of RGB values Default: colormap with full range of colors
`DithermapMode`	Enable MATLAB-generated dithermap	Values: `auto`, `manual` Default: `manual`
`FixedColors`	Colors not obtained from colormap	Values: m-by-3 matrix of RGB values (read only)
`MinColormap`	Minimum number of system color table entries to use	Values: scalar Default: 64
`ShareColors`	Allow MATLAB to share system color table slots	Values `on`, `off` Default: `on`
Specifying Transparency		
`Alphamap`	The figure alphamap	m-by-1 matrix of alpha values
Specifying the Renderer		
`BackingStore`	Enable off screen pixel buffering	Values: `on`, `off` Default: `on`
`DoubleBuffer`	Flash-free rendering for simple animations	Values: `on`, `off` Default: `off`

Property Name	Property Description	Property Value
Renderer	Rendering method used for screen and printing	Values: painters, zbuffer, OpenGL Default: automatic selection by MATLAB
General Information About the Figure		
Children	Handle of any uicontrol, uimenu, and uicontextmenu objects displayed in the figure	Values: vector of handles
FileName	Used by guide	String
Parent	The root object is the parent of all figures	Value: always 0
Selected	Indicate whether figure is in a “selected” state.	Values: on, off Default: on
Tag	User-specified label	Value: any string Default: '' (empty string)
Type	The type of graphics object (read only)	Value: the string 'figure'
UserData	User-specified data	Values: any matrix Default: [] (empty matrix)
RendererMode	Automatic or user-selected renderer	Values: auto, manual Default: auto
Information About Current State		
CurrentAxes	Handle of the current axes in this figure	Values: axes handle
CurrentCharacter	The last key pressed in this figure	Values: single character
CurrentObject	Handle of the current object in this figure	Values: graphics object handle

Property Name	Property Description	Property Value
CurrentPoint	Location of the last button click in this figure	Values: 2-element vector [x-coord, y-coord]
SelectionType	Mouse selection type	Values: normal, extended, alt, open
Callback Routine Execution		
BusyAction	Specify how to handle callback routine interruption	Values: cancel, queue Default: queue
ButtonDownFcn	Define a callback routine that executes when a mouse button is pressed on an unoccupied spot in the figure	Values: string Default: empty string
CloseRequestFcn	Define a callback routine that executes when you call the close command	Values: string Default: closereq
CreateFcn	Define a callback routine that executes when a figure is created	Values: string Default: empty string
DeleteFcn	Define a callback routine that executes when the figure is deleted (via close or delete)	Values: string Default: empty string
Interruptible	Determine if callback routine can be interrupted	Values: on, off Default: on (can be interrupted)
KeyPressFcn	Define a callback routine that executes when a key is pressed in the figure window	Values: string Default: empty string
ResizeFcn	Define a callback routine that executes when the figure is resized	Values: string Default: empty string
UIContextMenu	Associate a context menu with the figure	Values: handle of a Uicontrextmenu

Property Name	Property Description	Property Value
`WindowButtonDownFcn`	Define a callback routine that executes when you press the mouse button down in the figure	Values: string Default: empty string
`WindowButtonMotionFcn`	Define a callback routine that executes when you move the pointer in the figure	Values: string Default: empty string
`WindowButtonUpFcn`	Define a callback routine that executes when you release the mouse button	Values: string Default: empty string
Controlling Access to Objects		
`IntegerHandle`	Specify integer or noninteger figure handle	Values: `on`, `off` Default: `on` (integer handle)
`HandleVisibility`	Determine if figure handle is visible to users or not	Values: `on`, `callback`, `off` Default: `on`
`HitTest`	Determine if the figure can become the current object (see the figure `CurrentObject` property)	Values: `on`, `off` Default: `on`
`NextPlot`	Determine how to display additional graphics to this figure	Values: `add`, `replace`, `replacechildren` Default: `add`
Defining the Pointer		
`Pointer`	Select the pointer symbol	Values: `crosshair`, `arrow`, `watch`, `topl`, `topr`, `botl`, `botr`, `circle`, `cross`, `fleur`, `left`, `right`, `top`, `bottom`, `fullcrosshair`, `ibeam`, `custom` Default: `arrow`

Property Name	Property Description	Property Value
PointerShapeCData	Data that defines the pointer	Values: 16-by-16 matrix Default: set Pointer to custom and see
PointerShapeHotSpot	Specify the pointer active spot	Values: 2-element vector [row, column] Default: [1,1]
Properties That Affect Printing		
InvertHardcopy	Change figure colors for printing	Values: on, off Default: on
PaperOrientation	Horizontal or vertical paper orientation	Values: portrait, landscape Default: portrait
PaperPosition	Control positioning figure on printed page	Values: 4-element vector [left, bottom, width, height]
PaperPositionMode	Enable WYSIWYG printing of figure	Values: auto, manual Default: manual
PaperSize	Size of the current PaperType specified in PaperUnits	Values: [width, height]
PaperType	Select from standard paper sizes	Values: see property description Default: usletter
PaperUnits	Units used to specify the PaperSize and PaperPosition	Values: normalized, inches, centimeters, points Default: inches
Controlling the XWindows Display (UNIX only)		

Property Name	Property Description	Property Value
`XDisplay`	Specify display for MATLAB (UNIX only)	Values: display identifier Default: `:0.0`
`XVisual`	Select visual used by MATLAB (UNIX only)	Values: visual ID
`XVisualMode`	Auto or manual selection of visual (UNIX only)	Values: `auto`, `manual` Default: `auto`

Modifying Properties

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

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

To change the default value of properties see Settingcreating_plots Default Property Values.

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.

`BackingStore` `{on} | off`

Off screen pixel buffer. When `BackingStore` is `on`, MATLAB stores a copy of the figure window in an off-screen pixel buffer. When obscured parts of the figure window are exposed, MATLAB copies the window contents from this buffer rather than regenerating the objects on the screen. This increases the speed with which the screen is redrawn.

While refreshing the screen quickly is generally desirable, the buffers required do consume system memory. If memory limitations occur, you can set `BackingStore` to `off` to disable this feature and release the memory used by the buffers. If your computer does not support backingstore, setting the `BackingStore` property results in a warning message, but has no other effect.

Setting `BackingStore` to `off` can increase the speed of animations because it eliminates the need to draw into both an off-screen buffer and the figure window.

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, subsequently invoked callback routines 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

Button press callback function. A callback routine 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, axes, or axes child). Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

Children vector of handles

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

Clipping {on} | off

This property has no effect on figures.

CloseRequestFcn string

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 quiting MATLAB.
- The close operation executes the function defined by the figure `CloseRequestFcn`. The default function is named `closereq` and is predefined as:

```
shh = get(0,'ShowHiddenHandles');
set(0,'ShowHiddenHandles','on');
currFig = get(0,'CurrentFigure');
set(0,'ShowHiddenHandles',shh);
delete(currFig);
```

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.

You can set `CloseRequestFcn` to any string that is a valid MATLAB statement, including the name of an M-file. For example,

```
set(gcf,'CloseRequestFcn','disp(''This window is immortal'')')
```

This close request function never closes the figure window; it simply echoes "This window is immortal" on the command line. Unless the close request function calls `delete`, 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 more useful application of the close request function is to display a question dialog box asking the user to confirm the close operation. The following M-file illustrates how to do this.

```
% my_closereq
% User-defined close request function
% to display a question dialog box
```

```
selection = questdlg('Close Specified Figure?',...
                     'Close Request Function',...
                     'Yes','No','Yes');
switch selection,
   case 'Yes',
     delete(gcf)
   case 'No'
     return
end
```

Now assign this M-file to the `CloseRequestFcn` of a figure:

```
set(figure_handle,'CloseRequestFcn','my_closereq')
```

To make this M-file 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.

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

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 string

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

```
set(0,'DefaultFigureCreateFcn',...
      'set(gcbo,''IntegerHandle'',''off'')')
```

defines a default value on the root level that causes the created figure to use noninteger handles whenever you (or MATLAB) create a figure. MATLAB executes this routine after setting all properties for the figure. Setting this property on an existing figure 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`.

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.

CurrentMenu (Obsolete)

This property produces a warning message when queried. It has been superseded by the root `CallbackObject` property.

CurrentObject object handle

Handle of current object. MATLAB sets this property to the handle of the object that is under the current point (see the CurrentPoint property). This object is the front-most object in the stacking order. 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.

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.

In addition, 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 on a uicontrol object.

DeleteFcn string

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

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

Dithermap m-by-3 matrix of RGB values

Colormap used for true-color data on pseudocolor displays. This property defines a colormap that MATLAB uses to dither true-color CData for display on pseudocolor (8-bit or less) displays. MATLAB maps each RGB color defined as true-color CData to the closest color in the dithermap. The default Dithermap contains colors that span the full spectrum so any color values map reasonably well.

However, if the true-color data contains a wide range of shades in one color, you may achieve better results by defining your own dithermap. See the DithermapMode property.

DithermapMode auto | {manual}

MATLAB generated dithermap. In manual mode, MATLAB uses the colormap defined in the Dithermap property to display direct color on pseudocolor displays. When DithermapMode is auto, MATLAB generates a dithermap based on the colors currently displayed. This is useful if the default dithermap does not produce satisfactory results.

The process of generating the dithermap can be quite time consuming and is repeated whenever MATLAB re-renders the display (e.g., when you add a new object or resize the window). You can avoid unnecessary regeneration by setting this property back to `manual` and save the generated dithermap (which MATLAB loaded into the `Dithermap` property).

`DoubleBuffer` `on | {off}`

Flash-free rendering for simple animations. Double buffering is the process of drawing to an off-screen pixel buffer and then blitting 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 enable double buffering.

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

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

`FileName` String

GUI M-file. This property is used by the guide GUI builder to store the name of the generated M-file.

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

Non-colormap colors. Fixed colors define all colors appearing in a figure window that are not obtained from the figure colormap. These colors include axis lines and labels, the color of line, text, uicontrol, and uimenu objects, and any colors that you explicitly define, 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` and

`ShareColors` properties for information on how MATLAB shares colors between applications.)

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

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

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

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

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

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

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

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

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

HitTest {on} | off

Selectable by mouse click. HitTest determines if the 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 on the figure sets the CurrentObject to the empty matrix.

IntegerHandle {on} | off (GUIDE default 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 routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the ButtonDownFcn, KeyPressFcn, WindowButtonDownFcn, WindowButtonMotionFcn, and WindowButtonUpFcn are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback routine only when it encounters a drawnow, figure, getframe, or pause command in the routine. See the BusyAction property for related information.

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 may 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 string

Key press callback function. A callback routine invoked by a key press occurring in the figure window. You can define `KeyPressFcn` as any legal MATLAB expression or the name of an M-file.

The callback routine can query the figure's `CurrentCharacter` property to determine what particular key was pressed and thereby limit the callback execution to specific keys.

The callback routine can also query the root `PointerWindow` property to determine in which figure the key was pressed. Note that pressing a key while the pointer is in a particular figure window does not make that figure the current figure (i.e., the one referred by the `gcf` command).

MenuBar `none | {figure}` (GUIDE default none)

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

This property affects only built in menus. Menus defined with the `uimenu` command are not affected by this property.

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 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 other windows (including other MATLAB figure windows) will display in false colors.

`Name` string

Figure window title. This property specifies the title displayed in the figure window. By default, `Name` is empty and the figure title is displayed as `Figure No. 1`, `Figure No. 2`, and so on. When you set this parameter to a string, the figure title becomes `Figure No. 1: <`*`string`*`>`. See the `NumberTitle` property.

`NextPlot` `{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:

- `add` — use the current figure to display graphics (the default).
- `replace` — reset all figure properties, except `Position`, to their defaults and delete all figure children before displaying graphics (equivalent to `clf reset`).
- `replacechildren` — remove all child objects, but do not reset figure properties (equivalent to `clf`).

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

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

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

`PaperOrientation` `{portrait} | landscape`

Horizontal or vertical paper orientation. This property determines how printed figures are oriented 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.

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 the one of the following standard sizes.

Property Value	Size (Width x Height)
`usletter (default)`	8.5-by-11 inches
`uslegal`	11-by-14 inches
`tabloid`	11-by-17 inches
`A0`	841-by-1189mm
`A1`	594-by-841mm
`A2`	420-by-594mm
`A3`	297-by-420mm
`A4`	210-by-297mm
`A5`	148-by-210mm
`B0`	1029-by-1456mm
`B1`	728-by-1028mm
`B2`	514-by-728mm

Property Value	Size (Width x Height)
B3	364-by-514mm
B4	257-by-364mm
B5	182-by-257mm
arch-A	9-by-12 inches
arch-B	12-by-18 inches
arch-C	18-by-24 inches
arch-D	24-by-36 inches
arch-E	36-by-48 inches
A	8.5-by-11 inches
B	11-by-17 inches
C	17-by-22 inches
D	22-by-34 inches
E	34-by-43 inches

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

PaperUnits normalized | {inches} | centimeters | points

Hardcopy measurement units. This property specifies the units used to define the PaperPosition and PaperSize properties. All units are measured 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 it to its default value after completing your computation so as not to affect other functions that assume `PaperUnits` is set to the default value.

`Parent` handle

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

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

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

`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` 2-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. 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.

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.

`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` – MATLAB's original rendering method is faster when the figure contains only simple or small graphics objects.
- `zbuffer` – MATLAB draws graphics object faster and more accurately because objects are colored 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.

Using the OpenGL Renderer

Hardware vs. Software OpenGL Implementations

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

The hardware implementation makes use of 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, however, OpenGL generally provides superior performance to ZBuffer.

OpenGL Availability

OpenGL is available on all computers that MATLAB runs on. MATLAB automatically finds hardware versions of OpenGl if they are available. If the hardware version is not available, then MATLAB uses the software version.

The software versions that are available on different platforms are:

- On UNIX systems, MATLAB uses the software version of OpenGL that is included in the MATLAB distribution.
- On MS-Windows NT 4.0, OpenGL is available as part of the operating system.
- On MS-Windows 95, OpenGL is included in the Windows 95 OSR 2 release. If you do not have this release, the libraries are available on the Microsoft ftp site.

  Microsoft version is available at the URL:

  ```
  ftp://ftp.microsoft.com/softlib/mslfiles/opengl95.exe
  ```

  There is also a Silicon Graphics version of OpenGL for Windows 95 that is available at the URL:

  ```
  http://www.sgi.com
  ```

Tested Hardware Versions

On MS-Windows platforms, there are many graphics boards that accelerate OpenGL. The MathWorks has tested MATLAB on the AccelECLIPSE board from AccelGraphics.

On UNIX platforms, The MathWorks has tested MATLAB on Sparc Ultra with the Creator 3D board and Silicon Graphics computers running IRIX 6.4 or newer.

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
```

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.

OpenGL vs. Other MATLAB Renderers

There are some difference between drawings created with OpenGL and those created with the 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 will interpolate 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.

MATLAB issues a warning if you request nonsupported behavior.

Implementations of OpenGL Tested by The MathWorks

The following hardware versions have been tested:

- AccelECLIPSE by AccelGraphics
- Sol2/Creator 3D
- SGI

The following software versions have been tested:

- Mesa
- CosmoGL
- Microsoft's Windows 95 implementation
- NT 4.0

`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 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
- The figure contains no logarithmic axes
- MATLAB would select `zbuffer` based on figure contents
- Patch objects faces have no more than three vertices
- The figure contains less than 10 uicontrols
- No line objects use markers
- Phong lighting is not specified

Or

- Figure objects use 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 string

Window resize callback routine. MATLAB executes the specified callback routine whenever you resize the figure window. You can query the figure's Position property to determine the new size and position of the figure window. 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 MATLAB's 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.

`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 are generally associated with particular responses from the user interface software (e.g., single clicking on a graphics object places it in move or resize mode; double-clicking on a filename 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	MS-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 that the `ListBox` style of uicontrols set the figure `SelectionType` property to `normal` to indicate a single mouse click or to `open` to indicate a double mouse click. See `uicontrol` for information on how this property is set when you click on a uicontrol object.

ShareColors `{on} | off`

Share slots in system colortable with like colors. This property affects the way MATLAB stores the figure colormap in the system color table. By default, MATLAB looks at colors already defined and uses those slots to assign pixel colors. This leads to an efficient use of color resources (which are limited on systems capable of displaying 256 or less colors) and extends the number of figure windows that can simultaneously display correct colors.

However, in situations where you want to change the figure colormap quickly without causing MATLAB to re-render the displayed graphics objects, you should disable color sharing (set `ShareColors` to `off`). In this case, MATLAB can swap one colormap for another without changing pixel color assignments because all the slots in the system color table used for the first colormap are replaced with the corresponding color in the second colormap. (Note that this applies only in cases where both colormaps are the same length and where the computer hardware allows user modification of the system color table.)

Tag string (GUIDE sets this property)

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

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'))
```

Type `string (read only)`

Object class. This property identifies the kind of graphics object. For figure objects, `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` (Guide default `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.

WindowButtonDownFcn string

Button press callback function. Use this property to define a callback routine that MATLAB executes whenever you press a mouse button while the pointer is in the figure window. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

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

WindowButtonMotionFcn string

Mouse motion callback function. Use this property to define a callback routine that MATLAB executes whenever you move the pointer within the figure window. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

WindowButtonUpFcn string

Button release callback function. Use this property to define a callback routine that MATLAB executes whenever you release a mouse button. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

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` may not be called. You can prevent this problem by setting `Interruptible` to `on`.

WindowStyle `{normal} | modal`

Normal or modal 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.

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 instead of destroying it when you want to reuse it.

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.

Modal figures do not display uimenu children or built-in menus, 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.

Use modal figures to create dialog boxes that force the user to respond without being able to interact with other windows. Typing **Control C** at the MATLAB prompt causes all figures with WindowStyle modal to revert to WindowStyle normal, allowing you to type at the command line.

XDisplay display identifier (UNIX only)

Specify display for MATLAB. You can display figure windows on different displays using the 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 avail on your system, use the UNIX xdpyinfo command. From MATLAB, type

```
!xdpyinfo
```

The information returned will contain a line specifying the visual ID. For example,

```
visual id:    0x21
```

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

```
set(gcf,'XVisual','0x21')
```

XVisualMode `auto | manual`

Auto or manual selection of visual. `VisualMode` 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`.

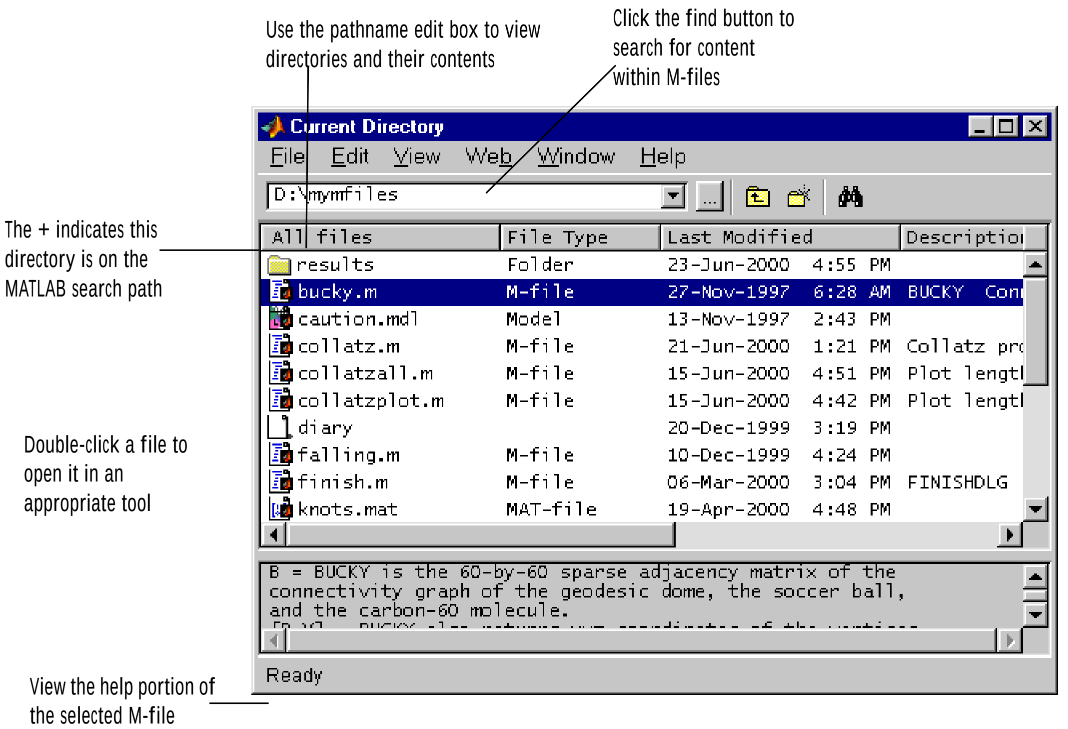
Purpose Display the Current Directory browser, a tool for viewing current directory files

Graphical Interface As an alternative to the `filebrowser` function, select **Current Directory** from the **View** menu in the MATLAB desktop.

Syntax

```
filebrowser
```

Description `filebrowser` displays the Current Directory browser.



See Also cd, pwd

fileparts

Purpose

Return filename parts

Syntax

```
[path,name,ext,ver] = fileparts('filename')
```

Description

`[path,name,ext,ver] = fileparts('filename')` returns the path, filename, extension, and version for the specified file. 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(path,[name ext ver])
```

Examples

This example returns the parts of `file` to `path`, `name`, `ext`, and `ver`.

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

[path,name,ext,ver] = fileparts(file)

path =
\home\user4\matlab

name =
classpath

ext =
.txt

ver =
     ''
```

See Also

`fullfile`

Purpose Filled two-dimensional 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.

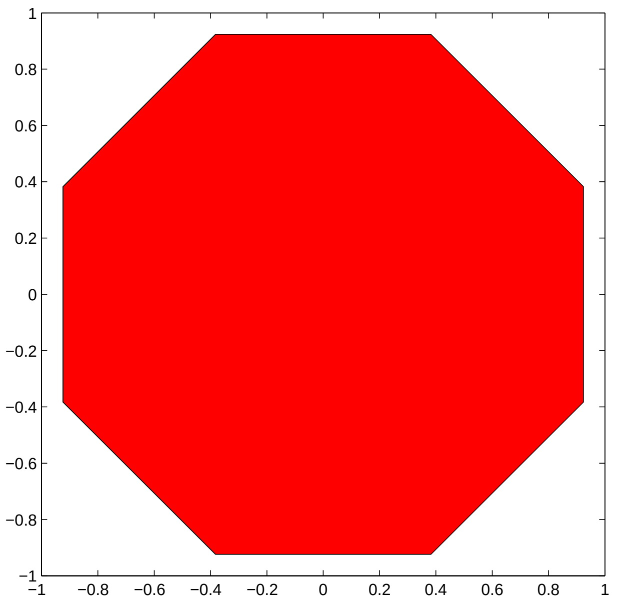
If C is a row vector, fill generates flat-shaded polygons where each element determines the color of the polygon defined by the respective column of the X and Y matrices. Each patch object's FaceColor property is set to 'flat'. Each row element becomes the CData property value for the nth patch object, 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

Purpose

Filled three-dimensional 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.

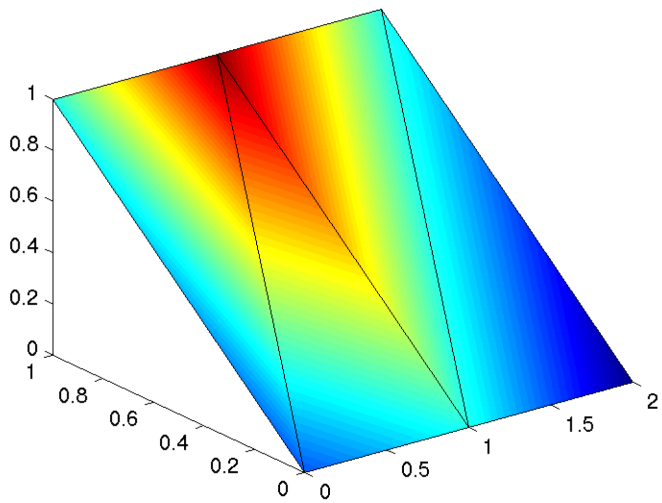
If C is a row vector, fill3 generates flat-shaded polygons and sets the FaceColor property of the patch objects to 'flat'. Each element becomes the CData property value for the respective patch object.

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`

Purpose

Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) 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. Output `zf` is a vector of `max(size(a),size(b))` or an array of such vectors, one for each column of `X`.

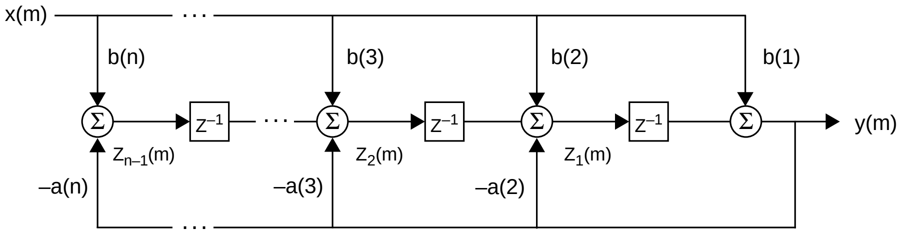
`[y,zf] = filter(b,a,X,zi)` accepts initial conditions and returns the final conditions, `zi` and `zf` respectively, of the filter delays. Input `zi` is a vector (or an array of vectors) of length `max(length(a),length(b))-1`.

`y = filter(b,a,X,zi,dim)` and

`[...] = filter(b,a,X,[],dim)` operate across the dimension `dim`.

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, and which handles both FIR and IIR filters [1].

The operation of `filter` at sample m is given by the time domain difference equations

$$
\begin{aligned}
y(m) &= b(1)x(m) + z_1(m-1)\\
z_1(m) &= b(2)x(m) + z_2(m-1) - a(2)y(m)\\
\vdots \quad &= \quad \vdots \qquad\qquad \vdots\\
z_{n-2}(m) &= b(n-1)x(m) + z_{n-1}(m-1) - a(n-1)y(m)\\
z_{n-1}(m) &= b(n)x(m) - a(n)y(m)
\end{aligned}
$$

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` 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

Two-dimensional digital filtering

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'` (the 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`

find

Purpose

Find indices and values of nonzero elements

Syntax

```
k = find(x)
[i,j] = find(X)
[i,j,v] = find(X)
```

Description

`k = find(X)` returns the indices of the array `X` that point to nonzero elements. If none is found, `find` returns an empty matrix.

`[i,j] = find(X)` returns the row and column indices of the nonzero entries in the matrix `X`. This is often used with sparse matrices.

`[i,j,v] = find(X)` returns a column vector `v` of the nonzero entries in `X`, as well as row and column indices.

In general, `find(X)` regards `X` as `X(:)`, which is the long column vector formed by concatenating the columns of `X`.

Examples

`[i,j,v] = find(X~=0)` produces a vector `v` with all `1`s, and returns the row and column indices.

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
```

And on a matrix

```
M = magic(3)

M =

     8     1     6
     3     5     7
     4     9     2

[i,j,v] = find(M > 6)

i =            j =             v =

     1             1               1
     3             2           1
     2             3           1
```

See Also `nonzeros`, `sparse`, `colon`, `logical operators`, `relational operators`

findall

Purpose Find handles of 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 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 off-screen 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 MATLAB to be off-screen 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 bringing an application from a larger monitor to a smaller one (or one with lower resolution). Windows visible on the larger monitor may appear off-screen on a smaller monitor. Using `findfigs` ensures that all windows appear on the screen.

findobj

Purpose
Locate graphics objects

Syntax

```
h = findobj
h = findobj('PropertyName',PropertyValue,...)
h = findobj(objhandles,...)
h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)
```

Description
`findobj` locates graphics objects and returns their handles. You can limit the search to objects with particular property values and along specific branches of the hierarchy.

`h = findobj` returns the handles of the root object and all its descendants.

`h = findobj('`*`PropertyName`*`',PropertyValue,...)` returns the handles of all graphics objects having the property *PropertyName*, set to the value `PropertyValue`. You can specify more than one property/value pair, in which case, `findobj` returns only those objects having all specified values.

`h = findobj(objhandles,...)` restricts the search to objects listed in `objhandles` and their descendants.

`h = findobj(objhandles,'flat','`*`PropertyName`*`',PropertyValue,...)` restricts the search to those objects listed in `objhandles` and does not search descendants.

Remarks
`findobj` returns an error if a handle refers to a non-existent 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 its handle being returned multiple times.

Examples
Find all line objects in the current axes:

```
h = findobj(gca,'Type','line')
```

See Also

copyobj, gcf, gca, gcbo, gco, get, set

Graphics objects include:

axes, figure, image, light, line, patch, surface, text, uicontrol, uimenu

findstr

Purpose Find one string within another

Syntax `k = findstr(`*str1*`,`*str2*`)`

Description `k = findstr(`*str1*`,`*str2*`)` finds the starting indices of any occurrences of the shorter string within the longer.

Examples

```
str1 = 'Find the starting indices of the shorter string.';
str2 = 'the';
findstr(str1,str2)

ans =
     6    30
```

See Also `strcmp`, `strmatch`, `strncmp`

Purpose MATLAB termination M-file

Description When MATLAB quits, it runs a script called `finish.m`, if it exists and is on the MATLAB search path. This is a file that you create yourself in order to have MATLAB perform any final tasks just prior to terminating. For example, you may 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:

- Select the close box  in the MATLAB Desktop
- Select **Exit MATLAB** from the desktop **File** menu
- Type `quit` or `exit` at the Command Window prompt

Remarks When using Handle Graphics 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 `toolbox/local`. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` to use it.

- `finishsav.m` - saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m` - displays a dialog allowing you to cancel quitting; it uses `quit cancel` and contains the following code.

```
button = questdlg('Ready to quit?', ...
                  'Exit Dialog','Yes','No','No');
switch button
  case 'Yes',
    disp('Exiting MATLAB');
      %Save variables to matlab.mat
      save
  case 'No',
    quit cancel;
end
```

See Also `quit`, `startup`

fix

Purpose

Round towards 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`

Purpose Flip array along a 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`

fliplr

Purpose

Flip matrices left-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`

Purpose Flip matrices up-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`

Purpose Round towards minus 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`

flops

Purpose Count floating-point operations

Description This is an obsolete function. With the incorporation of LAPACK in MATLAB version 6, counting floating-point operations is no longer practical.

Purpose A 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, is the speed profile of a submerged jet within a infinite tank. `flow` is useful for demonstrating `slice`, `interp3`, and for generating scalar volume data.

`v = flow` produces a 50-by-25-by-25 array.

`v = flow(n)` produces a 2n-by-n-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.

fmin

Purpose

Minimize a function of one variable

Note The `fmin` function was replaced by `fminbnd` in Release 11 (MATLAB 5.3). In Release 12 (MATLAB 6.0), `fmin` displays a warning message and calls `fminbnd`.

Syntax

```
x = fmin('fun',x1,x2)
x = fmin('fun',x1,x2,options)
x = fmin('fun',x1,x2,options,P1,P2, ...)
[x,options] = fmin(...)
```

Description

`x = fmin('`*fun*`',x1,x2)` returns a value of `x` which is a local minimizer of `fun(x)` in the interval $x_1 < x < x_2$.

`x = fmin('`*fun*`',x1,x2,options)` does the same as the above, but uses `options` control parameters.

`x = fmin('`*fun*`',x1,x2,options,P1,P2,...)` does the same as the above, but passes arguments to the objective function, `fun(x,P1,P2,...)`. Pass an empty matrix for `options` to use the default value.

`[x,options] = fmin(...)` returns, in `options(10)`, a count of the number of steps taken.

Arguments

`x1,x2`	Interval over which *function* is minimized.
`P1,P2...`	Arguments to be passed to *function*.
fun	A string containing the name of the function to be minimized.

`options` A vector of control parameters. Only three of the 18 components of `options` are referenced by `fmin`; Optimization Toolbox functions use the others. The three control `options` used by `fmin` are:

- `options(1)` — If this is nonzero, intermediate steps in the solution are displayed. The default value of `options(1)` is 0.
- `options(2)` — This is the termination tolerance. The default value is `1.e-4`.
- `options(14)` — This is the maximum number of steps. The default value is `500`.

Examples

`fmin('cos',3,4)` computes π to a few decimal places.

`fmin('cos',3,4,[1,1.e-12])` displays the steps taken to compute π to 12 decimal places.

To find the minimum of the function $f(x) = x^3 - 2x - 5$ on the interval (0, 2), write an M-file called `f.m`.

```
function y = f(x)
y = x.^3-2*x-5;
```

Then invoke `fmin` with

```
x = fmin('f', 0, 2)
```

The result is

```
x =
    0.8165
```

The value of the function at the minimum is

```
y = f(x)

y =
    -6.0887
```

Algorithm

The algorithm is based on golden section search and parabolic interpolation. A Fortran program implementing the same algorithms is given in [1].

See Also

`fmins`	Minimize a function of several variables
`fzero`	Find zero of a function of one variable

`foptions` in the Optimization Toolbox (or type `help optimset`).

References

[1] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

Purpose

Minimize a function of one variable on a fixed interval

Syntax

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
x = fminbnd(fun,x1,x2,options,P1,P2,...)
[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.

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) dislays output only if the function does not converge.
MaxFunEvals	Maximum number of function evaluations allowed.
MaxIter	Maximum number of iterations allowed.
TolX	Termination tolerance on x.

x = fminbnd(fun,x1,x2,options,P1,P2,...) provides for additional arguments, P1, P2, etc., which are passed to the objective function, fun(x,P1,P2,...). Use options=[] as a placeholder if no options are set.

[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`:

>0	Indicates that the function converged to a solution `x`.
0	Indicates that the maximum number of function evaluations was exceeded.
<0	Indicates that the function did not converge to a solution.

`[x,fval,exitflag,output] = fminbnd(...)` returns a structure `output` that contains information about the optimization:

`output.algorithm`	The algorithm used
`output.funcCount`	The number of function evaluations
`output.iterations`	The number of iterations taken

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.

```
x = fminbnd(@myfun,x0)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...          % Compute function value at x.
```

`fun` can also be an inline object.

```
x = fminbnd(inline('sin(x*x)'),x0);
```

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 an inline function. To find the minimum of the function $f(x) = x^3 - 2x - 5$ on the interval (0,2), create an inline object f

```
f = inline('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
```

Algorithm

The algorithm is based on Golden Section search and parabolic interpolation. A Fortran program implementing the same algorithm is given in [1].

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 (@), inline

References

[1] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

fmins

Purpose

Minimize a function of several variables

Note The `fmins` function was replaced by `fminsearch` in Release 11 (MATLAB 5.3). In Release 12 (MATLAB 6.0), `fmins` displays a warning message and calls `fminsearch`.

Syntax

```
x = fmins('fun',x0)
x = fmins('fun',x0,options)
x = fmins('fun',x0,options,[],P1,P2, ...)
[x,options] = fmins(...)
```

Description

`x = fmins('`*fun*`',x0)` returns a vector `x` which is a local minimizer of `fun(x)` near x_0.

`x = fmins('`*fun*`',x0,options)` does the same as the above, but uses `options` control parameters.

`x = fmins('`*fun*`',x0,options,[],P1,P2,...)` does the same as above, but passes arguments to the objective function, `fun(x,P1,P2, ...)`. Pass an empty matrix for `options` to use the default value.

`[x,options] = fmins(...)` returns, in `options(10)`, a count of the number of steps taken.

Arguments

`x0`	Starting vector.
`P1,P2...`	Arguments to be passed to *fun*.
`[]`	Argument needed to provide compatibility with `fminu` in the Optimization Toolbox.

fun	A string containing the name of the objective function to be minimized. `fun(x)` is a scalar valued function of a vector variable.
`options`	A vector of control parameters. Only four of the 18 components of `options` are referenced by `fmins`; Optimization Toolbox functions use the others. The four control `options` used by `fmins` are: • `options(1)` — If this is nonzero, intermediate steps in the solution are displayed. The default value of `options(1)` is 0. • `options(2)` and `options(3)` — These are the termination tolerances for `x` and `function(x)`, respectively. The default values are `1.e-4`. • `options(14)` — This is the maximum number of steps. The default value is `500`.

Examples

A classic test example for multidimensional minimization is the Rosenbrock banana function:

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

The minimum is at `(1,1)` and has the value `0`. The traditional starting point is `(-1.2,1)`. The M-file `banana.m` defines the function.

```
function f = banana(x)
f = 100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

The statements

```
[x,out] = fmins('banana',[-1.2, 1]);
x
out(10)
```

produce

```
x =

    1.0000    1.0000
```

```
ans =

     165
```

This indicates that the minimizer was found to at least four decimal places in 165 steps.

Move the location of the minimum to the point `[a,a^2]` by adding a second parameter to `banana.m`.

```
function f = banana(x,a)
if nargin < 2, a = 1; end
f = 100*(x(2)-x(1)^2)^2+(a-x(1))^2;
```

Then the statement

```
[x,out] = fmins('banana', [-1.2, 1], [0, 1.e-8], [], sqrt(2));
```

sets the new parameter to `sqrt(2)` and seeks the minimum to an accuracy higher than the default.

Algorithm

The algorithm is the Nelder-Mead simplex search described in the two references. It is a direct search method that does not require gradients or other derivative information. If `n` is the length of `x`, a simplex in `n`-dimensional space is characterized by the `n+1` distinct vectors which 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.

See Also

`fmin`	Minimize a function of one variable

`foptions` in the Optimization Toolbox (or type `help foptions`).

References

[1] Nelder, J. A. and R. Mead, "A Simplex Method for Function Minimization," *Computer Journal*, Vol. 7, p. 308-313.

[2] Dennis, J. E. Jr. and D. J. Woods, "New Computing Environments: Microcomputers in Large-Scale Computing," edited by A. Wouk, *SIAM*, 1987, pp. 116-122.

Purpose

Minimize a function of several variables

Syntax

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,options)
x = fminsearch(fun,x0,options,P1,P2,...)
[x,fval] = fminsearch(...)
[x,fval,exitflag] = fminsearch(...)
[x,fval,exitflag,output] = fminsearch(...)
```

Description

`fminsearch` finds the minimum of a scalar function of several variables, starting at an initial estimate. This is generally referred to as *unconstrained nonlinear optimization*.

`x = fminsearch(fun,x0)` starts at the point `x0` and finds a local minimum `x` of the function described in `fun`. `x0` can be a scalar, vector, or matrix.

`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) dislays output only if the function does not converge.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.
`TolX`	Termination tolerance on `x`.

`x = fminsearch(fun,x0,options,P1,P2,...)` passes the problem-dependent parameters `P1`, `P2`, etc., directly to the function `fun`. Use `options = []` as a placeholder if no options are set.

`[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`:

>0	Indicates that the function converged to a solution `x`.
0	Indicates that the maximum number of function evaluations was exceeded.
<0	Indicates that the function did not converge to a solution.

`[x,fval,exitflag,output] = fminsearch(...)` returns a structure `output` that contains information about the optimization:

`output.algorithm`	The algorithm used
`output.funcCount`	The number of function evaluations
`output.iterations`	The number of iterations taken

Arguments

`fun` is the function to be minimized. It 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.

```
x = fminsearch(@myfun,x0,A,b)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`fun` can also be an inline object.

```
x = fminsearch(inline('sin(x*x)'),x0,A,b);
```

Other arguments are described in the syntax descriptions above.

Examples

A classic test example for multidimensional minimization is the Rosenbrock banana function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

The minimum is at `(1,1)` and has the value `0`. The traditional starting point is `(-1.2,1)`. The M-file `banana.m` defines the function.

```
function f = banana(x)
f = 100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

The statement

```
[x,fval] = fminsearch(@banana,[-1.2, 1])
```

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.

Move the location of the minimum to the point `[a,a^2]` by adding a second parameter to `banana.m`.

```
function f = banana(x,a)
if nargin < 2, a = 1; end
f = 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), sqrt(2));
```

sets the new parameter to `sqrt(2)` and seeks the minimum to an accuracy higher than the default on `x`.

Algorithm

`fminsearch` uses the simplex search method of [1]. This is a direct search method that does not use numerical or analytic gradients.

If `n` is the length of `x`, a simplex in `n`-dimensional space is characterized by the `n+1` distinct vectors that are its vertices. In two-space, a simplex is a triangle; in three-space, it is a pyramid. At each step of the search, a new point in or near the current simplex is generated. The function value at the new point is compared with the function's values at the vertices of the simplex and, usually, one of the vertices is replaced by the new point, giving a new simplex. This step is repeated until the diameter of the simplex is less than the specified tolerance.

Limitations

`fminsearch` can often handle discontinuity, particularly if it does not occur near the solution. `fminsearch` may only give local solutions.

`fminsearch` only minimizes over the real numbers, that is, x must only consist of real numbers and $f(x)$ must only return real numbers. When x has complex variables, they must be split into real and imaginary parts.

See Also

`fminbnd`, `optimset`, `function_handle` (@), `inline`

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 a file or obtain information about open files

Syntax

```
fid = fopen(filename)
fid = fopen(filename,permission)
[fid,message] = fopen(filename,permission,machineformat)
fids = fopen('all')
[filename,permission, machineormat] = fopen(fid)
```

Description

`fid = fopen(filename)` opens the file `filename` for read access. (On PCs, `fopen` opens files for binary read access.)

`fid` is a scalar MATLAB integer, called a file identifier. You use the `fid` as the first argument to other file input/output routines. If `fopen` cannot open the file, it returns `-1`. Two file identifiers are automatically available and need not be opened. They are `fid=1` (standard output) and `fid=2` (standard error).

`fid = fopen(filename,permission)` opens the file `filename` in the mode specified by `permission`. `permission` can be:

`'r'`	Open file for reading (default).
`'w'`	Open file, or create new file, for writing; discard existing contents, if any.
`'a'`	Open file, or create new file, for writing; append data to the end of the file.
`'r+'`	Open file for reading and writing.
`'w+'`	Open file, or create a new file, for reading and writing; discard existing contents, if any.
`'a+'`	Open file, or create new file, for reading and writing; append data to the end of the file.
`'A'`	Append without automatic flushing; used with tape drives
`'W'`	Write without automatic flushing; used with tape drives

`filename` can be a `MATLABPATH` relative partial pathname if the file is opened for reading only. A relative path is always searched for first with respect to the current directory. If it is not found and reading only is specified or implied then fopen does an additional search of the `MATLABPATH`

Files can be opened in binary mode (the default) or in text mode. In binary mode, no characters are singled out for special treatment. In text mode on the PC, , the carriage return character preceding a newline character is deleted on input and added before the newline character on output. To open in text mode, add “t” to the permission string, for example `'rt'` and `'wt+'`. (On Unix, text and binary mode are the same so this has no effect. But on PC systems this is critical.)

`[fid,message] = fopen(filename,permission)` opens a file as above. If it cannot open the file, `fid` equals `-1` and `message` contains a system-dependent error message. If `fopen` successfully opens a file, the value of `message` is empty.

`[fid,message] = fopen(filename,permission,machineformat)` opens the specified file with the specified `permission` and treats data read using `fread` or data written using `fwrite` as having a format given by `machineformat`. `machineformat` is one of the following strings:

`'cray'` or `'c'`	Cray floating point with big-endian byte ordering
`'ieee-be'` or `'b'`	IEEE floating point with big-endian byte ordering
`'ieee-le'` or `'l'`	IEEE floating point with little-endian byte ordering
`'ieee-be.l64'` or `'s'`	IEEE floating point with big-endian byte ordering and 64-bit long data type
`'ieee-le.l64'` or `'a'`	IEEE floating point with little-endian byte ordering and 64-bit long data type
`'native'` or `'n'`	Numeric format of the machine on which MATLAB is running (the default).
`'vaxd'` or `'d'`	VAX D floating point and VAX ordering
`'vaxg'` or `'g'`	VAX G floating point and VAX ordering

`fids = fopen('all')` returns a row vector containing the file identifiers of all open files, not including `1` and `2` (standard output and standard error). The number of elements in the vector is equal to the number of open files.

`[filename,permission,machineformat] = fopen(fid)` returns the `filename`, `permission` string, and `machineformat` string associated with the specified file. An invalid `fid` returns empty strings for all output arguments.

The `'W'` and `'A'` permissions are designed for use with tape drives and do not automatically perform a flush of the current output buffer after output operations. For example, open a 1/4" cartridge tape on a SPARCstation for writing with no auto-flush:

```
fid = fopen('/dev/rst0','W')
```

Example

The example uses `fopen` to open a file and then passes the `fid`, returned by `fopen`, to other file I/O functions to read data from the file and then close the file.

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

See Also

`fclose`, `ferror`, `fprintf`, `fread`, `fscanf`, `fseek`, `ftell`, `fwrite`

fopen (serial)

Purpose

Connect a serial port object to the device

Syntax

```
fopen(obj)
```

Arguments

`obj`	A serial port object or an array of serial port objects.

Description

`fopen(obj)` connects `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.

```
s = serial('COM1');
```

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

See Also

Functions

fclose

Properties

BytesAvailable, BytesToOutput, Status, ValuesReceived, ValuesSent

for

Purpose Repeat statements a specific number of times

Syntax

```
for variable = expression
    statements
end
```

Description The general format is

```
for variable = expression
    statement
    ...
    statement
end
```

The columns of the *expression* are stored one at a time in the variable while the following statements, up to the `end`, are executed.

In practice, the *expression* is almost always of the form `scalar` : `scalar`, in which case its columns are simply scalars.

The scope of the `for` statement is always terminated with a matching `end`.

Examples Assume `n` has already been assigned a value. Create the Hilbert matrix, using `zeros` to preallocate the matrix to conserve memory:

```
a = zeros(n,n)  % Preallocate matrix
for i = 1:n
    for j = 1:n
        a(i,j) = 1/(i+j -1);
    end
end
```

Step `s` with increments of `-0.1`

```
for s = 1.0: -0.1: 0.0,..., 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 j = 1:n, V = A(:,j);..., end
```

except `j` is also set here.

See Also

`break`, `end`, `if`, `return`, `switch`, `while`

The colon operator `:`

format

Purpose

Control the display format for output

Graphical Interface

As an alternative to `format`, use preferences. Select **Preferences** from the **File** menu in the MATLAB desktop and use **Command Window** preferences.

Syntax

```
format
format type
format('type')
```

Description

MATLAB performs all computations in double precision. Use the `format` function to control the output format of the numeric values displayed in the Command Window. The `format` function affects only how numbers are displayed, not how MATLAB computes or saves them. The specified format applies only to the current session. To maintain a format across sessions, use MATLAB preferences.

`format` by itself, changes the output format to the default type, `short`, which is 5-digit scaled, fixed-point values.

`format` *type* changes the format to the specified *type*. The table below describes the allowable values for *type*. To see the current *type* file, use `get(0,'Format')`, or for `compact` versus `loose`, use `get(0,'FormatSpacing')`.

Value for type	Result	Example
`+`	+, -, blank	`+`
`bank`	Fixed dollars and cents	`3.14`

`compact`	Suppresses excess line feeds to show more output in a single screen. Contrast with `loose`.	`theta = pi/2`
		`theta=`
		`    1.5708`

`hex`	Hexadecimal	`400921fb54442d18`
`long`	15-digit scaled fixed point	`3.14159265358979`
`long e`	15-digit floating point	`3.141592653589793e+` `00`

Value for type	Result	Example
`long g`	Best of 15-digit fixed or floating point	`3.14159265358979`

`loose`	Adds linefeeds to make output more readable. Contrast with `compact`.	`theta = pi/2`
		
		`theta=`
		
		`1.5708`

`rat`	Ratio of small integers	`355/113`
`short`	5-digit scaled fixed point	`3.1416`
`short e`	5-digit floating point	`3.1416e+00`
`short g`	Best of 5-digit fixed or floating point	`3.1416`

`format('`*`type`*`')` is the function form of the syntax.

Examples

Change the format for `pi` to `long` by typing.

```
format long
```

View the result by typing

```
pi
```

and MATLAB returns

```
ans =
   3.14159265358979
```

View the current format by typing

```
get(0,'Format')
```

MATLAB returns

```
ans =
long
```

Set the format to `short e` by typing

```
format short e
```

or use the function form of the syntax

```
format('short','e')
```

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

`fprintf`, `num2str`, `rat`, `sprintf`, `spy`

Purpose Plot a function between specified limits

Syntax

```
fplot('function',limits)
fplot('function',limits,LineSpec)
fplot('function',limits,tol)
fplot('function',limits,tol,LineSpec)
fplot('function',limits,n)
[X,Y] = fplot('function',limits,...)
[...] = plot('function',limits,tol,n,LineSpec,P1,P2,...)
```

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('`*`function`*`',limits)` plots `'`*`function`*`'` between the limits specified by `limits`. `limits` is a vector specifying the x-axis limits (`[xmin xmax]`), or the x- and y-axis limits, (`[xmin xmax ymin ymax]`).

`'`*`function`*`'` must be the name of an M-file function or a string with variable `x` that may be passed to `eval`, such as `'sin(x)'`, `'diric(x,10)'` or `'[sin(x),cos(x)]'`.

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('`*`function`*`',limits,LineSpec)` plots `'`*`function`*`'` using the line specification `LineSpec`.

`fplot('`*`function`*`',limits,tol)` plots `'`*`function`*`'` using the relative error tolerance `tol` (The default is 2e–3, i.e., 0.2 percent accuracy).

fplot('*function*',limits,tol,LineSpec) plots '*function*' using the relative error tolerance tol and a line specification that determines line type, marker symbol, and color.

fplot('*function*',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.

[X,Y] = fplot('*function*',limits,...) returns the abscissas and ordinates for '*function*' in X and Y. No plot is drawn on the screen, however you can plot the function using plot(X,Y).

[...] = plot('*function*',limits,tol,n,LineSpec,P1,P2,...) enables you to pass parameters P1, P2, etc. directly to the function '*function*':

```
Y = function(X,P1,P2,...)
```

To use default values for tol, n, or LineSpec, you can pass in the empty matrix ([]).

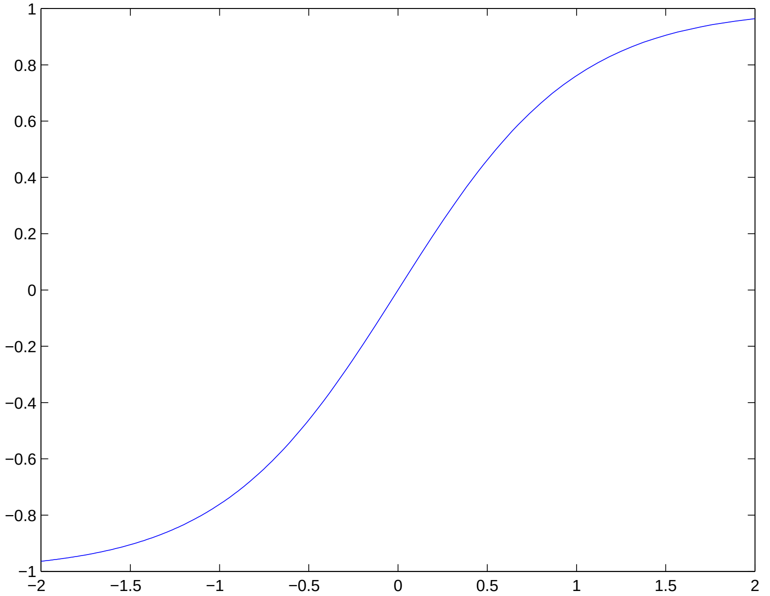
Remarks

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:

```
fplot('tanh',[-2 2])
```

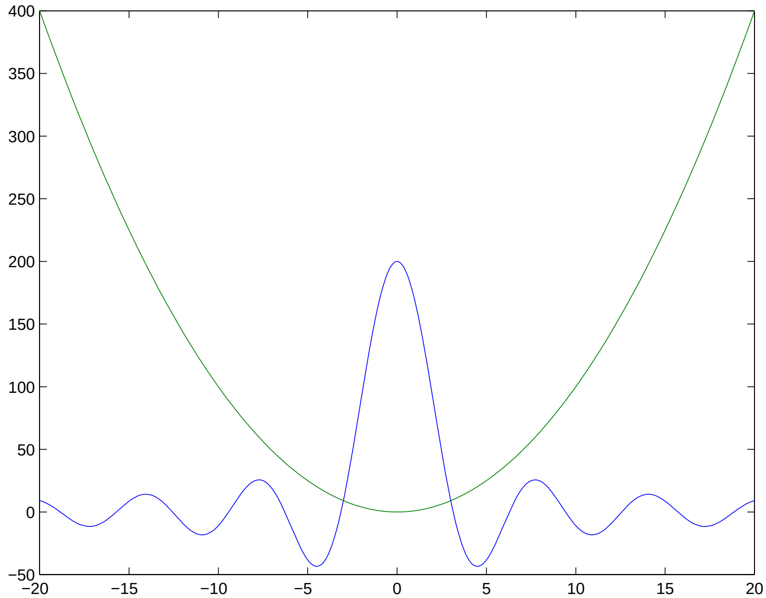


Create an M-file, `myfun`, that returns a two column matrix:

```
function Y = myfun(x)
Y(:,1) = 200*sin(x(:))./x(:);
Y(:,2) = x(:).^2;
```

Plot the function with the statement:

```
fplot('myfun',[–20 20]
```



Addition Examples

```
subplot(2,2,1);fplot('humps',[0 1])
subplot(2,2,2);fplot('abs(exp(-j*x*(0:9))*ones(10,1))',[0 2*pi])
subplot(2,2,3);fplot('[tan(x),sin(x),cos(x)]',2*pi*[-1 1 -1 1])
subplot(2,2,4);fplot('sin(1./x)',[0.01 0.1],1e-3)
```

See Also `eval`, `feval`, `LineSpec`, `plot`

Purpose Write formatted data to file

Syntax

```
count = fprintf(fid,format,A,...)
```

Description `count = fprintf(fid,format,A,...)` formats the data in the real part of matrix `A` (and in any additional matrix arguments) under control of the specified `format` string, and writes it to the file associated with file identifier `fid`. `fprintf` returns a count of the number of bytes written.

Argument `fid` is an integer file identifier obtained from `fopen`. (It may also be `1` for standard output (the screen) or `2` for standard error. See `fopen` for more information.) Omitting `fid` causes output to appear on the screen.

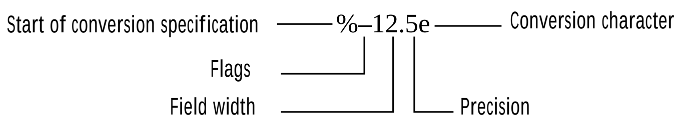
Format String

The `format` argument is a string containing C language conversion specifications. A conversion specification controls the notation, alignment, significant digits, field width, and other aspects of output format. The format string can contain escape characters to represent non-printing characters such as newline characters and tabs.

Conversion specifications begin with the `%` character and contain these optional and required elements:

- Flags (optional)
- Width and precision fields (optional)
- A subtype specifier (optional)
- Conversion character (required)

You specify these elements in the following order:



Flags

You can control the alignment of the output using any of these optional flags.

Character	Description	Example
A minus sign (–)	Left-justifies the converted argument in its field.	`%–5.2d`
A plus sign (+)	Always prints a sign character (+ or –).	`%+5.2d`
Zero (`0`)	Pad with zeros rather than spaces.	`%05.2d`

Field Width and Precision Specifications

You can control the width and precision of the output by including these options in the format string.

Character	Description	Example
Field width	A digit string specifying the minimum number of digits to be printed.	`%6f`
Precision	A digit string including a period (.) specifying the number of digits to be printed to the right of the decimal point.	`%6.2f`

Conversion Characters

Conversion characters specify the notation of the output.

Specifier	Description
`%c`	Single character
`%d`	Decimal notation (signed)
`%e`	Exponential notation (using a lowercase `e` as in `3.1415e+00`)
`%E`	Exponential notation (using an uppercase `E` as in `3.1415E+00`)

Specifier	Description
`%f`	Fixed-point notation
`%g`	The more compact of `%e` or `%f`, as defined in [2]. Insignificant zeros do not print.
`%G`	Same as `%g`, but using an uppercase `E`
`%o`	Octal notation (unsigned)
`%s`	String of characters
`%u`	Decimal notation (unsigned)
`%x`	Hexadecimal notation (using lowercase letters `a`–`f`)
`%X`	Hexadecimal notation (using uppercase letters `A`–`F`)

Conversion characters `%o`, `%u`, `%x`, and `%X` support subtype specifiers. See Remarks for more information.

Escape Characters

This table lists the escape character sequences you use to specify non-printing characters in a format specification.

Character	Description
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\\`	Backslash

Character	Description
\'' or '' (two single quotes)	Single quotation mark
%%	Percent character

Remarks

The `fprintf` function behaves like its ANSI C language namesake with these exceptions and extensions.

- If you use `fprintf` to convert a MATLAB double into an integer, and the double contains a value that cannot be represented as an integer (for example, it contains a fraction), MATLAB ignores the specified conversion and outputs the value in exponential format. To successfully perform this conversion, use the `fix`, `floor`, `ceil`, or `round` functions to change the value in the double into a value that can be represented as an integer before passing it to sprintf.
- The following, non-standard subtype specifiers are supported for the conversion characters `%o`, `%u`, `%x`, and `%X`.

b	The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like '`%bx`'.
t	The underlying C data type is a float rather than an unsigned integer.

  For example, to print a double value in hexidecimal use the format `'%bx'`
- The `fprintf` function is vectorized for nonscalar arguments. The function recycles the format string through the elements of `A` (columnwise) until all the elements are used up. The function then continues in a similar manner through any additional matrix arguments.

Examples

The statements

```
x = 0:.1:1;
y = [x; exp(x)];
```

```
fid = fopen('exp.txt','w');
fprintf(fid,'%6.2f %12.8f\n',y);
fclose(fid)
```

create a text file called `exp.txt` containing a short table of the exponential function:

```
0.00    1.00000000
0.10    1.10517092
...
1.00    2.71828183
```

The command

```
fprintf('A unit circle has circumference %g.\n',2*pi)
```

displays a line on the screen:

```
A unit circle has circumference 6.283186.
```

To insert a single quotation mark in a string, use two single quotation marks together. For example,

```
fprintf(1,'It''s Friday.\n')
```

displays on the screen:

```
It's Friday.
```

The commands

```
B = [8.8  7.7; 8800  7700]
fprintf(1,'X is %6.2f meters or %8.3f mm\n',9.9,9900,B)
```

display the lines:

```
X is 9.90 meters or 9900.000 mm
X is 8.80 meters or 8800.000 mm
X is 7.70 meters or 7700.000 mm
```

Explicitly convert MATLAB double-precision variables to integral values for use with an integral conversion specifier. For instance, to convert signed 32-bit data to hexadecimal format:

```
a = [6 10 14 44];
fprintf('%9X\n',a + (a<0)*2^32)
```

```
 6
 A
 E
2C
```

See Also

`fclose`, `ferror`, `fopen`, `fread`, `fscanf`, `fseek`, `ftell`, `fwrite`, `disp`

References

[1] Kernighan, B.W. and D.M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988.

[2] ANSI specification X3.159-1989: “Programming Language C,” ANSI, 1430 Broadway, New York, NY 10018.

Purpose

Write text to the device

Syntax

```
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
```

Arguments

`obj`	A serial port object.
`'cmd'`	The string written to the device.
`'format'`	C language conversion specification.
`'mode'`	Specifies whether data is written synchronously or asynchronously.

Description

`fprintf(obj,'cmd')` writes the string `cmd` to the device connected to `obj`. The default format is `%s\n`. The write operation is synchronous and blocks the command line until execution is complete.

`fprintf(obj,'`*format*`','cmd')` writes the string using the format specified by *format*. *format* is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sprintf` file I/O format specifications or a C manual for more information.

`fprintf(obj,'cmd','`*mode*`')` writes the string with command line access specified by *mode*. If *mode* is `sync`, `cmd` is written synchronously and the command line is blocked. If *mode* is `async`, `cmd` is written asynchronously and the command line is not blocked. If *mode* is not specified, the write operation is synchronous.

`fprintf(obj,'`*format*`','cmd','`*mode*`')` writes the string using the specified format. If *mode* is `sync`, `cmd` is written synchronously. If *mode* is `async`, `cmd` is written asynchronously.

Remarks

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

The ValuesSent property value is increased by the number of values written each time fprintf is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the OutputBufferSize property.

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

```
help serial/fprintf
```

Synchronous Versus Asynchronous Write Operations

By default, text is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *mode* input argument to be async. For asynchronous writes:

- The BytesToOutput property value is continuously updated to reflect the number of bytes in the output buffer.
- The M-file action function specified for the OutputEmptyAction 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 fprintf

A synchronous or asynchronous write operation using fprintf completes when:

- The specified data is written.
- The time specified by the Timeout property passes.

Additionally, you can stop an asynchronous write operation with the stopasync function.

Rules for Writing the Terminator

All occurrences of `\n` in `cmd` are replaced with the `Terminator` property value. Therefore, when using the default format `%s\n`, all commands written to the device will end with this property value. The terminator required by your device will be described in its documentation.

Example

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

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

Since the default format for `fprintf` is `%s\n`, the terminator specified by the `Terminator` property was automatically written. However, in some cases you may want to suppress writing the terminator. To do so, you must explicitly specify a format for the data that does not include the terminator.

```
fprintf(s,'%s','RS232?')
```

See Also

Functions

`fopen`, `fwrite`, `stopasync`

Properties

`BytesToOutput`, `OutputBufferSize`, `OutputEmptyAction`, `Status`, `TransferStatus`, `ValuesSent`

frame2im

Purpose Convert movie frame to indexed image

Syntax

```
[X,Map] = frame2im(F)
```

Description `[X,Map] = frame2im(F)` converts the single movie frame `F` into the indexed image `X` and associated colormap `Map`. The functions `getframe` and `im2frame` create a movie frame. If the frame contains truecolor data, then `Map` is empty.

See Also `getframe`, `im2frame`, `movie`

Purpose

Create and edit print frames for Simulink and Stateflow block diagrams

Syntax

```
frameedit
frameedit filename
```

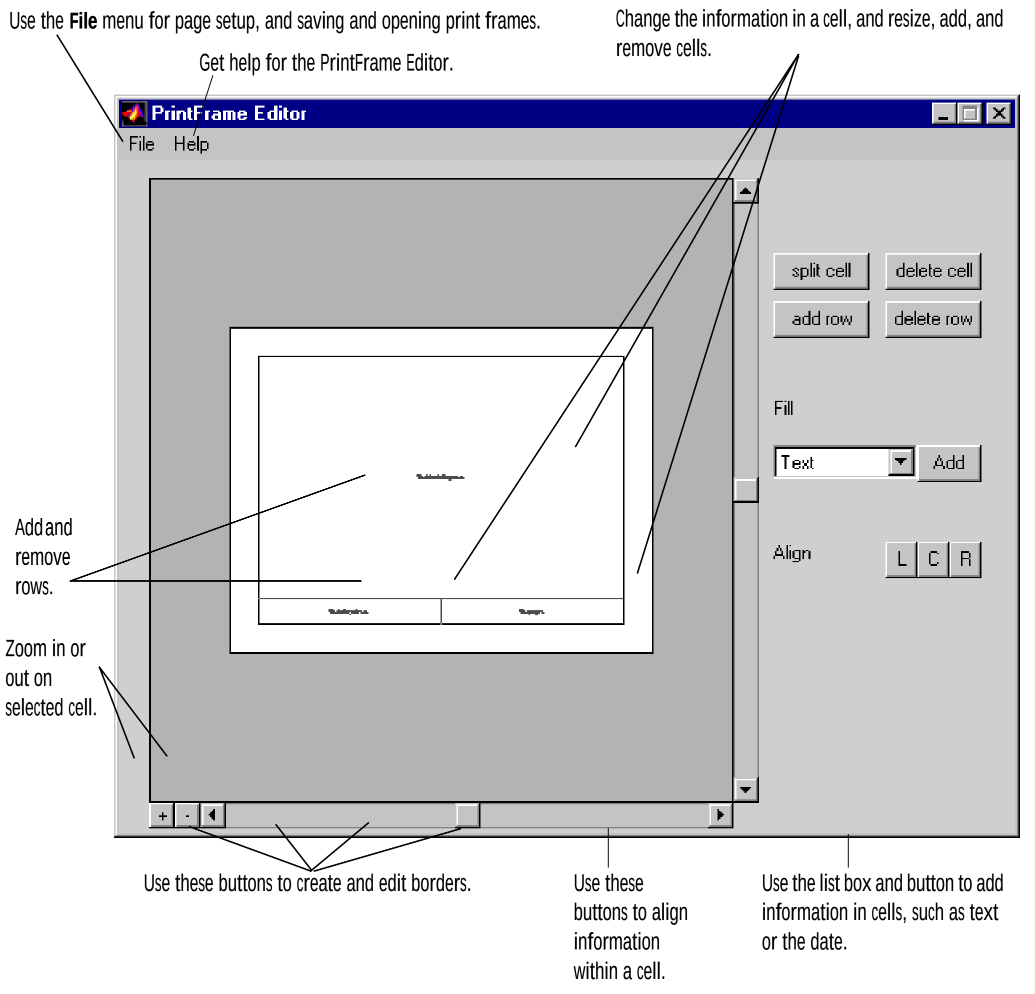
Description

`frameedit` starts the PrintFrame Editor, a graphical user interface you use to create borders for Simulink and Stateflow block diagrams. With no argument, `frameedit` opens the **PrintFrame Editor** window with a new file.

`frameedit filename` opens the **PrintFrame Editor** window with the specified filename, where `filename` is a figure file (`.fig`) previously created and saved using `frameedit`.

Remarks

This illustrates the main features of the PrintFrame Editor.



Closing the PrintFrame Editor

To close the **PrintFrame Editor** window, click the close box in the upper right corner, or select **Close** from the **File** menu.

Printing Simulink Block Diagrams with Print Frames

Select **Print** from the Simulink **File** menu. Check the **Frame** box and supply the filename for the print frame you want to use. Click **OK** in the **Print** dialog box.

Getting Help for the PrintFrame Editor

For further instructions on using the PrintFrame Editor, select **PrintFrame Editor Help** from the **Help** menu in the PrintFrame Editor.

fread

Purpose

Read binary data from file

Syntax

```
[A,count] = fread(fid,size,precision)
[A,count] = fread(fid,size,precision,skip)
```

Description

`[A,count] = fread(fid,size,precision)` reads binary data from the specified file and writes it into matrix `A`. Optional output argument `count` returns the number of elements successfully read. `fid` is an integer file identifier obtained from `fopen`.

`size` is an optional argument that determines how much data is read. If `size` is not specified, `fread` reads to the end of the file and the file pointer is at the end of the file (see `feof` for details). Valid options are:

`n`	Reads `n` elements into a column vector.
`inf`	Reads to the end of the file, resulting in a column vector containing the same number of elements as are in the file.
`[m,n]`	Reads enough elements to fill an `m`–by–`n` matrix, filling in elements in column order, padding with zeros if the file is too small to fill the matrix. `n` can be specified as `inf`, but `m` cannot.

`precision` is a string that specifies the format of the data to be read. It commonly contains a datatype specifier such as `int` or `float`, followed by an integer giving the size in bits. Any of the strings in the following table, either the MATLAB version or their C or Fortran equivalent, may be used. If precision is not specified, the default is `'uchar'`..

MATLAB	C or Fortran	Interpretation
`'schar'`	`'signed char'`	Signed character; 8 bits
`'uchar'`	`'unsigned char'`	Unsigned character; 8 bits
`'int8'`	`'integer*1'`	Integer; 8 bits
`'int16'`	`'integer*2'`	Integer; 16 bits
`'int32'`	`'integer*4'`	Integer; 32 bits

MATLAB	C or Fortran	Interpretation
'int64'	'integer*8'	Integer; 64 bits
'uint8'	'integer*1'	Unsigned integer; 8 bits
'uint16'	'integer*2'	Unsigned integer; 16 bits
'uint32'	'integer*4'	Unsigned integer; 32 bits
'uint64'	'integer*8'	Unsigned integer; 64 bits
'float32'	'real*4'	Floating-point; 32 bits
'float64'	'real*8'	Floating-point; 64 bits
'double'	'real*8'	Floating-point; 64 bits

The following platform dependent formats are also supported but they are not guaranteed to be the same size on all platforms.

MATLAB	C or Fortran	Interpretation
'char'	'char*1'	Character; 8 bits
'short'	'short'	Integer; 16 bits
'int'	'int'	Integer; 32 bits
'long'	'long'	Integer; 32 or 64 bits
'ushort'	'unsigned short'	Unsigned integer; 16 bits
'uint'	'unsigned int'	Unsigned integer; 32 bits
'ulong'	'unsigned long'	Unsigned integer; 32 or 64 bits
'float'	'float'	Floating-point; 32 bits

The following formats map to an input stream of bits rather than bytes.

MATLAB	C or Fortran	Interpretation
'bitN'	-	Signed integer; N bits ($1 \le N \le 64$)
'ubitN'	-	Unsigned integer; N bits ($1 \le N \le 64$)

By default, numeric values are returned in class 'double' arrays. To return numeric values stored in classes other thatn double, create your precision argument by first specifying your source format, then following it with the characters “=>”, and finally specifying your destination format. You are not reuiqred to use the exact name of a MATLAB class type for destination. (See class for details). fread translates the name to the most appropriate MATLAB class type. If the source and destination formats are the same, the following shorthand notation may be used:

```
*source
```

which means:

```
source=>source
```

This table shows some example precision format strings.

'uint8=>uint8'	Read in unsigned 8-bit integers and save them in an unsigned 8-bit integer array.
'*uint8'	Shorthand version of the above.
'bit4=>int8'	Read in signed 4-bit integers packed in bytes and save them in a signed 8-bit array. Each 4-bit integer becomes an 8-bit integer
'double=>real*4'	Read in doubles, convert and save as a 32-bit floating point array.

[A,count] = fread(fid,size,precision,skip) includes an optional skip argument that specifies the number of bytes to skip after each precision value is read. With skip is used, the precision string may contain a positive integer repetition factor of the form `'N*'` which prepends the source format specification, such as `'40*uchar'`.

Note Do not confuse the asterisk (*) used in the repetition factor with the asterisk used as precision format shorthand. The format string `'40*uchar'` is equivalent to `'40*uchar=>double'`, not `'40*uchar=>uchar'`.

When skip is specified, `fread` reads in at most a repetition factor number of values (default is 1), does a skip of input specified by the `skip` argument, reads in another block of values, does a skip of input, and so on, until `size` number of values have beenread. If `precision` is a bit format like `'bitN'` or `'ubitN'`, `skip` is specified in bits. Use the repetition factor with the `skip` argument to extract data in noncontiguous fields from fixed length records.

If `fread` reaches the end of the file and the current input stream does not contain enough bits to write out a complete matrix element of the specified precision, `fread` pads the last byte or element with zero bits until the full value is obtained. If an error occurs, reading is done up to the last full value.

Examples

For example,

```
type fread.m
```

displays the complete M-file containing this fread help entry. To simulate this command using fread, enter the following:

```
fid = fopen('fread.m','r');
F = fread(fid);
s = char(F')
```

In the example, the `fread` command assumes the default size, `inf`, and the default precision, `'uchar'`. `fread` reads the entire file, converting the unsigned characters into a column vector of class `'double'` (double precision floating point). To display the result as readable text, the `'double'` column vector is transposed to a row vector and converted to class `'char'` using the `char` function.

As another example,

```
s = fread(fid,120,'40*uchar=>uchar',8);
```

reads in 120 characters in blocks of 40, each separated by 8 characters. Note that the class type of `s` is `'uint8'` since it is the appropriate class corresponding to the destination format, `'uchar'`. Also, since 40 evenly divides 120, the last block read is a full block which means that a final skip will be done before the command is finished. If the last block read is not a full block then `fread` will not finish with a skip.

See `fopen` for informationabout reading Big and Little Endian files.

See Also

`fclose`, `ferror`, `fopen`, `fprintf`, `fread`, `fscanf`, `fseek`, `ftell`, `fwrite`

Purpose Read binary data from the device

Syntax

```
A = fread(obj,size)
A = fread(obj,size,'precision')
[A,count] = fread(...)
[A,count,msg] = fread(...)
```

Arguments

`obj`	A serial port object.
`size`	The number of values to read.
`'precision'`	The number of bits read for each value, and the interpretation of the bits as character, integer, or floating-point values.
`A`	Binary data returned from the device.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`A = fread(obj,size)` reads binary data from the device connected to `obj`, and returns the data to `A`. The maximum number of values to read is 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. 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
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

fread (serial)

See Also

Functions

`fgetl`, `fgets`, `fopen`, `fscanf`

Properties

`BytesAvailable`, `BytesAvailableAction`, `InputBufferSize`, `Status`, `Terminator`, `ValuesReceived`

Purpose Determine frequency spacing for frequency response

Syntax

```
[f1,f2] = freqspace(n)
[f1,f2] = freqspace([m n])
[x1,y1] = freqspace(...,'meshgrid')
f = freqspace(N)
f = freqspace(N,'whole')
```

Description

`freqspace` returns the implied frequency range for equally spaced frequency responses. `freqspace` is useful when creating desired frequency responses for various one- and two-dimensional applications.

`[f1,f2] = freqspace(n)` returns the two-dimensional frequency vectors `f1` and `f2` for an `n`-by-`n` matrix.

For `n` odd, both `f1` and `f2` are `[-n+1:2:n-1]/n`.

For `n` even, both `f1` and `f2` are `[-n:2:n-2]/n`.

`[f1,f2] = freqspace([m n])` returns the two-dimensional frequency vectors `f1` and `f2` for an `m`-by-`n` matrix.

`[x1,y1] = freqspace(...,'meshgrid')` is equivalent to

```
[f1,f2] = freqspace(...);
[x1,y1] = meshgrid(f1,f2);
```

`f = freqspace(N)` returns the one-dimensional frequency vector `f` assuming `N` evenly spaced points around the unit circle. For `N` even or odd, `f` is `(0:2/N:1)`. For `N` even, `freqspace` therefore returns `(N+2)/2` points. For `N` odd, it returns `(N+1)/2` points.

`f = freqspace(N,'whole')` returns `N` evenly spaced points around the whole unit circle. In this case, `f` is `0:2/N:2*(N-1)/N`.

See Also `meshgrid`

frewind

Purpose Move the file position indicator to the beginning of an open file

Syntax `frewind(fid)`

Description `frewind(fid)` sets the file position indicator to the beginning of the file specified by `fid`, an integer file identifier obtained from `fopen`.

Remarks Rewinding a `fid` associated with a tape device may not work even though `frewind` does not generate an error message.

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

Purpose Read formatted data from file

Syntax

```
A = fscanf(fid,format)
[A,count] = fscanf(fid,format,size)
```

Description `A = fscanf(fid,format)` reads all the data from the file specified by `fid`, converts it according to the specified `format` string, and returns it in matrix `A`. Argument `fid` is an integer file identifier obtained from `fopen`. `format` is a string specifying the format of the data to be read. See "Remarks" for details.

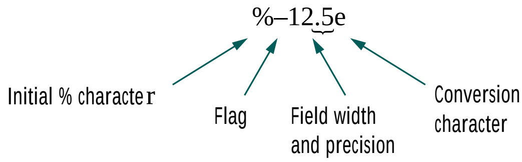
`[A,count] = fscanf(fid,format,size)` reads the amount of data specified by `size`, converts it according to the specified `format` string, and returns it along with a `count` of elements successfully read. `size` is an argument that determines how much data is read. Valid options are:

`n`	Read `n` elements into a column vector.
`inf`	Read to the end of the file, resulting in a column vector containing the same number of elements as are in the file.
`[m,n]`	Read enough elements to fill an `m`-by-`n` matrix, filling the matrix in column order. `n` can be `Inf`, but not `m`.

`fscanf` differs from its C language namesakes `scanf()` and `fscanf()` in an important respect — it is *vectorized* in order to return a matrix argument. The `format` string is cycled through the file until an end-of-file is reached or the amount of data specified by `size` is read in.

Remarks When MATLAB reads a specified file, it attempts to match the data in the file to the format string. If a match occurs, the data is written into the matrix in column order. If a partial match occurs, only the matching data is written to the matrix, and the read operation stops.

The `format` string consists of ordinary characters and/or conversion specifications. Conversion specifications indicate the type of data to be matched and involve the character %, optional width fields, and conversion characters, organized as shown below:



Add one or more of these characters between the % and the conversion character:

An asterisk (*)	Skip over the matched value, if the value is matched but not stored in the output matrix.
A digit string	Maximum field width.
A letter	The size of the receiving object; for example, `h` for short as in `%hd` for a short integer, or `l` for long as in `%ld` for a long integer or `%lg` for a double floating-point number.

Valid conversion characters are:

`%c`	Sequence of characters; number specified by field width
`%d`	Decimal numbers
`%e, %f, %g`	Floating-point numbers
`%i`	Signed integer
`%o`	Signed octal integer
`%s`	A series of non-white-space characters
`%u`	Signed decimal integer
`%x`	Signed hexadecimal integer
`[...]`	Sequence of characters (scanlist)

If `%s` is used, an element read may use several MATLAB matrix elements, each holding one character. Use `%c` to read space characters or `%s` to skip all white space.

Mixing character and numeric conversion specifications cause the resulting matrix to be numeric and any characters read to appear as their ASCII values, one character per MATLAB matrix element.

For more information about format strings, refer to the `scanf()` and `fscanf()` routines in a C language reference manual.

Examples

The example in `fprintf` generates an ASCII text file called `exp.txt` that looks like:

```
0.00    1.00000000
0.10    1.10517092
...
1.00    2.71828183
```

Read this ASCII file back into a two-column MATLAB matrix:

```
fid = fopen('exp.txt');
a = fscanf(fid,'%g %g',[2 inf]) % It has two rows now.
a = a';
fclose(fid)
```

See Also

`fgetl`, `fgets`, `fread`, `fprintf`, `fscanf`, `input`, `sscanf`, `textread`

fscanf (serial)

Purpose

Read data from the 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(...)
```

Arguments

`obj`	A serial port object.
`'`*`format`*`'`	C language conversion specification.
`size`	The number of values to read.
`A`	Data read from the device and formatted as text.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`A = fscanf(obj)` reads data from the device connected to `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.

```
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?')
```

Since 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`, `BytesAvailableAction`, `InputBufferSize`, `Status`, `Terminator`, `Timeout`

Purpose Set file position indicator

Syntax

```
status = fseek(fid,offset,origin)
```

Description

`status = fseek(fid,offset,origin)` repositions the file position indicator in the file with the given `fid` to the byte with the specified `offset` relative to `origin`.

Arguments

`fid`	An integer file identifier obtained from `fopen`.
`offset`	A value that is interpreted as follows:
	`offset > 0` Move position indicator `offset` bytes toward the end of the file.
	`offset = 0` Do not change position.
	`offset < 0` Move position indicator `offset` bytes toward the beginning of the file.
`origin`	A string whose legal values are:
	`'bof'` `–1`: Beginning of file.
	`'cof'` `0`: Current position in file.
	`'eof'` `1`: End of file.
`status`	A returned value that is `0` if the `fseek` operation is successful and `–1` if it fails. If an error occurs, use the function `ferror` to get more information.

See Also `fopen`, `ftell`

ftell

Purpose Get file position indicator

Syntax `position = ftell(fid)`

Description `position = ftell(fid)` returns the location of the file position indicator for the file specified by `fid`, an integer file identifier obtained from `fopen`. The `position` is a nonnegative integer specified in bytes from the beginning of the file. A returned value of `–1` for `position` indicates that the query was unsuccessful; use `ferror` to determine the nature of the error.

See Also `fclose`, `ferror`, `fopen`, `fprintf`, `fread`, `fscanf`, `fseek`, `fwrite`

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 (logical)
  S    200X200   318432   sparse array (logical)
```

See Also `sparse`

fullfile

Purpose

Build a full filename from parts

Syntax

```
fullfile('dir1','dir2',...,'filename')
f = fullfile('dir1','dir2',...,'filename')
```

Description

`fullfile(dir1,dir2,...,filename)` builds a full filename from the directories and filename specified. This is conceptually equivalent to

```
f = [dir1 dirsep dir2 dirsep ... dirsep filename]
```

except that care is taken to handle the cases when the directories begin or end with a directory separator.

Examples

To create the full filename from a disk name, directories, and filename,

```
f = fullfile('C:','Applications','matlab','myfun.m')
f =
C:\Applications\matlab\myfun.m
```

The following examples both produce the same result on UNIX, but only the second one works on all platforms.

`fullfile(matlabroot,'toolbox/matlab/general/Contents.m')` and

`fullfile(matlabroot,'toolbox','matlab','general','Contents.m')`

See Also

`fileparts`

Purpose Constructs a function name string from a function handle

Syntax

```
s = 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 also operates on MATLAB data structures that hold more than one function handle, (for example, arrays). It returns the names of all function handles in the data structure. These names are contained in a cell array of strings.

Examples To create a function name string from the function handle, `@humps`

```
funname = func2str(@humps)
funname =
humps
```

To create a cell array of function names from an array of function handles

```
func_array = func2str([@sin @cos @tan])
func_array =
    'sin'    'cos'    'tan'
```

See Also `function_handle`, `str2func`, `functions`

function

Purpose Function M-files

Description You add new functions to MATLAB's 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. For example, the existence of a file on disk called `stat.m` with

```
function [mean,stdev] = stat(x)
n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2/n));
```

defines a new function called `stat` that calculates the mean and standard deviation of a vector. 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. For example, `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;
```

Subfunctions are not visible outside the file where they are defined. 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. In general, if you input the name of something to MATLAB, the MATLAB interpreter:

1. Checks to see if the name is a variable.
2. Checks to see if the name is an internal function (`eig`, `sin`) that was not overloaded.
3. Checks to see if the name is a local function (local in sense of multifunction file).
4. Checks to see if the name is a function in a private directory.
5. Locates any and all occurrences of function in method directories and on the path. Order is of no importance.

At execution, MATLAB:

6. Checks to see if the name is wired to a specific function (2, 3, & 4 above)
7. Uses precedence rules to determine which instance from 5 above to call (we may default to an internal MATLAB function). Constructors have higher precedence than anything else.

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.

See Also

`nargin`, `nargout`, `pcode`, `varargin`, `varargout`, `what`

function_handle (@)

Purpose MATLAB data type that is a handle to a function

Syntax

```
handle = @function
```

Description `@function` returns a handle to the specified MATLAB function.

A function handle is a MATLAB data type that captures all the information about a function that MATLAB needs to execute, or *evaluate*, it. A function handle is typically passed in an argument list to other functions. These other functions then use the handle as a means to call the function for whom the handle was constructed.

Because many MATLAB functions are overloaded, a function handle often maps to a number of code sources (e.g., built-in code, M-files), that implement the function. A function handle stores the context of all of these overloaded sources, or *methods*, if they are on the MATLAB path at the time the handle is created.

You evaluate a function handle using the MATLAB `feval` command. If you pass a function handle as an argument into another function, then the receiving function must use `feval` to evaluate the function. When you evaluate an overloaded function handle, MATLAB selects and executes one of the overloaded methods whose context is stored in the handle.

As a standard MATLAB data type, a function handle can be manipulated and operated on in the same manner as other MATLAB data types.

Function handles enable you to:

- Pass a function reference to another function within the handle
- Reduce the number of files that define your functions
- Improve performance in repeated operations
- Ensure reliability when evaluating functions

Examples 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 (@humps, 0.3, 1)
x =
    0.6370
```

See Also str2func, func2str, functions

functions

Purpose Return information about a function handle

Syntax

```
f = functions(funhandle)
f = functions(@fun)
```

Description `f = functions(funhandle)` returns, in a MATLAB structure, the function name, type, and filename for the function handle stored in the variable, `funhandle`. For overloaded functions, it also returns a separate structure identifying the classes and M-files that overload the function.

`f = functions(@fun)` returns the same information for the function, `fun`.

Examples To obtain information on a function handle for the `display` function,

```
f = functions(@deblank)
f =
    function: 'deblank'
        type: 'overloaded'
        file: 'matlabroot\toolbox\matlab\strfun\deblank.m'
     methods: [1x1 struct]
```

The `methods` field is a separate structure containing one fieldname for each class that overloads the function. The value of each field is the path and name of the file that defines the method.

```
f = functions(@display);
f.methods

ans =

   polynom: '\home\user4\@polynom\display.m'
    inline: 'matlabroot\toolbox\matlab\funfun\@inline\display.m'
    serial: 'matlabroot\toolbox\matlab\iofun\@serial\display.m'
   avifile: 'matlabroot\toolbox\matlab\iofun\@avifile\display.m'
```

See Also `function_handle`

Purpose Evaluate general matrix function

Syntax

```
F = funm(A,fun)
[F,esterr] = funm(A,fun)
```

Description `F = funm(A,fun)` for a square matrix argument `A`, evaluates the matrix version of the function `fun`. For matrix exponentials, logarithms and square roots, use `expm(A)`, `logm(A)` and `sqrtm(A)` instead.

`[F,esterr] = funm(A,fun)` does not print any message, but returns a very rough estimate of the relative error in the computed result.

If `A` is symmetric or Hermitian, then its Schur form is diagonal and `funm` is able to produce an accurate result.

`L = logm(A)` uses `funm` to do its computations, but it can get more reliable error estimates by comparing `expm(L)` with `A`. `S = sqrtm(A)` and `E = expm(A)` use completely different algorithms.

Examples **Example 1.** `fun` can be specified using `@`:

```
F = funm(magic(3),@sin)
```

is the matrix sine of the 3-by-3 magic matrix.

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))`.

Algorithm `funm` uses a potentially unstable algorithm. If `A` is close to a matrix with multiple eigenvalues and poorly conditioned eigenvectors, `funm` may produce inaccurate results. An attempt is made to detect this situation and print a warning message. The error detector is sometimes too sensitive and a message is printed even though the the computed result is accurate.

The matrix functions are evaluated using Parlett's algorithm, which is described in [1].

See Also

`expm`, `logm`, `sqrtm`, `function_handle` (`@`)

References

[1] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Johns Hopkins University Press, 1983, p. 384.

[2] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review 20*, 1979, pp. 801-836.

Purpose Write binary data to a file

Syntax

```
count = fwrite(fid,A,precision)
count = fwrite(fid,A,precision,skip)
```

Description

`count = fwrite(fid,A,precision)` writes the elements of matrix `A` to the specified file, translating MATLAB values to the specified `precision`. The data is written to the file in column order, and a `count` is kept of the number of elements written successfully.

`fid` is an integer file identifier obtained from `fopen`, or `1` for standard output or `2` for standard error.

`precision` controls the form and size of the result. See `fread` for a list of allowed precisions. For `'bitN'` or `'ubitN'` precisions, `fwrite` sets all bits in `A` when the value is out-of-range.

`count = fwrite(fid,A,precision,skip)` includes an optional `skip` argument that specifies the number of bytes to skip before each `precision` value is written. With the `skip` argument present, `fwrite` skips and writes one value, skips and writes another value, etc. until all of `A` is written. If `precision` is a bit format like `'bitN'` or `'ubitN'`, `skip` is specified in bits. This is useful for inserting data into noncontiguous fields in fixed-length records.

Examples For example,

```
fid = fopen('magic5.bin','wb');
fwrite(fid,magic(5),'integer*4')
```

creates a 100-byte binary file, containing the 25 elements of the 5-by-5 magic square, stored as 4-byte integers.

See Also `fclose`, `ferror`, `fopen`, `fprintf`, `fread`, `fscanf`, `fseek`, `ftell`

fwrite (serial)

Purpose Write binary data to the device

Syntax

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A,'mode')
fwrite(obj,A,'precision','mode')
```

Arguments

`obj`	A serial port object.
`A`	The binary data written to the device.
`'`*`precision`*`'`	The number of bits written for each value, and the interpretation of the bits as character, integer, or floating-point values.
`'`*`mode`*`'`	Specifies whether data is written synchronously or asynchronously.

Description `fwrite(obj,A)` writes the binary data `A` to the device connected to `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
```

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 action function specified for the `OutputEmptyAction` 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
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`, `OutputEmptyAction`, `Status`, `Timeout`, `TransferStatus`, `ValuesSent`

fzero

Purpose

Find zero of a function of one variable

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)
x = fzero(fun,x0,options,P1,P2,...)
[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. 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.

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) dislays output only if the function does not converge.
`TolX`	Termination tolerance on `x`.

`x = fzero(fun,x0,options,P1,P2,...)` provides for additional arguments passed to the function, `fun`. Use `options = []` as a placeholder if no options are set.

`[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`:

>0	Indicates that the function found a zero `x`.
<0	No interval was found with a sign change, or a `NaN` or `Inf` function value was encountered during search for an interval containing a sign change, or a complex function value was encountered during the search for an interval containing a sign change.

`[x,fval,exitflag,output] = fzero(...)` returns a structure `output` that contains information about the optimization:

`output.algorithm`	The algorithm used
`output.funcCount`	The number of function evaluations
`output.iterations`	The number of iterations taken

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.

```
x = fzero(@myfun,x0)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`fun` can also be an inline object.

```
x = fzero(inline('sin(x*x)'),x0);
```

Other arguments are described in the syntax descriptions above.

Examples

Calculate π by finding the zero of the sine function near 3.

```
x = fzero(@sin,3)
x =
    3.1416
```

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.

To find a zero of the function $f(x) = x^3 - 2x - 5$

write an M-file called `f.m`.

```
function y = f(x)
y = x.^3-2*x-5;
```

To 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
```

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`, `function_handle` (@), `inline`, `optimset`

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.

gallery

Purpose Test matrices

Syntax

`[A,B,C,...] = gallery('tmfun',P1,P2,...)`
`gallery(3)` a badly conditioned 3-by-3 matrix
`gallery(5)` an interesting eigenvalue problem

Description

`[A,B,C,...] = gallery('tmfun',P1,P2,...)` returns the test matrices specified by string *tmfun*. *tmfun* 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.

The gallery holds over fifty different test matrix functions useful for testing algorithms and other purposes.

Test Matrices			
cauchy	chebspec	chebvand	chow
circul	clement	compar	condex
cycol	dorr	dramadah	fiedler
forsythe	frank	gearmat	grcar
hanowa	house	invhess	invol
ipjfact	jordbloc	kahan	kms
krylov	lauchli	lehmer	lesp
lotkin	minij	moler	neumann
orthog	parter	pei	poisson
prolate	randcolu	randcorr	rando
randhess	randsvd	redheff	riemann
ris	rosser	smoke	toeppd

Test Matrices (Continued)			
tridiag	triw	vander	wathen
wilk			

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 V 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. `alpha` and `delta` are scalars with default values `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` with $w = [1\ t\ t^2\ \dots\ t^{(n-1)}]$ is an eigenvalue of `C` and `w(n:-1:1)` is an eigenvector.

clement—Tridiagonal matrix with zero diagonal entries

`A = gallery('clement',n,sym)` returns an `n` by `n` tridiagonal matrix with zeros on its main diagonal and known eigenvalues. It is singular if order `n` is odd. About 64 percent of the entries of the inverse are zero. The eigenvalues include plus and minus the numbers `n-1`, `n-3`, `n-5`, `...`, as well as (for odd `n`) a final eigenvalue of `1` or `0`.

Argument `sym` determines whether the Clement matrix is symmetric. For `sym = 0` (the default) the matrix is nonsymmetric, while for `sym = 1`, it is symmetric.

compar—Comparison matrices

`A = gallery('compar',A,1)` returns `A` with each diagonal element replaced by its absolute value, and each off-diagonal element replaced by minus the absolute value of the largest element in absolute value in its row. However, if `A` is triangular `compar(A,1)` is too.

`gallery('compar',A)` is `diag(B) - tril(B,-1) - triu(B,1)`, where `B = abs(A)`. `compar(A)` is often denoted by $M(A)$ in the literature.

`gallery('compar',A,0)` is the same as `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` as follows:

`k = 1`	4-by-4	LINPACK (`rcond`)
`k = 2`	3-by-3	LINPACK (`rcond`)
`k = 3`	arbitrary	LINPACK (`rcond`) (independent of `theta`)
`k = 4`	`n` ≥ 4	SONEST (Higham 1988) (default)

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`. `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 a row diagonally dominant, tridiagonal order n 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 nth 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.

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+2a), ..., sin(w+Na)]`, 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] = 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`.

invhess—Inverse of an upper Hessenberg matrix

`A = gallery('invhess',x,y)`, where `x` is a length `n` vector and `y` 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(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`.

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 = 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 singular, row-diagonally dominant matrix resulting from discretizing the Neumann problem with the usual five-point operator on a regular mesh. Argument `n` is a perfect square integer $n = m^2$ or a two-element vector. `C` is sparse and has a one-dimensional null space with null vector `ones(n,1)`.

orthog—Orthogonal and nearly orthogonal matrices

`Q = gallery('orthog',n,k)` returns the 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 = -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:

- For `k = 0` (the default) `diag(x)` is initially subjected to a random two-sided orthogonal transformation, and then a sequence of Givens rotations is applied.
- For `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:

- For `k = 0` (the default) the diagonal matrix of eigenvalues is initially subjected to a random orthogonal similarity transformation, and then a sequence of Givens rotations is applied.
- For `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.

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` may 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.

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`^(1/2+epsilon)) 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`^(`-1/2`+epsilon)) for every epsilon > `0`.

The Riemann matrix is defined by:

```
A = B(2:n+1,2:n+1)
```

where `B(i,j) = i-1` if `i` divides `j`, and `B(i,j) = -1` otherwise.

The Riemann matrix has these properties:

- Each eigenvalue `e(i)` satisfies `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.

rosser—Classic symmetric eigenvalue test matrix

`A = rosser` returns the Rosser matrix. This matrix was a challenge for many matrix eigenvalue algorithms. But the Francis QR algorithm, as perfected by

Wilkinson and implemented in MATLAB, has no trouble with it. The matrix is 8-by-8 with integer elements. It has:

- A double eigenvalue
- Three nearly equal eigenvalues
- Dominant eigenvalues of opposite sign
- A zero eigenvalue
- A small, nonzero eigenvalue

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 `smoke(n,1)` are the nth roots of unity; those of `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-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.

vander—Vandermonde matrix

`A = gallery('vander',c)` returns the Vandermonde matrix whose second to last column is `c`. The `j`th column of a Vandermonde matrix is given by `A(:,j) = C^(n-j)`.

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

`[A,b] = gallery('wilk',n)` returns a different matrix or linear system depending on the value of `n`:

n	MATLAB Code	Result
`n = 3`	`[A,b] = gallery('wilk',3)`	Upper triangular system `Ux=b` illustrating inaccurate solution.
`n = 4`	`[A,b] = gallery('wilk',4)`	Lower triangular system `Lx=b`, ill-conditioned.

`n = 5`	`A = gallery('wilk',5)`	`hilb(6)(1:5,2:6)*1.8144`. A symmetric positive definite matrix.
`n = 21`	`A = gallery('wilk',21)`	`W21+`, tridiagonal matrix. Eigenvalue problem.

See Also `hadamard`, `hilb`, `invhilb`, `magic`, `wilkinson`

References 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. Additional detail on these matrices is documented in *The Test Matrix Toolbox for MATLAB* by N. J. Higham, September, 1995. This report is available via anonymous ftp from The MathWorks at `ftp://ftp.mathworks.com/pub/contrib/linalg/testmatrix/testmatrix.ps` or on the Web at `ftp://ftp.ma.man.ac.uk/pub/narep` or `http://www.ma.man.ac.uk/MCCM/MCCM.html`. Further background can be found in the book *Accuracy and Stability of Numerical Algorithms,* Nicholas J. Higham, SIAM, 1996.

Purpose

Gamma functions

Syntax

`Y = gamma(A)`	Gamma function
`Y = gammainc(X,A)`	Incomplete gamma function
`Y = gammaln(A)`	Logarithm of gamma function

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:

$$P(x, a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

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 = 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 Get current axes handle

Syntax

```
h = gca
```

Description `h = gca` returns the handle to the current axes for the current figure. If no axes exists, MATLAB 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.

The current axes is the target for graphics output when you create axes children. 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`, `delete`, `gcf`, `gcbo`, `gco`, `hold`, `subplot`, `findobj`

figure `CurrentAxes` property

gcbf

Purpose Get 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 Return the handle of the object whose callback is currently 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 MATLAB stores the handle of the object whose callback is executing in the root's `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`

gcd

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.

gcf

Purpose

Get 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, MATLAB 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

`axes`, `clf`, `close`, `delete`, `figure`, `gca`, `gcbo`, `gco`, `subplot`

root `CurrentFigure` property

Purpose Return 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. MATLAB 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

genpath

Purpose Generate a path string

Syntax

```
genpath
genpath directory
p = genpath('directory')
```

Description

`genpath` returns a path string formed by recursively adding all the directories below matlabroot/toolbox.

`genpath directory` returns a path string formed by recursively adding all the directories below `directory`.

`p = genpath('directory')` returns the path string to variable, `p`.

Examples

You generate a path that includes matlabroot\toolbox\images and all directories below that with the following command:

```
p = genpath(fullfile(matlabroot,'toolbox','images'))
p =
matlabroot\toolbox\images;matlabroot\toolbox\images\images;
matlabroot\toolbox\images\images\ja;matlabroot\toolbox\images\
imdemos;matlabroot\toolbox\images\imdemos\ja;
```

See Also `path`, `addpath`, `rmpath`

Purpose Get object properties

Syntax

```
get(h)
get(h,'PropertyName')
<m-by-n value cell array> = get(H,<property cell array>)
a = get(h)
a = get(0,'Factory')
a = get(0,'FactoryObjectTypePropertyName')
a = get(h,'Default')
a = get(h,'DefaultObjectTypePropertyName')
```

Description

`get(h)` returns all properties and their current values of the graphics object identified by the handle `h`.

`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. `h` must be a scalar. 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

Purpose Return serial port object properties

Syntax

```
get(obj)
out = get(obj)
out = get(obj,'PropertyName')
```

Arguments

`obj`	A serial port object or an array of serial port objects.
`'PropertyName'`	A property name or a cell array of property names.
`out`	A single property value, a structure of property values, or a cell array of property values.

Description

`get(obj)` returns all property names and their current values to the command line for `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`.

```
s = serial('COM1');
out1 = get(s);
out2 = get(s,{'BaudRate','DataBits'});
get(s,'Parity')
ans =
none
```

See Also

Functions

`set`

Purpose Get 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 the handle `h`. If the application-defined data does not exist, MATLAB returns an empty matrix in `value`.

`value = getappdata(h)` returns all application-defined data for the object with handle `h`.

See Also `setappdata`, `rmappdata`, `isappdata`

getenv

Purpose Get field of structure array

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, MATLAB 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 `computer`, `pwd`, `ver`, `path`

Purpose

Get 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.

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 i = 1:2
    name{i} = getfield(mystr,{i,1},'name');
end
name

name =

    'alice'    'gertrude'
```

See Also

`setfield`, `rmfield`, `fieldnames`

Purpose Get movie frame

Syntax

```
F = getframe
F = getframe(h)
F = getframe(h,rect)
[X,Map] = getframe(...)
```

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 the 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.

`F = 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)
```

`[X,Map] = getframe(...)` returns the frame as an indexed image matrix `X` and a colormap `Map`. This version is obsolete and is supported only for compatibility with earlier version of MATLAB. Since indexed images cannot always capture true color displays, you should use the single output argument form of `getframe`. To write code that is compatible with earlier version of

MATLAB and that can take advantage of true color support, use the following approach:

```
F = getframe(gcf);
[X,Map] = frame2im(f);
imshow(X,Map)
```

Remarks

Usually, `getframe` is 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)
```

To create movies that are compatible with earlier versions of MATLAB (before Release 11/MATLAB 5.3) use this approach:

```
M = moviein(n);
for j = 1:n
    plotting commands
    M(:,j) = getframe;
end
movie(M)
```

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.

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 `getframe`, `frame2im`, `image`, `im2frame`, `movie`, `moviein`

ginput

Purpose Input data using the mouse

Syntax

```
[x,y] = ginput(n)
[x,y] = ginput
[x,y,button] = ginput(...)
```

Description `ginput` enables you to select points from the figure using the mouse or arrow keys for cursor positioning. The figure must have focus before `ginput` receives input.

`[x,y] = ginput(n)` enables you to select `n` points from the current axes and returns the *x*- and *y*-coordinates in the column vectors `x` and `y`, respectively. You can 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.

Remarks If you select points from multiple axes, the results you get are relative to those axes coordinates systems.

Examples Pick 10 two-dimensional points from the figure window.

```
[x,y] = ginput(10)
```

Position the cursor with the mouse (or the arrow keys on terminals without a mouse, such as Tektronix emulators). Enter data points by pressing a mouse button or a key on the keyboard. To terminate input before entering 10 points, press the **Return** key.

See Also `gtext`

Purpose Define a global variable

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,

```
uicontrol('style','pushbutton','CallBack',...
'global MY_GLOBAL,disp(MY_GLOBAL),MY_GLOBAL = MY_GLOBAL+1,clear MY_GLOBAL',...
'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)
gmres(afun,b,restart,tol,maxit,m1fun,m2fun,x0,p1,p2,...)
[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 the column vector `b` must have length `n`. `A` can be a function `afun` such that `afun(x)` returns `A*x`.

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` iterations. If `restart` is `[]`, then `gmres` uses the default, `n`, which does not actually restart.

`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 iterations. If `maxit` is `[]`, then `gmres` uses the default, `min(n/restart,10)`.

`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 that 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.

`gmres(afun,b,restart,tol,maxit,m1fun,m2fun,x0,p1,p2,...)` passes parameters to functions `afun(x,p1,p2,...)`, `m1fun(x,p1,p2,...)`, and `m2fun(x,p1,p2,...)`.

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

Flag	Convergence
0	`gmres` converged to the desired tolerance `tol` within `maxit` iterations.
1	`gmres` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
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,[],[]);
gmres(10) converged at iteration 2(10) to a solution with relative
residual 1.9e-013
```

Alternatively, use this matrix-vector product function

```
function y = afun(x,n)
y = [0;
     x(1:n-1)] + [((n-1)/2:-1:0)';
     (1:(n-1)/2)'] .*x + [x(2:n);
     0];
```

and this preconditioner backsolve function

```
function y = mfun(r,n)
y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
```

as inputs to `gmres`

```
x1 = gmres(@afun,b,10,tol,maxit,@mfun,[],[],21);
```

Note that both `afun` and `mfun` must accept `gmres`'s extra input `n=21`.

Example 2.

```
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.

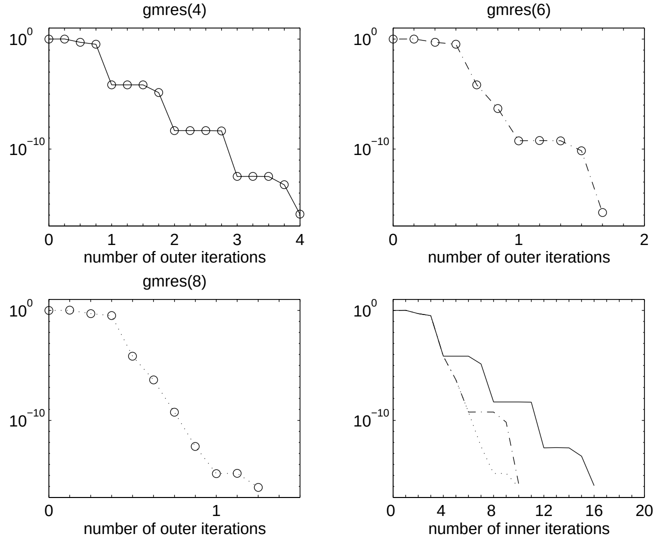
```
[L1,U1] = luinc(A,1e-5);
[x1,flag1] = gmres(A,b,5,1e-6,5,L1,U1);
```

`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`, `luinc`, `minres`, `pcg`, `qmr`, `symmlq`

`@` (function handle), `\` (backslash)

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] 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 set of nodes using an adjacency matrix

Synopsis

```
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 or an *n*-by-3 matrix, where *n* is the number of nodes and each coordinate pair or triple represents one node.

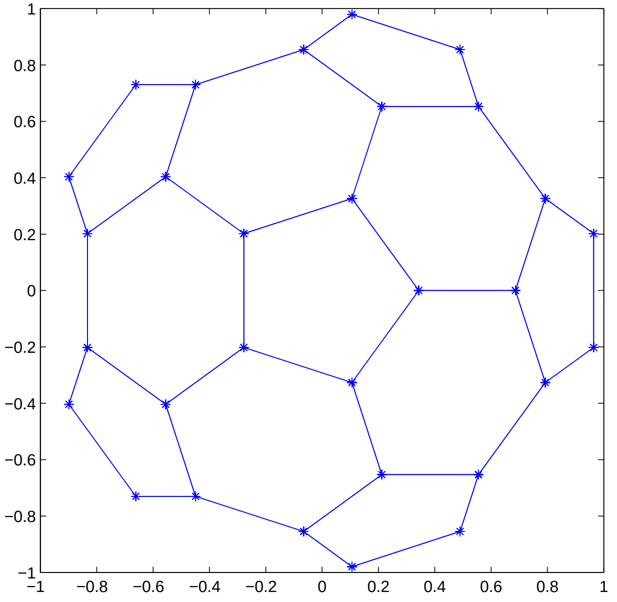
`gplot(A,Coordinates,`*`LineSpec`*`)` plots the nodes using the line type, marker symbol, and color specified by `LineSpec`.

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 joined, `A(i,j)` or `A(j,i)` are 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`

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, numerical gradients (differences) can be computed for functions with any number of variables. For a function of N variables, *F(x,y,z,...)*,

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j} + \frac{\partial F}{\partial z}\hat{k} + \ldots$$

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 the x 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 the `x` (column) direction. `FY` corresponds to $\partial F / \partial y$, the differences in the y (row) 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.

[...] = gradient(F,h) where h is a scalar uses h as the spacing between points in each direction.

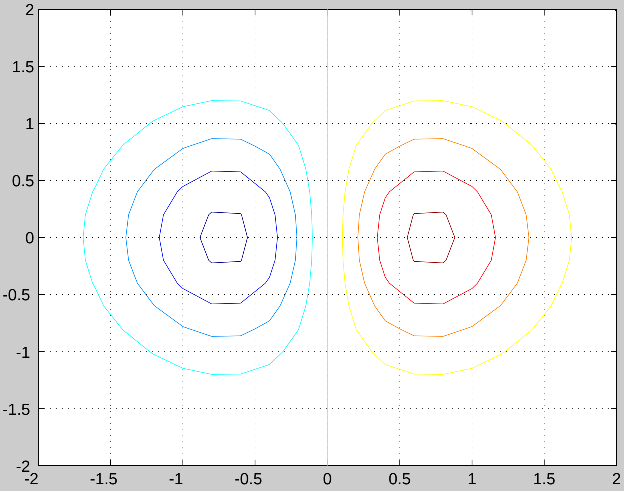
[...] = gradient(F,h1,h2,...) with N spacing parameters specifies the spacing for each dimension of F.

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(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`

Purpose Grid lines for two- and three-dimensional plots

Syntax

```
grid on
grid off
grid
grid(axes_handle,...)
```

Description The `grid` function turns the current axes' grid lines on and off.

`grid on` adds grid lines to the current axes.

`grid off` removes grid lines from the current axes.

`grid` toggles the 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 current axes.

See Also `axes`, `plot`

The `XGrid`, `YGrid`, and `ZGrid` properties of axes objects.

Purpose Data gridding

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.

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).

Algorithm

The griddata(...,'v4') command uses the method documented in [1]. The other methods are based on Delaunay triangulation (see delaunay).

Examples

Sample a function at 100 random points between ±2.0:

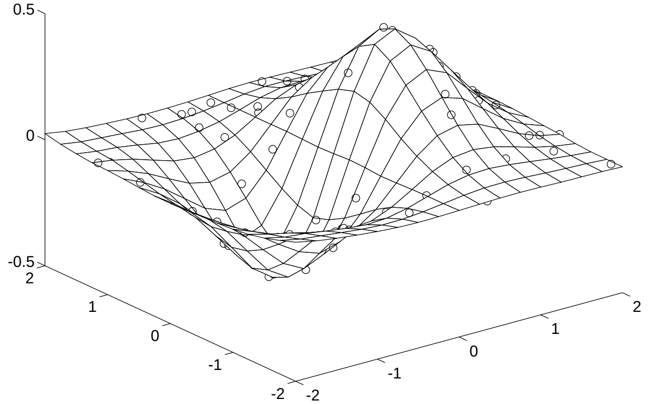
```
rand('seed',0)
x = rand(100,1)*4-2; y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

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`, `griddata3`, `griddatan`, `interp2`, `meshgrid`

References

[1] Sandwell, David T., “Biharmonic Spline Interpolation of GEOS-3 and SEASAT Altimeter Data”, *Geophysical Research Letters*, 2, 139-142,1987.

[2] Watson, David E., *Contouring: A Guide to the Analysis and Display of Spatial Data*, Tarrytown, NY: Pergamon (Elsevier Science, Inc.): 1992.

Purpose

Data gridding and hypersurface fitting for 3-D data

Syntax

```
w = griddata3(x,y,z,v,xi,yi,zi)
w = griddata3(...,'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(...,'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

Note All the methods are based on a Delaunay triangulation of the data that uses `qhull` [1]. For information about `qhull`, see `http://www.geom.umn.edu/software/qhull/`. For copyright information, see `http://www.geom.umn.edu/software/download/COPYING.html`.

See Also

`delaunayn`, `griddata`, `griddatan`, `meshgrid`

Reference

[1] National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center), University of Minnesota. 1993.

griddatan

Purpose

Data gridding and hypersurface fitting (dimension >= 2)

Syntax

```
yi = griddatan(X,y,xi)
yi = griddatan(...,'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`-D 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`-D 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`).

`[...] = griddatan(...,'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.

Note `griddatan` calls `delaunayn`, which is based on `qhull` [1]. For information about `qhull`, see `http://www.geom.umn.edu/software/qhull/`. For copyright information, see `http://www.geom.umn.edu/software/download/COPYING.html`.

See Also

`delaunayn`, `griddata`, `griddata3`, `meshgrid`

Reference

[1] National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center), University of Minnesota. 1993.

Purpose Generalized singular value decomposition

Syntax

```
[U,V,X,C,S] = gsvd(A,B)
[U,V,X,C,S] = gsvd(A,B,0)
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 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.

Reference

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

See Also

`svd`

gtext

Purpose Mouse placement of text in two-dimensional view

Syntax

```
gtext('string')
h = gtext('string')
```

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.

`h = gtext('`*`string`*`')` returns the handle to a text graphics object after you place `'`*`string`*`'` on the plot at the selected location.

Remarks As you move the pointer into a figure window, the pointer becomes a crosshair 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`

Purpose Store or retrieve application data

Syntax

```
guidata(object_handle,data)
data = guidata(object_handle)
```

Description `guidata(object_handle,data)` stores the specified data in the figure's application data. If `object_handle` is not a figure handle, then the object's parent figure is used.

`data = guidata(object_handle)` returns previously stored data, or an empty matrix if nothing has been stored.

`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.

`guidata` is particularly useful in conjunction with `guihandles`, which creates a structure in the figure's application data containing the handles of all the components in a GUI.

Examples In this example, `guidata` is used to save a structure on a GUI figure's application data from within the initialization section of the application M-file. This structure is initially created by `guihandles` and then used to save additional data as well.

```
data = guihandles(figure_handle); % create structure of handles
data.numberOfErrors = 0; % add some additional data
guidata(figure_handle,data) % save the structure
```

You can recall the data from within a subfunction callback routine and then save the structure again:

```
data = guidata(gcbo); % get the structure in the subfunction
data.numberOfErrors = data.numberOfErrors + 1;
guidata(gcbo,data) % save the changes to the structure
```

See Also

`guide`, `guihandles`, `getappdata`, `setappdata`

Purpose Start the GUI Layout Editor

Syntax

```
guide
guide('filename.fig')
guide(figure_handles)
```

Description `guide` displays the GUI Layout Editor open to a new untitled FIG-file.

`guide('filename.fig')` opens the FIG-file named `filename.fig`. You can specify the path to a file not on your MATLAB path.

`guide('figure_handles')` opens FIG-files in the Layout Editor for each existing figure listed in `figure_handles`. MATLAB copies the contents of each figure into the FIG-file, with the exception of axes children (image, light, line, patch, rectangle, surface, and text objects), which are not copied.

See Also `inspect`

Creating GUIs

guihandles

Purpose Create a 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 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.

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`

Purpose HDF interface

Syntax

```
hdf*(functstr,param1,param2,...)
```

Description MATLAB provides a set of functions that enable you to access the HDF library developed and supported by the National Center for Supercomputing Applications (NCSA). MATLAB supports all or a portion of these HDF interfaces: SD, V, VS, AN, DRF8, DF24, H, HE, and HD.

To use these functions you must be familiar with the HDF library. Documentation for the library is available on the NCSA HDF Web page at `http://hdf.ncsa.uiuc.edu`. MATLAB additionally provides extensive command line help for each of the provided functions.

This table lists the interface-specific HDF functions in MATLAB.

Function	Interface
`hdfan`	Multifile annotation
`hdfdf24`	24-bit raster image
`hdfdfr8`	8-bit raster image
`hdfgd`	HDF-EOS GD interface
`hdfh`	HDF H interface
`hdfhd`	HDF HD interface
`hdfhe`	HDF HE interface
`hdfml`	Gateway utilities
`hdfpt`	HDF-EOS PT interface
`hdfsd`	Multifile scientific data set
`hdfsw`	HDF-EOS SW interface
`hdfv`	Vgroup
`hdfvf`	Vdata VF functions

hdf

Function	Interface
hdfvh	Vdata VH functions
hdfvs	Vdata VS functions

See Also

imfinfo, imread, imwrite, int8, int16, int32, single, uint8, uint16, uint32

Purpose Display help for MATLAB functions in the Command Window

Syntax

```
help
help /
help function
help toolbox/
help toolbox/function
```

Description `help` lists all primary help topics in the Command Window. Each main help topic corresponds to a directory name on MATLAB's search path.

`help /` lists all of the operators and special characters, along with their descriptions.

`help function` displays M-file help, a brief description and the syntax, for `function` in the Command Window. If `function` is overloaded, `help` displays the M-file help for the first `function` found on the search path, and lists the overloaded functions.

`help toolbox/` displays the contents file for the specified directory, `toolbox`. It is not necessary to give the full pathname of the directory; the last component, or the last several components, is sufficient.

`help toolbox/function` displays the M-file help for `function` that belongs to the specified `toolbox`.

Note M-file help displayed in the Command Window uses uppercase characters for the function and variable names to make them stand out from the rest of the text. When typing function names, however, always use the corresponding lowercase characters since MATLAB is case sensitive and all function names are actually in lowercase.

Remarks

Creating Online Help for Your Own M-Files

MATLAB's help system, like MATLAB itself, is highly extensible. You can write help descriptions for your own M-files and toolboxes using the same self-documenting method that MATLAB's M-files and toolboxes use.

The help function lists all help topics by displaying the first line (the H1 line) of the contents files in each directory on MATLAB's search path. The contents files are the M-files named Contents.m within each directory.

Typing help topic, where topic is a directory name, displays the comment lines in the Contents.m file located in that directory. If a contents file does not exist, help displays the H1 lines of all the files in the directory.

Typing help topic, where topic is a function name, displays help for the function by listing the first contiguous comment lines in the M-file topic.m.

Create self-documenting online help for your own M-files by entering text on one or more contiguous comment lines, beginning with the second line of the file (first line if it is a script). For example, an abridged version of the M-file angle.m provided with MATLAB could contain

```
function p = angle(h)
% ANGLE Polar angle.
% ANGLE(H) returns the phase angles, in radians, of a matrix
% with complex elements. Use ABS for the magnitudes.
p = atan2(imag(h),real(h));
```

When you execute help angle, lines 2, 3, and 4 display. These lines are the first block of contiguous comment lines. The help system ignores comment lines that appear later in an M-file, after any executable statements or after a blank line.

The first comment line in any M-file (the H1 line) is special. It should contain the function name and a brief description of the function. The lookfor function searches and displays this line, and help displays these lines in directories that do not contain a Contents.m file.

Creating Contents Files for Your Own M-File Directories

A Contents.m file is provided for each M-file directory included with the MATLAB software. If you create directories in which to store your own M-files, you should create Contents.m files for them too. To do so, simply follow the format used in an existing Contents.m file.

Examples

Typing

```
help datafun
```

displays help for the datafun directory.

Typing

```
help fft
```

displays help for the `fft` function.

To prevent long descriptions from scrolling off the screen before you have time to read them, enter `more on`, and then enter the `help` function.

See Also

`doc`, `helpbrowser`, `helpwin`, `lookfor`, `more`, `partialpath`, `path`, `what`, `which`

helpbrowser

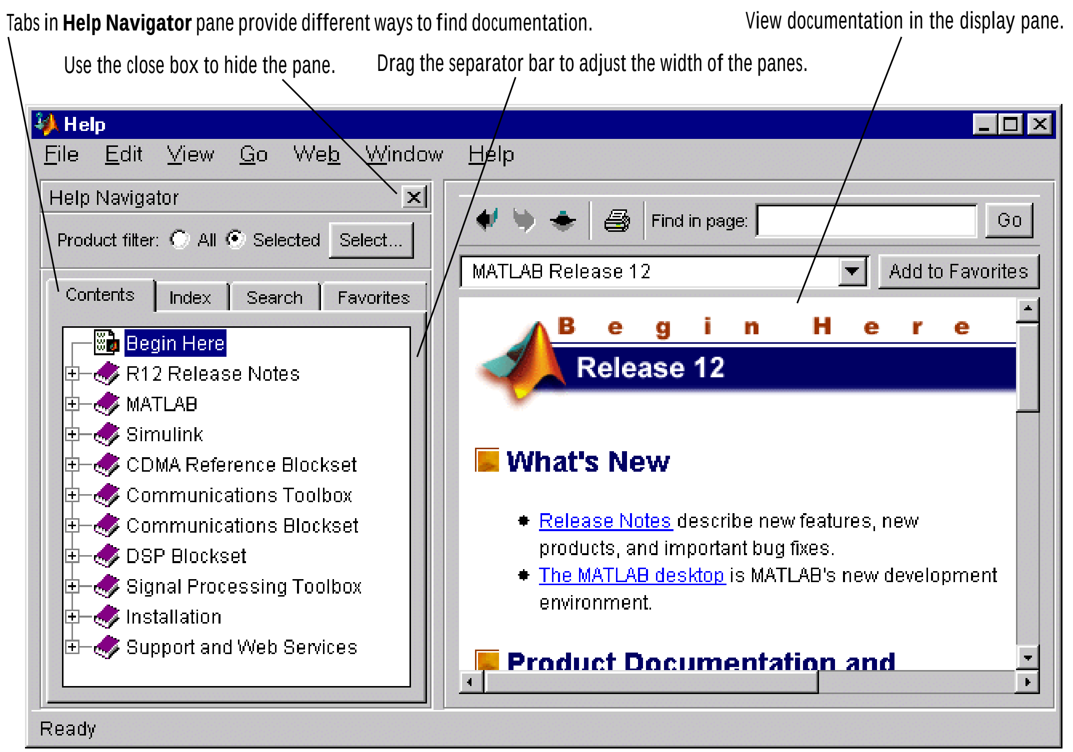
Purpose Display the MATLAB Help browser, providing access to extensive online help

Graphical Interface As an alternative to the `helpbrowser` function, select **Help** from the **View** menu or click the help button on the toolbar in the MATLAB desktop.

Syntax

```
helpbrowser
```

Description `helpbrowser` displays the Help browser, providing direct access to a comprehensive library of online help, including reference pages and manuals. If the Help browser was previously opened in the current session, it shows the last page viewed; otherwise it shows the “Begin Here” page. See “Using the Help Browser” for details.



See Also doc, docopt, help, helpdesk, helpwin, lookfor, web

helpdesk

Purpose Display the Help browser

Syntax `helpdesk`

Description `helpdesk` displays the Help browser and shows the “Begin Here” 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.

See Also `helpbrowser`

Purpose

Create a help dialog box

Syntax

```
helpdlg
helpdlg('helpstring')
helpdlg('helpstring','dlgname')
h = helpdlg(...)
```

Description

`helpdlg` creates a help dialog box or brings the named help dialog box to the front.

`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

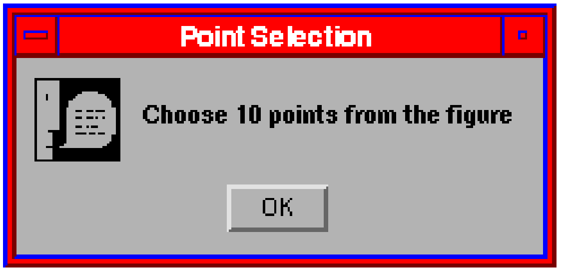
MATLAB wraps the text in `'helpstring'` to fit the width of the dialog box. The dialog box remains on your screen until you press the OK button or the **Return** key. After pressing the button, the help dialog box disappears.

Examples

The statement,

```
helpdlg('Choose 10 points from the figure','Point Selection');
```

displays this dialog box:



See Also `dialog`, `errordlg`, `questdlg`, `warndlg`

Purpose Display M-file help and 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 access M-file help for the functions. You cannot follow links in the `helpwin` list of functions if MATLAB is busy.

`helpwin topic` displays help information for the topic in the Help browser. If `topic` is a directory, it displays all functions in the directory. If `topic` is a function, it displays M-file help for that function. From the page, you can access a list of directories (the **Default Topics** link) as well as the reference page help for the function (the **Go to online doc** link). You cannot follow links in the `helpwin` list of functions if MATLAB is busy.

Examples Typing

```
helpwin datafun
```

displays the functions in the `datafun` directory and a brief description of each.

Typing

```
helpwin fft
```

displays the M-file help for the `fft` function in the Help browser.

See Also `doc`, `docopt`, `help`, `helpbrowser`, `lookfor`, `web`

hess

Purpose Hessenberg form of a matrix

Syntax

```
[P,H] = hess(A)
H = hess(A)
```

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))`.

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 `hess` uses 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`)

Matrix A	Routine
Complex Hermitian	`ZHETRD` `ZHETRD`, `ZUNGTR` (with output `P`)
Complex non-Hermitian	`ZGEHRD` `ZGEHRD`, `ZUNGHR` (with output `P`)

See Also

`eig`, `qz`, `schur`

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*, Third Edition, SIAM, Philadelphia, 1999.

hex2dec

Purpose Hexadecimal to decimal number conversion

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 Hexadecimal to double number conversion

Syntax `f = hex2num('`*hex_value*`')`

Description `f = hex2num('`*hex_value*`')` converts *hex_value* to the IEEE double-precision floating-point number it represents. `NaN`, `Inf`, and denormalized numbers are all handled correctly. Fewer than 16 characters are padded on the right with zeros.

Examples

```
f = hex2num('400921fb54442d18')

f =

    3.14159265358979
```

See Also `format`, `hex2dec`, `sprintf`

hgload

Purpose Loads Handle Graphics object from a file

Syntax

```
h = hgload('filename')
h = hgload('filename','all')
```

Description

`h = hgload('filename')` loads a handle graphics object and its children if any from the file specified by `filename`. If `filename` contains no extension, then MATLAB adds the "`.fig`" extension.

`h = hgload('filename','all')` overrides the default behavior, which does not reload non-serializable objects saved in the file. These objects include the default toolbars and default menus.

Non-serializable objects 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` insures that any non-serializable objects contained in the file are also reloaded.

Note that by default, `hgsave` excludes non- serializable objects from the fig-file unless you use the `all` flag.

See Also

`hgsave`, `open`

Purpose Saves a Handle Graphics object hierarchy to a file

Syntax

```
hgsave('filename')
hgsave(h,'filename')
hgsave('filename','all')
```

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 MATLAB adds the extension ".`fig`". If `h` is a vector, none of the handles in `h` may be ancestors or descendents of any other handles in `h`.

`hgsave('filename','all')` overrides the default behavior, which does not save non-serializable objects. Non-serializable 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` insures that non-serializable objects are also saved.

Note: the default behavior of `hgload` is to ignore non- serializable objects in the file at load time. This behavior can be overwritten using the `all` argument with `hgload`.

See Also `hgload`, `open`

hidden

Purpose Remove hidden lines from a 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`

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
```

Algorithm See the M-file for a good example of efficient MATLAB programming where conventional `for` loops are replaced by vectorized statements.

See Also `invhilb`

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

Purpose

Histogram plot

Syntax

```
n = hist(Y)
n = hist(Y,x)
n = hist(Y,nbins)
[n,xout] = hist(...)
```

Description

A histogram shows the distribution of data values.

`n = hist(Y)` bins the elements in `Y` into 10 equally spaced containers and returns the number of elements in each container. If `Y` is a matrix, `hist` works down the columns.

`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`. 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, `hist` 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`.

Remarks

All elements in vector `Y` or in one column of matrix `Y` are grouped according to their numeric range. Each group is shown as one bin.

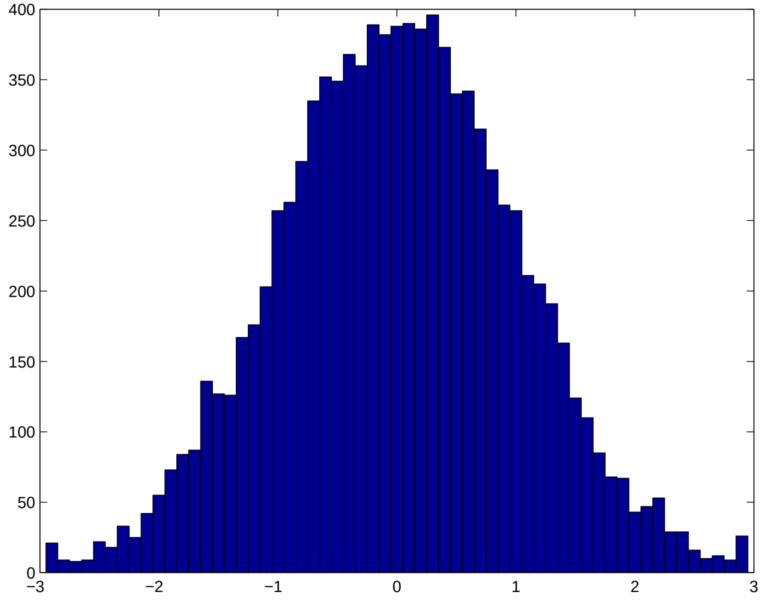
The histogram's *x*-axis reflects the range of values in `Y`. The histogram's *y*-axis shows the number of elements that fall within the groups; therefore, the *y*-axis ranges from `0` to the greatest number of elements deposited in any bin.

The histogram is created with a patch graphics object. If you want to change the color of the graph, you can set patch properties. See the "Example" section for more information. By default, the graph color is controlled by the current colormap, which maps the bin color to the first color in the colormap.

Examples

Generate a bell-curve histogram from Gaussian data.

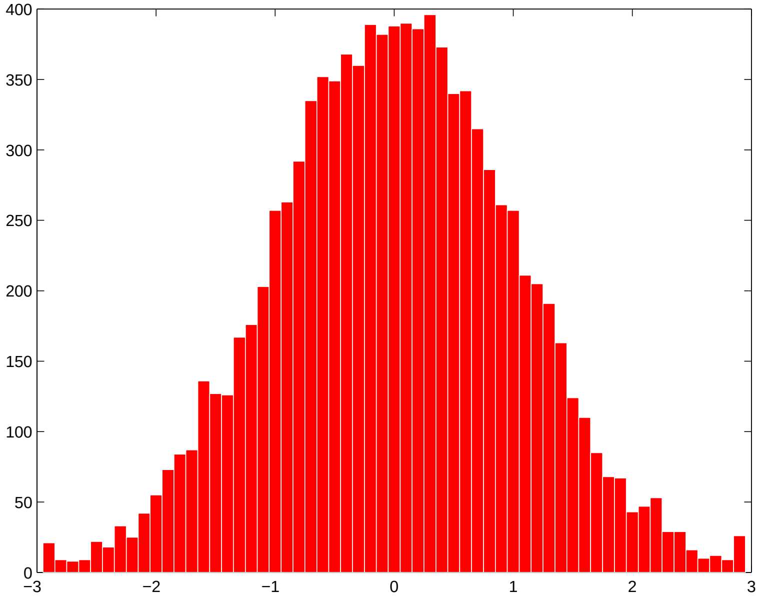
```
x = -2.9:0.1:2.9;
y = randn(10000,1);
hist(y,x)
```



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`, `patch`, `stairs`

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 non-decreasing values). `n` is a `length(edges)` vector containing these counts.

`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 non-singleton 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.

See Also `hist`

hold

Purpose Hold current graph in the figure

Syntax

```
hold on
hold off
hold
```

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` toggles the hold state between adding to the graph and replacing the graph.

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.

Purpose Move the cursor to the upper left corner of the Command Window

Syntax

```
home
```

Description `home` moves the cursor to the upper-left corner of the Command Window and clears the screen. You can use the scroll bar to see the history of previous functions.

Examples Use `home` in an M-file to return the cursor to the upper-left corner of the screen.

See Also `clc`

hsv2rgb

Purpose

Convert HSV colormap to RGB colormap

Syntax

```
M = hsv2rgb(H)
```

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.

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([hue saturation value])` where `hue` is a linear ramp from 0 to 1, and `saturation` and `value` are all 1's.

See Also

`brighten`, `colormap`, `rgb2hsv`

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`

Purpose Conditionally execute statements

Syntax

```
if expression
    statements
end
if expression1
    statements
elseif expression2
    statements
else
    statements
end
```

Description `if` conditionally executes statements.

The simple form is:

```
if expression
    statements
end
```

More complicated forms use `else` or `elseif`. Each `if` must be paired with a matching `end`.

Arguments

expression A MATLAB expression, usually consisting of smaller expressions or variables joined by relational operators (==, <, >, <=, >=, or ~=). Two examples are: `count < limit` and `(height - offset) >= 0`.
Expressions may also include logical functions, as in: `isreal(A)`.
Simple expressions can be combined by logical operators (&,|,~) into compound expressions such as: `(count < limit) & ((height - offset) >= 0)`.

statements One or more MATLAB statements to be executed only if the *expression* is *true* (or nonzero). See Examples for information about how nonscalar variables are evaluated.

Examples Here is an example showing `if`, `else`, and `elseif`:

```
for i = 1:n
    for j = 1:n
        if i == j
            a(i,j) = 2;
        elseif abs([i j]) == 1
            a(i,j) = 1;
        else
            a(i,j) = 0;
        end
    end
end
```

Such expressions are evaluated as *false* unless every element-wise comparison evaluates as *true*. Thus, given matrices `A` and `B`:

```
A =                 B =
     1     0             1     1
     2     3             3     4
```

The expression:

`A < B`	Evaluates as *false*	Since `A(1,1)` is not less than `B(1,1)`.
`A < (B+1)`	Evaluates as *true*	Since no element of `A` is greater than the corresponding element of `B`.
`A & B`	Evaluates as *false*	Since `A(1,2) | B(1,2)` is *false*.
`5 > B`	Evaluates as *true*	Since every element of `B` is less than `5`.

See Also

`break`, `else`, `end`, `for`, `return`, `switch`, `while`

Purpose

Inverse one-dimensional fast Fourier transform

Syntax

```
y = ifft(X)
y = ifft(X,n)
y = ifft(X,[],dim)
y = ifft(X,n,dim)
```

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.

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`.

For any `X`, `ifft(fft(X))` equals `X` to within roundoff error. If `X` is real, `ifft(fft(X))` may have small imaginary parts.

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.

See Also

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

`fft`, `ifft2`, `ifftn`, `ifftshift`

Purpose Inverse two-dimensional fast Fourier transform

Syntax

```
Y = ifft2(X)
Y = ifft2(X,m,n)
```

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`.

`Y = ifft2(X,m,n)` returns the `m`-by-`n` inverse fast Fourier transform of matrix `X`.

For any `X`, `ifft2(fft2(X))` equals `X` to within roundoff error. If `X` is real, `ifft2(fft2(X))` may have small imaginary parts.

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.

See Also `dftmtx` and `freqz` in the Signal Processing Toolbox, and:

`fft2`, `fftshift`, `ifft`, `ifftn`, `ifftshift`

ifftn

Purpose

Inverse multidimensional fast Fourier transform

Syntax

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

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`.

`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`.

Remarks

For any `X`, `ifftn(fftn(X))` equals `X` within roundoff error. If `X` is real, `ifftn(fftn(X))` may have small imaginary parts.

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.

See Also

`fftn`, `ifft`, `ifft2`, `ifftshift`

Purpose Inverse FFT shift

Syntax

```
ifftshift(X)
ifftshift(X,dim)
```

Description `ifftshift(X)` undoes the results of `fftshift`.

If `X` is a vector, `iffshift(X)` swaps the left and right halves of `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`.

See Also `fft`, `fft2`, `fftn`, `fftshift`

im2frame

Purpose

Convert indexed or truecolor image into movie format

Syntax

```
F = im2frame(X,Map)
F = im2frame(truecolor_array)
```

Description

`F = im2frame(X,Map)` converts the indexed image `X` and associated colormap `Map` into a movie frame F. `X` can be of type `double` or `uint8` and must be specified with a colormap (`Map`). The colormap must have three columns, and must contain double-precision values between 0 and 1 (no NaNs or Infs). The values in `X` must contain legal indices into that colormap.

`uint8` images must have 0-based indices (so if the colormap is 7 rows long, `X` must contain values between 0 and 6). If it is `double`, it must contain 1-based indices into the colormap (so if the colormap is 7 rows long, X must contain values between 1 and 7). NaNs and Infs are illegal, as are negative

values or 0.

`F = im2frame(`*truecolor_array*`)` converts the truecolor image into a movie frame F. *truecolor_array* must be an m-by-n-by-3 array of RGB values. It can be of type `double` or `uint8` with values in the range:

- `0 <= double <= 1`
- `0 <= uint8 <= 255`

Example

Use `im2frame` to convert a sequence of images into a movie.

```
F(1) = im2frame(X1,map);
F(2) = im2frame(X2,map);
...
F(n) = im2frame(Xn,map);
movie(F)
```

See Also

`getframe`, `frame2im`, `movie`

Purpose Imaginary part of a complex number

Syntax

```
Y = imag(Z)
```

Description `Y = imag(Z)` returns the imaginary part of the elements of array `Z`.

Examples

```
imag(2+3i)

ans =

    3
```

See Also `conj`, `i`, `j`, `real`

image

Purpose Display image object

Syntax

```
image(C)
image(x,y,C)
image(...,'PropertyName',PropertyValue,...)
image('PropertyName',PropertyValue,...)  Formal synatx – PN/PV only
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.

`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 , 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 MATLAB 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 method than the double-precision (64-bit) floating-point numbers that MATLAB typically uses. However, it is necessary for MATLAB to interpret

8-bit and 16-bit image data differently from 64-bit data. This table summarizes these differences.

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] (`unit8`) or [0, 65535] (`uint16`); colormap is an m-by-3 array of floating-point values in the range [0, 1]
truecolor (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] (`unit8`) or [0, 65535] (`uint16`)

Indexed Images

In an indexed image of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. In a `uint8` or `uint16` indexed image, there is an offset; the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on.

If you want to convert a `uint8` or `uint16` indexed image to `double`, you need to add 1 to the result. For example,

```
X64 = double(X8) + 1;
```

or

```
X64 = double(X16) + 1;
```

To convert from `double` to `uint8` or `unit16`, 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));
```

The order of the operations must be as shown in these examples, because you cannot perform mathematical operations on uint8 or uint16 arrays.

When you write an indexed image using imwrite, MATLAB automatically converts the values if necessary.

Colormaps

Colormaps in MATLAB are alway 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 does not support colormaps with integer values. imread and imwrite automatically convert colormap values when reading and writing files.

True Color Images

In a truecolor image of class double, the data values are floating-point numbers in the range [0, 1]. In a truecolor image of class uint8, the data values are integers in the range [0, 255] and for truecolor image of class uint16 the data values are integers in the range [0, 65535]

If you want to convert a truecolor image from one data type to the other, you must rescale the data. For example, this statement converts a uint8 truecolor image to double,

```
RGB64 = double(RGB8)/255;
```

or for uint16 images,

```
RGB64 = double(RGB16)/65535;
```

This statement converts a double truecolor image to uint8.

```
RGB8 = uint8(round(RGB64*255));
```

or for uint16 images,

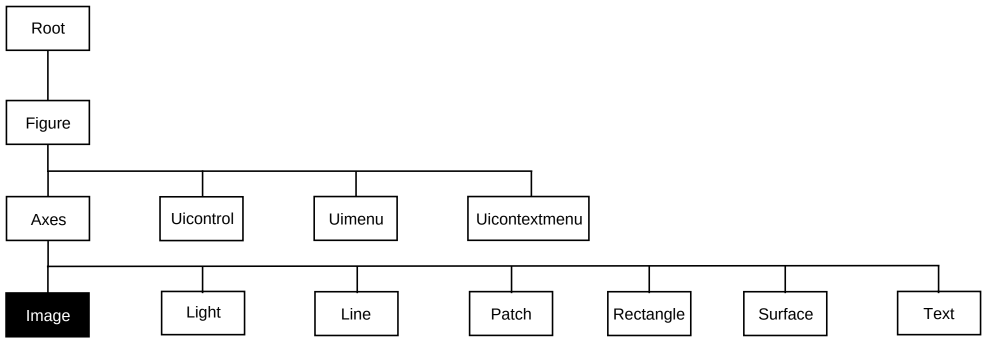
```
RGB16 = uint16(round(RGB64*65535));
```

The order of the operations must be as shown in these examples, because you cannot perform mathematical operations on uint8 or uint16 arrays.

When you write a truecolor image using imwrite, MATLAB automatically converts the values if necessary.

image

Object Hierarchy



Setting Default Properties

You can set default image properties on the axes, figure, and root 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.

The following table lists all image properties and provides a brief description of each. The property name links take you to an expanded description of the properties.

Property Name	Property Description	Property Value
Data Defining the Object		
`CData`	The image data	Values: matrix or m-by-n-by-3 array Default: enter `image;axis image ij` and see
`CDataMapping`	Specify the mapping of data to colormap	Values: `scaled`, `direct` Default: `direct`

Property Name	Property Description	Property Value
XData	Control placement of image along *x*-axis	Values: [min max] Default: [1 size(CData,2)]
YData	Control placement of image along *y*-axis	Values: [min max] Default: [1 size(CData,1)]
Specifying Transparency		
alphadata	Transparency data	m-by-n matrix of double or uint8 Default: 1 (opaque)
alphadatamapping	Transparency mapping method	none, direct, scaled Default: none
Controlling the Appearance		
Clipping	Clipping to axes rectangle	Values: on, off Default: on
EraseMode	Method of drawing and erasing the image (useful for animation)	Values: normal, none, xor, background Default: normal
SelectionHighlight	Highlight image when selected (Selected property set to on)	Values: on, off Default: on
Visible	Make the image visible or invisible	Values: on, off Default: on
Controlling Access to Objects		
HandleVisibility	Determines if and when the the line's handle is visible to other functions	Values: on, callback, off Default: on
HitTest	Determine if image can become the current object (see the figure CurrentObject property)	Values: on, off Default: on
General Information About the Image		
Children	Image objects have no children	Values: [] (empty matrix)

Property Name	Property Description	Property Value
Parent	The parent of an image object is always an axes object	Value: axes handle
Selected	Indicate whether image is in a “selected” state.	Values: on, off Default: on
Tag	User-specified label	Value: any string Default: '' (empty string)
Type	The type of graphics object (read only)	Value: the string 'image'
UserData	User-specified data	Value: any matrix Default: [] (empty matrix)
Properties Related to Callback Routine Execution		
BusyAction	Specify how to handle callback routine interruption	Values: cancel, queue Default: queue
ButtonDownFcn	Define a callback routine that executes when a mouse button is pressed on over the image	Values: string Default: empty string
CreateFcn	Define a callback routine that executes when an image is created	Values: string Default: empty string
DeleteFcn	Define a callback routine that executes when the image is deleted (via close or delete)	Values: string Default: empty string
Interruptible	Determine if callback routine can be interrupted	Values: on, off Default: on (can be interrupted)
UIContextMenu	Associate a context menu with the image	Values: handle of a uicontextmenu

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 element in the image data. The AlphaData can be of class double or uint8.

MATLAB determines the transparency in one of three ways:

- Using the elements of AlphaData as transparency values (AlphaDataMapping set to none, the default).
- 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).

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 unit8 integers, then the indexing begins at 0 (i.e., MATLAB maps a value of 0 to the first alpha value in the alphamap).

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, subsequently invoked callback routes 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

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the image object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

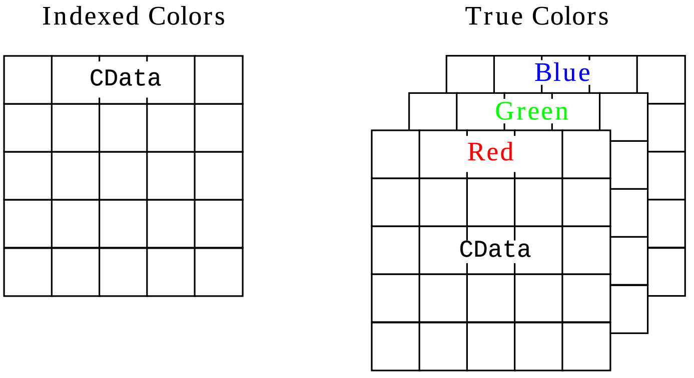
`CData` matrix or m-by-n-by-3 array

The image data. A matrix 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)
- 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.

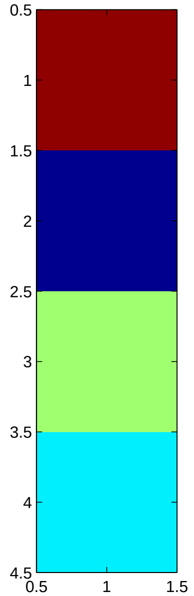
A true color specification for `CData` requires an m-by-n-by-3 array of RGB values. The first page contains the red component, the second page the green component, and the third page the blue component of each element in the image. RGB values range from 0 to 1. The following picture illustrates the relative 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.

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 display outside the axes rectangle. For example, if you create an image, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger image, it extends beyond the axis limits.

CreateFcn string

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. 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`.

DeleteFcn string

Delete image callback routine. A callback routine that executes when you delete the image object (i.e., when you issue a `delete` command or clear the axes or figure containing the image). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

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

EraseMode `{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase image objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects redraw 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 image when it is moved or changed. 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 image 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 image. However, the image's color depends on the color of whatever is beneath it on the display.
- `background` – Erase the image 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 image, but images are always properly colored.

Printing with Non-normal 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., XORing 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 non-normal 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 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 provide 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 evaling a user-typed string), and so temporarily hides its own handles during the execution of that function.

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

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

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

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

`HitTest` `{on} | off`

Selectable by mouse click. `HitTest` determines if the image 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 image. If `HitTest` is `off`, clicking on the image selects the object below it (which maybe the axes containing it).

`Interruptible` `{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an image 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.

`Parent` handle of parent axes

Image's parent. The handle of the image object's parent axes. You can move an image object to another axes by changing this property to the new axes handle.

`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 highlight when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

Tag string

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

Type string (read only)

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

UserData matrix

User specified data. This property can be any data you want to associate with the image object. The image does not use this property, but you can access it using `set` and `get`.

Visible `{on} | off`

Image visibility. By default, image objects are visible. Setting this property to `off` prevents the image 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 elements one unit apart.

See Also

`colormap`, `imfinfo`, `imread`, `imwrite`, `pcolor`, `newplot`, `surface`

Displaying Bit-Mapped Images in Visualzation Techniques.

Purpose Scale data and display an image object

Syntax

```
imagesc(C)
imagesc(x,y,C)
imagesc(...,clims)
h = imagesc(...)
```

Description The `imagesc` function scales image data to the full range of the current colormap and displays the image. (See the illustration on the following page.)

`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`.

`imagesc(...,clims)` normalizes the values in `C` to the range specified by `clims` and displays `C` as an image. `clims` is a two-element vector that limits the range of data values in `C`. These values map to the full range of values in the current colormap.

`h = imagesc(...)` returns the handle for an image graphics object.

Remarks `x` and `y` do not affect the elements in `C`; they only affect the annotation of the axes. If `length(x) > 2` or `length(y) > 2`, `imagesc` ignores all except the first and last elements of the respective vector.

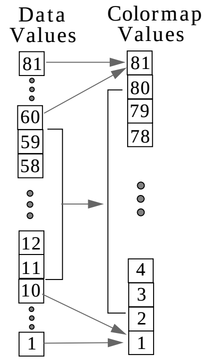
Algorithm `imagesc` creates an image with `CDataMapping` set to `scaled`, and sets the axes `CLim` property to the value passed in `clims`.

Examples

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 to the right.

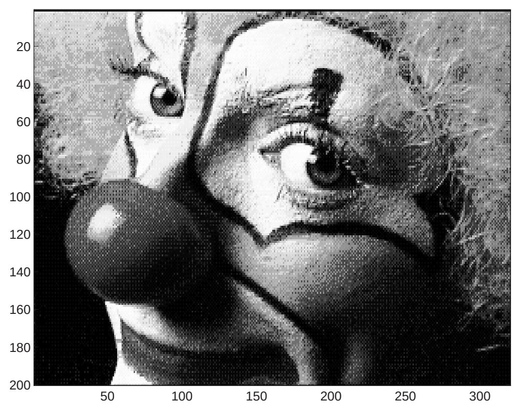


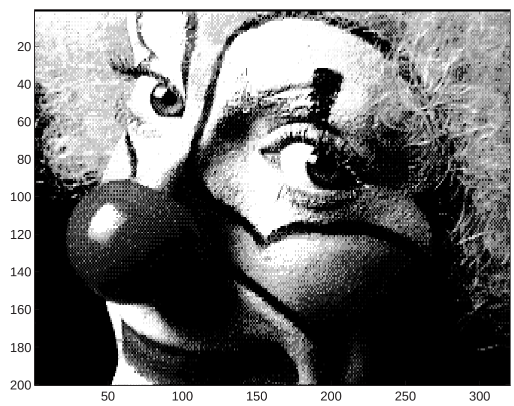
The left image maps to the gray colormap using the statements

```
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`, `colorbar`

Purpose Information about graphics file

Syntax

```
info = imfinfo(filename,fmt)
info = imfinfo(filename)
```

Description

`info = imfinfo(filename,`*`fmt`*`)` returns a structure whose fields contain information about an image in a graphics file. `filename` is a string that specifies the name of the graphics file, and `fmt` is a string that specifies the format of the file. The file must be in the current directory or in a directory on the MATLAB path. If `imfinfo` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

This table lists the possible values for *fmt*.

Format	File Type
'bmp'	Windows Bitmap (BMP)
'cur'	Windows Cursor resources (CUR)
'hdf'	Hierarchical Data Format (HDF)
'ico'	Windows Icon resources (ICO)
'jpg' or 'jpeg'	Joint Photographic Experts Group (JPEG)
'pcx'	Windows Paintbrush (PCX)
'png'	Portable Network Graphics (PNG)
'tif' or 'tiff'	Tagged Image File Format (TIFF)
'xwd'	X Windows Dump (XWD)

If `filename` is a TIFF or HDF file containing more than one image, `info` is a structure array with one element (i.e., an individual structure) for each image in the file. For example, `info(3)` would contain information about the third image in the file.

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 fields and describes their values.

Field	Value
`Filename`	A string containing the name of the file; if the file is not in the current directory, the string contains the full pathname of the file
`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 JPEG and TIFF files, the three-letter variant is returned
`FormatVersion`	A string or number describing the version of the format
`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; either `'truecolor'` for a truecolor RGB image, `'grayscale'` for a grayscale intensity image, or `'indexed'` for an indexed image

`info = imfinfo(filename)` attempts to infer the format of the file from its contents.

Example

```
info = imfinfo('canoe.tif')

info =

                        Filename:'canoe.tif'
```

```
                     FileModDate: '25-Oct-1996 22:10:39'
                        FileSize: 69708
                          Format: 'tif'
                   FormatVersion: []
                           Width: 346
                          Height: 207
                        BitDepth: 8
                       ColorType: 'indexed'
                 FormatSignature: [73 73 42 0]
                       ByteOrder: 'little-endian'
                  NewSubfileType: 0
                   BitsPerSample: 8
                     Compression: 'PackBits'
       PhotometricInterpretation: 'RGB Palette'
                    StripOffsets: [  9x1  double]
                 SamplesPerPixel: 1
                    RowsPerStrip: 23
                 StripByteCounts: [  9x1  double]
                     XResolution: 72
                     YResolution: 72
                  ResolutionUnit: 'Inch'
                        Colormap: [256x3  double]
             PlanarConfiguration: 'Chunky'
                       TileWidth: []
                      TileLength: []
                     TileOffsets: []
                  TileByteCounts: []
                     Orientation: 1
                       FillOrder: 1
                GrayResponseUnit: 0.0100
                  MaxSampleValue: 255
                  MinSampleValue: 0
                    Thresholding: 1
```

See Also

`imread`, `imwrite`

import

Purpose

Add a package or class to the current Java import list for the MATLAB command environment or for the calling function

Syntax

```
import package_name.*
import class_name
import cls_or_pkg_name1 cls_or_pkg_name2...
import
L = import
```

Description

`import` *package_name*`.*` adds all the classes in *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` command operates exclusively on the import list of the function from which it is invoked. When invoked at the command prompt, `import` uses the import list for the MATLAB command environment. If `import` is used in a script invoked from a function, it affects the import list of the function. If `import` is used in a script that is invoked from the command prompt, it affects the import list for the command environment.

The import list of a function is persistent across calls to that function and is only cleared when the function is cleared.

To clear the current import list, use the following command.

```
clear import
```

This command may only be invoked at the command prompt. Attempting to use `clear import` within a function results in an error.

Remarks

The only reason for using `import` is to allow your code to refer to each imported class with the immediate 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

This example shows importing and using the single class, `java.lang.String`, and two complete packages, `java.util` and `java.awt`.

```
import java.lang.String
import java.util.* java.awt.*
f = Frame;                % Create java.awt.Frame object
s = String('hello');      % Create java.lang.String object
methods Enumeration       % List java.util.Enumeration methods
```

See Also

`clear`

importdata

Purpose Load data from disk file.

Syntax

```
importdata('filename')
A = importdata('filename')
importdata('filename','delimiter')
```

Description `importdata('filename')` loads data from `filename` into the workspace.

`A = importdata('filename')` loads data from `filename` into `A`.

`A = importdata('filename','delimiter')` loads data from `filename` using delimiter as the column separator (if text). Use `'\t'` for tab.

Remarks `importdata` looks at the file extension to determine which helper function to use. If it can recognize the file extension, `importdata` calls the appropriate helper function, specifying the maximum number of output arguments. If it cannot recognize the file extension, `importdata` calls `finfo` to determine which helper function to use. If no helper function is defined for this file extension, `importdata` treats the file as delimited text. `importdata` removes from the result empty outputs returned from the helper function.

Examples

```
s = importdata('ding.wav')
s =

   data: [11554x1 double]
     fs: 22050
```

See Also `load`

Purpose

Read image from graphics files

Syntax

```
A = imread(filename,fmt)
[X,map] = imread(filename,fmt)
[...] = imread(filename)
[...] = imread(...,idx)    (CUR, ICO, and TIFF only)
[...] = imread(...,ref)    (HDF only)
[...] = imread(...,'BackgroundColor',BG)  (PNG only)
[A,map,alpha] = imread(...)  (PNG only)
```

Description

`A = imread(filename,`*`fmt`*`)` reads a grayscale or truecolor image named `filename` into `A`. If the file contains a grayscale intensity image, `A` is a two-dimensional array. If the file contains a truecolor (RGB) image, `A` is a three-dimensional (m-by-n-by-3) array.

`[X,map] = imread(filename,`*`fmt`*`)` reads the indexed image in `filename` into `X` and its associated colormap into `map`. The colormap values are rescaled to the range [0,1]. `A` and `map` are two-dimensional arrays.

`[...] = imread(filename)` attempts to infer the format of the file from its content.

`filename` is a string that specifies the name of the graphics file, and *fmt* is a string that specifies the format of the file. If the file is not in the current directory or in a directory in the MATLAB path, specify the full pathname for a location on your system. If `imread` cannot find a file named `filename`, it looks for a file named `filename.fmt`. If you do not specify a string for *fmt*, the toolbox will try to discern the format of the file by checking the file header.

This table lists the possible values for *fmt*.

Format	**File Type**
`'bmp'`	Windows Bitmap (BMP)
`'cur'`	Windows Cursor resources (CUR)
`'hdf'`	Hierarchical Data Format (HDF)
`'ico'`	Windows Icon resources (ICO)

Format	File Type
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)
`'pcx'`	Windows Paintbrush (PCX)
`‘png’`	Portable Network Graphics (PNG)
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF)
`'xwd'`	X Windows Dump (XWD)

Special Case Syntax:

TIFF-Specific Syntax

`[...] = 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.

PNG-Specific Syntax

The discussion in this section is only relevant to PNG files that contain transparent pixels. A PNG file does not necessarily contain transparency data. Transparent pixels, when they exist, will be identified by one of two components: a *transparency chunk* or an *alpha channel*. (A PNG file can only have one of these components, not both.)

The transparency chunk identifies which pixel values will be treated as transparent, e.g., if the value in the transparency chunk of an 8-bit image is 0.5020, all pixels in the image with the color 0.5020 can be displayed as transparent. An alpha channel is an array with the same number of pixels as are in the image, which indicates the transparency status of each corresponding pixel in the image (transparent or nontransparent).

Another potential PNG component related to transparency is the *background color chunk*, which (if present) defines a color value that can be used behind all transparent pixels. This section identifies the default behavior of the toolbox for reading PNG images that contain either a transparency chunk or an alpha channel, and describes how you can override it.

Case 1. You do not ask to output the alpha channel and do not specify a background color to use. For example,

```
[A,map] = imread(filename);
A = imread(filename);
```

If the PNG file contains a background color chunk, the transparent pixels will be composited against the specified background color.

If the PNG file does not contain a background color chunk, the transparent pixels will be composited against `0` for grayscale (black), `1` for indexed (first color in map), or `[0 0 0]` for RGB (black).

Case 2. You do not ask to output the alpha channel but you specify the background color parameter in your call. For example,

```
[...] = imread(...,'BackgroundColor',bg);
```

The transparent pixels will be composited against the specified color. The form of `bg` depends on whether the file contains an indexed, intensity (grayscale), or RGB image. If the input image is indexed, `bg` should be an integer in the range `[1,P]` where `P` is the colormap length. If the input image is intensity, `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].

There is one exception to the toolbox's behavior of using your background color. If you set background to `'none'` no compositing will be performed. For example,

```
[...] = imread(...,'Back','none');
```

Note If you specify a background color, you *cannot* output the alpha channel.

Case 3. You ask to get the alpha channel as an output variable. For example,

```
[A,map,alpha] = imread(filename);
[A,map,alpha] = imread(filename,fmt);
```

No compositing is performed; the alpha channel will be stored separately from the image (not merged into the image as in cases 1 and 2). This form of `imread` returns the alpha channel if one is present, and also returns the image and any associated colormap. If there is no alpha channel, `alpha` returns `[]`. If there is no colormap, or the image is grayscale or truecolor, map may be empty.

HDF-Specific Syntax

`[...] = imread(...,ref)` reads in one image from a multi-image HDF 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 HDF file the reference numbers do not necessarily correspond to the order of the images in the file. You can use `imfinfo` to match up image order with reference number.) If you omit this argument, `imread` reads the first image in the file.

CUR- and ICO-Specific Syntax

`[...] = 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 the Image Processing Toolbox, you can use the `imresize` function.

Format Support

This table summarizes the types of images that `imread` can read.

Format	Variants
BMP	1-bit, 4-bit, 8-bit, and 24-bit uncompressed images; 4-bit and 8-bit run-length encoded (RLE) images
CUR	1-bit, 4-bit, and 8-bit uncompressed images
HDF	8-bit raster image datasets, with or without associated colormap; 24-bit raster image datasets
ICO	1-bit, 4-bit, and 8-bit uncompressed images

Format	Variants
JPEG	Any baseline JPEG image (8 or 24-bit); JPEG images with some commonly used extensions
PCX	1-bit, 8-bit, and 24-bit images
PNG	Any PNG image, including 1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit indexed images; 24-bit and 48-bit RGB images
TIFF	Any baseline TIFF image, including 1-bit, 8-bit, and 24-bit uncompressed images; 1-bit, 8-bit, 16-bit, and 24-bit images with packbits compression; 1-bit images with CCITT compression; also 16-bit grayscale, 16-bit indexed, and 48-bit RGB images.
XWD	1-bit and 8-bit ZPixmaps; XYBitmaps; 1-bit XYPixmaps

Class Support

In most of the image file formats supported by `imread`, pixels are stored using eight or fewer bits per color plane. When reading such a file, the class of the output (`A` or `X`) is `uint8`. `imread` also supports reading 16-bit-per-pixel data from TIFF and PNG files; for such image files, the class of the output (`A` or `X`) is `uint16`. Note that 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`.

Remarks

`imread` is a function in MATLAB.

Examples

This example reads the sixth image in a TIFF file.

```
[X,map] = imread('flowers.tif',6);
```

This example reads the fourth image in an HDF file.

```
info = imfinfo('skull.hdf');
[X,map] = imread('skull.hdf',info(4).Reference);
```

This example reads a 24-bit PNG image and sets any of its fully transparent (alpha channel) pixels to red.

```
bg = [255 0 0];
```

```
A = imread('image.png','BackgroundColor',bg);
```

This example returns the alpha channel (if any) of a PNG image.

```
[A,map,alpha] = imread('image.png');
```

This example reads an ICO image, applies a transparency mask, and then displays the image.

```
[a,b,c] = imread('myicon.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
imshow(uint8(d), b2)
```

See Also

`double`, `fread`, `imfinfo`, `imwrite`, `uint8`, `uint16`

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 in `A` to `filename` in the format specified by *fmt*. `A` can be either a grayscale image (M-by-N) or a truecolor image (M-by-N-by-3). If `A` is of class `uint8 or uint16`, `imwrite` writes the actual values in the array to the file. If `A` is of class `double`, `imwrite` rescales the values in the array before writing, using `uint8(round(255*A))`. This operation converts the floating-point numbers in the range [0,1] to 8-bit integers in the range [0,255].

`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)`. (See note below for an exception.) `map` must be a valid MATLAB colormap of class `double`; `imwrite` rescales the values in `map` using `uint8(round(255*map))`. Note that most image file formats do not support colormaps with more than 256 entries.

Note If the image is `double`, and you specify PNG as the output format and a bit depth of 16 bpp, the values in the array will be offset using `uint16(X-1)`.

`imwrite(...,filename)` writes the image to `filename`, inferring the format to use from the filename's extension. The extension must be one of the legal values for `fmt`.

`imwrite(...,Param1,Val1,Param2,Val2...)` specifies parameters that control various characteristics of the output file. Parameter settings can currently be made for HDF, PNG, JPEG, and TIFF files. For example, if you are writing a JPEG file, you can set the "quality" of the JPEG compression. For the lists of parameters available for each format, see the tables below.

`filename` is a string that specifies the name of the output file, and *fmt* is a string that specifies the format of the file.

This table lists the possible values for `fmt`.

Format	File Type
`'bmp'`	Windows Bitmap (BMP)
`'hdf'`	Hierarchical Data Format (HDF)
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)
`'pcx'`	Windows Paintbrush (PCX)
`'png'`	Portable Network Graphics (PNG)
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF)
`'xwd'`	X Windows Dump (XWD)

This table describes the available parameters for HDF files.

Parameter	Values	Default
`'Compression'`	One of these strings: `'none'` (the default), `'rle'`, `'jpeg'`. `'rle'` is valid only for grayscale and indexed images. `'jpeg'` is valid only for grayscale and RGB images.	`'rle'`
`'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.	75
`'WriteMode'`	One of these strings: `'overwrite'` (the default), or `'append'`.	`'overwrite'`

This table describes the available parameters for JPEG files.

Parameter	Values	Default
'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

This table describes the available parameters for TIFF files.

Parameter	Values	Default
'Compression'	One of these strings: 'none', 'packbits', 'ccitt', 'fax3', or 'fax4'. The 'ccitt', 'fax3', and 'fax4' compression schemes are valid for binary images only.	'ccitt' for binary images; 'packbits' for nonbinary images
'Description'	Any string; fills in the ImageDescription field returned by imfinfo.	empty
'Resolution'	A two-element vector containing the XResolution and YResolution, or a scalar indicating both resolutions.	72
'WriteMode'	One of these strings: 'overwrite' or 'append'	'overwrite'

This table describes the available parameters for PNG files.

Parameter	Values	Default
'Author'	A string	Empty
'Description'	A string	Empty
'Copyright'	A string	Empty
'CreationTime'	A string	Empty
'Software'	A string	Empty
'Disclaimer'	A string	Empty

Parameter	Values	Default
'Warning'	A string	Empty
'Source'	A string	Empty
'Comment'	A string	Empty
'InterlaceType'	Either 'none' or 'adam7'	'none'
'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.	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
'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 will represent 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. You cannot specify 'Transparency' and 'Alpha' at the same time.	Empty

Parameter	Values	Default
'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].	Empty
'Gamma'	A nonnegative scalar indicating the file gamma	Empty
'Chromaticities'	An eight-element vector [wx wy rx ry gx gy bx by] that specifies the reference white point and the primary chromaticities	Empty
'XResolution'	A scalar indicating the number of pixels/unit in the horizontal direction	Empty
'YResolution'	A scalar indicating the number of pixels/unit in the vertical direction	Empty
'ResolutionUnit'	Either 'unknown' or 'meter'	Empty
'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].	Empty
'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	Empty

In addition to these PNG parameters, you can use any parameter name that satisfies the PNG specification for keywords, including only printable characters, 80 characters or fewer, and 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.

Format Support

This table summarizes the types of images that `imwrite` can write.

Format	Variants
BMP	8-bit uncompressed images with associated colormap; 24-bit uncompressed images
HDF	8-bit raster image datasets, with or without associated colormap, 24-bit raster image datasets; uncompressed or with RLE or JPEG compression.
JPEG	Baseline JPEG images (8 or 24-bit). **Note:** Indexed images are converted to RGB before writing out JPEG files, because the JPEG format does not support indexed images.
PCX	8-bit images
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 with or without alpha channels
TIFF	Baseline TIFF images, including 1-bit, 8-bit, and 24-bit uncompressed images; 1-bit, 8-bit, and 24-bit images with packbits compression; 1-bit images with CCITT 1D, Group 3, and Group 4 compression.
XWD	8-bit ZPixmaps

Class Support

Most of the supported image file formats store `uint8` data. PNG and TIFF additionally support `uint16` data. For grayscale and RGB images, if the data array is `double`, the assumed dynamic range is [0,1]. The data array is automatically scaled by 255 before being written out as `uint8`. If the data array is `uint8` or `uint16` (PNG and TIFF only), then it is written out without scaling as `uint8` or `uint16`, respectively.

Note If a logical double or uint8 is written to a PNG or TIFF file, it is assumed to be a binary image and will be written with a bit depth of 1.

For indexed images, if the index array is double, then the indices are first converted to zero-based indices by subtracting 1 from each element, and then they are written out as uint8. If the index array is uint8 or uint16 (PNG and TIFF only), then it is written out without modification as uint8 or uint16, respectively. When writing PNG files, you can override this behavior with the 'BitDepth' parameter; see the PNG table in this imwrite reference for details.

Remarks

imwrite is a function in MATLAB.

Example

This example appends an indexed image X and its colormap map to an existing uncompressed multipage HDF file named flowers.hdf.

```
imwrite(X,map,'flowers.hdf','Compression','none',...
'WriteMode','append')
```

See Also

fwrite, imfinfo, imread

ind2rgb

Purpose Convert an indexed image to an RGB image

Syntax `RGB = ind2rgb(X,map)`

Description `RGB = ind2rgb(X,map)` converts thematrix `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-3 array of class `double`.

See Also `image`

Purpose Subscripts from linear index

Syntax

```
[I,J] = ind2sub(siz,IND)
[I1,I2,I3,...,In] = ind2sub(siz,IND)
```

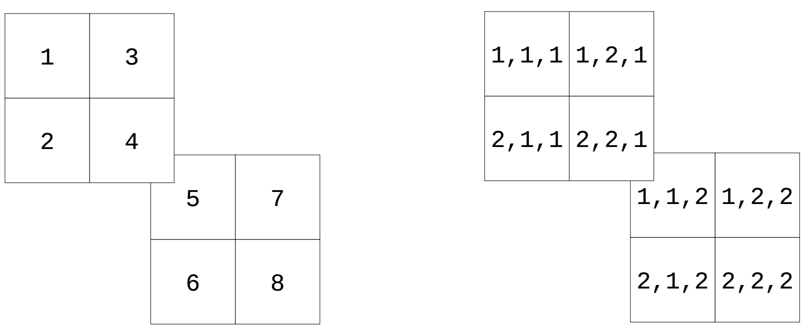
Description The ind2sub command determines the equivalent subscript values corresponding to a single index into an array.

[I,J] = ind2sub(*siz*,IND) returns the arrays I and J containing the equivalent row and column subscripts corresponding to the index matrix IND for a matrix of size *siz*.

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*.

Examples The mapping from linear indexes to subscript equivalents for a 2-by-2-by-2 array is:



See Also sub2ind, find

Inf

Purpose Infinity

Syntax `inf`

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.

Examples `1/0`, `1.e1000`, `2^1000`, and `exp(1000)` all produce `Inf`.

`log(0)` produces `-Inf`.

`Inf-Inf` and `Inf/Inf` both produce `NaN`, Not-a-Number.

See Also `is*`, `NaN`

Purpose

Inferior class relationship

Syntax

```
inferiorto('class1','class2',...)
```

Description

The `inferiorto` function establishes a hierarchy which determines the order in which MATLAB calls object methods.

`inferiorto('class1','class2',...)` invoked within a class constructor method (say `myclass.m`) indicates that `myclass`'s method should not be invoked if a function is called with an object of class `myclass` and one or more objects of class `class1`, `class2`, and so on.

Remarks

Suppose `A` is of class `'class_a'`, `B` is of class `'class_b'` and `C` is of class `'class_c'`. Also suppose the constructor `class_c.m` contains the statement: `inferiorto('class_a')`. Then `e = fun(a,c)` or `e = fun(c,a)` invokes `class_a/fun`.

If a function is called with two objects having an unspecified relationship, the two objects are considered to have equal precedence, and the leftmost object's method is called. So, `fun(b,c)` calls `class_b/fun`, while `fun(c,b)` calls `class_c/fun`.

See Also

`superiorto`

info

Purpose

Display contact information about The MathWorks or toolbox Readme files

Syntax

```
info
info <toolbox>
```

Description

`info` displays contact information about The MathWorks in the Command Window, including phone and fax numbers and e-mail addresses.

`info toolbox` displays in the Help browser the Readme file for the specified toolbox, which contains information about problems from previous releases that have been fixed in the current release.

Purpose Construct an inline object

Syntax

```
g = inline(expr)
g = inline(expr,arg1,arg2, ...)
g = 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

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
```

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)ans =
```

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

Return functions in memory

Syntax

```
M = inmem
[M,X] = inmem
[M,X,J] = inmem
```

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.

Examples

This example lists the M-files that are required to run `erf`.

```
clear all;        % Clear the workspace
erf(0.5);
M = inmem

M =

    'repmat'
    'erfcore'
    'erf'
```

See Also

`clear`

Purpose

Detect points inside a polygonal region

Syntax

```
IN = inpolygon(X,Y,xv,yv)
```

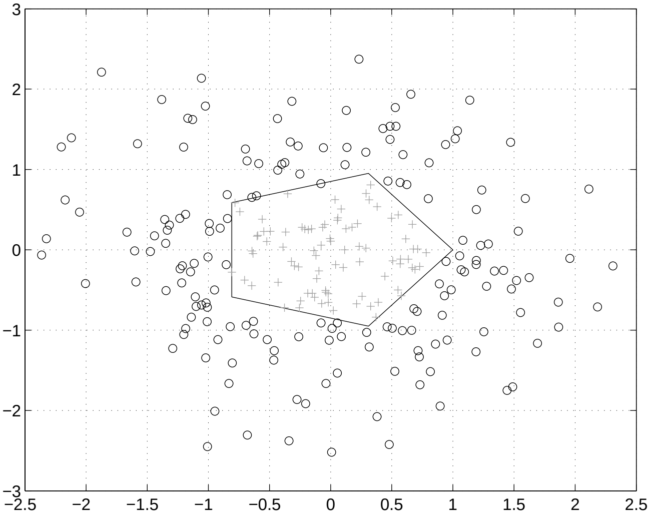
Description

`IN = inpolygon(X,Y,xv,yv)` returns a matrix `IN` the same size as `X` and `Y`. Each element of `IN` is assigned one of the values 1, 0.5 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
`IN(p,q) = 0.5`	If `(X(p,q),Y(p,q))` is on the polygon boundary
`IN(p,q) = 0`	If `(X(p,q),Y(p,q))` is outside the polygonal region

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')
```



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 may 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 `'\\'`.

Examples

Press **Return** to select a default value by detecting an empty matrix:

```
i = input('Do you want more? Y/N [Y]: ','s');
if isempty(i)
    i = 'Y';
end
```

See Also

`keyboard`, `menu`, `ginput`, `uicontrol`

Purpose

Create input dialog box

Syntax

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,title)
answer = inputdlg(prompt,title,lineNo)
answer = inputdlg(prompt,title,lineNo,defAns)
answer = inputdlg(prompt,title,lineNo,defAns,Resize)
```

Description

`answer = inputdlg(prompt)` creates a modal dialog box and returns user inputs in the cell array. `prompt` is a cell array containing prompt strings.

`answer = inputdlg(prompt,title)` `title` specifies a title for the dialog box.

`answer = inputdlg(prompt,title,lineNo)` `lineNo` specifies the number of lines for each user entered value. `lineNo` can be a scalar, column vector, or matrix.

- If `lineNo` is a scalar, it applies to all prompts.
- If `lineNo` is a column vector, each element specifies the number of lines of input for a prompt.
- If `lineNo` 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,title,lineNo,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.

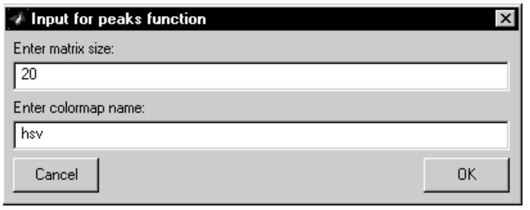
`answer = inputdlg(prompt,title,lineNo,defAns,Resize)` `Resize` specifies whether or not the dialog box can be resized. Permissible values are `'on'` and `'off'` where `'on'` means that the dialog box can be resized and that the dialog box is not modal.

Example

Create a dialog box to input an integer and colormap name. Allow one line for each value.

```
prompt = {'Enter matrix size:','Enter colormap name:'};
title  = 'Input for peaks function';
```

```
lines= 1;
def    = {'20','hsv'};
answer = inputdlg(prompt,title,lines,def);
```



See Also `dialog`, `errordlg`, `helpdlg`, `questdlg`, `warndlg`

Purpose Input argument name

Syntax `inputname(argnum)`

Description This command can be used only inside the body of a function.

`inputname(argnum)` returns the workspace variable name corresponding to the argument number *argnum*. If the input argument has no name (for example, if it is an expression instead of a variable), the `inputname` command returns the empty string (`''`).

Examples Suppose the function `myfun.m` is defined as:

```
function c = myfun(a,b)
disp(sprintf('First calling variable is "%s".',inputname(1))
```

Then

```
x = 5; y = 3; myfun(x,y)
```

produces

```
First calling variable is "x".
```

But

```
myfun(pi+1,pi-1)
```

produces

```
First calling variable is "".
```

See Also `nargin`, `nargout`, `nargchk`

inspect

Purpose Start the Property Inspector

Syntax

```
inspect
```

Description `inspect` displays the Property Inspector, which enables you to inspect and set the properties of any object you select in the figure window or Layout Editor.

See Also `guide`

Purpose Display event information when an event occurs

Syntax

```
instraction(obj, event)
```

Arguments

`obj`	An serial port object.
`event`	The event that caused the action to execute.

Description `instraction(obj, event)` displays a message that contains the event type, the time the event occurred, and the name of the serial port object 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 `instraction` as a template from which you create action functions that suit your specific application needs.

Example The following example creates the serial port objects `s`, and configures `s` to execute `instraction` when an output-empty event occurs. The event occurs after the `*IDN?` command is written to the instrument.

```
s = serial('COM1');
set(s,'OutputEmptyAction','instraction')
fopen(s)
fprintf(s,'*IDN?','async')
```

The resulting display from `instraction` is shown below.

```
OutputEmpty event occurred at 08:37:49 for the object:
Serial-COM1.
```

Read the identification information from the input buffer and end the serial port session.

```
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

instrfind

Purpose Return serial port objects from memory to the MATLAB workspace

Syntax

```
out = instrfind
out = instrfind('PropertyName',PropertyValue,...)
out = instrfind(S)
out = instrfind(obj,'PropertyName',PropertyValue,...)
```

Arguments

`'`*`PropertyName`*`'`	A property name for `obj`.
`PropertyValue`	A property value supported by *PropertyName*.
`S`	A structure of property names and property values.
`obj`	A serial port object, or an array of serial port objects.
`out`	An array of serial port objects.

Description

`out = instrfind` returns all valid serial port objects as an array to `out`.

`out = instrfind('`*`PropertyName`*`',PropertyValue,...)` returns an array of serial port objects whose property names and property values match those specified.

`out = instrfind(S)` returns an array of serial port objects whose property names and property values match those defined in the structure `S`. The field names of `S` are the property names, while the field values are the associated property values.

`out = instrfind(obj,'`*`PropertyName`*`',PropertyValue,...)` restricts the search for matching property name/property value pairs to the serial port objects listed in `obj`.

Remarks

Refer to “Displaying Property Names and Property Values” on page 8-21 for a list of serial port object properties that you can use with `instrfind`.

You must specify property values using the same format as the `get` function returns. For example, if `get` returns the `Name` property value as `MyObject`, `instrfind` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of string values. For example, `instrfind` will find an object with a `Parity` property value of `Even` or `even`.

You can use property name/property value string pairs, structures, and cell array pairs in the same call to `instrfind`.

Example

Suppose you create the following two serial port objects.

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s2,'BaudRate',4800)
```

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;
```

See Also

Functions

`clear`, `get`

int2str

Purpose Integer to string conversion

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)
```

Description

`i = int*(x)` converts the vector `x` into a signed integer. `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`	-32768 to 32767	Signed 16-bit integer	2	`int16`
`int32`	-2147483648 to 2147483647	Signed 32-bit integer	4	`int32`

A value of `x` above or below the range for a class is mapped to one of the endpoints of the range. If `x` is already a signed integer of the same class, `int*` has no effect.

The `int*` class is primarily meant to store integer values. Most operations that manipulate arrays without changing their elements are defined (examples are `reshape`, `size`, the logical and relational operators, subscripted assignment, and subscripted reference). No math operations except for `sum` are defined for `int*` since such operations are ambiguous on the boundary of the set (for example, they could wrap or truncate there). You can define 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.

See Also

`double`, `single`, `uint8`, `uint16`, `uint32`

interp1

Purpose One-dimensional 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,extrapval)
```

Description

`yi = interp1(x,Y,xi)` returns vector `yi` containing elements corresponding to the elements of `xi` and determined by interpolation within vectors `x` and `Y`. The vector `x` specifies the points at which the data `Y` is given. If `Y` is a matrix, then the interpolation is performed for each column of `Y` and `yi` is `length(xi)`-by-`size(Y,2)`.

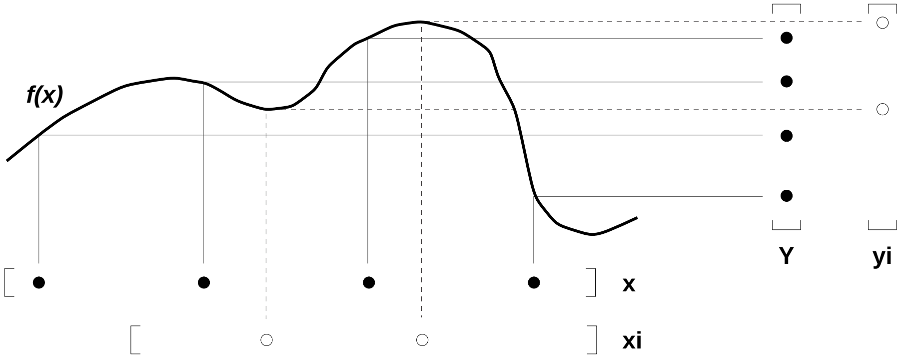
`yi = interp1(Y,xi)` assumes that `x = 1:N`, where `N` is the length of `Y` for vector `Y`, or `size(Y,1)` for matrix `Y`.

`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

If any element of `xi` is outside the interval spanned by `x`, the specified interpolation method is used for extrapolation. Alternatively, `yi = interp1(x,Y,xi,method,extrapval)` replaces extrapolated values with `extrapval`. `NaN` is often used for `extrapval`.

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`.

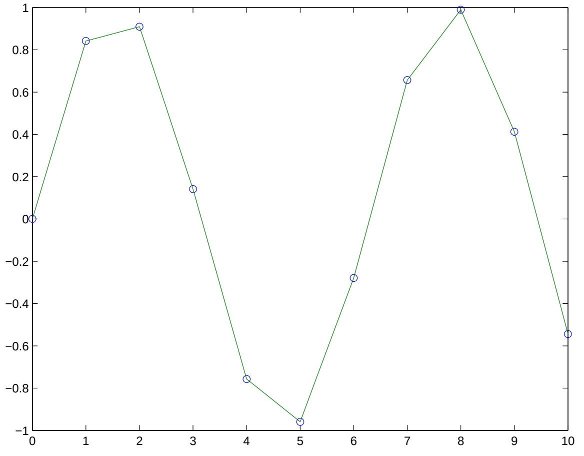


Interpolation is the same operation as *table lookup*. Described in table lookup terms, the *table* is [x,Y] and interp1 *looks up* the elements of xi in x, and, based upon their locations, returns values yi interpolated within the elements of Y.

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. Here are two vectors representing the census years from 1900 to 1990 and the corresponding United States population in millions of people.

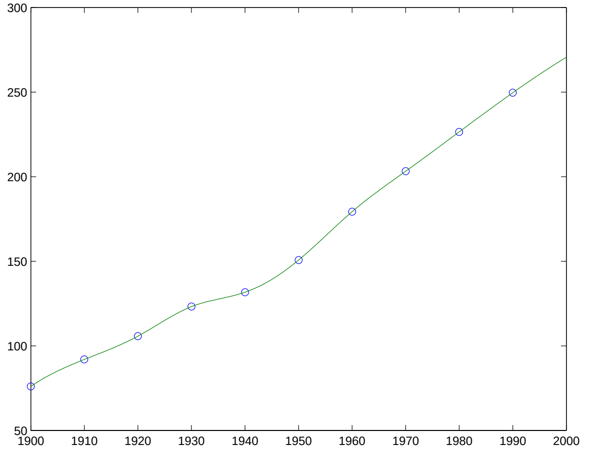
```
t = 1900:10:1990;
p = [75.995  91.972  105.711  123.203  131.669...
     150.697  179.323  203.212  226.505  249.633];
```

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
```

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.

See Also

`interpft`, `interp2`, `interp3`, `interpn`, `pchip`, `spline`

References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

Purpose

Two-dimensional 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)
```

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)`.

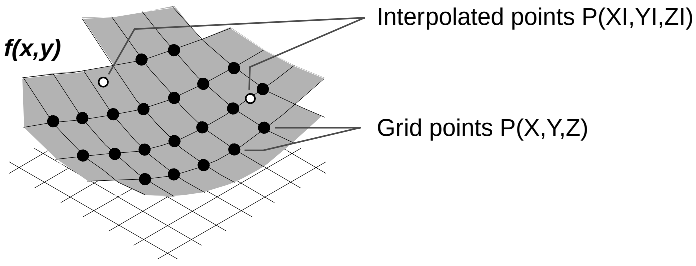
`ZI = interp2(X,Y,Z,XI,YI,`*`method`*`)` specifies an alternative interpolation method:

- `'linear'` for bilinear interpolation (default)
- `'nearest'` for nearest neighbor interpolation
- `'spline'` for cubic spline interpolation
- `'cubic'` for bicubic interpolation

All interpolation methods require that `X` and `Y` be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. 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'`.

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.



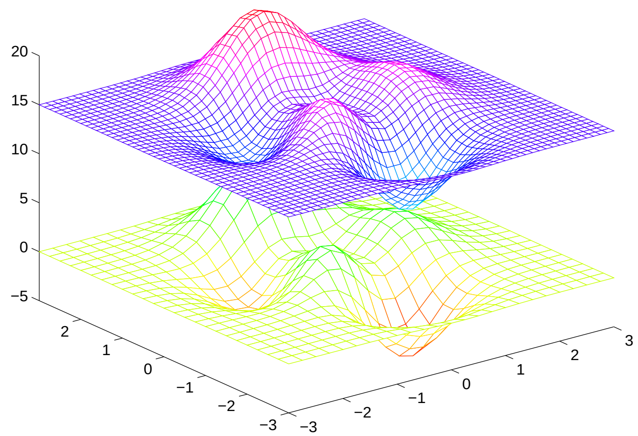
Interpolation is the same operation as table lookup. Described in table lookup terms, the table is `tab = [NaN,Y; X,Z]` and `interp2` looks up the elements of `XI` in `X`, `YI` in `Y`, and, based upon their location, returns values `ZI` interpolated within the elements of `Z`.

Examples

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])
```



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, interp3, interpn, meshgrid

interp3

Purpose

Three-dimensional 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)
```

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:

`'linear'`	Linear interpolation (default)
`'cubic'`	Cubic interpolation
`'spline'`	Cubic spline interpolation
`'nearest'`	Nearest neighbor interpolation

Discussion

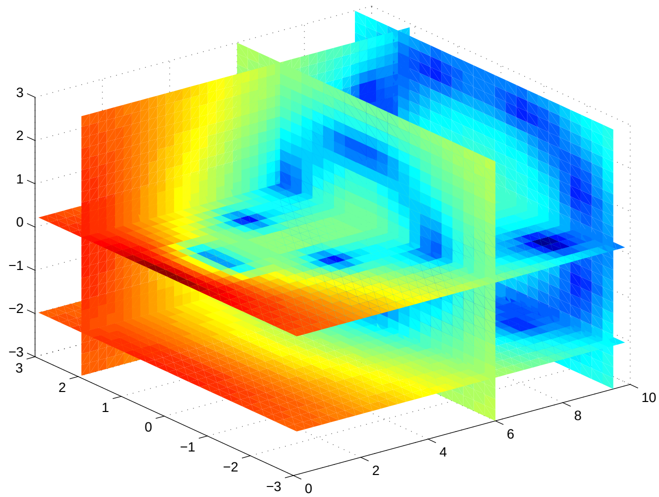
All the interpolation methods require that `X`,`Y` and `Z` be monotonic and have the same format (“plaid”) as if they were created using `meshgrid`. `X`, `Y`, and `Z` can be non-uniformly spaced. For faster interpolation when `X`, `Y`, and `Z` are equally spaced and monotonic, use the methods '*`linear`', '*`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); % V is 31-by-41-by-27
slice(xi,yi,zi,vi,[6 9.5],2,[-2 .2]), shading flat
```



See Also

`interp1`, `interp2`, `interpn`, `meshgrid`

interpft

Purpose

One-dimensional interpolation using the 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.

See Also

`interp1`

Purpose Multidimensional 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)
```

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`-D `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-D 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,1)` is the same as `interpn(V)`.

`VI = interpn(...,method)` specifies alternative methods:

`'linear'`	Linear interpolation (default)
`'cubic'`	Cubic interpolation
`'spline'`	Cubic spline interpolation
`'nearest'`	Nearest neighbor interpolation

Discussion

All the interpolation methods require that `X1`,`X2`, and `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`'.

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 stream line 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)

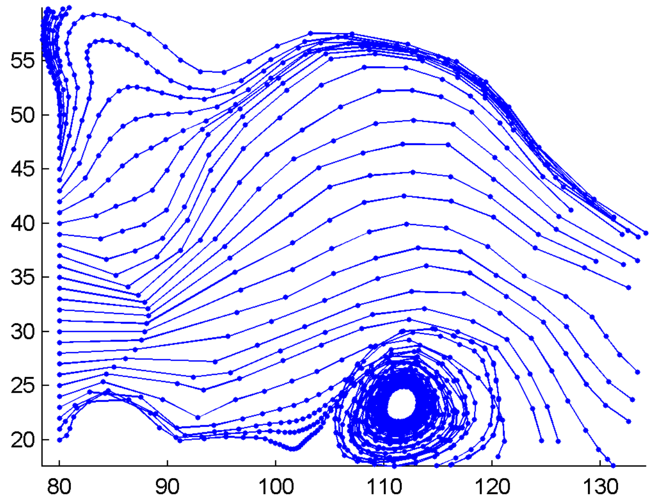
interpstreamspeed(...,sf) uses sf to scale the magnitude of the vector data and therefore controls the number of interpolated vertices. For example, if sf is 3, then interpstreamspeed creates only one third of the vertices.

vertsout = interpstreamspeed(...) returns a cell array of vertex arrays.

Examples

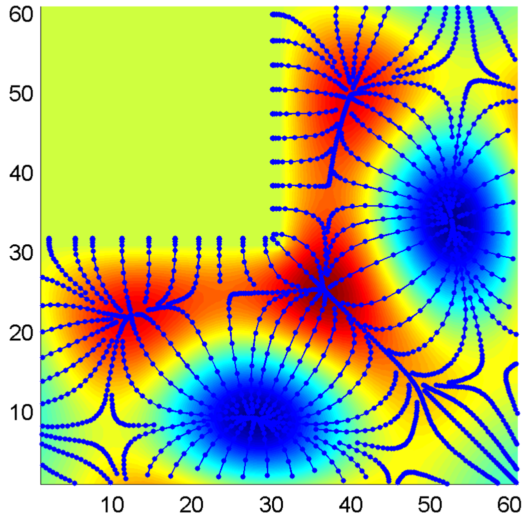
This example draws stream lines using the vertices returned by interpstreamspeed. Dot markers indicate the location of each vertex. This example enables you to visualize the relative speeds of the flow data. Stream lines having widely space vertices indicate faster flow; those with closely spaced vertices 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 stream lines whose vertex spacing indicates the value of the gradient along the stream line.

```
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

Purpose

Set intersection of two vectors

Syntax

```
c = intersect(a,b)
c = intersect(A,B,'rows')
[c,ia,ib] = intersect(...)
```

Description

`c = intersect(a,b)` returns the values common to both `a` and `b`. The resulting vector is sorted in ascending order. In set theoretic terms, this is $a \cap b$. `a` and `b` can be cell arrays of strings.

`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`.

`[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,:)`).

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`, `setdiff`, `setxor`, `union`, `unique`

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

`inv` uses LAPACK routines to compute the matrix inverse:

Matrix	Routine
Real	DLANGE, DGETRF, DGECON, DGETRI
Complex	ZLANGE, ZGETRF, ZGECON, ZGETRI

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*, Third Edition, SIAM, Philadelphia, 1999.

invhilb

Purpose Inverse of the 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.

ipermute

Purpose Inverse permute the dimensions of a multidimensional 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 Detect state

Syntax

```
k = iscell(C)
k = iscellstr(S)
k = ischar(S)
k = isempty(A)
k = isequal(A,B,...)
k = isfield(S,'field')
TF = isfinite(A)
k = isglobal(NAME)
TF = isinf(A)
TF = isletter('str')
k = islogical(A)
TF = isnan(A)
k = isnumeric(A)
k = isobject(A)
TF = isprime(A)
k = isreal(A)
TF = isspace('str')
k = issparse(S)
k = isstruct(S)
k = isstudent
k = isunix
```

Description

`k = iscell(C)` returns logical true (`1`) if `C` is a cell array and logical false (`0`) otherwise.

`k = iscellstr(S)` returns logical true (`1`) if `S` is a cell array of strings and logical false (`0`) otherwise. A cell array of strings is a cell array where every element is a character array.

`k = ischar(S)` returns logical true (`1`) if `S` is a character array and logical false (`0`) otherwise.

`k = isempty(A)` returns logical true (`1`) if `A` is an empty array and logical false (`0`) otherwise. An empty array has at least one dimension of size zero, for example, 0-by-0 or 0-by-5.

`k = isequal(A,B,...)` returns logical true (`1`) if the input arrays are the same type and size and hold the same contents, and logical false (`0`) otherwise.

`k = isfield(S,'field')` returns logical true (`1`) if *field* is the name of a field in the structure array `S`.

`TF = isfinite(A)` returns an array the same size as `A` containing logical true (`1`) where the elements of the array `A` are finite and logical false (`0`) where they are infinite or `NaN`.

For any A, exactly one of the three quantities isfinite(A), isinf(A), and isnan(A) is equal to one.

k = isglobal(NAME) returns logical true (1) if NAME has been declared to be a global variable, and logical false (0) if it has not been so declared.

TF = isinf(A) returns an array the same size as A containing logical true (1) where the elements of A are +Inf or -Inf and logical false (0) where they are not.

TF = isletter('*str*') returns an array the same size as '*str*' containing logical true (1) where the elements of *str* are letters of the alphabet and logical false (0) where they are not.

k = islogical(A) returns logical true (1) if A is a logical array and logical false (0) otherwise.

TF = isnan(A) returns an array the same size as A containing logical true (1) where the elements of A are NaNs and logical false (0) where they are not.

k = isnumeric(A) returns logical true (1) if A is a numeric array and logical false (0) otherwise. For example, sparse arrays, and double-precision arrays are numeric while strings, cell arrays, and structure arrays are not.

k = isobject(A) returns logical true (1) if A is an object and logical false (0) otherwise.

TF = isprime(A) returns an array the same size as A containing logical true (1) for the elements of A which are prime, and logical false (0) otherwise.

k = isreal(A) returns logical true (1) if all elements of A are real numbers, and logical false (0) if either A is not a numeric array, or if any element of A has a nonzero imaginary component. Since strings are a subclass of numeric arrays, isreal always returns 1 for a string input.

Because MATLAB 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.

`TF = isspace('str')` returns an array the same size as `'str'` containing logical true (`1`) where the elements of *str* are ASCII white spaces and logical false (`0`) where they are not. White spaces in ASCII are space, newline, carriage return, tab, vertical tab, or formfeed characters.

`k = issparse(S)` returns logical true (`1`) if the storage class of `S` is sparse and logical false (`0`) otherwise.

`k = isstruct(S)` returns logical true (`1`) if `S` is a structure and logical false (`0`) otherwise.

`k = isstudent` returns logical true (`1`) for student editions of MATLAB and logical false (`0`) for commercial editions.

`k = isunix` returns logical true (`1`) for UNIX versions of MATLAB and logical false (`0`) otherwise.

Examples

```
s = 'A1,B2,C3';

isletter(s)
ans =
     1   0   0   1   0   0   1   0


B = rand(2,2,2);
B(:,:,:) = [];

isempty(B)
ans =
     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`.

Let

```
a = [-2  -1  0  1  2]
```

Then

```
isfinite(1./a) =  [1  1  0  1  1]
isinf(1./a)  =  [0  0  1  0  0]
isnan(1./a)  =  [0  0  0  0  0]
```

and

```
isfinite(0./a) =  [1  1  0  1  1]
isinf(0./a)  =  [0  0  0  0  0]
isnan(0./a)  =  [0  0  1  0  0]
```

See Also

isappdata, ishandle, ishold, isjava, iskeyword, ismember, isstr, isvalid, isvarname

Purpose Detect an object of a given MATLAB class or Java class

Syntax

```
K = isa(obj,'class_name')
```

Description `K = isa(obj,'class_name')` returns logical true (`1`) if `obj` is of class (or a subclass of) *`class_name`*, and logical false (`0`) otherwise.

The argument `obj` is a MATLAB object or a Java object. The argument *`class_name`* is the name of a MATLAB (predefined or user-defined) or a Java class. Predefined MATLAB classes include:

`cell`	Cell array
`char`	Characters array
`double`	Double-precision floating-point array
`function_handle`	Function Handle
`int8`	8-bit signed integer array
`int16`	16-bit signed integer array
`int32`	32-bit signed integer array
`numeric`	Integer or floating-point array
`single`	Single-precision floating-point array
`sparse`	2-D real (or complex) sparse array
`struct`	Structure array
`uint8`	8-bit unsigned integer array
`uint16`	16-bit unsigned integer array
`uint32`	32-bit unsigned integer array

You cannot use `isa` to identify a logical value. Use `islogical` for this instead.

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.

See Also `getappdata`, `rmappdata`, `setappdata`

ishandle

Purpose

Determines if values are valid graphics object handles

Syntax

```
array = ishandle(h)
```

Description

`array = ishandle(h)` returns an array that contains `1`'s where the elements of `h` are valid graphics handles and `0`'s where they are not.

Examples

Determine whether the handles previously returned by `fill` remain handles of existing graphical objects:

```
X = rand(4); Y = rand(4);
h = fill(X,Y,'blue')
.
.
.
delete(h(3))
.
.
.
ishandle(h)
ans =
    1
    1
    0
    1
```

See Also

`findobj`

Purpose

Return hold state

Syntax

```
k = ishold
```

Description

`k = ishold` returns the hold state of the current axes. If `hold` is `on` `k = 1`, if `hold` is `off`, `k = 0`.

Examples

`ishold` is useful in graphics M-files where you want to perform a particular action only if `hold` is not `on`. For example, these statements set the view to 3-D only if `hold` is `off`:

```
if ~ishold
   view(3);
end
```

See Also

`axes`, `figure`, `hold`, `newplot`

isjava

Purpose Test whether an object is a Java object

Syntax `B = isjava(obj)`

Description `B = isjava(obj)` returns `1` if `obj` is a Java object, and returns `0` if it is not.

See Also `isa`, `isstruct`, `iscell`, `isnumeric`, `isobject`, `islogical`, `is*`

Purpose Test if string is a MATLAB keyword

Syntax

```
iskeyword 's'
iskeyword
R = iskeyword('s')
```

Description `iskeyword 's'` returns a `1` if string, `s`, is a keyword in the MATLAB language. Returns `0`, otherwise.

`iskeyword` returns a list of all MATLAB keywords.

`R = iskeyword('s')` returns a `1` in `R` if string, `s`, is a MATLAB keyword.

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'
    'continue'
    'else'
    'elseif'
    'end'
    'for'
    'function'
    'global'
    'if'
    'otherwise'
    'persistent'
    'return'
    'switch'
    'try'
    'while'
```

See Also

isvarname

Purpose Detect members of a set

Syntax

```
k = ismember(a,S)
k = ismember(A,S,'rows')
```

Description

`k = ismember(a,S)` returns an vector the same length as `a` containing logical true (`1`) where the elements of `a` are in the set `S`, and logical false (`0`) elsewhere. In set theoretic terms, `k` is 1 where $a \in S$. `a` and `S` can be cell arrays of strings.

`k = 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.

Examples

```
set = [0 2 4 6 8 10 12 14 16 18 20];
a = reshape(1:5,[5 1])

a =

     1
     2
     3
     4
     5

ismember(a,set)

ans =

     0
     1
     0
     1
     0
```

See Also `intersect`, `setdiff`, `setxor`, `union`, `unique`

isocaps

Purpose

Compute isosurface end-cap geometry

Syntax

```
fvc = isocaps(X,Y,Z,V,isovalue)
fvc = isocaps(V,isovalue)
fvc = isocaps(...,'enclose')
fvc = isocaps(...,'whichplane')
[f,v,c] = isocaps(...)
isocaps(...)
```

Description

`fvc = isocaps(X,Y,Z,V,isovalue)` computes isosurface end cap geometry for the volume data `V` at isosurface value `isovalue`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`.

The struct `fvc` contains the face, vertex, and color data for the end caps and can be passed directly to the `patch` command.

`fvc = isocaps(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isocaps(...,'`*enclose*`')` specifies whether the end caps enclose data values above or below the value specified in `isovalue`. The string *enclose* can be either `above` (default) or `below`.

`fvc = isocaps(...,'`*whichplane*`')` specifies on which planes to draw the end caps. Possible values for *whichplane* are: `all` (default), `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, or `zmax`.

`[f,v,c] = isocaps(...)` returns the face, vertex, and color data for the end caps in three arrays instead of the struct `fvc`.

`isocaps(...)` without output arguments draws a patch with the computed faces, vertices, and colors.

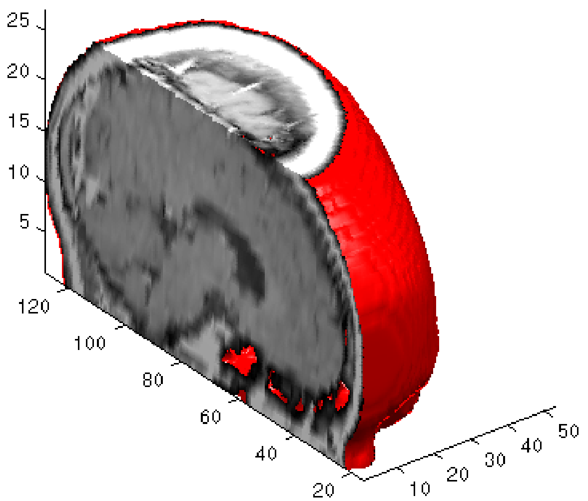
Examples

This example uses a data set that is a collection of MRI slices of a human skull. It illustrates the use of `isocaps` to draw the end caps on this cut-away 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

Purpose

Calculates 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 (truecolor).

`nc = isocolors(C,vertices)`, `nc = isocolors(R,G,B,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(C)`.

`nc = isocolors(...,PatchHandle)` uses the vertices from the patch identified by `PatchHandle`.

`isocolors(...,PatchHandle)` sets the `FaceVertexCData` property of the patch specified by `PatchHandle` to the computed colors.

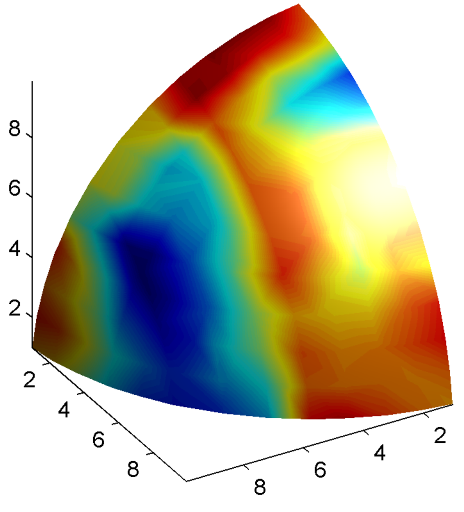
Examples

Indexed Color Data

This example displays an isosurface and colors it with random data using indexed color. (See "Interpolating in Indexed Color vs. Truecolor" 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));
```

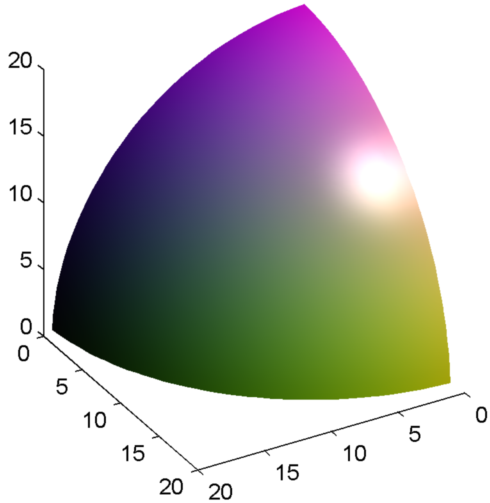
```
isonormals(x,y,z,data,p);
isocolors(x,y,z,cdata,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);axis tight
camlight; lighting phong;
```



Truecolor Data

This example displays an isosurface and colors it with truecolor (RGB) data.

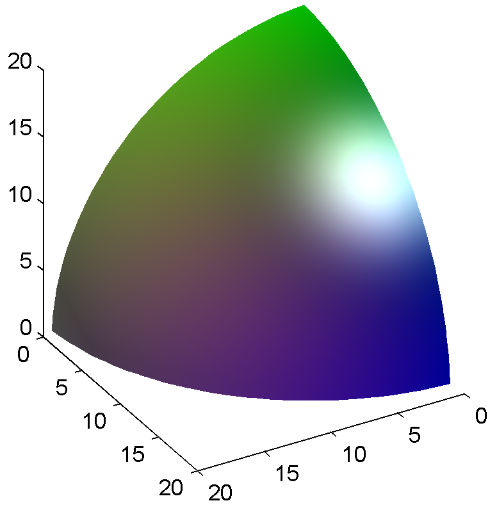
```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(x,y,z,data,20));
isonormals(x,y,z,data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
isocolors(x,y,z,r/20,g/20,b/20,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);
camlight; lighting phong;
```



Modified Truecolor Data

This example uses `isocolors` to calculate the truecolor 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.

Purpose

Compute normals of isosurface vertices

Syntax

```
n = isonormals(X,Y,Z,V,vertices)
n = isonormals(V,vertices)
n = isonormals(V,p), n = isonormals(X,Y,Z,V,p)
n = isonormals(...,'negate')
isonormals(V,p), 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');
```

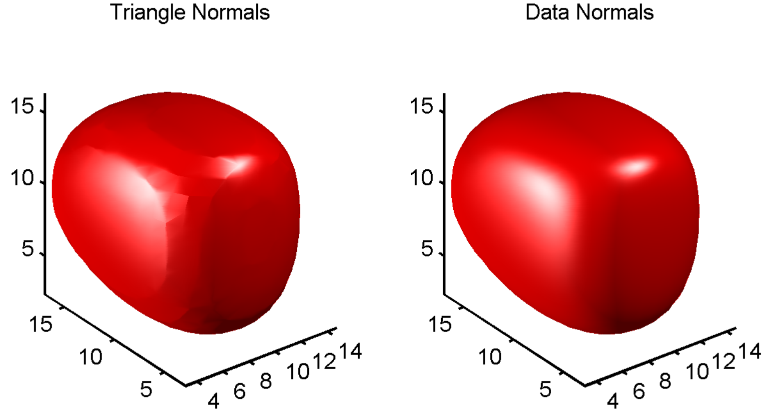
Draw an isosurface from the volume data and add lights. This isosurface uses triangle normals (`patch`, `isosurface`, `view`, `daspect`, `axis`, `camlight`, `lighting`, `title`):

```
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`

isosurface

Purpose

Extract isosurface data from volume data

Syntax

```
fv = isosurface(X,Y,Z,V,isovalue)
fv = isosurface(V,isovalue)
fv = isosurface(X,Y,Z,V), fv = isosurface(X,Y,Z,V)
fvc = isosurface(...,colors)
fv = isosurface(...,'noshare')
fv = isosurface(...,'verbose')
[f,v] = 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`. 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(...)` returns the faces and vertices in two arrays instead of a struct.

`isosurface(...)` with no output arguments creates a patch using the computed faces and vertices.

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

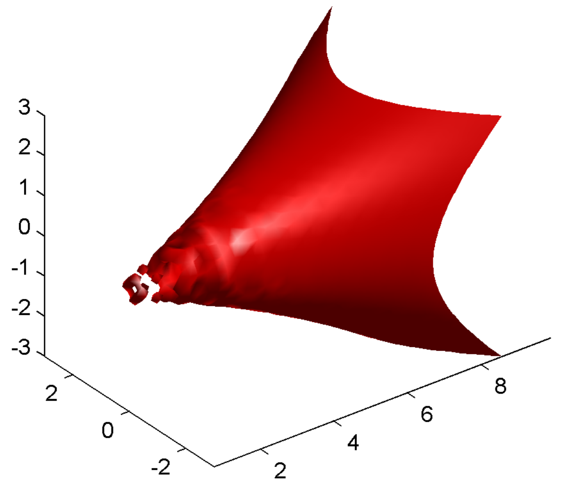
```
[f,v] = isosurface(X,Y,Z,V,isovalue);
patch('Faces',f,'Vertices',v)
```

Examples

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
```



See Also isonormals, isocaps, reducepatch, reducevolume, shrinkfaces, smooth3, subvolume

Purpose Detect strings

Description This MATLAB 4 function has been renamed `ischar` in MATLAB 5.

See Also `is*`

isvalid

Purpose

Determine if serial port objects are valid

Syntax

```
out = isvalid(obj)
```

Arguments

`obj`	A serial port object or array of serial port objects.
`out`	A logical array.

Description

`out = isvalid(obj)` returns the logical array `out`, which contains a `0` where the elements of `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. Since 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 Test if string is a valid variable name

Syntax

```
isvarname 's'
R = isvarname('s')
```

Description `isvarname 's'` returns a `1` if string, `s`, is a valid MATLAB variable name. Returns `0`, otherwise. A valid variable name is a character string of letters, digits, and underscores, totaling not more than 32 characters and beginning with a letter.

`R = isvarname('s')` returns a 1 in R if string, s, is a valid variable name.

Examples You can use `isvarname` without parenthesis if you pass a single string argument. If you are building strings from various pieces, place the construction in parenthesis.

```
isvarname foo
ans =
   1

isvarname 'Monday 23'
ans =
   0

d = date;
isvarname(['Monday_',d(1:2)])
ans =
   1
```

See Also `iskeyword`, `is*`

j

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

Constructs a 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 i = 1:4
   for j = 1:5
   dblArray(i,j) = java.lang.Double((i*10) + j);
   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`

javaMethod

Purpose Invokes a Java method

Syntax

```
X = javaMethod('method_name','class_name’,x1,...,xn)
X = javaMethod('method_name',J,x1,...,xn)
```

Description

`javaMethod('method_name','class_name’,x1,...,xn)` invokes the `static` method `method_name` in the class `class_name`, with the argument list that matches `x1,...,xn`.

`javaMethod('method_name',J,x1,...,xn)` invokes the nonstatic method `method_name` on the object `J`, with the argument list that matches `x1,...,xn`.

Remarks

Using the `javaMethod` function enables you 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 MATLAB. 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`. The default MATLAB syntax for invoking a Java method is somewhat simpler and is preferable for most applications. Use `javaMethod` primarily for the two cases described above.

Examples

To invoke the `static` Java method `isNaN` on class, `java.lang.Double`, use

```
javaMethod('isNaN','java.lang.Double',2.2)
```

The following example invokes the nonstatic method `setTitle`, where `frameObj` is a `java.awt.Frame` object.

```
frameObj = java.awt.Frame;
javaMethod('setTitle', frameObj, 'New Title');
```

See Also `javaArray`, `javaObject`, `import`, `methods`, `isjava`

javaObject

Purpose Constructs a Java object

Syntax

```
J = javaObject('class_name',x1,...,xn)
```

Description

`javaObject('class_name',x1,...,xn)` invokes the Java constructor for class `'class_name'` 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

Using the `javaObject` function enables you 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, you would use

```
strObj = java.lang.String('hello');
```

Note Typically, you will not need to use `javaObject`. The default MATLAB syntax for instantiating a Java class is somewhat simpler and is preferable for most applications. Use `javaObject` primarily for the two 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`

keyboard

Purpose Invoke the keyboard in an M-file

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`, `return`

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:

lasterr

Purpose

Last error message

Syntax

```
str = lasterr
lasterr('')
```

Description

`str = lasterr` returns the last error message generated by MATLAB.

`lasterr('')` resets `lasterr` so it returns an empty matrix until the next error occurs.

Examples

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 catchfcn
l = lasterr;
j = findstr(l,'Inner matrix dimensions');
if ~isempty(j)
    disp('Wrong dimensions for matrix multiply')
else
    k = findstr(l,'Undefined function or variable');
    if ~isempty(k)
        disp('At least one operand does not exist')
    end
end
```

The `lasterr` function is useful in conjunction with the two-argument form of the `eval` function:

```
eval('string','catchstr')
```

or the `try ... catch...end` statements. The `catch` action examines the `lasterr` string to determine the cause of the error and takes appropriate action.

The `eval` function evaluates *string* and returns if no error occurs. If an error occurs, `eval` executes *catchstr*. Using `eval` with the `catchfcn` function above:

```
clear
A = [1  2  3; 6  7  2; 0  -1  5];
B = [9  5  6; 0  4  9];
```

```
eval('A*B','catchfcn')
```

MATLAB responds with `Wrong dimensions for matrix multiply`.

See Also `error`, `eval`

lastwarn

Purpose Last warning message

Syntax

```
lastwarn
lastwarn('')
lastwarn('string')
```

Description `lastwarn` returns a string containing the last warning message issued by MATLAB.

`lastwarn('')` resets the `lastwarn` function so that it will return an empty string matrix until the next warning is encountered.

`lastwarn('string')` sets the last warning message to `'string'`. The last warning message is updated regardless of whether `warning` is `on` or `off`.

See Also `lasterr`, `warning`

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`

legend

Purpose Display a legend on graphs

Syntax

```
legend('string1','string2',...)
legend(h,'string1','string2',...)
legend(string_matrix)
legend(h,string_matrix)
legend(axes_handle,...)
legend('off')
legend(h,...)
legend(...,pos)
h = legend(...)
[legend_handle,object_handles] = legend(...)
```

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.

`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 handles in the vector `h`, using the specified strings to label the corresponding graphics object (line, bar, etc.).

`legend(string_matrix)` adds a legend containing the rows of the matrix `string_matrix` as labels. This is the same as `legend(string_matrix(1,:),string_matrix(2,:),...)`.

`legend(h,string_matrix)` associates each row of the matrix `string_matrix` with the corresponding graphics object in the vector `h`.

`legend(axes_handle,...)` displays the legend for the axes specified by `axes_handle`.

`legend('off'),legend(axes_handle,'off')` removes the legend from the current axes or the axes specified by `axes_hanlde`.

`legend_handle = legend` returns the handle to the legend on the current axes or an empty vector if no legend exists.

`legend` with no arguments refreshes all the legends in the current figure.

`legend(legend_handle)` refreshes the specified legend.

`legend(...,pos)` uses `pos` to determine where to place the legend.

- `pos = -1` places the legend outside the axes boundary on the right side.
- `pos = 0` places the legend inside the axes boundary, obscuring as few points as possible.
- `pos = 1` places the legend in the upper-right corner of the axes (default).
- `pos = 2` places the legend in the upper-left corner of the axes.
- `pos = 3` places the legend in the lower-left corner of the axes.
- `pos = 4` places the legend in the lower-right corner of the axes.

`[legend_handle,object_handles] = legend(...)` returns the handle of the legend (`legend_handle`), which is an axes graphics object and the handles of the line, patch and text graphics objects (`object_handles`) used in the legend. These handles enable you to modify the properties of the respective objects.

Remarks

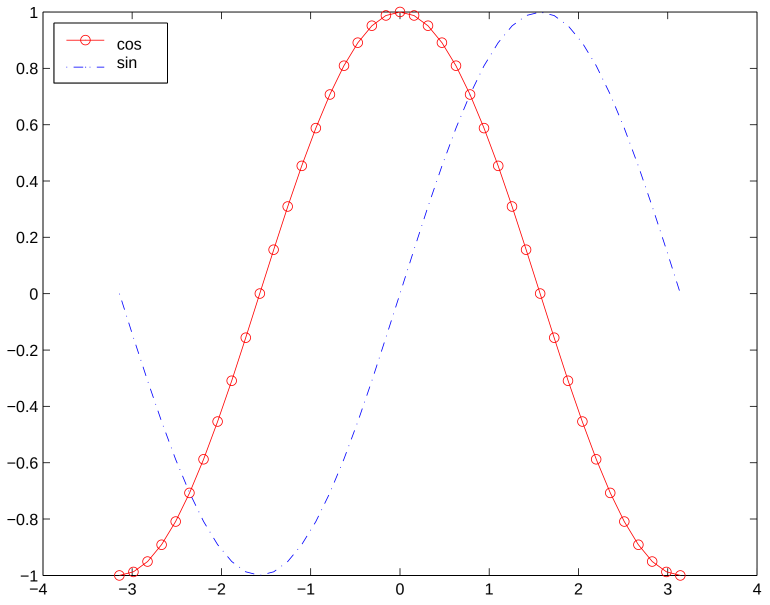
`legend` associates strings with the objects in the axes in the same order that they are listed in the axes `Children` property. By default, the legend annotates the current axes.

MATLAB displays only one legend per axes. `legend` positions the legend based on a variety of factors, such as what objects the legend obscures. You 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 on a label allows you to edit the label.

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','sin',2);
```



In this example, the `plot` command specifies a solid, red line (`'-r'`) for the cosine function and a dash-dot, blue line (`'-.b'`) for the sine function.

See Also

`LineSpec`, `plot`

Purpose Associated Legendre functions

Syntax

```
P = legendre(n,X)
S = legendre(n,X,'sch')
```

Definition 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:

$$P_n(x) = \frac{1}{2^n n!}\left[\frac{d}{dx}^n(x^2-1)^n\right]$$

The Schmidt seminormalized associated Legendre functions are related to the nonnormalized associated Legendre functions $P_n^m(x)$ by:

$$S_n^m(x) = (-1)^m\sqrt{\frac{2(n-m)!}{(n+m)!}}\,P_n^m(x)$$

where $m > 0$.

Description `P = legendre(n,X)` computes the associated Legendre functions of degree `n` and order `m = 0,1,...,n`, evaluated at `X`. Argument `n` must be a scalar integer less than 256, and `X` must contain real values in the domain $-1 \le x \le 1$.

The returned array `P` has one more dimension than `X`, and each element `P(m+1,d1,d2...)` contains the associated Legendre function of degree `n` and order `m` evaluated at `X(d1,d2...)`.

If X is a vector, then P is a matrix of the form:

$$\begin{array}{llll} P_2^0(x(1)) & P_2^0(x(2)) & P_2^0(x(3)) & \ldots \\ P_2^1(x(1)) & P_2^1(x(2)) & P_2^1(x(3)) & \ldots \\ P_2^2(x(1)) & P_2^2(x(2)) & P_2^2(x(3)) & \ldots \end{array}$$

`S = legendre(...,'sch')` computes the Schmidt seminormalized associated Legendre functions $S_n^m(x)$.

Examples

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

Note that this matrix is of the form shown at the bottom of the previous page.

Given,

```
X = rand(2,4,5); N = 2;
P = legendre(N,X)
```

Then `size(P)` is 3-by-2-by-4-by-5, and `P(:,1,2,3)` is the same as `legendre(n,X(1,2,3))`.

Purpose Length of vector

Syntax `n = length(X)`

Description The statement `length(X)` is equivalent to `max(size(X))` for nonempty arrays and `0` for empty arrays.

`n = length(X)` returns the size of the longest dimension of `X`. If `X` is a vector, this is the same as its length.

Examples

```
x = ones(1,8);
n = length(x)

n =

     8

x = rand(2,10,3);
n = length(x)

n =

    10
```

See Also `ndims`, `size`

length (serial)

Purpose Length of serial port object array

Syntax `length(obj)`

Arguments

`obj`	A serial port object or an array of serial port objects.

Description `length(obj)` returns the length of `obj`. It is equivalent to the command `max(size(obj))`.

See Also

Functions

`size`

Purpose Show the license number for MATLAB

Syntax `license`

Description `license` shows the license number for MATLAB, as a string. It returns `demo` for demonstration versions and `unknown` if the license number cannot be determined.

See Also `version`

light

Purpose Create a light object

Syntax

```
light('PropertyName',PropertyValue,...)
handle = light(...)
```

Description `light` creates a light object in the current axes. lights affect only patch and surface object.

`light('PropertyName',PropertyValue,...)` creates a light object using the specified values for the named properties. MATLAB 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.

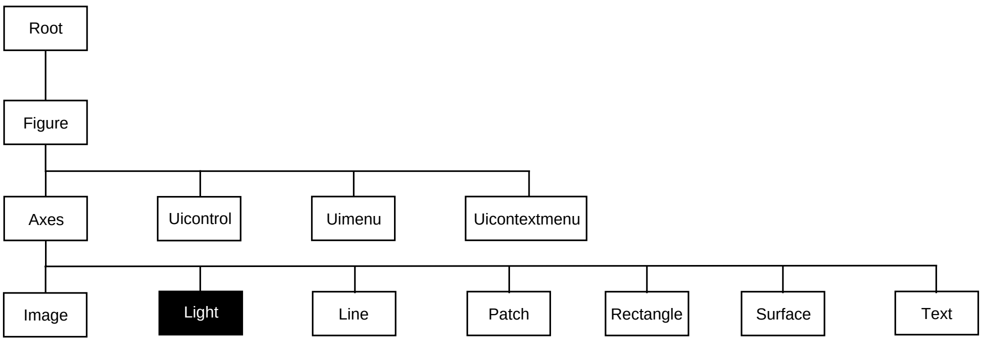
You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see `set` and `get` for examples of how to specify these data types).

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’);
```

Object Hierarchy



Setting Default Properties

You can set default light properties on the axes, figure, and root 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.

The following table lists all light properties and provides a brief description of each. The property name links take you to an expanded description of the properties.

Property Name	Property Description	Property Value
Defining the Light		
`Color`	Color of the light produced by the light object	Values: `ColorSpec`
`Position`	Location of light in the axes	Values: x-, y-, z-coordinates in axes units Default: `[1 0 1]`
`Style`	Parallel or divergent light source	Values: `infinite`, `local`

Property Name	Property Description	Property Value
Controlling the Appearance		
`SelectionHighlight`	This property is not used by light objects	Values: `on`, `off` Default: `on`
`Visible`	Make the effects of the light visible or invisible	Values: `on`, `off` Default: `on`
Controlling Access to Objects		
`HandleVisibility`	Determines if and when the the line's handle is visible to other functions	Values: `on`, `callback`, `off` Default: `on`
`HitTest`	This property is not used by light objects	Values: `on`, `off` Default: `on`
General Information About the Light		
`Children`	Light objects have no children	Values: `[]` (empty matrix)
`Parent`	The parent of a light object is always an axes object	Value: axes handle
`Selected`	This property is not used by light objects	Values: `on`, `off` Default: `on`
`Tag`	User-specified label	Value: any string Default: `''` (empty string)
`Type`	The type of graphics object (read only)	Value: the string `'light'`
`UserData`	User-specified data	Values: any matrix Default: `[]` (empty matrix)
Properties Related to Callback Routine Execution		
`BusyAction`	Specify how to handle callback routine interruption	Values: `cancel`, `queue` Default: `queue`
`ButtonDownFcn`	This property is not used by light objects	Values: string Default: empty string

Property Name	Property Description	Property Value
Controlling the Appearance		
SelectionHighlight	This property is not used by light objects	Values: on, off Default: on
Visible	Make the effects of the light visible or invisible	Values: on, off Default: on
Controlling Access to Objects		
HandleVisibility	Determines if and when the the line's handle is visible to other functions	Values: on, callback, off Default: on
HitTest	This property is not used by light objects	Values: on, off Default: on
General Information About the Light		
Children	Light objects have no children	Values: [] (empty matrix)
Parent	The parent of a light object is always an axes object	Value: axes handle
Selected	This property is not used by light objects	Values: on, off Default: on
Tag	User-specified label	Value: any string Default: '' (empty string)
Type	The type of graphics object (read only)	Value: the string 'light'
UserData	User-specified data	Values: any matrix Default: [] (empty matrix)
Properties Related to Callback Routine Execution		
BusyAction	Specify how to handle callback routine interruption	Values: cancel, queue Default: queue
ButtonDownFcn	This property is not used by light objects	Values: string Default: empty string

Property Name	Property Description	Property Value
`CreateFcn`	Define a callback routine that executes when a light is created	Values: string (command or M-file name) Default: empty string
`DeleteFcn`	Define a callback routine that executes when the light is deleted (via `close` or `delete`)	Values: string (command or M-file name) Default: empty string
`Interruptible`	Determine if callback routine can be interrupted	Values: `on`, `off` Default: `on` (can be interrupted)
`UIContextMenu`	This property is not used by light objects	Values: handle of a Uicontrextmenu

Modifying Properties

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

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

To change the default value of properties see Setting Default Property Values.

Light Property Descriptions

This section lists property names along with the type of values each accepts.

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, subsequently invoked callback routes 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

This property is not useful 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 three-element RGB vector or one of MATLAB's predefined names. See the `ColorSpec` reference page for more information.

CreateFcn string

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a light object. You must define this property as a default value for lights. For example, the statement,

```
set(0,'DefaultLightCreateFcn','set(gcf,''Colormap'',hsv)')
```

sets the current figure colormap to `hsv` whenever you create a light object. MATLAB executes this routine after setting all light properties. Setting this property on an existing light 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`.

DeleteFcn string

Delete light callback routine. A callback routine that executes when you delete the light object (i.e., when you issue a `delete` command or clear the axes or figure containing the light). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

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

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 evaling 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

Light objects parent. The handle of the light object's parent axes. You can move a light object to another axes by changing this property to the new axes handle.

`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 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).

See Also `lighting`, `material`, `patch`, `surface`

lightangle

Purpose Create or position a light object in spherical coordinates

Syntax

```
lightangle(az,el)
light_handle = lightangle(az,el)
lightangle(light_handle,az,el)
[ax 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 into `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`

Purpose Select the 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.

`lighting flat` selects flat lighting.

`lighting gouraund` selects gouraud lighting.

`lighting phong` selects phong lighting.

`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 `light`, `material`, `patch`, `surface`

lin2mu

Purpose Convert linear audio signal to mu-law

Syntax `mu = lin2mu(y)`

Description `mu = lin2mu(y)` converts linear audio signal amplitudes in the range $-1 \leq Y \leq 1$ to mu-law encoded “flints” in the range $0 \leq u \leq 255$.

See Also `auwrite`, `mu2lin`

Purpose Create line object

Syntax

```
line(X,Y)
line(X,Y,Z)
line(X,Y,Z,'PropertyName',PropertyValue,...)
line('PropertyName',PropertyValue,...)  low-level–PN/PV pairs only
h = line(...)
```

Description

`line` creates a line object in the current axes. You can specify the color, width, line style, and marker type, as well as other characteristics.

The `line` function has two forms:

- Automatic color and line style cycling. When you specify matrix coordinate data using the informal syntax (i.e., the first three arguments are interpreted as the coordinates),

  ```
  line(X,Y,Z)
  ```

  MATLAB cycles through the axes `ColorOrder` and `LineStyleOrder` property values the way the `plot` function does. However, unlike `plot`, `line` does not call the `newplot` function.
- Purely low-level behavior. When you call `line` with only property name/property value pairs,

  ```
  line('XData',x,'YData',y,'ZData',z)
  ```

  MATLAB draws a line object in the current axes using the default line color (see the `colordef` function for information on color defaults). Note that you cannot specify matrix coordinate data with the low-level form of the `line` function.

`line(X,Y)` adds the line defined in vectors `X` and `Y` to the current axes. If `X` and `Y` are matrices of the same size, `line` draws one line per column.

`line(X,Y,Z)` creates lines in three-dimensional coordinates.

`line(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,'`*`PropertyName`*`',PropertyValue,...)` 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 paris. 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 setting 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.

Examples

This example uses the `line` function to add a shadow to plotted data. First, plot some data and save the line's handle:

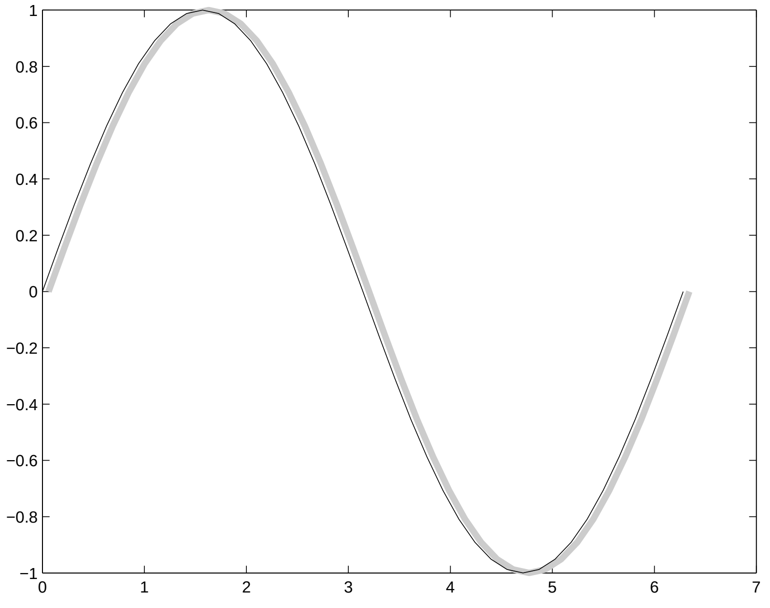
```
t = 0:pi/20:2*pi;
hline1 = plot(t,sin(t),'k');
```

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, pop the first line to the front:

```
set(gca,'Children',[hline1 hline2])
```



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))
```

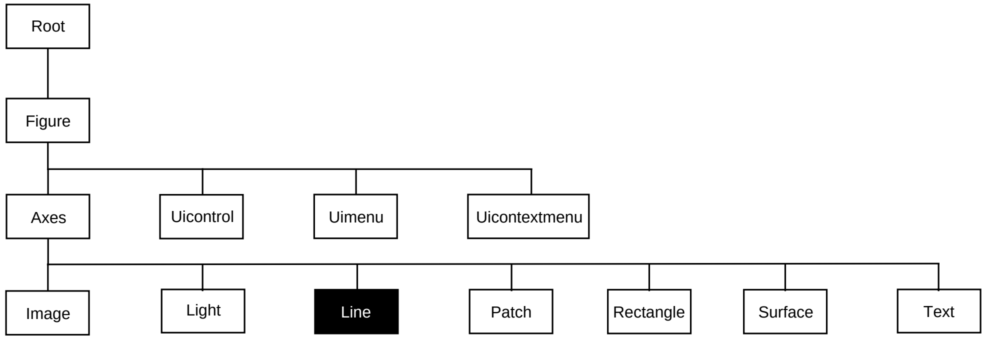
This also applies to the case when just one or two matrices have one row. For example, the statement,

```
line(rand(2,4),rand(2,4),rand(1,4))
```

is equivalent to:

```
line(rand(4,2),rand(4,2),rand(4,1))
```

Object Hierarchy



Setting Default Properties

You can set default line properties on the axes, figure, and root 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.

The following table lists all light properties and provides a brief description of each. The property name links take you to an expanded description of the properties.

Property Name	Property Description	Property Value
Data Defining the Object		
XData	The *x*-coordinates defining the line	Values: vector or matrix Default: [0 1]
YData	The *y*-coordinates defining the line	Values: vector or matrix Default: [0 1]
ZData	The *z*-coordinates defining the line	Values: vector or matrix Default: [] empty matrix
Defining Line Styles and Markers		
LineStyle	Select from five line styles.	Values: –, ––, :, –., none Default: –
LineWidth	The width of the line in points	Values: scalar Default: 0.5 points
Marker	Marker symbol to plot at data points	Values: see Marker property Default: none
MarkerEdgeColor	Color of marker or the edge color for filled markers	Values: ColorSpec, none, auto Default: auto
MarkerFaceColor	Fill color for markers that are closed shapes	Values: ColorSpec, none, auto Default: none
MarkerSize	Size of marker in points	Values: size in points Default: 6
Controlling the Appearance		
Clipping	Clipping to axes rectangle	Values: on, off Default: on
EraseMode	Method of drawing and erasing the line (useful for animation)	Values: normal, none, xor, background Default: normal
SelectionHighlight	Highlight line when selected (Selected property set to on)	Values: on, off Default: on

Property Name	Property Description	Property Value
`Visible`	Make the line visible or invisible	Values: `on`, `off` Default: `on`
`Color`	Color of the line	`ColorSpec`
Controlling Access to Objects		
`HandleVisibility`	Determines if and when the the line's handle is visible to other functions	Values: `on`, `callback`, `off` Default: `on`
`HitTest`	Determines if the line can become the current object (see the figure `CurrentObject` property)	Values: `on`, `off` Default: `on`
General Information About the Line		
`Children`	Line objects have no children	Values: `[]` (empty matrix)
`Parent`	The parent of a line object is always an axes object	Value: axes handle
`Selected`	Indicate whether the line is in a "selected" state.	Values: `on`, `off` Default: `on`
`Tag`	User-specified label	Value: any string Default: `''` (empty string)
`Type`	The type of graphics object (read only)	Value: the string `'line'`
`UserData`	User-specified data	Values: any matrix Default: `[]` (empty matrix)
Properties Related to Callback Routine Execution		
`BusyAction`	Specify how to handle callback routine interruption	Values: `cancel`, `queue` Default: `queue`
`ButtonDownFcn`	Define a callback routine that executes when a mouse button is pressed on over the line	Values: string Default: `''` (empty string)

Property Name	Property Description	Property Value
`CreateFcn`	Define a callback routine that executes when a line is created	Values: string Default: `''` (empty string)
`DeleteFcn`	Define a callback routine that executes when the line is deleted (via `close` or `delete`)	Values: string Default: `''` (empty string)
`Interruptible`	Determine if callback routine can be interrupted	Values: `on`, `off` Default: `on` (can be interrupted)
`UIContextMenu`	Associate a context menu with the line	Values: handle of a Uicontextmenu

Line Properties

Modifying Properties

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

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

To change the default value of properties see Setting Default Property Values.

Line Property Descriptions

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

`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, subsequently invoked callback routes 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

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the line object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

`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 display outside the axes plot box. This can occur if you create a line, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a longer line.

`Color` `ColorSpec`

Line color. A three-element RGB vector or one of MATLAB's predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

`CreateFcn` string

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a line object. You must define this property as a default value for lines. For example, the statement,

```
set(0,'DefaultLineCreateFcn','set(gca,''LineStyleOrder'',''-.|--'')')
```

defines a default value on the root level that sets the axes `LineStyleOrder` whenever you create a line object. MATLAB executes this routine after setting all line properties. Setting this property on an existing line 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`.

`DeleteFcn` string

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

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

`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 redraw 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 Non-normal 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., XORing 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 non-normal 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 on 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 evaling 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 propertes. 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
`s`	square

Marker Specifier	Description
`d`	diamond
`^`	upward pointing triangle
`v`	downward pointing triangle
`>`	right pointing triangle
`<`	left pointing triangle
`p`	five-pointed star (pentagram)
`h`	six-pointed star (hexagram)
`none`	no marker (default)

`MarkerEdgeColor` `ColorSpec | none | {auto}`

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

Line's parent. The handle of the line object's parent axes. You can move a line object to another axes by changing this property to the new axes handle.

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 highlight 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 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 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 have the same number of rows. (See Examples).

YData vector or matrix of coordinates

Y-coordinates. A vector of *y*-coordinates defining the line. `XData` and `ZData` must 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.

See Also

`axes`,`newplot`, `plot`, `plot3`

LineSpec

Purpose

Line specification syntax

Description

This page describes how to specify the properties of lines used for plotting. MATLAB enables you to define many characteristics including:

- Line style
- Line width
- Color
- Marker type
- Marker size
- Marker face and edge coloring (for filled markers)

MATLAB defines string specifiers for line styles, marker types, and colors. The following tables list these specifiers.

Line Style Specifiers

Specifier	Line Style
-	solid line (default)
--	dashed line
:	dotted line
-.	dash-dot line

Marker Specifiers

Specifier	Marker Type
+	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)

Color Specifiers

Specifier	Color
`r`	red
`g`	green
`b`	blue
`c`	cyan
`m`	magenta
`y`	yellow
`k`	black
`w`	white

Many plotting commands 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.

If you specify a marker, but not a line style, MATLAB plots only the markers. 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:

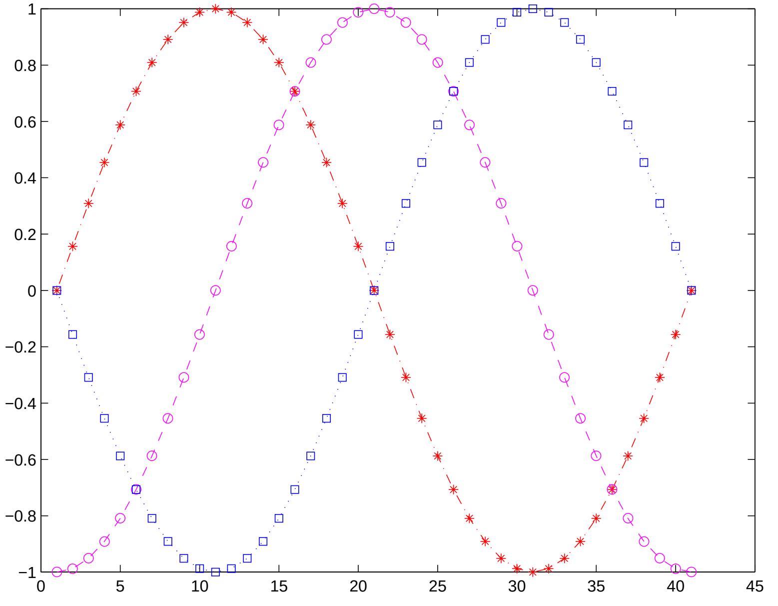
- `LineWidth` – specifies the width (in points) of the line
- `MarkerEdgeColor` – specifies the color of the marker or the edge color forfilled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- `MarkerFaceColor` – specifies the color of the face of filled markers.
- `MarkerSize`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

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 `ColorSpec` for more information on color.

Examples

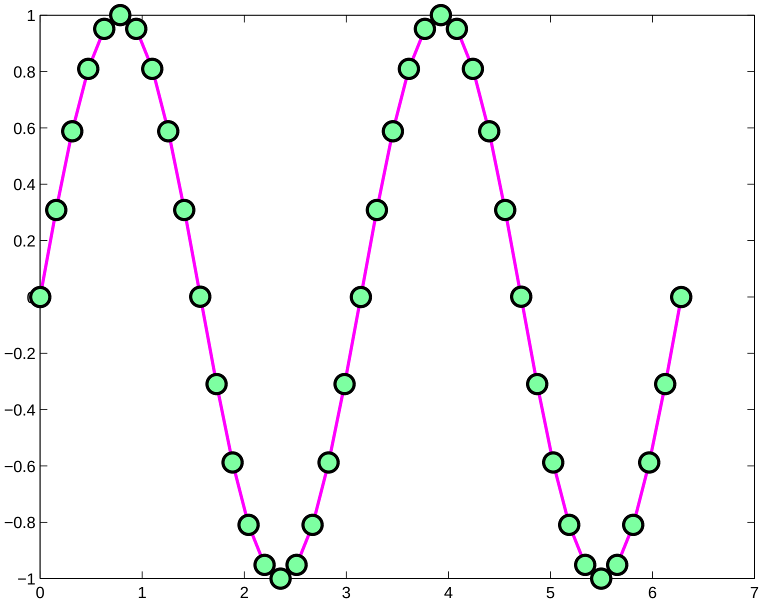
Plot the sine function over three different ranges using different line styles, colors, and markers.

```
t = 0:pi/20:2*pi;
plot(t,sin(t),'-.r*')
hold on
plot(sin(t-pi/2),'--mo')
plot(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 `line`, `plot`, `patch`, `set`, `surface`, axes `LineStyleOrder` property

linspace

Purpose Generate linearly spaced vectors

Syntax

```
y = linspace(a,b)
y = linspace(a,b,n)
```

Description The `linspace` function generates linearly spaced vectors. It is similar to the colon operator ":", but gives direct control over the number of points.

`y = linspace(a,b)` generates a row vector `y` of 100 points linearly spaced between and including `a` and `b`.

`y = linspace(a,b,n)` generates a row vector `y` of `n` points linearly spaced between and including `a` and `b`.

See Also `logspace`

The colon operator `:`

Purpose

Create 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.
`'fus'`	Frame/uicontrol spacing, in pixels. Default is 8.
`'ffs'`	Frame/figure spacing, in pixels. Default is 8.

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)
```

See Also

`dir`

Purpose

Load workspace variables from disk

Syntax

```
load
load filename
load filename X Y Z
load filename –ascii
load filename –mat
S = load(...)
```

Description

`load` loads all the variables from the MAT-file `matlab.mat`, if it exists, and returns an error if it doesn't exist.

`load filename` loads all the variables from `filename` given a full pathname or a `MATLABPATH` relative partial pathname. If `filename` has no extension, `load` looks for file named `filename` or `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 from the MAT-file. The wildcard '*' loads variables that match a pattern (MAT-file only).

`load –ascii filename` or `load –mat filename` forces `load` to treat the file as either an ASCII file or a MAT-file, regardless of file extension. With `-ascii`, `load` returns an error if the file is not numeric text. With `-mat`, `load` returns an error if the file is not a MAT-file.

`load filename.ext` reads ASCII files that contain rows of space-separated values. The resulting data is placed into a variable with the same name as the file (without the extension). ASCII files may contain MATLAB comments (lines that begin with `%`).

If `filename` is a MAT-file, `load` creates the requested variables from `filename` in the workspace. If `filename` is not a MAT-file, load creates a double precision array with a name based on `filename`. load replaces leading underscores or digits in `filename` with an `X` and replaces other non-alphabetic character with underscores. The text file must be organized as a rectangular table of numbers, separated by blanks, with one row per line, and an equal number of elements in each row.

`S = load(...)` returns the contents of a MAT-file in the variable `S`. If the file is a MAT-file, `S` is a struct containing fields that match the variables in retrieved. When the file contains ASCII data, `S` is a double-precision array.

Use the functional form of `load`, such as `load('filename')`, when the file name is stored in a string, when an output argument is requested, or if `filename` contains spaces. To specify an command line option with this functional form, specify the option as a string argument, including the hyphen. For example,

```
load('myfile.dat','-mat')
```

Remarks

MAT-files are double-precision binary MATLAB format files created by the `save` command and readable by the `load` command. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the disparate formats allow. They can also be manipulated by other programs, external to MATLAB.

The Application Program Interface Libraries contain C- and Fortran-callable routines to read and write MAT-files from external programs.

See Also

`fprintf`, `fscanf`, `partialpath`, `save`, `spconvert`

Purpose Load serial port objects and variables into the MATLAB workspace

Syntax

```
load filename
load filename obj1 obj2...
out = load('filename','obj1','obj2',...)
```

Arguments

`filename`	The MAT-file name.
`obj1 obj2...`	Serial port objects or arrays of serial port objects.
`out`	A structure containing the specified serial port objects.

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.

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

```
help serial/private/load
```

Example

Suppose you create the serial port objects `s1` and `s2`, configure a few properties for `s1`, and connect both objects to their instruments.

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s1,'Parity','mark','DataBits',7)
fopen(s1)
fopen(s2)
```

Save s1 and s2 to the file MyObject.mat, and then load the objects into the workspace using new variables.

```
save MyObject s1 s2
news1 = load MyObject s1
news2 = load('MyObject','s2')
```

Values for read-only properties are restored to their default values upon loading, while all other properties values are honored.

```
get(news1,{'Parity','DataBits','Status'})
ans =
    'mark'    [7]    'closed'
get(news2,{'Parity','DataBits','Status'})
ans =
    'none'    [8]    'closed'
```

See Also

Functions

save

Properties

Status

Purpose

User-defined extension of the `load` function for user objects

Syntax

```
b = loadobj(a)
```

Description

`b = loadobj(a)` extends the `load` function for user objects. When an object is loaded from a MAT file, the `load` function calls the `loadobj` method for the object's class if it is defined. The `loadobj` method must have the calling sequence shown; the input argument `a` is the object as loaded from the MAT file and the output argument `b` is the object that the `load` function will load into the workspace.

These steps describe how an object is loaded from a MAT file into the workspace:

1. The `load` function detects the object `a` in the MAT file.
2. The `load` function looks in the current workspace for an object of the same class as the object `a`. If there isn't an object of the same class in the workspace, `load` calls the default constructor, registering an object of that class in the workspace. The default constructor is the constructor function called with no input arguments.
3. The `load` function checks to see if the structure of the object `a` matches the structure of the object registered in the workspace. If the objects match, `a` is loaded. If the objects don't match, `load` converts `a` to a structure variable.
4. The `load` function calls the `loadobj` method for the object's class if it is defined. `load` passes the object `a` to the `loadobj` method as an input argument. Note, the format of the object `a` is dependent on the results of step 3 (object or structure). The output argument of `loadobj`, `b`, is loaded into the workspace in place of the object `a`.

Remarks

`loadobj` can be overloaded only for user objects. `load` will not call `loadobj` for built-in datatypes (such as `double`).

`loadobj` is invoked separately for each object in the MAT file. The `load` function recursively descends cell arrays and structures applying the `loadobj` method to each object encountered.

See Also

`load`, `save`, `saveobj`

log

Purpose Natural logarithm

Syntax `Y = log(X)`

Description The `log` function operates element-wise on arrays. Its domain includes complex and negative numbers, which may lead to unexpected results if used unintentionally.

`Y = log(X)` returns the natural logarithm of the elements of `X`. For complex or negative z, where $z = x + y*i$, the complex logarithm is returned:

```
log(z) = log(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`

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`

log10

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 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, returns the values of `A` at the indices where the real part of `B` is nonzero. `B` must be the same size as `A`.

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 double array, `A`, to a strictly 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))
??? Index into matrix is negative or zero.
```

See Also `islogical`, `logical operators`

Logical Operators & | ~

Purpose Logical operations

Syntax

```
A & B
A | B
~A
```

Description The symbols &, |, and ~ are the logical operators AND, OR, and NOT. They work element-wise on arrays, with 0 representing logical false (F), and anything nonzero representing logical true (T). The & operator does a logical AND, the | operator does a logical OR, and ~A complements the elements of A. The function xor(A,B) implements the exclusive OR operation. Truth tables for these operators and functions follow.

Inputs		**and**	**or**	**xor**	**NOT**
A	B	A&B	A|B	xor(A,B)	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

The precedence for the logical operators with respect to each other is:

1. not has the highest precedence.
2. and and or have equal precedence, and are evaluated from left to right.

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

and	A&B	and(A,B)
or	A|B	or(A,B)
not	~A	not(A)

Precedence of & and |

MATLAB's left to right execution precedence causes a|b&c to be equivalent to (a|b)&c. However, in most programming languages, a|b&c is equivalent to a|(b&c), that is, & takes precedence over |. To ensure compatibility with future versions of MATLAB, you should use parentheses to explicity specify the intended precedence of statements containing combinations of & and |.

Examples

Here are two examples that illustrate the precedence of the logical operators to each other:

```
1 | 0 & 0 = 0
0 & 0 | 1 = 1
```

See Also

`all`, `any`, `find`, `logical`, `xor`

The relational operators: <, <=, >, >=, ==, ~=ì

loglog

Purpose Log-log scale plot

Syntax

```
loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,LineSpec,...)
loglog(...,'PropertyName',PropertyValue,...)
h = loglog(...)
```

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 line graphics objects created by `loglog`. See the `line` reference page for more information.

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

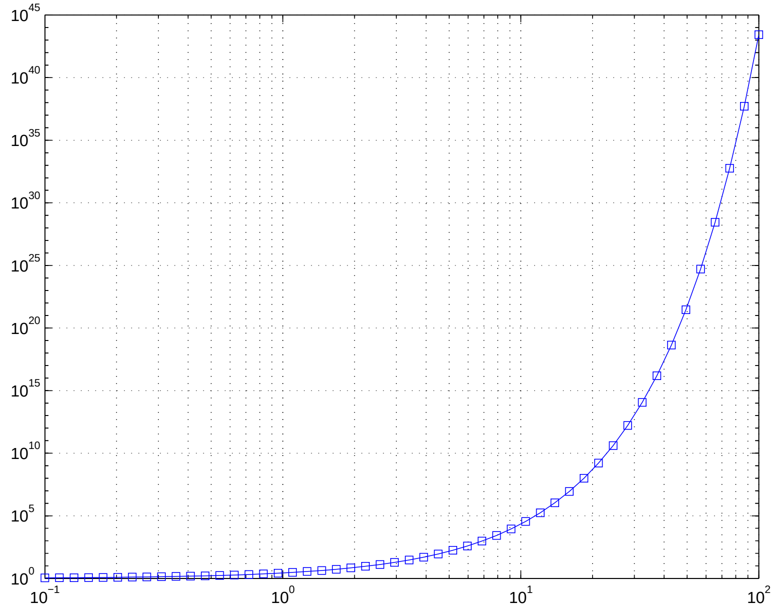
Remarks

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.

Examples

Create a simple loglog plot with square markers.

```
x = logspace(-1,2);
loglog(x,exp(x),'-s')
grid on
```



See Also

line, LineSpec, plot, semilogx, semilogy

logm

Purpose

Matrix logarithm

Syntax

```
Y = logm(X)
[Y,esterr] = logm(X)
```

Description

`Y = logm(X)` returns the matrix logarithm: the inverse function of `expm(X)`. Complex results are produced if `X` has negative eigenvalues. A warning message is printed if the computed `expm(Y)` is not close to `X`.

`[Y,esterr] = logm(X)` does not print any warning message, but returns an estimate of the relative residual, `norm(expm(Y)-X)/norm(X)`.

Remarks

If `X` is real symmetric or complex Hermitian, then so is `logm(X)`.

Some matrices, like `X = [0 1; 0 0]`, do not have any logarithms, real or complex, and `logm` cannot be expected to produce one.

Limitations

For most matrices:

```
logm(expm(X)) = X = expm(logm(X))
```

These identities may fail for some `X`. For example, if the computed eigenvalues of `X` include an exact zero, then `logm(X)` generates infinity. Or, if the elements of `X` are too large, `expm(X)` may overflow.

Examples

Suppose `A` is the 3-by-3 matrix

```
     1     1     0
     0     0     2
     0     0    -1
```

and `X = expm(A)` is

```
X =

    2.7183    1.7183    1.0862
         0    1.0000    1.2642
         0         0    0.3679
```

Then `A = logm(X)` produces the original matrix `A`.

```
A =
```

```
    1.0000    1.0000    0.0000
         0         0    2.0000
         0         0   -1.0000
```

But `log(X)` involves taking the logarithm of zero, and so produces

```
ans =

    1.0000    0.5413    0.0826
      -Inf         0    0.2345
      -Inf      -Inf   -1.0000
```

Algorithm

The matrix functions are evaluated using an algorithm due to Parlett, which is described in [1]. The algorithm uses the Schur factorization of the matrix and may give poor results or break down completely when the matrix has repeated eigenvalues. A warning message is printed when the results may be inaccurate.

See Also

`expm`, `funm`, `sqrtm`

References

[1] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Johns Hopkins University Press, 1983, p. 384.

[2] Moler, C. B. and C. F. Van Loan, “Nineteen Dubious Ways to Compute the Exponential of a Matrix,” *SIAM Review* 20, 1979,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 the specified keyword in all help entries

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 MATLAB's 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 directory, `which` finds the directory containing a given function or file, and `lookfor` finds all functions in all directories that might have something to do with a given keyword.

Even more extensive than the `lookfor` function is the Find feature in the Current Directory browser. It looks for all occurrences of a specified word in all the M-files in the current directory. See "Finding and Replacing Content Within Files" for instructions.

See Also `dir`, `doc`, `filebrowser`, `help`, `helpdesk`, `helpwin`, `what`, `which`, `who`

lower

Purpose Convert string to lower case

Syntax

```
t = lower('str')
B = lower(A)
```

Description `t = lower('`*`str`*`')` returns the string formed by converting any upper-case characters in *str* to the corresponding lower-case 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: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

See Also `upper`

Purpose List directory on UNIX

Syntax

```
ls
```

Description `ls` displays the results of the `ls` command on UNIX. You can pass any flags to `ls` that your operating system supports. On UNIX, `ls` returns a `\n` delimited string of filenames. On all other platforms, `ls` executes `dir`.

See Also `dir`

lscov

Purpose Least squares solution in the presence of known covariance

Syntax

```
x = lscov(A,b,V)
[x,dx] = lscov(A,b,V)
```

Description `x = lscov(A,b,V)` returns the vector `x` that solves `A*x = b + e` where `e` is normally distributed with zero mean and covariance `V`. Matrix `A` must be `m`-by-`n` where `m > n`. This is the over-determined least squares problem with covariance `V`. The solution is found without inverting `V`.

`[x,dx] = lscov(A,b,V)` returns the standard errors of `x` in `dx`. The standard statistical formula for the standard error of the coefficients is:

```
mse = B'*(inv(V)-inv(V)*A*inv(A'*inv(V)*A)*A'*inv(V))*B./(m-n)
dx = sqrt(diag(inv(A'*inv(V)*A)*mse))
```

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`.

See Also `lsqnonneg`, `qr`

The arithmetic operator `\`

Reference

Strang, G., *Introduction to Applied Mathematics*, Wellesley-Cambridge, 1986, p. 398.

Purpose

Linear least squares with nonnegativity constraints

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) dislays 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, `C*x-d`.

`[x,resnorm,residual,exitflag] = lsqnonneg(...)` returns a value `exitflag` that describes the exit condition of `lsqnonneg`:

>0	Indicates that the function converged to a solution `x`.
0	Indicates that the iteration count was exceeded. Increasing the tolerance (`TolX` parameter in `options`) may lead to a solution.

`[x,resnorm,residual,exitflag,output] = lsqnonneg(...)` returns a structure `output` that contains information about the operation:

`output.algorithm`	The algorithm used
`output.iterations`	The number of iterations taken

`[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)` returns the dual vector (Lagrange multipliers) `lambda`, where `lambda(i)<=0` when `x(i)` is (approximately) `0`, and `lambda(i)` is (approximately) `0` when `x(i)>0`.

Examples

Compare the unconstrained least squares solution to the `lsqnonneg` solution for a 4-by-2 problem:

```
C = [
    0.0372    0.2869
    0.6861    0.7071
    0.6233    0.6245
    0.6344    0.6170];

d = [
    0.8587
    0.1781
    0.0747
    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.

lsqr

Purpose LSQR implementation of Conjugate Gradients on the Normal Equations

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)
lsqr(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...)
[x,flag] = lsqr(A,b,...)
[x,flag,relres] = lsqr(A,b,...)
[x,flag,relres,iter] = lsqr(A,b,...)
[x,flag,relres,iter,resvec] = lsqr(A,b,...)
```

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 the column vector `b` must have length `m`. `A` can be a function `afun` such that `afun(x)` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

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. If `maxit` is `[]`, then `lsqr` uses the default, `min([m,n,20])`.

`lsqr(A,b,tol,maxit,M1)` 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 `x = M*y`. If `M` is `[]` then `lsqr` applies no preconditioner. `M` can be a function `mfun` such that `mfun(x)` 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.

`lsqr(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...)` passes parameters `p1,p2,...` to functions `afun(x,p1,p2,...)` and `afun(x,p1,p2,...,'transp')` and similarly to the preconditioner functions `m1fun` and `m2fun`.

`[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 the `flag` output is specified.

`[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)`.

Examples

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
```

```
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = lsqr(A,b,tol,maxit,M1,M2,[]);
lsqr converged at iteration 12 to a solution with relative
residual 3.5e-009
```

Alternatively, use this matrix-vector product function

```
function y = afun(x,n,transp_flag)
if (nargin > 2) & strcmp(transp_flag,'transp')
   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);
else
   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
```

as input to `lsqr`.

```
x1 = lsqr(@afun,b,tol,maxit,M1,M2,[],n);
```

See Also

`bicg`, `bicgstab`, `cgs`, `gmres`, `minres`, `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 LU matrix factorization

Syntax

```
[L,U] = lu(X)
[L,U,P] = lu(X)
lu(X)
lu(X, thresh)
```

Description The `lu` function expresses a square matrix `X` 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.

`[L,U] = lu(X)` returns an upper triangular matrix in `U` and a psychologically lower triangular matrix (i.e., a product of lower triangular and permutation matrices) in `L`, so that `X = L*U`.

`[L,U,P] = lu(X)` returns an upper triangular matrix in `U`, a lower triangular matrix in `L`, and a permutation matrix in `P`, so that `L*U = P*X`.

`lu(X)` returns the output from the LAPACK routine `DGETRF` or `ZGETRF`.

`lu(X,thresh)` controls pivoting for sparse matrices, where `thresh` is a pivot threshold in `[0,1]`. Pivoting occurs when the diagonal entry in a column has magnitude less than `thresh` times the magnitude of any sub-diagonal entry in that column. `thresh = 0` forces diagonal pivoting. `thresh = 1` is the default.

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

`X`	Square matrix to be factored.
`thresh`	Pivot threshold for sparse matrices. Valid values are `in [0,1]`. The default is `1`.
`L`	A factor of `X`. Depending on the form of the function, `L` is either lower triangular, or else the product of a lower triangular matrix with a permutation matrix `P`.

U An upper triangular matrix that is a factor of X.

P The permutation matrix satisfying the equation L*U = P*X.

Examples

Start with

```
A =
     1    2    3
     4    5    6
     7    8    0
```

To see the LU factorization, call lu with two output arguments:

```
[L,U] = lu(A)

L =

     0.1429    1.0000         0
     0.5714    0.5000    1.0000
     1.0000         0         0

U =

     7.0000    8.0000    0.0000
          0    0.8571    3.0000
          0         0    4.5000
```

Notice that L is a permutation of a lower triangular matrix that has 1's on the permuted diagonal, and that U is upper triangular. To check that the factorization does its job, compute the product:

```
L*U
```

which returns the original A. Using three arguments on the left-hand side to get the permutation matrix as well

```
[L,U,P] = lu(A)
```

returns the same value of U, but L is reordered:

```
L =

     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
```

To verify that `L*U` is a permuted version of `A`, compute `L*U` and subtract it from `P*A`:

```
P*A - L*U
```

The inverse of the example matrix, `X = inv(A)`, is actually computed from the inverses of the triangular factors:

```
X = inv(U)*inv(L)
```

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
```

Algorithm

`lu` uses the subroutines `DGETRF` (real) and `ZGETRF` (complex) from LAPACK.

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*, Third Edition, SIAM, Philadelphia, 1999.

Purpose Incomplete LU matrix factorizations

Syntax

```
luinc(X,'0')
[L,U] = luinc(X,'0')
[L,U,P] = luinc(X,'0')
luinc(X,droptol)
luinc(X,options)
[L,U] = luinc(X,options)
[L,U] = luinc(X,droptol)
[L,U,P] = luinc(X,options)
[L,U,P] = luinc(X,droptol)
```

Description

`luinc` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`luinc(X,'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 `X`, and their product agrees with the permuted `X` over its sparsity pattern. `luinc(X,'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(X,'0')) = nnz(X)`, with the possible exception of some zeros due to cancellation.

`[L,U] = luinc(X,'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 `X` 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(X)+n`, where `X` is `n-by-n`.

The product `L*U` agrees with `X` over its sparsity pattern. `(L*U).*spones(X)-X` has entries of the order of `eps`.

`[L,U,P] = luinc(X,'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 the permuted `X`

```
spones(L) = spones(tril(P*X))
```

with the possible exceptions of 1s on the diagonal of L where P*X may be zero, and zeros in L due to cancellation where P*X may be nonzero. U has the same sparsity pattern as the upper triangle of P*X

```
spones(U) = spones(triu(P*X))
```

with the possible exceptions of zeros in U due to cancellation where P*X may be nonzero. The product L*U agrees within rounding error with the permuted matrix P*X over its sparsity pattern. (L*U).*spones(P*X)-P*X has entries of the order of eps.

luinc(X,droptol) computes the incomplete LU factorization of any sparse matrix using a drop tolerance. droptol must be a non-negative scalar. luinc(X,droptol) produces an approximation to the complete LU factors returned by lu(X). 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(X).

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 X)

```
droptol*norm(X(:,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(X,options) specifies a structure with up to four fields that may be used in any combination: droptol, milu, udiag, thresh. Additional fields of options are ignored.

droptol is the drop tolerance of the incomplete factorization.

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.

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` is the 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 desribed in greater detail in `lu`.

`luinc(X,options)` is the same as `luinc(X,droptol)` if options has `droptol` as its only field.

`[L,U] = luinc(X,options)` returns a permutation of a unit lower triangular matrix in `L` and an upper trianglar matrix in `U`. The product `L*U` is an approximation to `X`. `luinc(X,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] = luinc(X,options)` is the same as `luinc(X,droptol)` if options has `droptol` as its only field.

`[L,U,P] = luinc(X,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((X:,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(X(:,j))/U(j,j).
```

The product `L*U` is an approximation to the permuted `P*X`.

`[L,U,P] = luinc(X,options)` is the same as `[L,U,P] = luinc(X,droptol)` if `options` has `droptol` as its only field.

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.

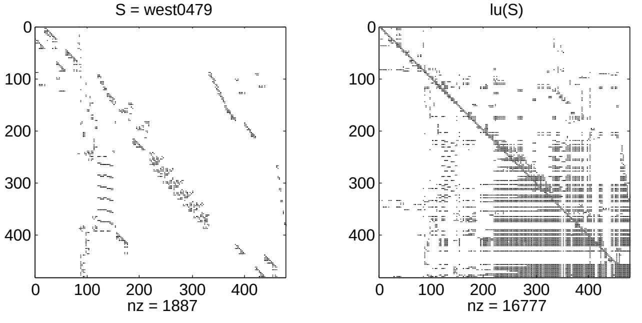
Limitations

luinc(X,'0') works on square matrices only.

Examples

Start with a sparse matrix and compute its LU factorization.

```
load west0479;
S = west0479;
LU = lu(S);
```

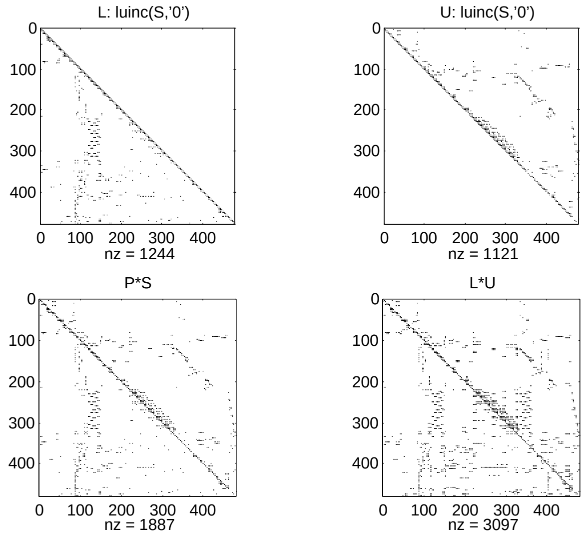


Compute the incomplete LU factorization of level 0.

```
[L,U,P] = luinc(S,'0');
D = (L*U).*spones(P*S)-P*S;
```

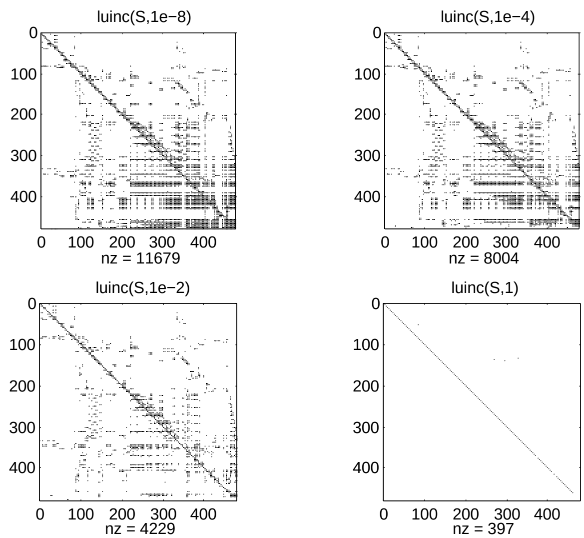
spones(U) and spones(triu(P*S)) are identical.

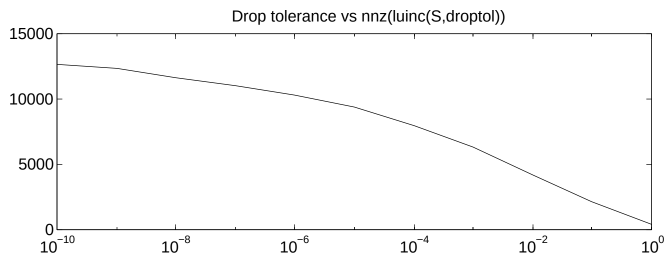
spones(L) and spones(tril(P*S)) disagree at 73 places on the diagonal, where L is 1 and P*S is 0, and also at position (206,113), where L is 0 due to cancellation, and P*S is -1. D has entries of the order of eps.

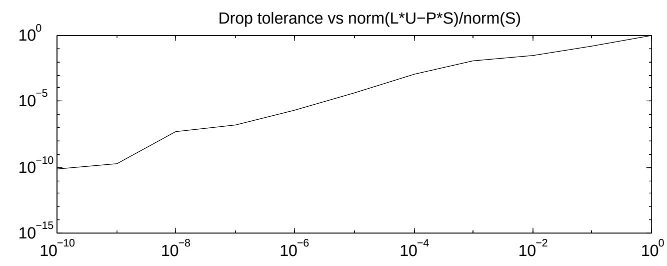



```
[IL0,IU0,IP0] = luinc(S,0);
[IL1,IU1,IP1] = luinc(S,1e-10);
                    .
                    .
                    .
```

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)
nz = 11679
luinc(S,1e−4)
nz = 8004
luinc(S,1e−2)
nz = 4229
luinc(S,1)
nz = 397


Drop tolerance vs nnz(luinc(S,droptol))


Drop tolerance vs norm(L*U−P*S)/norm(S)

Algorithm

`luinc(X,'0')` is based on the "KJI" variant of the LU factorization with partial pivoting. Updates are made only to positions which are nonzero in `X`.

`luinc(X,droptol)` and `luinc(X,options)` are based on the column-oriented `lu` for sparse matrices.

See Also

`lu`, `cholinc`, `bicg`

References

Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

magic

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:

- one for odd `n`
- one for even `n` not divisible by four
- one for even `n` divisible by four.

To make this apparent, type:

```
for n = 3:20
    A = magic(n);
    plot(A,'-');
    r(n) = rank(A);
end
r
```

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`

mat2str

Purpose Convert a matrix into a string

Syntax

```
str = mat2str(A)
str = mat2str(A,n)
```

Description `str = mat2str(A)` converts matrix `A` into a string, suitable for input to the `eval` function, using full precision.

`str = mat2str(A,n)` converts matrix `A` using `n` digits of precision.

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 Consider the matrix:

```
A =
     1     2
     3     4
```

The statement

```
b = mat2str(A)
```

produces:

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

where `b` is a string of 11 characters, including the square brackets, spaces, and a semicolon.

`eval(mat2str(A))` reproduces `A`.

See Also `int2str`, `sprintf`, `str2num`

Purpose

Controls the 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 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, 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`

Purpose Start MATLAB (UNIX systems only)

Syntax

```
matlab [-h|-help] | [-n] [-arch | -ext | -arch/ext]
[-c licensefile] [-display Xdisplay | -nodisplay]
[-nosplash] [-mwvisual visualid] [-debug]
[-nodesktop | -nojvm] [-runtime] [-check_malloc]
[-r MATLAB_command] [-Ddebugger [options]]
```

Description

`matlab` is a Bourne shell script that starts the MATLAB executable. (In this document, `matlab` refers to this script; MATLAB refers to the application program). The script determines, filters, and passes information to the executable through two mechanisms:

- Arguments to the executable
- Variables passed through the environment to the executable

There are three ways in which you can control these two mechanisms:

- By specifying command-line options to the script
- By assigning values in the MATLAB startup file `.matlab6rc.sh`
- By assigning values to environment variables before executing the script

All arguments that ultimately get passed to the executable must be passed in as arguments to the script.

To affect the values of variables passed through the environment can be more complicated. Not all variables can be affected in the same way. The following decision hierarchy (from highest to lowest precedence) is used whenever possible to determine the final values of the environment variables passed to the MATLAB executable.

1. An argument has been passed to the script that controls the variable.
2. The variable is assigned in the .matlab6rc.sh startup file.
3. The variable is assigned in the environment before the script was executed.
4. The variable is assigned a default value in the script.

When invoked, `matlab` looks for the first occurrence of `.matlab6rc.sh` in the current directory, in the home directory (`$HOME`) and in the `$MATLAB/bin` directory, where the template version of `.matlab6rc.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 system wide, 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 essential environment variables and standard assignment behavior. See the comments and the code in the `.matlab6rc.sh` file for more information relevant to the affect this file has on these variables. Note that not all variables are referenced in the shipping `.matlab6rc.sh`.

Variable	Definition and Standard Assignment Behavior
`ARCH`	The machine architecture. The value ARCH passed in 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 `.matlab6rc.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. `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 five sublists each of which could be empty. The first sublist is defined in .matlab6rc.sh as LDPATH_PREFIX. The second sublist is computed in the script and includes directories inside the MATLAB root directory and relevant Java directories. The third sublist contains any nonempty value of LD_LIBRARY_PATH from the environment possibly augmented in .matlab6rc.sh. The fourth sublist contains system libraries. The final sublist is defined in .matlab6rc.sh as LDPATH_SUFFIX.
LM_LICENSE_FILE	The FLEXlm license variable. The license file passed with the -c argument to the script is used, otherwise the value set in .matlab6rc.sh. In general, the final value is a colon separated list of license files and/or port@host entries. The shipping .matlab6rc.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 $MATLAB/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 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 .matlab6rc.sh. MATLABdefault is not reset in the shipping .matlab6rc.sh.
MATLAB_DEBUG	Normally set to the name of debugger. The -Ddebugger argument passed in 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 .matlab6rc.sh is used first then any nonempty value from the environment. There is no value set in the shipping .matlab6rc.sh so that environment alone is used.
MATLAB_MEM_MGR	Turns on MATLAB memory integrity checking. The -check_malloc argument passed in to the script sets this variable to 'debug'. Otherwise, a nonempty value set in .matlab6rc.sh is used, or a nonempty value in the environment is used. If a nonempty value is not found, the variable is not exported to the environment.
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.

Variable	Definition and Standard Assignment Behavior
SHELL	The shell to use when "!" or a Unix command is issued in MATLAB. This is taken from the environment unless SHELL is reset in .matlab6rc.sh. SHELL is not reset in the shipping .matlab6rc.sh. If SHELL is empty or not defined then MATLAB uses /bin/sh internally.
TOOLBOX	Path of the toolbox directory. A nonempty value in the environment is used first. Otherwise, $MATLAB/toolbox, computed by the script is used, unless TOOLBOX is reset in .matlab6rc.sh. TOOLBOX is not reset in the shipping .matlab6rc.sh.
XAPPLRESDIR	The X application resource directory. A nonempty value in the environment is used first unless XAPPLRESDIR is reset in .matlab6rc.sh. Otherwise, $MATLAB/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 .matlab6rc.sh. Otherwise, $MATLAB/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 may 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.

Options

The following table describes `matlab` command line options, grouped by function.

Option	Function
Diagnostic Options	
`-h |-help`	Display `matlab` command usage. MATLAB is not started when you specify this option.
`-n`	Display all the final values of the environment variables and arguments passed to the MATLAB executable as well as other diagnostic information. MATLAB is not started when you specify this option.
Select Which MATLAB Executable to Run	
`-arch`	Run MATLAB assuming architecture `arch`.
`-ext`	Run the version of MATLAB with extension `ext`, if it exists.
`-arch/ext`	Run the version of MATLAB with extension `ext,` if it exists, assuming `arch` identifies the architecture.
Affect the Environment Variables Passed to the MATLAB Executable	
`-c licensefile`	Set the value of the `LM_LICENSE_FILE` environment variable to `licensefile`. `licensefile` can be a colon separated list of files or `port@host` entries, or both. For more information, see `LM_LICENSE_FILE` in the variable table.

Option	Function
-check_malloc	Set the value of the MATLAB_MEM_MGR environment variable to 'debug'. This turns on MATLAB memory integrity checking. For more information, see MATLAB_MEM_MGR in the variable table.
-display *Xserver*	Define the X display used for MATLAB output, *Xserver* has the form hostname:display. For example, `matlab -display falstaff:0` causes MATLAB output to be displayed on the host named falstaff. This setting supersedes the value of the DISPLAY environment variable and the value of the DISPLAY variable defined in .matlab5rc.sh.
Passed Without Change to the MATLAB Executable	
-debug	Provide debugging information, especially for X-based problems. Note that you should use this option only in conjunction with a Technical Support representative from The MathWorks, Inc.
-mwvisual visualid	The default X visual to use for figure windows.
-nodesktop	Do not start the MATLAB desktop. Use the current window for commands. The Java Virtual Machine (JVM) will be started.
-nojvm	Shut off all Java support by not starting the Java Virtual Machine (JVM). In particular, the MATLAB desktop will not be started.
-nosplash	Do not display the splash screen during startup.

Option	Function
`-r MATLAB_command`	Start MATLAB and execute the MATLAB command. An arbitrary MATLAB command string can be used here. You will have to use proper quoting to make the input legal for both the shell and MATLAB.
`-runtime`	Run MATLAB in Runtime Server mode. Required only if MATLAB is run from the Runtime Development Kit.
Debugging	
`-D`*debugger* `[options]`	Start MATLAB with the specified `debugger` (e.g. dbx, gdb, dde, xdb, cvd). A full path can be specified for debugger. The options cover ONLY those that go after the executable to be debugged in the syntax of the actual debug command and for most debuggers this is very limited. To customize your debugging session use a startup file. See your debugger documentation for details. The `MATLAB_DEBUG` environment variable is set to the filename part of the debugger argument. For more information, see `MATLAB_DEBUG` in the variable table above.

Any arguments that the `matlab` script cannot recognize are passed without change to the MATLAB executable. If the MATLAB executable does not recognize the arguments they will be ignored without warning.

If you are debugging, these arguments should be used as part of a command inside the debugger like `run` and not used when running the `matlab` script. If any of the options are placed before the `-Ddebugger` argument, they will be handled as if they were part of the options after the `-Ddebugger` argument and will be treated as illegal options by most debuggers.

See Also `mex`

Purpose

MATLAB startup M-file

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 MATLAB's search path.

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 Windows shortcut for MATLAB.

Examples

Example – 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

`quit`, `startup`
"Startup Options" in "Working Environment for MATLAB"

matlabroot

Purpose

Return root directory of MATLAB installation

Syntax

```
matlabroot
rd = matlabroot
```

Description

`matlabroot` returns the name of the directory in which the MATLAB software is installed.

`rd = matlabroot` returns the name of the directory in which the MATLAB software is installed and assigns it to `rd`.

Examples

```
fullfile(matlabroot,'toolbox','matlab','general','')
```

produces a full path to the `toolbox/matlab/general` directory that is correct for the platform it is executed on.

Purpose Maximum elements of an 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`.

`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 modulus, computed with `max(abs(A))`. The `max` function ignores `NaN`s.

See Also `isnan`, `mean`, `median`, `min`, `sort`

mean

Purpose

Average or mean value of arrays

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*.

Examples

```
A = [1 2 4 4; 3 4 6 6; 5 6 8 8; 5 6 8 8];
mean(A)
ans =
    3.5000    4.5000    6.5000    6.5000

mean(A,2)
ans =
    2.7500
    4.7500
    6.7500
    6.7500
```

See Also

`corrcoef`, `cov`, `max`, `median`, `min`, `std`

Purpose Median value of arrays

Syntax

```
M = median(A)
M = median(A,dim)
```

Description `M = median(A)` returns the median values of the elements along different dimensions of an array.

If `A` is a vector, `median(A)` returns the median value of `A`.

If `A` is a matrix, `median(A)` treats the columns of `A` as vectors, returning a row vector of median values.

If `A` is a multidimensional array, `median(A)` treats the values along the first nonsingleton dimension as vectors, returning an array of median values.

`M = median(A,`*`dim`*`)` returns the median values for elements along the dimension of `A` specified by scalar *`dim`*.

Examples

```
A = [1 2 4 4; 3 4 6 6; 5 6 8 8; 5 6 8 8];
median(A)

ans =

     4     5     7     7

median(A,2)

ans =

     3
     5
     7
     7
```

See Also `corrcoef`, `cov`, `max`, `mean`, `min`, `std`

memory

Purpose Help for memory limitations

Description If the `out of memory` error message is encountered, there is no more room in memory for new variables. You must free up some space before you may proceed. One way to free up space is to use the `clear` function to remove some of the variables residing in memory. Another is to issue the `pack` command to compress data in memory. This opens up larger contiguous blocks of memory for you to use.

Here are some additional system specific tips:

Windows: Increase virtual memory by using System in the Control Panel.

UNIX: Ask your system manager to increase your swap space.

See Also `clear`, `pack`

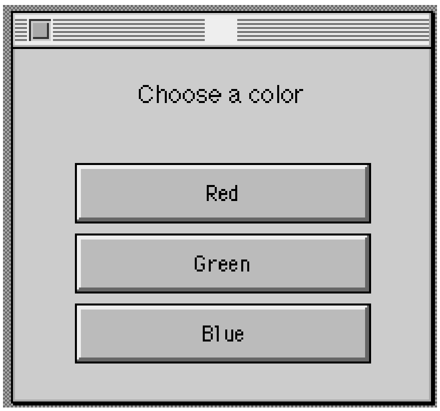
Purpose Generate a menu of choices for user input

Syntax `k = menu('`*`mtitle`*`','`*`opt1`*`','`*`opt2`*`',...,'`*`optn`*`')`

Description `k = menu('mtitle','`*`opt1`*`','`*`opt2`*`',...,'`*`optn`*`')` displays the menu whose title is in the string variable `'`*`mtitle`*`'` and whose choices are string variables *'opt1'*, *'opt2'*, and so on. `menu` returns the value you entered.

Remarks To call `menu` from another ui-object, set that object's `Interruptible` property to `'yes'`. For more information, see the *MATLAB Graphics Guide*.

Examples `k = menu('Choose a color','Red','Green','Blue')` displays



After input is accepted, use `k` to control the color of a graph.

```
color = ['r','g','b']
plot(t,s,color(k))
```

See Also `input`, `uicontrol`

mesh, meshc, meshz

Purpose

Mesh plots

Syntax

```
mesh(X,Y,Z)
mesh(Z)
mesh(...,C)
mesh(...,'PropertyName',PropertyValue,...)
meshc(...)
meshz(...)
h = mesh(...)
h = meshc(...)
h = meshz(...)
```

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.

`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 surface graphics object.

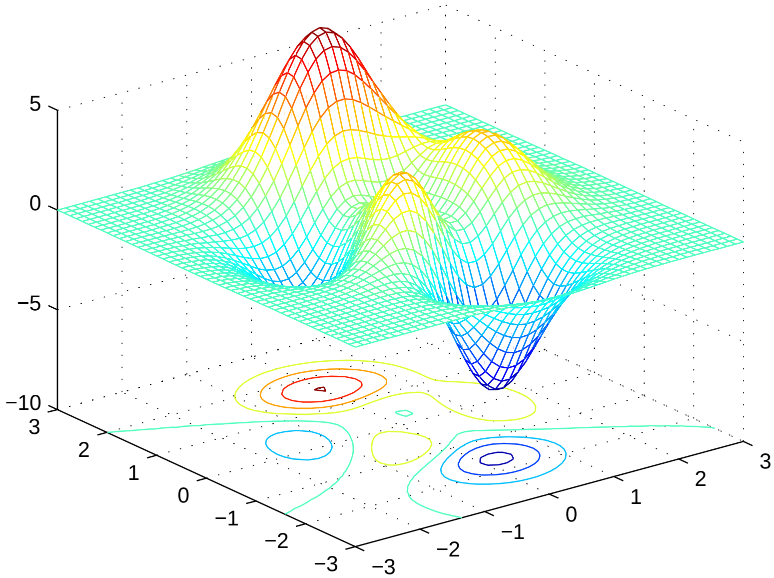
Remarks

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:

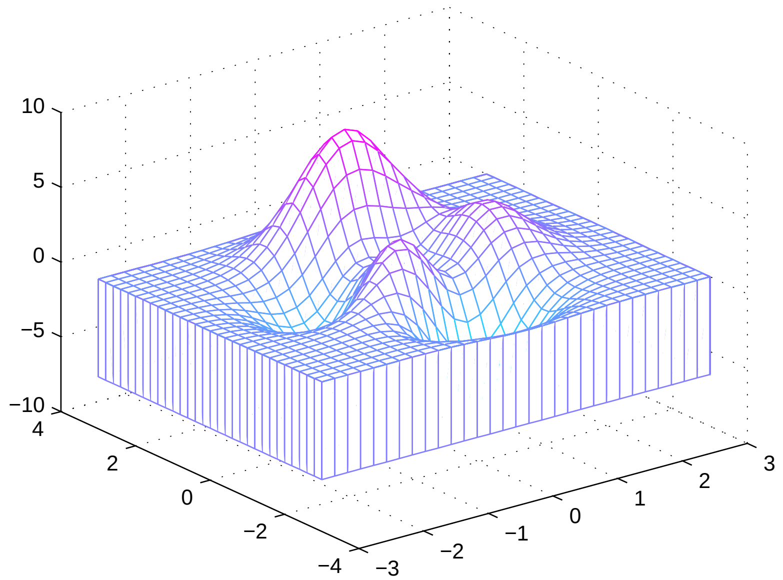
```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshc(X,Y,Z);
axis([-3 3 -3 3 -10 5])
```



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 setting of the axes XLimMode, YLimMode, and ZLimMode properties determine the axis limits. axis sets these properties.

The range of C, or the current setting 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. MATLAB's default behavior computes 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`, sruface, `surf`, `surfc`, `surfl`, `waterfall`

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`.

meshgrid

Purpose Generate X and Y matrices for three-dimensional 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
```

See Also griddata, mesh, ndgrid, slice, surf

methods

Purpose Display method names

Syntax

```
n = methods class_name
n = methods class_name -full
```

Description

n = methods('class_name') returns, in a cell array of strings, the names of all methods for the MATLAB or Java class with the name class_name.

n = methods('class_name','-full') returns, in a cell array of strings, the full description of the methods defined for the class, including inheritance information and, for Java methods, attributes and signatures. For any overloaded method, the returned array includes a description of each of its signatures. If class_name represents a MATLAB class, then inheritance information is returned only if that class has been instantiated.

Examples

To display a full description of all methods on Java object java.awt.Dimension

```
methods java.awt.Dimension -full

Methods for class java.awt.Dimension:
Dimension()
Dimension(java.awt.Dimension)
Dimension(int,int)
java.lang.Class getClass() % Inherited from java.lang.Object
int hashCode() % Inherited from java.lang.Object
boolean equals(java.lang.Object)
java.lang.String toString()
void notify() % Inherited from java.lang.Object
void notifyAll() % Inherited from java.lang.Object
void wait(long) throws java.lang.InterruptedException % Inherited
from java.lang.Object
void wait(long,int) throws java.lang.InterruptedException %
Inherited from java.lang.Object
void wait() throws java.lang.InterruptedException % Inherited
from java.lang.Object
java.awt.Dimension getSize()
void setSize(java.awt.Dimension)
void setSize(int,int)
```

See Also `methodsview`, `help`, `what`, `which`

methodsview

Purpose

Displays information on all methods implemented by a class.

Syntax

```
methodsview package_name.class_name
methodsview class_name
```

Description

`methodsview package_name.class_name` displays information describing the Java class, `class_name`, that is available from the package of Java classes, `package_name`.

`methodsview class_name` displays information describing the imported Java or MATLAB class, `class_name`.

MATLAB creates a new window in response to the `methodsview` command. This window displays all of the methods defined in the specified class. For each of these methods, the following additional information is supplied:

- Name of the method
- Method type qualifiers (for example, `abstract` or `synchronized`)
- Data type returned by the method
- Arguments passed to the method
- Possible exceptions thrown
- Parent of the specified class

Examples

The following command lists information on all methods in the `java.awt.MenuItem` class.

```
methodsview java.awt.MenuItem
```

mexext

Purpose Return the MEX-filename extension

Syntax `ext = mexext`

Description `ext = mexext` returns the filename extension for the current platform.

Examples

```
ext = mexext

ext =
dll
```

mfilename

Purpose The name of the currently running M-file

Syntax `mfilename`

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.

When called from the command line, `mfilename` returns an empty matrix.

Purpose Minimum elements of an 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`.

`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 modulus, computed with `min(abs(A))`. The `min` function ignores `NaN`s.

See Also `max`, `mean`, `median`, `sort`

minres

Purpose

Minimum Residual method

Syntax

```
x = minres(A,b)
minres(A,b,tol)
minres(A,b,tol,maxit)
minres(A,b,tol.maxit,M)
minres(A,b,tol,maxit,M1,M2)
minres(A,b,tol,maxit,M1,M2,x0)
minres(afun,b,tol,maxit,mifun,m2fun,x0,p1,p2,...)
[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. The column vector `b` must have length `n`. `A` can be a function `afun` such that `afun(x)` returns `A*x`.

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 that 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.

minres(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...) passes parameters p1,p2,... to functions afun(x,p1,p2,...), m1fun(x,p1,p2,...), and m2fun(x,p1,p2,...).

[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
```

Alternatively, use this matrix-vector product function

```
function y = afun(x,n)
y = 4 * x;
y(2:n) = y(2:n) - 2 * x(1:n-1);
y(1:n-1) = y(1:n-1) - 2 * x(2:n);
```

as input to minres.

```
x1 = minres(@afun,b,tol,maxit,M1,[],n);
```

Example 2.

Use a symmetric indefinite matrix that fails with pcg.

```
A = diag([20:-1:1, -1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or too
large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, 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), / (slash),

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., “Solution of Sparse Indefinite Systems of Linear Equations.” *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

mislocked

Purpose True if M-file cannot be cleared

Syntax

```
mislocked
mislocked(fun)
```

Description

`mislocked` by itself is 1 if the currently running M-file is locked and 0 otherwise.

`mislocked(`*fun*`)` is 1 if the function named *fun* is locked in memory and 0 otherwise. Locked M-files cannot be removed with the `clear` function.

See Also `mlock`, `munlock`

Purpose

Make a new directory

Graphical Interface

As an alternative to the mkdir function, you can use the Current Directory browser to create new folders. To open it, select **Current Directory** from the **View** menu in the MATLAB desktop.

Syntax

```
mkdir dirname
mkdir parentdir dirname
status = mkdir(...,'dirname')
[status,msg] = mkdir(...,'dirname')
```

Description

`mkdir dirname` creates the directory `dirname` in the current directory. It returns a `status` of `1` if the new directory is created successfully, `2` if it already exists. Otherwise, it returns `0`.

`mkdir parentdir dirname` creates the directory `dirname` in the existing directory `parentdir`.

`status = mkdir(...,'dirname')` returns `status` and also returns a nonempty error message string in `msg` when an error occurs.

`[status,msg] = mkdir(...,'dirname')` returns `status` and also returns a nonempty error message string in `msg` when an error occurs.

Examples

To create a subdirectory of `testdata` called `newdir`,

```
mkdir ..\testdata newdir
```

This second attempt to create the same directory fails with an error message.

```
[status,msg] = mkdir('..\testdata','newdir')
status =
     2
msg =
Directory or file newdir already exists in ..\testdata
```

See Also

`copyfile`, `filebrowser`

mlock

Purpose Prevent M-file clearing

Syntax

```
mlock
```

Description `mlock` locks the currently running M-file in memory so that subsequent `clear` functions do not remove it.

Use the `munlock` function to return the M-file to its normal, clearable state.

Locking an M-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. This can be checked 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`

Purpose Modulus (signed remainder after division)

Syntax

```
M = mod(X,Y)
```

Definition `mod(x,y)` is x mod y.

Description `M = mod(X,Y)` returns the remainder `X - Y.*floor(X./Y)` for nonzero `Y`, and returns `X` otherwise. `mod(X,Y)` always differs from `X` by a multiple of `Y`.

Remarks So long as operands `X` and `Y` are of the same sign, the function `mod(X,Y)` returns the same result as does `rem(X,Y)`. However, for positive `X` and `Y`,

```
mod(-x,y) = rem(-x,y)+y
```

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
```

Limitations Arguments `X` and `Y` should be integers. Due to the inexact representation of floating-point numbers on a computer, real (or complex) inputs may lead to unexpected results.

See Also `rem`

more

Purpose

Control paged output for the Command Window

Syntax

```
more off
more on
more(n)
```

Description

`more off` disables paging of the output in the MATLAB Command Window.

`more on` enables paging of the output in the MATLAB Command Window.

`more(n)` displays `n` lines per page.

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.
Space bar	Advance to the next page of output.
q (for quit) key	Terminate display of the text.

By default, `more` is disabled. When enabled, `more` defaults to displaying 23 lines per page.

See Also

`diary`

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 can be any of the following strings:

- `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
- `southwest` – bottom left corner
- `center` – center of screen
- `onscreen` – nearest location with respect to current location that is on screen

The *position* argument can also be a two-element vector `[h,v]`, where 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

movegui

movegui('*position*') move the callback figure (gcbf) or the current figure (gcf) to the specified position.

movegui(h) moves the figure identified by the handle h to the onscreen position.

movegui moves the callback figure (gcbf) or the current figure (gcf) to the onscreen position. This is useful as a string-based CreateFcn callback for a saved figure. It ensures the figure appears on screen when reloaded, regardless of its saved position.

Examples

This example demonstrates the usefulness of movegui to ensure that saved GUIs appear on screen when reloaded, regardless of the target computer's screen sizes and resolution. It creates a figure off the screen, assigns movegui as its CreateFcn callback, then saves and reloads the figure.

```
f = figure('Position',[10000,10000,400,300]);
set(f,'CreateFcn','movegui')
hgsave(f,'onscreenfig')
close(f)
f2 = hgload('onscreenfig');
```

See Also

guide

Creating GUIs

Purpose Play recorded movie frames

Syntax

```
movie(M)
movie(M,n)
movie(M,n,fps)
movie(h,...)
movie(h,M,n,fps,loc)
```

Description `movie` plays the movie defined by a matrix whose columns are movie frames (usually produced by `getframe`).

`movie(M)` plays the movie in matrix `M` once.

`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 comprise 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 in the figure or axes identified by the handle `h`.

`movie(h,M,n,fps,loc)` specifies 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 and in units of pixels, regardless of the object's `Units` property.

Remarks The movie function displays each frame as it loads the data into memory, and then plays the movie. This eliminates long delays with a blank screen when you load a memory-intensive movie. The movie's load cycle is not considered one of the movie repetitions.

Examples 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 twenty times
movie(F,20)
```

See Also

`getframe`, `frame2im`, `im2frame`

`guide`

Creating GUIs

Purpose Create an 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`.

`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).	There is no default colormap.
`'compression'`	A text string specifying which compression codec to use.	

	On Windows:
	`'Indeo3'`
	`'Indeo5'`
	`'Cinepak'`
	`'MSVC'`
	`'RLE'`
	`'None'`
	
	On Unix:
	'None'
		`'Indeo3'`, on Windows. `'None'` on Unix.

	To use a custom compression codec, specify the four-character code that identifies the codec (typically included in the codec documentation). The `addframe` function reports an error if it can not find the specified custom compressor.	

Parameter	Value	Default
'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 key frames per second.
'name'	A descriptive name for the video stream. This parameter must be no greater than 64 characters long.	The default is the `filename`.
'quality'	A number between 0 and 100. This parameter has no effect on uncompressed movies. Higher quality numbers result in higher video quality and larger file sizes. Lower quality numbers result in lower video quality and smaller file sizes.	75

See Also

`avifile`, `aviread`, `aviinfo`, `movie`

Purpose

Allocate matrix for movie frames

Syntax

```
M = moviein(n)
M = moviein(n,h)
M = moviein(n,h,rect)
```

Description

`moviein` allocates an appropriately sized matrix for the `getframe` function.

`M = moviein(n)` creates matrix `M` having `n` columns to store `n` frames of a movie based on the size of the current axes.

`M = moviein(n,h)` specifies a handle for a valid figure or axes graphics object on which to base the memory requirement. You must use the same handle with `getframe`. If you want to capture the axis in the frames, specify `h` as the handle of the figure.

`M = moviein(n,h,rect)` specifies the rectangular area from which to copy the bitmap, relative to the lower-left corner of the figure or axes graphics object identified by `h`. `rect = [left bottom width height]`, where `left` and `bottom` specify the lower-left corner of the rectangle, and width and height specify the dimensions of the rectangle. Components of `rect` are in pixel units. You must use the same handle and rectangle with `getframe`.

Remarks

`moviein` is no longer meeded as of MATLAB Release 11 (5.3). In earlier versions, pre-allocating a movie increased performance, but there is no longer a need to do this.

See Also

`getframe`, `movie`

msgbox

Purpose Display message box

Syntax

```
msgbox(message)
msgbox(message,title)
msgbox(message,title,'icon')
msgbox(message,title,'custom',iconData,iconCmap)
msgbox(...,'createMode')
h = msgbox(...)
```

Description

`msgbox(message)` creates a message box that automatically wraps `message` to fit an appropriately sized figure. `message` is a string vector, string matrix, or cell array.

`msgbox(message,title)` specifies the title of the message box.

`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

`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.

`msgbox(...,'createMode')` specifies whether the message box is modal or nonmodal, and if it is nonmodal, whether to replace another message box with the same `title`. Valid values for `'createMode'` are `'modal'`, `'non-modal'`, and `'replace'`.

`h = msgbox(...)` returns the handle of the box in `h`, which is a handle to a Figure graphics object.

See Also `dialog`, `errordlg`, `inputdlg`, `helpdlg`, `questdlg`, `textwrap`, `warndlg`

Purpose Convert mu-law audio signal to linear

Syntax

```
y = mu2lin(mu)
```

Description `y = mu2lin(mu)` converts mu-law encoded 8-bit audio signals, stored as "flints" in the range $0 \leq mu \leq 255$, to linear signal amplitude in the range `-s < Y < s` where `s = 32124/32768 ~= .9803`. The input `mu` is often obtained using `fread(...,'uchar')` to read byte-encoded audio files. "Flints" are MATLAB's integers - floating-point numbers whose values are integers.

See Also `auread`, `lin2mu`

munlock

Purpose Allow M-file clearing

Syntax

```
munlock
munlock fun
munlock('fun')
```

Description munlock unlocks the currently running M-file in memory so that subsequent clear functions can remove it.

munlock fun unlocks the M-file named fun from memory. By default, M-files are unlocked so that changes to the M-file are picked up. Calls to munlock are needed only to unlock M-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. This can be checked 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

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.

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

Logical operations involving `NaN`s always return false, except ~= (not equal). Consequently, the statement `NaN` ~= `NaN` is true while the statement `NaN` == `NaN` is false.

See Also

Inf

nargchk

Purpose

Check number of input arguments

Syntax

```
msg = nargchk(low,high,number)
```

Description

The `nargchk` function often is used inside an M-file to check that the correct number of arguments have been passed.

`msg = nargchk(low,high,number)` returns an error message if `number` is less than `low` or greater than `high`. If `number` is between `low` and `high` (inclusive), `nargchk` returns an empty matrix.

Arguments

Input arguments to `nargchk` are

`low, high`	The minimum and maximum number of input arguments that should be passed.
`number`	The number of arguments actually passed, as determined by the `nargin` function.

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

`nargin`, `nargout`

Purpose Number of function arguments

Syntax

```
n = nargin
n = nargin('fun')
n = nargout
n = 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 M-file function *fun* or `-1` if the function has a variable of input arguments.

`nargout` returns the number of output arguments specified for a function.

`nargout('`*fun*`')` returns the number of declared outputs for the M-file function *fun*.

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(fname,lims,npts,angl,subdiv)
% MYPLOT  Plot a function.
% MYPLOT(fname,lims,npts,angl,subdiv)
%     The first two input arguments are
%     required; the other three have default values.
 ...
if nargin < 5, subdiv = 20; end
if nargin < 4, angl = 10; end
if nargin < 3, npts = 25; end
 ...
if nargout == 0
     plot(x,y)
else
     x0 = x;
```

```
        y0 = y;
    end
```

See Also `inputname`, `nargchk`

Purpose Validate number of output arguments

Syntax `msg = nargoutchk(low,high,n)`

Description `msg = nargoutchk(low,high,n)` returns an appropriate error message if `n` is not between `low` and `high`. If the number of output arguments is within the specified range, `nargoutchk` returns an empty matrix.

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 i=1:nout, varargout(i) = {s(i)}; end
else
    disp(msg)
end
```

See Also `inputname`, `nargchk`, `nargin`, `nargout`, `varargout`

nchoosek

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.

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

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

See Also

`perms`

Purpose

Generate arrays for multidimensional 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 `X`*i* are copies of elements of the vector `x`*i*.

`[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

Determine where to draw graphics objects

Syntax

```
newplot
h = newplot
```

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.

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 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 `nextplot` 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
`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.

NextPlot	What Happens
`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.
`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`	Removes all child objects (regardless of the setting of the `HandleVisibility` property) and resets 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.

Purpose

Next power of two

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`

nnls

Purpose

Nonnegative least squares

Note The `nnls` function was replaced by `lsqnonneg` in Release 11 (MATLAB 5.3). In Release 12 (MATLAB 6.0), `nnls` displays a warning message and calls `lsqnonneg`.

Syntax

```
x = nnls(A,b)
x = nnls(A,b,tol)
[x,w] = nnls(A,b)
[x,w] = nnls(A,b,tol)
```

Description

`x = nnls(A,b)` solves the system of equations $Ax = b$ in a least squares sense, subject to the constraint that the solution vector `x` has nonnegative elements: $x_j > 0,\ j = 1, 2, \ldots n$. The solution `x` minimizes $\|(Ax = b)\|$ subject to $x \geq 0$.

`x = nnls(A,b,tol)` solves the system of equations, and specifies a tolerance `tol`. By default, `tol` is: `max(size(A))*norm(A,1)*eps`.

`[x,w] = nnls(A,b)` also returns the dual vector `w`, where $w_i \leq 0$ when $x_i = 0$ and $w_i \cong 0$ when $x_i > 0$.

`[x,w] = nnls(A,b,tol)` solves the system of equations, returns the dual vector `w`, and specifies a tolerance `tol`.

Examples

Compare the unconstrained least squares solution to the `nnls` solution for a 4-by-2 problem:

```
A =
    0.0372    0.2869
    0.6861    0.7071
    0.6233    0.6245
    0.6344    0.6170

b =

    0.8587
    0.1781
    0.0747
```

```
    0.8405

[A\b nnls(A,b)] =

    -2.5627         0
     3.1108    0.6929


[norm(A*(a\b)-b) norm(A*nnls(a,b)-b)] =

     0.6674 0.9118
```

The solution from nnls does not fit as well, but has no negative components.

Algorithm

The nnls function uses the algorithm described in [1], Chapter 23. The algorithm starts with a set of possible basis vectors, computes the associated dual vector w, and selects the basis vector corresponding to the maximum value in w to swap out of the basis in exchange for another possible candidate, until $w \le 0$.

See Also

\ Matrix left division (backslash)

References

[1] Lawson, C. L. and R. J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, 1974, Chapter 23.

nnz

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`

Purpose Change EraseMode of all objects to normal

Syntax

```
noanimate(state,fig_handle)
noanimate(state)
```

Description noanimate(state,fig_handle) sets the EraseMode of all image, line, patch surface, and text graphics object 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

nonzeros

Purpose

Nonzero matrix elements

Syntax

```
s = nonzeros(A)
```

Description

`s = nonzeros(A)` returns a full column vector of the nonzero elements in `A`, ordered by columns.

This gives the `s`, but not the `i` and `j`, from `[i,j,s] = find(A)`. Generally,

```
length(s) = nnz(A) ≤ nzmax(A) ≤ prod(size(A))
```

See Also

`find`, `isa`, `nnz`, `nzmax`, `size`, `whos`

norm

Purpose

Vector and matrix norms

Syntax

```
n = norm(A)
n = norm(A,p)
```

Description

The *norm* of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. The `norm` function calculates several different types of matrix norms:

`n = norm(A)` returns the largest singular value of `A`, `max(svd(A))`.

`n = norm(A,`*p*`)` returns a different kind of norm, depending on the value of *p*:

If *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, slightly different rules apply:

`norm(A,`*p*`)`	Returns `sum(abs(A).^`*p*`)^(1/`*p*`)`, for any $1 \le p \le \infty$.
`norm(A)`	Returns `norm(A,2)`.
`norm(A,inf)`	Returns `max(abs(A))`.
`norm(A,-inf)`	Returns `min(abs(A))`.

Remarks

To obtain the root-mean-square (RMS) value, use `norm(A)/sqrt(n)`. Note that `norm(A)`, where `A` is an `n`-element vector, is the length of `A`.

See Also

`cond`, `condest`, `normest`, `rcond`, `svd`

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 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`

null

Purpose Null space of a matrix

Syntax `B = null(A)`

Description `B = null(A)` returns an orthonormal basis for the null space of `A`.

Remarks `B'*B = I`, `A*B` has negligible elements, and (if `B` is not equal to the empty matrix) the number of columns of `B` is the nullity of `A`.

Example

```
A =
     1     2     3
     1     2     3
     1     2     3

null(A)

ans =
   -0.1559    0.9509
   -0.7971   -0.2809
    0.5834   -0.1297

null(A,'r')

ans =
    -2    -3
     1     0
     0     1
```

See Also `orth`, `qr`, `svd`

Purpose Convert a numeric array into a cell array

Syntax

```
c = num2cell(A)
c = num2cell(A,dims)
```

Description `c = num2cell(A)` converts the matrix `A` into a cell array by placing each element of `A` into a separate cell. Cell array `c` will be the same size as matrix `A`.

`c = num2cell(A,dims)` converts the matrix `A` into a cell array by placing the dimensions specified by `dims` into separate cells. `C` will be the same size as `A` except that the dimensions matching `dims` will be 1.

Examples The statement

```
num2cell(A,2)
```

places the rows of `A` into separate cells. Similarly

```
num2cell(A,[1 3])
```

places the column-depth pages of `A` into separate cells.

See Also `cat`

num2str

Purpose

Number to string conversion

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*. By default, this is `'%11.4g'`, which signifies four significant digits in exponential or fixed-point notation, whichever is shorter. (See `fprintf` for format string details).

Examples

`num2str(pi)` is `3.142`.

`num2str(eps)` is `2.22e-16`.

`num2str(magic(2))` produces the string matrix

```
1  3
4  2
```

See Also

`fprintf`, `int2str`, `sprintf`

Purpose Number of elements in a matrix

Syntax

```
n = numel(a)
```

Description `n = numel(a)` returns the scalar count, `n`, of the number of elements in the matrix, `a`.

`numel(a)` gives the same answer as `prod(size(a))`. However, if the `size` function has been overloaded, `prod(size(a))` may not provide an accurate count.

Note It is strongly recommended that you not overload the `numel` function.

`numel` can also be used with `subsref` to determine the number of values that will be returned from a particular call to `subsref`. See the second example below to see how to use this.

Examples Create a 4-by-4-by-2 matrix. `numel` counts 32 elments 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
```

In this example, `numel` indicates that `stockobj(3)` references six values. The indexed reference to `stockobj` is made using `subsref`.

```
n = numel(stockobj(3))
n =
    6
```

Calling `subsref` on `stockobj(3)` does indeed return six values.

```
stockobj(3)
ans =
    1.0417    5.2000
    7.0000   39.0400
    4.2200   56.4340
```

See Also

`size`, `prod`, `subsref`

Purpose Amount of storage allocated for nonzero matrix elements

Syntax

```
n = nzmax(S)
```

Description `n = nzmax(S)` returns the amount of storage allocated for nonzero elements.

If `S` is a sparse matrix...	`nzmax(S)` is the number of storage locations allocated for the nonzero elements in `S`.
If `S` is a full matrix...	`nzmax(S) = prod(size(S))`.

Often, `nnz(S)` and `nzmax(S)` are the same. But if `S` is created by an operation which produces fill-in matrix elements, such as sparse matrix multiplication or sparse LU factorization, more storage may be allocated than is actually required, and `nzmax(S)` reflects this. Alternatively, `sparse(i,j,s,m,n,nzmax)` or its simpler form, `spalloc(m,n,nzmax)`, can set `nzmax` in anticipation of later fill-in.

See Also `find`, `isa`, `nnz`, `nonzeros`, `size`, `whos`

ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb

Purpose Solve initial value problems for ordinary differential equations (ODEs)

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y] = solver(odefun,tspan,y0,options,p1,p2...)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
```

where `solver` is one of `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.

Arguments

`odefun`	A function that evaluates the right-hand side of the differential equations. 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]`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.
`y0`	A vector of initial conditions.
`options`	Optional integration argument created using the `odeset` function. See `odeset` for details.
`p1,p2...`	Optional parameters to be passed to `odefun`.

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`. 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 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]`.

`[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). See `odeset` for details.

`[T,Y] =` *solver*`(odefun,tspan,y0,options,p1,p2...)` solves as above, passing the additional parameters `p1,p2...` to the function `odefun`, whenever it is called. Use `options = []` as a place holder if no options are set.

`[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. This is done by setting the `Events` property to, say, `@EVENTS`, and creating a function `[value,isterminal,direction] = EVENTS(t,y)`. For the `i`th event function:

- `value(i)` is the value of the function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only 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.

If you specify an output function as the value of the `OutputFcn` property, the solver will call it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When the solver is called with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimnesional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, all components of the solution are passed to the output function, but 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) and otherwise, 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 differential algebraic equation. 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.
`ode23`	Nonstiff	Low	If using crude error tolerances or solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	If using stringent error tolerances or solving a computationally intensive ODE file.
`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	If the problem is only moderately stiff and 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 1-1038 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and Improving ODE Solver Performance in the

"Differential Equations" section of the *Mathematical Analysis* MATLAB documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`RelTol`, `AbsTol`, `NormControl`	√	√	√	√	√	√	√
`OutputFcn`, `OutputSel`, `Refine`, `Stats`	√	√	√	√	√	√	√
`Events`	√	√	√	√	√	√	√
`MaxStep`, `InitialStep`	√	√	√	√	√	√	√
`Jacobian`, `JPattern`, `Vectorized`	—	—	—	√	√	√	√

`Mass`
`MStateDependence`
`MvPattern`
`MassSingular`
	√
	√
	—
	—
		√
		√
		—
		—
			√
			√
			—
			—
				√
				√
				√
				√
					√
					—
					—
					—
						√
						√
						√
						√
							√
							√
							√
							—

`InitialSlope`	—	—	—	√	—	√	—
`MaxOrder`, `BDF`	—	—	—	√	—	—	—

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
```

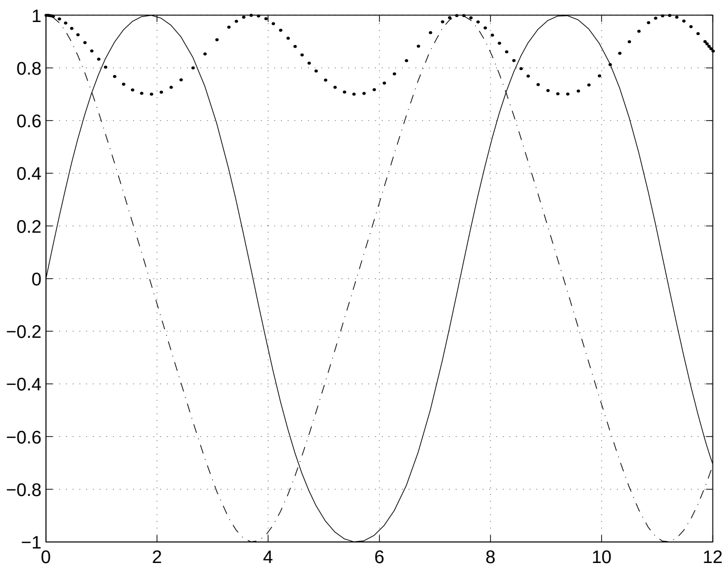
```
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval of `[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.

$$y'_1 = y_2 \qquad y_1(0) = 0$$
$$y'_2 = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 1$$

To simulate this system, create a function vdp1000 containing the equations

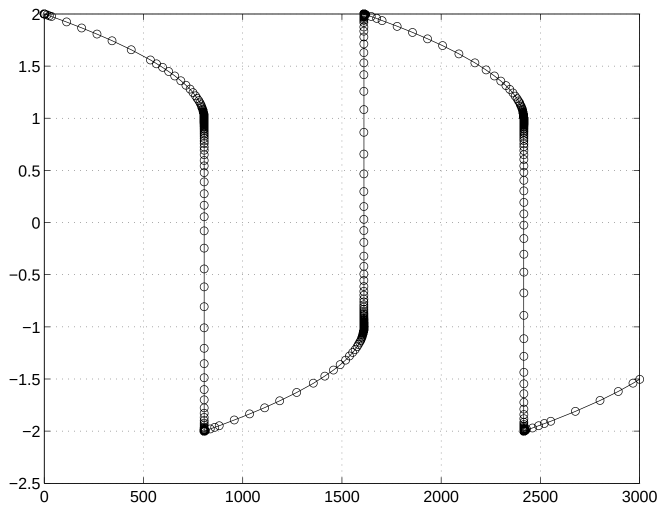
```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

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'):
```



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

@(function_handle), odeset, odeget

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 a differential equation problem for ordinary differential equation (ODE) 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 any of the ODE solvers or `odeset`.

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 MATLAB's 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

$$M(t, y)y' = f(t, y)$$

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 2-1044).

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.

$$y'_1 = y_2$$

$$y'_2 = \mu(1 - y_1^2) y_2 - y_1$$

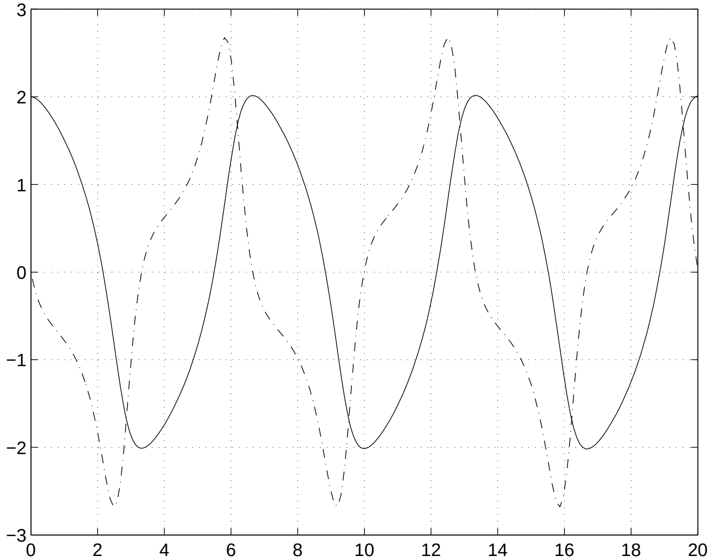
The M-file

```
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 Extract properties from options structure created with odeset

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 input to ordinary differential equation (ODE) 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 ODE solvers. The ODE solvers can integrate systems of differential equations of one of these forms

$$y' = f(t, y)$$

or

$$M(t, y)y' = f(t, y)$$

See below for information about the integration parameters.

`options = odeset('name1',value1,'name2',value2,...)` creates an integrator options structure in which the named properties have the specified values. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify a property name. Case is ignored for property names.

`options = odeset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`.

`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.

Properties

The available properties depend on the ODE solver used. There are several categories of properties:

- Error tolerance

- Solver output
- Jacobian matrix
- Event location
- Mass matrix and differential-algebraic equations (DAEs)
- Step size
- `ode15s`

Note This reference page describes the ODE properties for MATLAB, Version 6. 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`.

Table 1-1: Error Tolerance Properties

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	A relative error tolerance that applies to all components of the solution vector. The estimated error in each integration step satisfies `e(i)` ≤ `max(RelTol*abs(y(i)),AbsTol(i))`.
`AbsTol`	Positive scalar or vector {`1e-6`}	The absolute error tolerance. If scalar, the tolerance applies to all components of the solution vector. Otherwise the tolerances apply to corresponding components.
`NormControl`	`on` | {`off`}	Control error relative to norm of solution. 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 component-wise error control.

Table 1-2: Solver Output Properties

Property	Value		Description
OutputFcn	Function	Installable output function. The ODE solvers provide sample functions that you can use or modify:	
		odeplot	Time series plotting (default)
		odephas2	Two-dimensional phase plane plotting
		odephas3	Three-dimensional phase plane plotting
		odeprint	Print solution as it is computed
		To create or modify an output function, see ODE Solver Output Properties in the “Differential Equations” section of the MATLAB documentation.	
OutputSel	Vector of indices	Specifies the components of the solution vector that the solver passes to the output function.	
Refine	Positive integer	Produces smoother output, increasing the number of output points by the specified factor. The default value is 1 in all solvers except ode45, where it is 4. Refine doesn't apply if length(tspan) > 2.	
Stats	on | {off}	Specifies whether the solver should display statistics about the computational cost of the integration.	

Table 1-3: Jacobian Matrix Properties (for ode15s, ode23s, ode23t, and ode23tb)

Property	Value	Description
`Jacobian`	Function | constant matrix	Jacobian function. Set this property to `@FJac` (if a function `FJac(t,y)` returns $\partial f/\partial y$) or to the constant value of $\partial f/\partial y$.
`JPattern`	Sparse matrix of {0,1}	Sparsity pattern. 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.
`Vectorized`	`on` | {`off`}	Vectorized ODE function. Set this property `on` to inform the stiff solver that the ODE function `F` is coded so that `F(t,[y1 y2 ...])` returns the vector `[F(t,y1) F(t,y2) ...]`. That is, your ODE function can pass to the solver a whole array of column vectors at once. A stiff function calls your ODE function in a vectorized manner only if it is generating Jacobians numerically (the default behavior) and you have used `odeset` to set `Vectorized` to `on`.

Table 1-4: Event Location Property

Property	Value	Description
`Events`	Function	Locate events. Set this property to `@Events`, where `Events` is the event function. See the ODE solvers for details.

Table 1-5: Mass Matrix and DAE-Related Properties

Property	Value	Description
`Mass`	Constant matrix | function	For problems $My' = f(t, y)$ set this property to the value of the constant mass matrix m. For problems $M(t, y)y' = f(t, y)$, set this property to `@Mfun`, where `Mfun` is a function that evaluates the mass matrix $M(t, y)$.
`MStateDependence`	`none` | `{weak}` | `strong`	Dependence of the mass matrix on y. 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. For use with all solvers except `ode23s`.
`MvPattern`	Sparse matrix	$\partial(M(t, y)v)/\partial y$ sparsity pattern. 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`.
`MassSingular`	`yes` | `no` | `{maybe}`	Indicates whether the mass matrix is singular. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. For use with the `ode15s` and `ode23t` solvers.
`InitialSlope`	Vector	Consistent initial slope yp_0, where yp_0 satisfies $M(t_0, y_0)yp_0 = f(t_0, y_0)$. For use with the `ode15s` and `ode23t` solvers when solving DAEs.

Table 1-6: Step Size Properties

Property	Value	Description
`MaxStep`	Positive scalar	An upper bound on the magnitude of the step size that the solver uses. The default is one-tenth of the `tspan` interval.
`InitialStep`	Positive scalar	Suggested initial step size. The solver tries this first, but if too large an error results, the solver uses a smaller step size.

In addition there are two options that apply only to the `ode15s` solver.

Table 1-7: ode15s Properties

Property	Value	Description
`MaxOrder`	1 | 2 | 3 | 4 | {5}	The maximum order formula used.
`BDF`	`on` | `{off}`	Set `on` to specify that `ode15s` should use the backward differentiation formulas (BDFs) instead of the default numerical differentiation formulas (NDFs).

See Also

`@(function_handle)`, `odeget`, `ode45`, `ode23`, `ode23t`, `ode23tb`, `ode113`, `ode15s`, `ode23s`

ones

Purpose

Create an array of all ones

Syntax

```
Y = ones(n)
Y = ones(m,n)
Y = ones([m n])
Y = ones(d1,d2,d3...)
Y = ones([d1 d2 d3...])
Y = ones(size(A))
```

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(d1,d2,d3...)` or `Y = ones([d1 d2 d3...])` returns an array of `1`s with dimensions `d1`-by-`d2`-by-`d3`-by-`...`.

`Y = ones(size(A))` returns an array of `1`s that is the same size as `A`.

See Also

`eye`, `rand`, `randn`, `zeros`

Purpose

Open files based on extension

Syntax

```
open('name')
```

Description

`open('name')` opens the file `name`, where the specific action upon opening depends on the type of file that `name` is.

name	Action
variable	Open array `name` in the Array Editor (the array must be numeric); `open` calls `openvar`
figure file (`*.fig`)	Open figure in a figure window
HTML file (`*.html`)	Open HTML document in Help browser
M-file (`name.m`)	Open M-file `name` in M-file Editor
model (`name.mdl`)	Open model `name` in Simulink
p-file (`name.p`)	Open the corresponding M-file, `name.m`, if it exists, in the Editor
other extensions (`name.custom`)	Open `name.custom` by calling the helper function `opencustom`, where `opencustom` is a user-defined function.

Remarks

Behavior When name Does Not Have an Extension

If `name` does not contain a file extension, `open` opens the object returned by `which(name)`, where `name` is a variable, function, or model. If there is no matching helper function found, `open` uses the default editor.

If `name` does not contain a file extension and there is a matching filename without an extension, `open` opens the file in the editor. If it does not find a matching file without an extension, `open` looks for an M-file with the same name on the path, and if found, opens it in the editor.

To handle a variable, `open` calls the function `openvar`.

Create Custom open

Create your own opencustom functions to change the way standard file types are handled or to set up handlers for new file types. open calls the opencustom function it finds on the path.

Examples

Example 1 – No File Extension Specified

If testdata exists on the path,

```
open('testdata')
```

opens testdata in the editor.

If testdata does not exist, but testdata.m is on the path,

```
open('testdata')
```

opens testdata.m in the editor.

Example 2 – No File Extension Specified, M-file and Model Files Present

If testdata.m and testdata.mdl are both present on the search path, and you type

```
open('testdata')
```

testdata.mdl opens in Simulink. This is because model files take precedence over M-files, which you can see by typing

```
which('testdata')
```

It returns the file that takes precedence, in this case

```
testdata.mdl
```

Example 3 – Customized open

open('mychart.cht') calls opencht('myfigure.cht'), where opencht is a user-created function that uses .cht files.

See Also

hgload, hgsave, load, openvar, save, saveas

Purpose

Open new copy or raise existing copy of saved figure

Syntax

```
openfig('filename.fig','new')
openfig('filename.fig','reuse')
openfig('filename.fig')
figure_handle = openfig(...)
```

Description

`openfig` is designed for use with GUI figures. Use this function to:

- Open the FIG-file creating the GUI and ensure it is displayed on screen. This provides compatibility with different screen sizes and resolutions.
- Control whether MATLAB displays one or multiple instances of the GUI at any given time.
- Return the handle of the figure created, which is typically hidden for GUIs figures.

`openfig('`*filename*`.fig','new')` opens the figure contained in the FIG-file, *filename*`.fig`, and ensures it is visible and positioned completely on screen. You do not have to specify the full path to the FIG-file as long as it is on your MATLAB path. The `.fig` extension is optional.

`openfig('`*filename*`.fig','reuse')` opens the figure contained in the FIG-file only if a copy is not currently open; otherwise `openfig` brings the existing copy forward, making sure it is still visible and completely on screen.

`openfig('`*filename*`.fig')` is the same as `openfig('`*filename*`.fig','new')`.

`figure_handle = openfig(...)` returns the handle to the figure.

Remarks

If the FIG-file contains an invisible figure, `openfig` returns its handle and leaves it invisible. The caller should make the figure visible when appropriate.

See Also

`guide`, `guihandles`, `movegui`, `open`, `hgload`, `save`

opengl

Purpose Change automatic selection mode of OpenGL rendering

Syntax

```
opengl selection_mode
```

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 auto selection 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 auto selection state.

`opengl info` prints information with the version and vendor of the OpenGL on your system.

Note that the auto selection state only specifies that OpenGL should or not be considered for rendering, it does not explicitly set the rendering to OpenGL. This can be done by setting the `Renderer` property of figure to `OpenGL`. For example,

```
set(gcf,'Renderer','OpenGL')
```

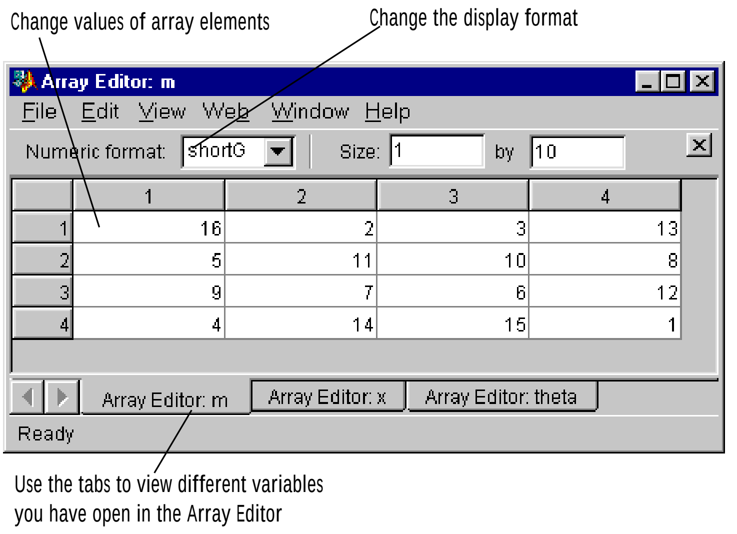
Purpose Open workspace variable in the Array Editor for graphical editing

Graphical Interface As an alternative to the openvar function, double-click on a variable in the Workspace browser.

Syntax

```
openvar('name')
```

Description `openvar('name')` opens the workspace variable name in the Array Editor for graphical debugging. The array must be numeric.



See Also `load`, `save`, `workspace`

optimget

Purpose

Get optimization options structure parameter 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 parameter structure

Syntax

```
options = optimset('param1',value1,'param2',value2,...)
optimset
options = optimset
options = optimset(optimfun)
options = optimset(oldopts,'param1',value1,...)
options = optimset(oldopts,newopts)
```

Description

`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`.

Parameters

Optimization parameters used by MATLAB functions and Optimization Toolbox functions:

Parameter	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'` dislays output only if the function does not converge.
`MaxFunEvals`	positive integer	Maximum number of function evaluations allowed.
`MaxIter`	positive integer	Maximum number of iterations allowed.
`TolFun`	positive scalar	Termination tolerance on the function value.
`TolX`	positive scalar	Termination tolerance on x .

Optimization parameters used by Optimization Toolbox functions (for more information about individual parameters, see "Optimization Options Parameters" in the *Optimization Toolbox User's Guide*, and the optimization functions that use these parameters):

Property	Value	Description
`DerivativeCheck`	`'on'` | {`'off'`}	Compare user-supplied analytic derivatives (gradients or Jacobian) to finite differencing derivatives.
`Diagnostics`	`'on'` | {`'off'`}	Print diagnostic information about the function to be minimized or solved.
`DiffMaxChange`	positive scalar | {`1e-1`}	Maximum change in variables for finite difference derivatives.

Property	Value	Description
`DiffMinChange`	positive scalar | {`1e-8`}	Minimum change in variables for finite difference derivatives.
`GoalsExactAchieve`	positive scalar integer | {`0`}	Number of goals to achieve exactly (do not over- or underachieve).
`GradConstr`	`'on'` | {`'off'`}	Gradients for nonlinear constraints defined by the user.
`GradObj`	`'on'` | {`'off'`}	Gradient(s) for objective function(s) defined by the user.
`Hessian`	`'on'` | {`'off'`}	Hessian for the objective function defined by the user.
`HessMult`	function | {`[]`}	Hessian multiply function defined by the user.
`HessPattern`	sparse matrix |{sparse matrix of all ones}	Sparsity pattern of the Hessian for finite differencing. The size of the matrix is n-by-n, where n is the number of elements in `x0`, the starting point.
`HessUpdate`	{`'bfgs'`} | `'dfp'` | `'gillmurray'` | `'steepdesc'`	Quasi-Newton updating scheme.
`Jacobian`	`'on'` | {`'off'`}	Jacobian for the objective function defined by the user.
`JacobMult`	function | {`[]`}	Jacobian multiply function defined by the user.
`JacobPattern`	sparse matrix |{sparse matrix of all ones}	Sparsity pattern of the Jacobian for finite differencing. The size of the matrix is m-by-n, where m is the number of values in the first argument returned by the user-specified function `fun`, and n is the number of elements in `x0`, the starting point.

Property	Value	Description
LargeScale	{'on'} | 'off'	Use large-scale algorithm if possible.
LevenbergMarquardt	'on' | {'off'}	Chooses Levenberg-Marquardt over Gauss-Newton algorithm.
LineSearchType	'cubicpoly' | {'quadcubic'}	Line search algorithm choice.
MaxPCGIter	positive integer	Maximum number of PCG iterations allowed. The default is the greater of 1 and floor(n/2)) where n is the number of elements in x0, the starting point.
MeritFunction	'singleobj' | {'multiobj'}	Use goal attainment/minimax merit function (multiobjective) vs. fmincon (single objective).
MinAbsMax	positive scalar integer | {0}	Number of $F(x)$ to minimize the worst case absolute values
PrecondBandWidth	positive integer | {0} | Inf	Upper bandwidth of preconditioner for PCG.
TolCon	positive scalar	Termination tolerance on the constraint violation.
TolPCG	positive scalar | {0.1}	Termination tolerance on the PCG iteration.
TypicalX	vector of all ones	Typical x values. The length of the vector is equal to the number of elements in x0, the starting point.

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

`optimget`, `fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`

orient

Purpose

Set paper orientation for printed output

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, either `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 MATLAB's 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`, `set`

`PaperOrientation`, `PaperPosition`, `PaperSize`, `PaperType`, and `PaperUnits` properties of figure graphics objects.

orth

Purpose Range space of a 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

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`) match 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`

Index

Symbols

Numerics

A

B

M

N

Q

R