MATLAB®

The Language of Technical Computing

- Computation
- Visualization
- Programming

Function Reference
Volume 2: F - O

Version 7

The MathWorks

How to Contact The MathWorks:

`www.mathworks.com`	Web
`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
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

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

MATLAB Function Reference Volume 2: F - O

Printing History:	December 1996	First printing	For MATLAB 5.0 (Release 8)
	June 1997	Online only	Revised for MATLAB 5.1 (Release 9)
	October 1997	Online only	Revised for MATLAB 5.2 (Release 10)
	January 1999	Online only	Revised for MATLAB 5.3 (Release 11
	June 1999	Second printing	For MATLAB 5.3 (Release 11)
	June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
	July 2002	Online only	Revised for 6.5 (Release 13)
	June 2004	Online only	Revised for 7.0 (Release 14)

Contents

1 Functions — Categorical List

Functions — Categorical List

The MATLAB® Function Reference contains descriptions of all MATLAB commands and functions.

Select a category from the following table to see a list of related functions.

Desktop Tools and Development Environment	Startup, Command Window, help, editing and debugging, tuning, other general functions
Mathematics	Arrays and matrices, linear algebra, data analysis, other areas of mathematics
Programming and Data Types	Function/expression evaluation, program control, function handles, object oriented programming, error handling, operators, data types, dates and times, timers
File I/O	General and low-level file I/O, plus specific file formats, like audio, spreadsheet, HDF, images
Graphics	Line plots, annotating graphs, specialized plots, images, printing, Handle Graphics®
3-D Visualization	Surface and mesh plots, view control, lighting and transparency, volume visualization.
Creating Graphical User Interface	GUIDE, programming graphical user interfaces.
External Interfaces	Java, COM, Serial Port functions.

See Simulink®, Stateflow®, Real-Time Workshop®, and the individual toolboxes for lists of their functions

Desktop Tools and Development Environment

General functions for working in MATLAB, including functions for startup, Command Window, help, and editing and debugging.

"Startup and Shutdown"	Startup and shutdown options
"Command Window and History"	Controlling Command Window and History
"Help for Using MATLAB"	Finding information
"Workspace, Search Path, and File Operations"	File, search path, variable management
"Programming Tools"	Editing and debugging, source control, Notebook
"System"	Identifying current computer, license, product version, and more

Startup and Shutdown

`exit`	Terminate MATLAB (same as `quit`)
`finish`	MATLAB termination M-file
`genpath`	Generate a path string
`matlab`	Start MATLAB (UNIX systems)
`matlab`	Start MATLAB (Windows systems)
`matlabrc`	MATLAB startup M-file for single user systems or administrators
`prefdir`	Return directory containing preferences, history, and layout files
`preferences`	Display Preferences dialog box for MATLAB and related products
`quit`	Terminate MATLAB
`startup`	MATLAB startup M-file for user-defined options

Command Window and History

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

Help for Using MATLAB

`doc`	Display online documentation in MATLAB Help browser
`demo`	Access product demos via Help browser
`docopt`	Web browser for UNIX platforms
`docsearch`	Open Help browser Search pane and run search for specified term
`help`	Display help for MATLAB functions in Command Window
`helpbrowser`	Display Help browser for access to full online documentation and demos
`helpwin`	Provide access to and display M-file help for all functions
`info`	Display Release Notes for MathWorks products
`lookfor`	Search for specified keyword in all `help` entries
`playshow`	Run published M-file demo
`support`	Open MathWorks Technical Support Web page
`web`	Open Web site or file in Web browser or Help browser
`whatsnew`	Display Release Notes for MathWorks products

Workspace, Search Path, and File Operations

- “Workspace”
- “Search Path”
- “File Operations”

Workspace

assignin	Assign value to workspace variable
clear	Remove items from workspace, freeing up system memory
evalin	Execute string containing MATLAB expression in a workspace
exist	Check if variables or functions are defined
openvar	Open workspace variable in Array Editor for graphical editing
pack	Consolidate workspace memory
uiimport	Open Import Wizard, the graphical user interface to import data
which	Locate functions and files
who, whos	List variables in the workspace
workspace	Display Workspace browser, a tool for managing the workspace

Search Path

addpath	Add directories to MATLAB search path
genpath	Generate path string
partialpath	Partial pathname
path	View or change the MATLAB directory search path
path2rc	Replaced by `savepath`
pathdef	List of directories in the MATLAB search path
pathsep	Return path separator for current platform
pathtool	Open **Set Path** dialog box to view and change MATLAB path
restoredefaultpath	Restore the default search path
rmpath	Remove directories from MATLAB search path
savepath	Save current MATLAB search path to `pathdef.m` file

File Operations

cd	Change working directory
copyfile	Copy file or directory
delete	Delete files or graphics objects
dir	Display directory listing
exist	Check if variables or functions are defined
fileattrib	Set or get attributes of file or directory
filebrowser	Display Current Directory browser, a tool for viewing files
lookfor	Search for specified keyword in all `help` entries
ls	List directory on UNIX
matlabroot	Return root directory of MATLAB installation
mkdir	Make new directory
movefile	Move file or directory
pwd	Display current directory
recycle	Set option to move deleted files to recycle folder
rehash	Refresh function and file system path caches
rmdir	Remove directory

`type`	List file
`web`	Open Web site or file in Web browser or Help browser
`what`	List MATLAB specific files in current directory
`which`	Locate functions and files

See also "File I/O" functions.

Programming Tools

- "Editing and Debugging"
- "Performance Improvement and Tuning Tools and Techniques"
- "Source Control"
- "Publishing"

Editing and Debugging

`dbclear`	Clear breakpoints
`dbcont`	Resume execution
`dbdown`	Change local workspace context
`dbquit`	Quit debug mode
`dbstack`	Display function call stack
`dbstatus`	List all breakpoints
`dbstep`	Execute one or more lines from current breakpoint
`dbstop`	Set breakpoints
`dbtype`	List M-file with line numbers
`dbup`	Change local workspace context
`debug`	M-file debugging functions
`edit`	Edit or create M-file
`keyboard`	Invoke the keyboard in an M-file

Performance Improvement and Tuning Tools and Techniques

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

Source Control

`checkin`	Check file into source control system
`checkout`	Check file out of source control system
`cmopts`	Get name of source control system
`customverctrl`	Allow custom source control system
`undocheckout`	Undo previous checkout from source control system
`verctrl`	Version control operations on PC platforms

Publishing

`notebook`	Open M-book in Microsoft Word (Windows only)
`publish`	Run M-file containing cells, and save results to file of specified type

System

`computer`	Identify information about computer on which MATLAB is running
`javachk`	Generate error message based on Java feature support
`license`	Show license number for MATLAB
`prefdir`	Return directory containing preferences, history, and layout files
`usejava`	Determine if a Java feature is supported in MATLAB
`ver`	Display version information for MathWorks products
`version`	Get MATLAB version number

Mathematics

Functions for working with arrays and matrices, linear algebra, data analysis, and other areas of mathematics.

"Arrays and Matrices"	Basic array operators and operations, creation of elementary and specialized arrays and matrices
"Linear Algebra"	Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization
"Elementary Math"	Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math
"Data Analysis and Fourier Transforms"	Descriptive statistics, finite differences, correlation, filtering and convolution, fourier transforms
"Polynomials"	Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion
"Interpolation and Computational Geometry"	Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation
"Coordinate System Conversion"	Conversions between Cartesian and polar or spherical coordinates
"Nonlinear Numerical Methods"	Differential equations, optimization, integration
"Specialized Math"	Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions
"Sparse Matrices"	Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations
"Math Constants"	Pi, imaginary unit, infinity, Not-a-Number, largest and smallest positive floating point numbers, floating point relative accuracy

Arrays and Matrices

- "Basic Information"
- "Operators"
- "Operations and Manipulation"
- "Elementary Matrices and Arrays"
- "Specialized Matrices"

Basic Information

`disp`	Display array
`display`	Display array
`isempty`	True for empty matrix
`isequal`	True if arrays are identical
`isfloat`	True for floating-point arrays
`isinteger`	True for integer arrays
`islogical`	True for logical array
`isnumeric`	True for numeric arrays
`isscalar`	True for scalars
`issparse`	True for sparse matrix
`isvector`	True for vectors
`length`	Length of vector
`ndims`	Number of dimensions
`numel`	Number of elements
`size`	Size of matrix

Operators

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

Operations and Manipulation

`:` (colon)	Index into array, rearrange array
`accumarray`	Construct an array with accumulation
`blkdiag`	Block diagonal concatenation
`cat`	Concatenate arrays
`cross`	Vector cross product
`cumprod`	Cumulative product
`cumsum`	Cumulative sum
`diag`	Diagonal matrices and diagonals of matrix
`dot`	Vector dot product
`end`	Last index
`find`	Find indices of nonzero elements
`fliplr`	Flip matrices left-right
`flipud`	Flip matrices up-down
`flipdim`	Flip matrix along specified dimension
`horzcat`	Horizontal concatenation
`ind2sub`	Multiple subscripts from linear index
`ipermute`	Inverse permute dimensions of multidimensional array
`kron`	Kronecker tensor product
`max`	Maximum value of array
`min`	Minimum value of array
`permute`	Rearrange dimensions of multidimensional array
`prod`	Product of array elements
`repmat`	Replicate and tile array
`reshape`	Reshape array
`rot90`	Rotate matrix 90 degrees
`sort`	Sort array elements in ascending or descending order
`sortrows`	Sort rows in ascending order
`sum`	Sum of array elements
`sqrtm`	Matrix square root
`sub2ind`	Linear index from multiple subscripts
`tril`	Lower triangular part of matrix
`triu`	Upper triangular part of matrix
`vertcat`	Vertical concatenation

See also “Linear Algebra” for other matrix operations.
See also “Elementary Math” for other array operations.

Elementary Matrices and Arrays

`:` (colon)	Regularly spaced vector
`blkdiag`	Construct block diagonal matrix from input arguments
`diag`	Diagonal matrices and diagonals of matrix
`eye`	Identity matrix
`freqspace`	Frequency spacing for frequency response
`linspace`	Generate linearly spaced vectors
`logspace`	Generate logarithmically spaced vectors
`meshgrid`	Generate X and Y matrices for three-dimensional plots
`ndgrid`	Arrays for multidimensional functions and interpolation
`ones`	Create array of all ones
`rand`	Uniformly distributed random numbers and arrays
`randn`	Normally distributed random numbers and arrays
`repmat`	Replicate and tile array
`zeros`	Create array of all zeros

Specialized Matrices

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

Linear Algebra

- "Matrix Analysis"
- "Linear Equations"
- "Eigenvalues and Singular Values"
- "Matrix Logarithms and Exponentials"
- "Factorization"

Matrix Analysis

`cond`	Condition number with respect to inversion
`condeig`	Condition number with respect to eigenvalues
`det`	Determinant
`norm`	Matrix or vector norm
`normest`	Estimate matrix 2-norm
`null`	Null space
`orth`	Orthogonalization
`rank`	Matrix rank
`rcond`	Matrix reciprocal condition number estimate
`rref`	Reduced row echelon form
`subspace`	Angle between two subspaces
`trace`	Sum of diagonal elements

Linear Equations

`\` and `/`	Linear equation solution
`chol`	Cholesky factorization
`cholinc`	Incomplete Cholesky factorization
`cond`	Condition number with respect to inversion
`condest`	1-norm condition number estimate
`funm`	Evaluate general matrix function
`inv`	Matrix inverse
`linsolve`	Solve linear systems of equations
`lscov`	Least squares solution in presence of known covariance
`lsqnonneg`	Nonnegative least squares
`lu`	LU matrix factorization
`luinc`	Incomplete LU factorization
`pinv`	Moore-Penrose pseudoinverse of matrix
`qr`	Orthogonal-triangular decomposition
`rcond`	Matrix reciprocal condition number estimate

Eigenvalues and Singular Values

`balance`	Improve accuracy of computed eigenvalues
`cdf2rdf`	Convert complex diagonal form to real block diagonal form
`condeig`	Condition number with respect to eigenvalues
`eig`	Eigenvalues and eigenvectors
`eigs`	Eigenvalues and eigenvectors of sparse matrix
`gsvd`	Generalized singular value decomposition
`hess`	Hessenberg form of matrix
`poly`	Polynomial with specified roots
`polyeig`	Polynomial eigenvalue problem
`qz`	QZ factorization for generalized eigenvalues

`rsf2csf`	Convert real Schur form to complex Schur form
`schur`	Schur decomposition
`svd`	Singular value decomposition
`svds`	Singular values and vectors of sparse matrix

Matrix Logarithms and Exponentials

`expm`	Matrix exponential
`logm`	Matrix logarithm
`sqrtm`	Matrix square root

Factorization

`balance`	Diagonal scaling to improve eigenvalue accuracy
`cdf2rdf`	Complex diagonal form to real block diagonal form
`chol`	Cholesky factorization
`cholinc`	Incomplete Cholesky factorization
`cholupdate`	Rank 1 update to Cholesky factorization
`lu`	LU matrix factorization
`luinc`	Incomplete LU factorization
`planerot`	Givens plane rotation
`qr`	Orthogonal-triangular decomposition
`qrdelete`	Delete column or row from QR factorization
`qrinsert`	Insert column or row into QR factorization
`qrupdate`	Rank 1 update to QR factorization
`qz`	QZ factorization for generalized eigenvalues
`rsf2csf`	Real block diagonal form to complex diagonal form

Elementary Math

- “Trigonometric”
- “Exponential”
- “Complex”
- “Rounding and Remainder”
- “Discrete Math (e.g., Prime Factors)”

Trigonometric

acos	Inverse cosine
acosd	Inverse cosine, degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent
acotd	Inverse cotangent, degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant
acscd	Inverse cosecant, degrees
acsch	Inverse hyperbolic cosecant
asec	Inverse secant
asecd	Inverse secant, degrees
asech	Inverse hyperbolic secant
asin	Inverse sine
asind	Inverse sine, degrees
asinh	Inverse hyperbolic sine
atan	Inverse tangent
atand	Inverse tangent, degrees
atanh	Inverse hyperbolic tangent
atan2	Four-quadrant inverse tangent
cos	Cosine
cosd	Cosine, degrees
cosh	Hyperbolic cosine
cot	Cotangent
cotd	Cotangent, degrees
coth	Hyperbolic cotangent
csc	Cosecant
cscd	Cosecant, degrees
csch	Hyperbolic cosecant
sec	Secant
secd	Secant, degrees
sech	Hyperbolic secant
sin	Sine
sind	Sine, degrees
sinh	Hyperbolic sine
tan	Tangent
tand	Tangent, degrees
tanh	Hyperbolic tangent

Exponential

`exp`	Exponential
`expm1`	Exponential of x minus 1
`log`	Natural logarithm
`log1p`	Logarithm of 1+x
`log2`	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
`log10`	Common (base 10) logarithm
`nextpow2`	Next higher power of 2
`pow2`	Base 2 power and scale floating-point number
`reallog`	Natural logarithm for nonnegative real arrays
`realpow`	Array power for real-only output
`realsqrt`	Square root for nonnegative real arrays
`sqrt`	Square root
`nthroot`	Real nth root

Complex

`abs`	Absolute value
`angle`	Phase angle
`complex`	Construct complex data from real and imaginary parts
`conj`	Complex conjugate
`cplxpair`	Sort numbers into complex conjugate pairs
`i`	Imaginary unit
`imag`	Complex imaginary part
`isreal`	True for real array
`j`	Imaginary unit
`real`	Complex real part
`sign`	Signum
`unwrap`	Unwrap phase angle

Rounding and Remainder

`fix`	Round towards zero
`floor`	Round towards minus infinity
`ceil`	Round towards plus infinity
`round`	Round towards nearest integer
`mod`	Modulus after division
`rem`	Remainder after division

Discrete Math (e.g., Prime Factors)

`factor`	Prime factors
`factorial`	Factorial function
`gcd`	Greatest common divisor
`isprime`	True for prime numbers
`lcm`	Least common multiple
`nchoosek`	All combinations of N elements taken K at a time
`perms`	All possible permutations
`primes`	Generate list of prime numbers
`rat`, `rats`	Rational fraction approximation

Data Analysis and Fourier Transforms

- “Basic Operations”
- “Finite Differences”
- “Correlation”
- “Filtering and Convolution”
- “Fourier Transforms”

Basic Operations

`cumprod`	Cumulative product
`cumsum`	Cumulative sum
`cumtrapz`	Cumulative trapezoidal numerical integration
`max`	Maximum elements of array
`mean`	Average or mean value of arrays
`median`	Median value of arrays
`min`	Minimum elements of array
`prod`	Product of array elements
`sort`	Sort array elements in ascending or descending 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
`subspace`	Angle between two subspaces

Filtering and Convolution

`conv`	Convolution and polynomial multiplication
`conv2`	Two-dimensional convolution
`convn`	N-dimensional convolution
`deconv`	Deconvolution and polynomial division
`detrend`	Linear trend removal
`filter`	Filter data with 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
`fft`	One-dimensional discrete Fourier transform
`fft2`	Two-dimensional discrete Fourier transform
`fftn`	N-dimensional discrete Fourier Transform
`fftshift`	Shift DC component of discrete Fourier transform to center of spectrum
`fftw`	Interface to the FFTW library run-time algorithm for tuning FFTs
`ifft`	Inverse one-dimensional discrete Fourier transform
`ifft2`	Inverse two-dimensional discrete Fourier transform
`ifftn`	Inverse multidimensional discrete Fourier transform
`ifftshift`	Inverse fast Fourier transform shift
`nextpow2`	Next power of two
`unwrap`	Correct phase angles

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

Interpolation and Computational Geometry

- “Interpolation”
- “Delaunay Triangulation and Tessellation”
- “Convex Hull”
- “Voronoi Diagrams”
- “Domain Generation”

Interpolation

`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 fast Fourier transform method
`interpn`	Multidimensional data interpolation (table lookup)
`meshgrid`	Generate X and Y matrices for three-dimensional plots
`mkpp`	Make piecewise polynomial
`ndgrid`	Generate arrays for multidimensional functions and interpolation
`pchip`	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)
`ppval`	Piecewise polynomial evaluation
`spline`	Cubic spline data interpolation
`tsearchn`	Multidimensional closest simplex search
`unmkpp`	Piecewise polynomial details

Delaunay Triangulation and Tessellation

`delaunay`	Delaunay triangulation
`delaunay3`	Three-dimensional Delaunay tessellation
`delaunayn`	Multidimensional Delaunay tessellation
`dsearch`	Search for nearest point
`dsearchn`	Multidimensional closest point search
`tetramesh`	Tetrahedron mesh plot
`trimesh`	Triangular mesh plot
`triplot`	Two-dimensional triangular plot
`trisurf`	Triangular surface plot
`tsearch`	Search for enclosing Delaunay triangle
`tsearchn`	Multidimensional closest simplex search

Convex Hull

`convhull`	Convex hull
`convhulln`	Multidimensional convex hull
`patch`	Create patch graphics object
`plot`	Linear two-dimensional plot
`trisurf`	Triangular surface plot

Voronoi Diagrams

`dsearch`	Search for nearest point
`patch`	Create patch graphics object
`plot`	Linear two-dimensional plot
`voronoi`	Voronoi diagram
`voronoin`	Multidimensional Voronoi diagrams

Domain Generation

`meshgrid`	Generate X and Y matrices for three-dimensional plots
`ndgrid`	Generate arrays for multidimensional functions and interpolation

Coordinate System Conversion

Cartesian

`cart2sph`	Transform Cartesian to spherical coordinates
`cart2pol`	Transform Cartesian to polar coordinates
`pol2cart`	Transform polar to Cartesian coordinates
`sph2cart`	Transform spherical to Cartesian coordinates

Nonlinear Numerical Methods

- “Ordinary Differential Equations (IVP)”
- “Delay Differential Equations”
- “Boundary Value Problems”
- “Partial Differential Equations”
- “Optimization”
- “Numerical Integration (Quadrature)”

Ordinary Differential Equations (IVP)

`ode113`	Solve non-stiff differential equations, variable order method
`ode15i`	Solve fully implicit differential equations, variable order method
`ode15s`	Solve stiff ODEs and DAEs Index 1, variable order method
`ode23`	Solve non-stiff differential equations, low order method
`ode23s`	Solve stiff differential equations, low order method
`ode23t`	Solve moderately stiff ODEs and DAEs Index 1, trapezoidal rule
`ode23tb`	Solve stiff differential equations, low order method
`ode45`	Solve non-stiff differential equations, medium order method
`odextend`	Extend the solution of an initial value problem
`odeget`	Get ODE `options` parameters
`odeset`	Create/alter ODE `options` structure
`decic`	Compute consistent initial conditions for `ode15i`
`deval`	Evaluate solution of differential equation problem

Delay Differential Equations

`dde23`	Solve delay differential equations with constant delays
`ddeget`	Get DDE options parameters
`ddeset`	Create/alter DDE `options` structure
`deval`	Evaluate solution of differential equation problem

Boundary Value Problems

`bvp4c`	Solve boundary value problems for ODEs
`bvpget`	Get BVP `options` parameters
`bvpset`	Create/alter BVP `options` structure
`deval`	Evaluate solution of differential equation problem

Partial Differential Equations

`pdepe`	Solve initial-boundary value problems for parabolic-elliptic PDEs
`pdeval`	Evaluates by interpolation solution computed by `pdepe`

Optimization

`fminbnd`	Scalar bounded nonlinear function minimization
`fminsearch`	Multidimensional unconstrained nonlinear minimization, by Nelder-Mead direct search method
`fzero`	Scalar nonlinear zero finding
`lsqnonneg`	Linear least squares with nonnegativity constraints
`optimset`	Create or alter optimization `options` structure
`optimget`	Get optimization parameters from `options` structure

Numerical Integration (Quadrature)

`quad`	Numerically evaluate integral, adaptive Simpson quadrature (low order)
`quadl`	Numerically evaluate integral, adaptive Lobatto quadrature (high order)
`quadv`	Vectorized quadrature
`dblquad`	Numerically evaluate double integral
`triplequad`	Numerically evaluate triple integral

Specialized Math

`airy`	Airy functions
`besselh`	Bessel functions of third kind (Hankel functions)
`besseli`	Modified Bessel function of first kind
`besselj`	Bessel function of first kind
`besselk`	Modified Bessel function of second kind
`bessely`	Bessel function of second kind
`beta`	Beta function
`betainc`	Incomplete beta function
`betaln`	Logarithm of beta function
`ellipj`	Jacobi elliptic functions
`ellipke`	Complete elliptic integrals of first and second kind
`erf`	Error function
`erfc`	Complementary error function
`erfcinv`	Inverse complementary error function
`erfcx`	Scaled complementary error function
`erfinv`	Inverse error function
`expint`	Exponential integral
`gamma`	Gamma function
`gammainc`	Incomplete gamma function
`gammaln`	Logarithm of gamma function
`legendre`	Associated Legendre functions
`psi`	Psi (polygamma) function

Sparse Matrices

- "Elementary Sparse Matrices"
- "Full to Sparse Conversion"
- "Working with Sparse Matrices"
- "Reordering Algorithms"
- "Linear Algebra"
- "Linear Equations (Iterative Methods)"
- "Tree Operations"

Elementary Sparse Matrices

`spdiags`	Sparse matrix formed from diagonals
`speye`	Sparse identity matrix
`sprand`	Sparse uniformly distributed random matrix
`sprandn`	Sparse normally distributed random matrix
`sprandsym`	Sparse random symmetric matrix

Full to Sparse Conversion

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

Working with Sparse Matrices

`issparse`	True for sparse matrix
`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 matrix elements
`spones`	Replace nonzero sparse matrix elements with ones
`spparms`	Set parameters for sparse matrix routines
`spy`	Visualize sparsity pattern

Reordering Algorithms

`colamd`	Column approximate minimum degree permutation
`colmmd`	Column minimum degree permutation
`colperm`	Column permutation
`dmperm`	Dulmage-Mendelsohn permutation
`randperm`	Random permutation
`symamd`	Symmetric approximate minimum degree permutation
`symmmd`	Symmetric minimum degree permutation
`symrcm`	Symmetric reverse Cuthill-McKee permutation

Linear Algebra

`cholinc`	Incomplete Cholesky factorization
`condest`	1-norm condition number estimate
`eigs`	Eigenvalues and eigenvectors of sparse matrix
`luinc`	Incomplete LU factorization
`normest`	Estimate matrix 2-norm
`sprank`	Structural rank
`svds`	Singular values and vectors of sparse matrix

Linear Equations (Iterative Methods)

`bicg`	BiConjugate Gradients method
`bicgstab`	BiConjugate Gradients Stabilized method
`cgs`	Conjugate Gradients Squared method
`gmres`	Generalized Minimum Residual method
`lsqr`	LSQR implementation of Conjugate Gradients on Normal Equations
`minres`	Minimum Residual method
`pcg`	Preconditioned Conjugate Gradients method
`qmr`	Quasi-Minimal Residual method
`spaugment`	Form least squares augmented system
`symmlq`	Symmetric LQ method

Tree Operations

`etree`	Elimination tree
`etreeplot`	Plot elimination tree
`gplot`	Plot graph, as in “graph theory”
`symbfact`	Symbolic factorization analysis
`treelayout`	Lay out tree or forest
`treeplot`	Plot picture of tree

Math Constants

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

Programming and Data Types

Functions to store and operate on data at either the MATLAB command line or in programs and scripts. Functions to write, manage, and execute MATLAB programs.

"Data Types"	Numeric, character, structures, cell arrays, and data type conversion
"Arrays"	Basic array operations and manipulation
"Operators and Operations"	Special characters and arithmetic, bit-wise, relational, logical, set, date and time operations
"Programming in MATLAB"	M-files, function/expression evaluation, program control, function handles, object oriented programming, error handling

Data Types

- "Numeric"
- "Characters and Strings"
- "Structures"
- "Cell Arrays"
- "Data Type Conversion"
- "Determine Data Type"

Numeric

`[ ]`	Array constructor
`cat`	Concatenate arrays
`class`	Return object's class name (e.g., numeric)
`find`	Find indices and values of nonzero array elements
`intmax`	Largest possible value of specified integer type
`intmin`	Smallest possible value of specified integer type
`intwarning`	Enable or disable integer warnings
`ipermute`	Inverse permute dimensions of multidimensional array
`isa`	Determine if item is object of given class (e.g., numeric)
`isequal`	Determine if arrays are numerically equal
`isequalwithequalnans`	Test for equality, treating `NaN`s as equal
`isnumeric`	Determine if item is numeric array
`isreal`	Determine if all array elements are real numbers
`isscalar`	True for scalars (1-by-1 matrices)
`isvector`	True for vectors (1-by-N or N-by-1 matrices)
`permute`	Rearrange dimensions of multidimensional array
`realmax`	Largest positive floating-point number
`realmin`	Smallest positive floating-point number
`reshape`	Reshape array
`squeeze`	Remove singleton dimensions from array
`zeros`	Create array of all zeros

Characters and Strings

Description of Strings in MATLAB

`strings`	Describes MATLAB string handling

Creating and Manipulating Strings

`blanks`	Create string of blanks
`char`	Create character array (string)
`cellstr`	Create cell array of strings from character array
`datestr`	Convert to date string format
`deblank`	Strip trailing blanks from the end of string
`lower`	Convert string to lower case
`sprintf`	Write formatted data to string
`sscanf`	Read string under format control
`strcat`	String concatenation

`strjust`	Justify character array
`strread`	Read formatted data from string
`strrep`	String search and replace
`strtrim`	Remove leading and trailing whitespace from string
`strvcat`	Vertical concatenation of strings
`upper`	Convert string to upper case

Comparing and Searching Strings

`class`	Return object's class name (e.g., char)
`findstr`	Find string within another, longer string
`isa`	Determine if item is object of given class (e.g., char)
`iscellstr`	Determine if item is cell array of strings
`ischar`	Determine if item is character array
`isletter`	Detect array elements that are letters of the alphabet
`isscalar`	True for scalars (1-by-1 matrices)
`isspace`	Detect elements that are ASCII white spaces
`isstrprop`	Determine content of each element of string
`isvector`	True for vectors (1-by-N or N-by-1 matrices)
`regexp`	Match regular expression
`regexpi`	Match regular expression, ignoring case
`regexprep`	Replace string using regular expression
`strcmp`	Compare strings
`strcmpi`	Compare strings, ignoring case
`strfind`	Find one string within another
`strmatch`	Find possible matches for string
`strncmp`	Compare first `n` characters of strings
`strncmpi`	Compare first `n` characters of strings, ignoring case
`strtok`	First token in string

Evaluating String Expressions

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

Structures

`cell2struct`	Cell array to structure array conversion
`class`	Return object's class name (e.g., struct)
`deal`	Deal inputs to outputs
`fieldnames`	Field names of structure
`isa`	Determine if item is object of given class (e.g., struct)
`isequal`	Determine if arrays are numerically equal
`isfield`	Determine if item is structure array field
`isscalar`	True for scalars (1-by-1 matrices)
`isstruct`	Determine if item is structure array
`isvector`	True for vectors (1-by-N or N-by-1 matrices)
`orderfields`	Order fields of a structure array
`rmfield`	Remove structure fields
`struct`	Create structure array
`struct2cell`	Structure to cell array conversion

Cell Arrays

`{ }`	Construct cell array
`cell`	Construct cell array
`cellfun`	Apply function to each element in cell array
`cellstr`	Create cell array of strings from character array
`cell2mat`	Convert cell array of matrices into single matrix
`cell2struct`	Cell array to structure array conversion
`celldisp`	Display cell array contents
`cellplot`	Graphically display structure of cell arrays
`class`	Return object's class name (e.g., cell)
`deal`	Deal inputs to outputs
`isa`	Determine if item is object of given class (e.g., cell)
`iscell`	Determine if item is cell array
`iscellstr`	Determine if item is cell array of strings
`isequal`	Determine if arrays are numerically equal
`isscalar`	True for scalars (1-by-1 matrices)
`isvector`	True for vectors (1-by-N or N-by-1 matrices)
`mat2cell`	Divide matrix up into cell array of matrices
`num2cell`	Convert numeric array into cell array
`struct2cell`	Structure to cell array conversion

Data Type Conversion

Numeric

`double`	Convert to double-precision
`int8`	Convert to signed 8-bit integer
`int16`	Convert to signed 16-bit integer
`int32`	Convert to signed 32-bit integer
`int64`	Convert to signed 64-bit integer
`single`	Convert to single-precision
`uint8`	Convert to unsigned 8-bit integer
`uint16`	Convert to unsigned 16-bit integer
`uint32`	Convert to unsigned 32-bit integer
`uint64`	Convert to unsigned 64-bit integer

String to Numeric

`base2dec`	Convert base `N` number string to decimal number
`bin2dec`	Convert binary number string to decimal number
`hex2dec`	Convert hexadecimal number string to decimal number
`hex2num`	Convert hexadecimal number string to double number
`str2double`	Convert string to double-precision number
`str2num`	Convert string to number

Numeric to String

`char`	Convert to character array (string)
`dec2base`	Convert decimal to base `N` number in string
`dec2bin`	Convert decimal to binary number in string
`dec2hex`	Convert decimal to hexadecimal number in string
`int2str`	Convert integer to string
`mat2str`	Convert a matrix to string
`num2str`	Convert number to string

Other Conversions

`cell2mat`	Convert cell array of matrices into single matrix
`cell2struct`	Convert cell array to structure array
`datestr`	Convert serial date number to string
`func2str`	Convert function handle to function name string
`logical`	Convert numeric to logical array
`mat2cell`	Divide matrix up into cell array of matrices
`num2cell`	Convert a numeric array to cell array
`str2func`	Convert function name string to function handle
`struct2cell`	Convert structure to cell array

Determine Data Type

`is*`	Detect state
`isa`	Determine if item is object of given class
`iscell`	Determine if item is cell array
`iscellstr`	Determine if item is cell array of strings
`ischar`	Determine if item is character array
`isfield`	Determine if item is character array
`isfloat`	True for floating-point arrays
`isinteger`	True for integer arrays
`isjava`	Determine if item is Java object
`islogical`	Determine if item is logical array
`isnumeric`	Determine if item is numeric array
`isobject`	Determine if item is MATLAB OOPs object
`isreal`	Determine if all array elements are real numbers
`isstruct`	Determine if item is MATLAB structure array

Arrays

- "Array Operations"
- "Basic Array Information"
- "Array Manipulation"
- "Elementary Arrays"

Array Operations

`[ ]`	Array constructor
`,`	Array row element separator
`;`	Array column element separator
`:`	Specify range of array elements
`end`	Indicate last index of array
`+`	Addition or unary plus
`-`	Subtraction or unary minus
`.*`	Array multiplication
`./`	Array right division
`.\`	Array left division
`.^`	Array power
`.'`	Array (nonconjugated) transpose

Basic Array Information

`disp`	Display text or array
`display`	Overloaded method to display text or array
`isempty`	Determine if array is empty
`isequal`	Determine if arrays are numerically equal
`isequalwithequalnans`	Test for equality, treating `NaN`s as equal
`islogical`	Determine if item is logical array
`isnumeric`	Determine if item is numeric array
`isscalar`	Determine if item is a scalar
`isvector`	Determine if item is a vector
`length`	Length of vector
`ndims`	Number of array dimensions
`numel`	Number of elements in matrix or cell array
`size`	Array dimensions

Array Manipulation

`:`	Specify range of array elements
`blkdiag`	Construct block diagonal matrix from input arguments
`cat`	Concatenate arrays
`circshift`	Shift array circularly
`find`	Find indices and values of nonzero elements
`fliplr`	Flip matrices left-right
`flipud`	Flip matrices up-down
`flipdim`	Flip array along specified dimension
`horzcat`	Horizontal concatenation
`ind2sub`	Subscripts from linear index
`ipermute`	Inverse permute dimensions of multidimensional array
`permute`	Rearrange dimensions of multidimensional array
`repmat`	Replicate and tile array
`reshape`	Reshape array
`rot90`	Rotate matrix 90 degrees
`shiftdim`	Shift dimensions
`sort`	Sort array elements in ascending or descending order
`sortrows`	Sort rows in ascending order
`squeeze`	Remove singleton dimensions
`sub2ind`	Single index from subscripts
`vertcat`	Horizontal concatenation

Elementary Arrays

`:`	Regularly spaced vector
`blkdiag`	Construct block diagonal matrix from input arguments
`eye`	Identity matrix
`linspace`	Generate linearly spaced vectors
`logspace`	Generate logarithmically spaced vectors
`meshgrid`	Generate X and Y matrices for three-dimensional plots
`ndgrid`	Generate arrays for multidimensional functions and interpolation
`ones`	Create array of all ones
`rand`	Uniformly distributed random numbers and arrays
`randn`	Normally distributed random numbers and arrays
`zeros`	Create array of all zeros

Operators and Operations

- "Special Characters"
- "Arithmetic Operations"
- "Bit-wise Operations"
- "Relational Operations"
- "Logical Operations"
- "Set Operations"
- "Date and Time Operations"

Special Characters

`:`	Specify range of array elements
`( )`	Pass function arguments, or prioritize operations
`[ ]`	Construct array
`{ }`	Construct cell array
`.`	Decimal point, or structure field separator
`...`	Continue statement to next line
`,`	Array row element separator
`;`	Array column element separator
`%`	Insert comment line into code
`!`	Command to operating system
`=`	Assignment

Arithmetic Operations

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

Bit-wise Operations

`bitand`	Bit-wise AND
`bitcmp`	Bit-wise complement
`bitor`	Bit-wise OR
`bitmax`	Maximum floating-point integer
`bitset`	Set bit at specified position
`bitshift`	Bit-wise shift
`bitget`	Get bit at specified position
`bitxor`	Bit-wise XOR

Relational Operations

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

Logical Operations

`&&`	Logical AND
`||`	Logical OR
`&`	Logical AND for arrays
`|`	Logical OR for arrays
`~`	Logical NOT
`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzero elements
`false`	False array
`find`	Find indices and values of nonzero elements
`is*`	Detect state
`isa`	Determine if item is object of given class
`iskeyword`	Determine if string is MATLAB keyword
`isvarname`	Determine if string is valid variable name
`logical`	Convert numeric values to logical
`true`	True array
`xor`	Logical EXCLUSIVE OR

Set Operations

`intersect`	Set intersection of two vectors
`ismember`	Detect members of set
`setdiff`	Return set difference of two vectors
`issorted`	Determine if set elements are in sorted order
`setxor`	Set exclusive or of two vectors
`union`	Set union of two vectors
`unique`	Unique elements of vector

Date and Time Operations

`addtodate`	Modify particular field of date number
`calendar`	Calendar for specified month
`clock`	Current time as date vector
`cputime`	Elapsed CPU time
`date`	Current date string
`datenum`	Serial date number
`datestr`	Convert serial date number to string
`datevec`	Date components
`eomday`	End of month
`etime`	Elapsed time
`now`	Current date and time
`tic, toc`	Stopwatch timer
`weekday`	Day of the week

Programming in MATLAB

M-File Functions and Scripts

`( )`	Pass function arguments
`%`	Insert comment line into code
`...`	Continue statement to next line
`depfun`	List dependent functions of M-file or P-file
`depdir`	List dependent directories of M-file or P-file
`echo`	Echo M-files during execution
`function`	Function M-files
`input`	Request user input
`inputname`	Input argument name
`mfilename`	Name of currently running M-file
`namelengthmax`	Return maximum identifier length
`nargin`	Number of function input arguments
`nargout`	Number of function output arguments
`nargchk`	Check number of input arguments
`nargoutchk`	Validate number of output arguments
`pcode`	Create preparsed pseudocode file (P-file)
`script`	Describes script M-file
`varargin`	Accept variable number of arguments
`varargout`	Return variable number of arguments

Evaluation of Expressions and Functions

`builtin`	Execute built-in function from overloaded method
`cellfun`	Apply function to each element in cell array
`echo`	Echo M-files during execution
`eval`	Interpret strings containing MATLAB expressions
`evalc`	Evaluate MATLAB expression with capture
`evalin`	Evaluate expression in workspace
`feval`	Evaluate function
`iskeyword`	Determine if item is MATLAB keyword
`isvarname`	Determine if item is valid variable name
`pause`	Halt execution temporarily
`run`	Run script that is not on current path
`script`	Describes script M-file
`symvar`	Determine symbolic variables in expression
`tic`, `toc`	Stopwatch timer

Timer Functions

`delete`	Delete timer object from memory
`disp`	Display information about timer object
`get`	Retrieve information about timer object properties
`isvalid`	Determine if timer object is valid
`set`	Display or set timer object properties
`start`	Start a timer
`startat`	Start a timer at a specific timer
`stop`	Stop a timer
`timer`	Create a timer object
`timerfind`	Return an array of all visible timer objects in memory
`timerfindall`	Return an array of all timer objects in memory
`wait`	Block command line until timer completes

Variables and Functions in Memory

`assignin`	Assign value to workspace variable
`genvarname`	Construct valid variable name from string
`global`	Define global variables
`inmem`	Return names of functions in memory
`isglobal`	Determine if item is global variable
`mislocked`	True if M-file cannot be cleared
`mlock`	Prevent clearing M-file from memory
`munlock`	Allow clearing M-file from memory
`namelengthmax`	Return maximum identifier length
`pack`	Consolidate workspace memory
`persistent`	Define persistent variable
`rehash`	Refresh function and file system caches

Control Flow

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

Function Handles

`class`	Return object's class name (e.g. function_handle)
`feval`	Evaluate function
`function_handle`	Describes function handle data type
`functions`	Return information about function handle
`func2str`	Constructs function name string from function handle
`isa`	Determine if item is object of given class (e.g. function_handle)
`isequal`	Determine if function handles are equal
`str2func`	Constructs function handle from function name string

Object-Oriented Programming

MATLAB Classes and Objects

`class`	Create object or return class of object
`fieldnames`	List public fields belonging to object,
`inferiorto`	Establish inferior class relationship
`isa`	Determine if item is object of given class
`isobject`	Determine if item is MATLAB OOPs object
`loadobj`	User-defined extension of `load` function for user objects
`methods`	Display information on class methods
`methodsview`	Display information on class methods in separate window
`saveobj`	User-defined extension of `save` function for user objects
`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`
`substruct`	Create structure argument for subsasgn or subsref
`superiorto`	Establish superior class relationship

Java Classes and Objects

`cell`	Convert Java array object to cell array
`class`	Return class name of Java object
`clear`	Clear Java import list or Java class definitions
`depfun`	List Java classes used by M-file
`exist`	Determine if item is Java class
`fieldnames`	List public fields belonging to object
`im2java`	Convert image to instance of Java image object
`import`	Add package or class to current Java import list
`inmem`	List names of Java classes loaded into memory
`isa`	Determine if item is object of given class
`isjava`	Determine if item is Java object
`javaaddpath`	Add entries to dynamic Java class path
javaArray	Construct Java array
`javachk`	Generate error message based on Java feature support
`javaclasspath`	Set and get dynamic Java class path
javaMethod	Invoke Java method
javaObject	Construct Java object
`javarmpath`	Remove entries from dynamic Java class path
`methods`	Display information on class methods
`methodsview`	Display information on class methods in separate window
`usejava`	Determine if a Java feature is supported in MATLAB
`which`	Display package and class name for method

Error Handling

`catch`	Begin `catch` block of `try/catch` statement
`error`	Display error message
`ferror`	Query MATLAB about errors in file input or output
`intwarning`	Enable or disable integer warnings
`lasterr`	Return last error message generated by MATLAB
`lasterror`	Last error message and related information
`lastwarn`	Return last warning message issued by MATLAB
`rethrow`	Reissue error
`try`	Begin `try` block of `try/catch` statement
`warning`	Display warning message

MEX Programming

`dbmex`	Enable MEX-file debugging
`inmem`	Return names of currently loaded MEX-files
`mex`	Compile MEX-function from C or Fortran source code
`mexext`	Return MEX-filename extension

File I/O

Functions to read and write data to files of different format types.

"Filename Construction"	Get path, directory, filename information; construct filenames
"Opening, Loading, Saving Files"	Open files; transfer data between files and MATLAB workspace
"Low-Level File I/O"	Low-level operations that use a file identifier (e.g., fopen, fseek, fread)
"Text Files"	Delimited or formatted I/O to text files
"XML Documents"	Documents written in Extensible Markup Language
"Spreadsheets"	Excel and Lotus 123 files
"Scientific Data"	CDF, FITS, HDF formats
"Audio and Audio/Video"	General audio functions; SparcStation, WAVE, AVI files
"Images"	Graphics files
"Internet Exchange"	URL, zip, and e-mail

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

Filename Construction

`fileparts`	Return parts of filename
`filesep`	Return directory separator for this platform
`fullfile`	Build full filename from parts
`tempdir`	Return name of system's temporary directory
`tempname`	Return unique string for use as temporary filename

Opening, Loading, Saving Files

`importdata`	Load data from various types of files
`load`	Load all or specific data from MAT or ASCII file
`open`	Open files of various types using appropriate editor or program
`save`	Save all or specific data to MAT or ASCII file
`uiimport`	Open Import Wizard, the graphical user interface to import data
`winopen`	Open file in appropriate application (Windows only)

Low-Level File I/O

`fclose`	Close one or more open files
`feof`	Test for end-of-file
`ferror`	Query MATLAB about errors in file input or output
`fgetl`	Return next line of file as string without line terminator(s)
`fgets`	Return next line of file as string with line terminator(s)
`fopen`	Open file or obtain information about open files
`fprintf`	Write formatted data to file
`fread`	Read binary data from file
`frewind`	Rewind open file
`fscanf`	Read formatted data from file
`fseek`	Set file position indicator
`ftell`	Get file position indicator
`fwrite`	Write binary data to file

Text Files

`csvread`	Read numeric data from text file, using comma delimiter
`csvwrite`	Write numeric data to text file, using comma delimiter
`dlmread`	Read numeric data from text file, specifying your own delimiter
`dlmwrite`	Write numeric data to text file, specifying your own delimiter
`textread`	Read data from text file, write to multiple outputs
`textscan`	Read data from text file, convert and write to cell array

XML Documents

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

Spreadsheets

Microsoft Excel Functions

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

Lotus123 Functions

`wk1read`	Read Lotus123 WK1 spreadsheet file into matrix
`wk1write`	Write matrix to Lotus123 WK1 spreadsheet file

Scientific Data

Common Data Format (CDF)

`cdfepoch`	Convert MATLAB date number or date string into CDF epoch
`cdfinfo`	Return information about CDF file
`cdfread`	Read CDF file
`cdfwrite`	Write CDF file

Flexible Image Transport System

`fitsinfo`	Return information about FITS file
`fitsread`	Read FITS file

Hierarchical Data Format (HDF)

`hdf`	Interface to HDF4 files
`hdfinfo`	Return information about HDF4 or HDF-EOS file
`hdfread`	Read HDF4 file
`hdftool`	Start HDF4 Import Tool
`hdf5`	Describes HDF5 data type objects
`hdf5info`	Return information about HDF5 file
`hdf5read`	Read HDF5 file
`hdf5write`	Write data to file in HDF5 format

Band-Interleaved Data

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

Audio and Audio/Video

General

`audioplayer`	Create audio player object
`audiorecorder`	Perform real-time audio capture
`beep`	Produce beep sound
`lin2mu`	Convert linear audio signal to mu-law
`mmfileinfo`	Information about a multimedia file
`mu2lin`	Convert mu-law audio signal to linear
`sound`	Convert vector into sound
`soundsc`	Scale data and play as sound

SPARCstation-Specific Sound Functions

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

Microsoft WAVE Sound Functions

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

Audio/Video Interleaved (AVI) Functions

`addframe`	Add frame to AVI file
`avifile`	Create new AVI file
`aviinfo`	Return information about AVI file
`aviread`	Read AVI file
`close`	Close AVI file
`movie2avi`	Create AVI movie from MATLAB movie

Images

`im2java`	Convert image to instance of Java image object
`imfinfo`	Return information about graphics file
`imread`	Read image from graphics file
`imwrite`	Write image to graphics file

Internet Exchange

`ftp`	Connect to FTP server, creating an FTP object
`sendmail`	Send e-mail message (attachments optional) to list of addresses
`unzip`	Extract contents of zip file
`urlread`	Read contents at URL
`urlwrite`	Save contents of URL to file
`zip`	Create compressed version of files in zip format

Graphics

2-D graphs, specialized plots (e.g., pie charts, histograms, and contour plots), function plotters, and Handle Graphics functions.

Basic Plots and Graphs	Linear line plots, log and semilog plots
Annotating Plots	Titles, axes labels, legends, mathematical symbols
Specialized Plotting	Bar graphs, histograms, pie charts, contour plots, function plotters
Bit-Mapped Images	Display image object, read and write graphics file, convert to movie frames
Printing	Printing and exporting figures to standard formats
Handle Graphics	Creating graphics objects, setting properties, finding handles

Basic Plots and Graphs

`box`	Axis box for 2-D and 3-D plots
`errorbar`	Plot graph with error bars
`hold`	Hold current graph
`LineSpec`	Line specification syntax
`loglog`	Plot using log-log scales
`polar`	Polar coordinate plot
`plot`	Plot vectors or matrices.
`plot3`	Plot lines and points in 3-D space
`plotyy`	Plot graphs with Y tick labels on the left and right
`semilogx`	Semi-log scale plot
`semilogy`	Semi-log scale plot
`subplot`	Create axes in tiled positions

Plotting Tools

`figurepalette`	Display figure palette on figure
`pan`	Turn panning on or off.
`plotbrowser`	Display plot browser on figure
`plottools`	Start plotting tools
`propertyeditor`	Display property editor on figure
`zoom`	Turn zooming on or off

Annotating Plots

`annotation`	Create annotation objects
`clabel`	Add contour labels to contour plot
`datetick`	Date formatted tick labels
`gtext`	Place text on 2-D graph using mouse
`legend`	Graph legend for lines and patches
`texlabel`	Produce the TeX format from character string
`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

Annotation Object Properties

`arrow`	Properties for annotation arrows
`doublearrow`	Properties for double-headed annotation arrows
`ellipse`	Properties for annotation ellipses
`line`	Properties for annotation lines
`rectangle`	Properties for annotation rectangles
`textarrow`	Properties for annotation textbox

Specialized Plotting

- "Area, Bar, and Pie Plots"
- "Contour Plots"
- "Direction and Velocity Plots"
- "Discrete Data Plots"
- "Function Plots"
- "Histograms"
- "Polygons and Surfaces"
- "Scatter/Bubble Plots"
- "Animation"

Area, Bar, and Pie Plots

`area`	Area plot
`bar`	Vertical bar chart
`barh`	Horizontal bar chart
`bar3`	Vertical 3-D bar chart
`bar3h`	Horizontal 3-D bar chart
`pareto`	Pareto char
`pie`	Pie plot
`pie3`	3-D pie plot

Contour Plots

`contour`	Contour (level curves) plot
`contour3`	3-D contour plot
`contourc`	Contour computation
`contourf`	Filled contour plot
`ezcontour`	Easy to use contour plotter
`ezcontourf`	Easy to use filled contour plotter

Direction and Velocity Plots

`comet`	Comet plot
`comet3`	3-D comet plot
`compass`	Compass plot
`feather`	Feather plot
`quiver`	Quiver (or velocity) plot
`quiver3`	3-D quiver (or velocity) plot

Discrete Data Plots

`stem`	Plot discrete sequence data
`stem3`	Plot discrete surface data
`stairs`	Stairstep graph

Function Plots

`ezcontour`	Easy to use contour plotter
`ezcontourf`	Easy to use filled contour plotter
`ezmesh`	Easy to use 3-D mesh plotter
`ezmeshc`	Easy to use combination mesh/contour plotter
`ezplot`	Easy to use function plotter
`ezplot3`	Easy to use 3-D parametric curve plotter
`ezpolar`	Easy to use polar coordinate plotter
`ezsurf`	Easy to use 3-D colored surface plotter
`ezsurfc`	Easy to use combination surface/contour plotter
`fplot`	Plot a function

Histograms

`hist`	Plot histograms
`histc`	Histogram count
`rose`	Plot rose or angle histogram

Polygons and Surfaces

`convhull`	Convex hull
`cylinder`	Generate cylinder
`delaunay`	Delaunay triangulation
`dsearch`	Search Delaunay triangulation for nearest point
`ellipsoid`	Generate ellipsoid
`fill`	Draw filled 2-D polygons
`fill3`	Draw filled 3-D polygons in 3-space
`inpolygon`	True for points inside a polygonal region
`pcolor`	Pseudocolor (checkerboard) plot
`polyarea`	Area of polygon
`ribbon`	Ribbon plot
`slice`	Volumetric slice plot
`sphere`	Generate sphere
`tsearch`	Search for enclosing Delaunay triangle
`voronoi`	Voronoi diagram
`waterfall`	Waterfall plot

Scatter/Bubble Plots

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

Animation

`frame2im`	Convert movie frame to indexed image
`getframe`	Capture movie frame
`im2frame`	Convert image to movie frame
`movie`	Play recorded movie frames
`noanimate`	Change `EraseMode` of all objects to `normal`

Bit-Mapped Images

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

Printing

`frameedit`	Edit print frame for Simulink and Stateflow diagram
`orient`	Hardcopy paper orientation
`pagesetupdlg`	Page setup dialog box
`print`	Print graph or save graph to file
`printdlg`	Print dialog box
`printopt`	Configure local printer defaults
`printpreview`	Preview figure to be printed
`saveas`	Save figure to graphic file

Handle Graphics

- Finding and Identifying Graphics Objects
- Object Creation Functions
- Figure Windows
- Axes Operations

Finding and Identifying Graphics Objects

`allchild`	Find all children of specified objects
`ancestor`	Find ancestor of graphics object
`copyobj`	Make copy of graphics object and its children
`delete`	Delete files or graphics objects
`findall`	Find all graphics objects (including hidden handles)
`figflag`	Test if figure is on screen
`findfigs`	Display off-screen visible figure windows
`findobj`	Find objects with specified property values
`gca`	Get current Axes handle
`gcbo`	Return object whose callback is currently executing
`gcbf`	Return handle of figure containing callback object
`gco`	Return handle of current object
`get`	Get object properties
`ishandle`	True if value is valid object handle
`set`	Set object properties

Object Creation Functions

`axes`	Create axes object
`figure`	Create figure (graph) windows
`hggroup`	Create a group object
`hgtransform`	Create a group to transform
`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)
`rootobject`	List of root properties
`surface`	Create surface (quadrilaterals)
`text`	Create text object (character strings)
`uicontextmenu`	Create context menu (popup associated with object)

Plot Objects

`areaseries`	Property list
`barseries`	Property list
`contourgroup`	Property list
`errorbarseries`	Property list
`lineseries`	Property list
`quivergroup`	Property list
`scattergroup`	Property list
`stairseries`	Property list
`stemseries`	Property list
`surfaceplot`	Property list

Figure Windows

`clc`	Clear figure window
`clf`	Clear figure
`close`	Close specified window
`closereq`	Default close request function
`drawnow`	Complete any pending drawing
`figflag`	Test if figure is on screen
`gcf`	Get current figure handle
`hgload`	Load graphics object hierarchy from a FIG-file
`hgsave`	Save graphics object hierarchy to a FIG-file
`newplot`	Graphics M-file preamble for `NextPlot` property
`opengl`	Change automatic selection mode of OpenGL rendering
`refresh`	Refresh figure
`saveas`	Save figure or model to desired output format

Axes Operations

`axis`	Plot axis scaling and appearance
`box`	Display axes border
`cla`	Clear Axes
`gca`	Get current Axes handle
`grid`	Grid lines for 2-D and 3-D plots
`ishold`	Get the current hold state
`makehgtform`	Create a transform matrix

Operating on Object Properties

`get`	Get object properties
linkaxes	Synchronize limits of specified axes
linkprop	Maintain same value for corresponding properties
`set`	Set object properties

3-D Visualization

Create and manipulate graphics that display 2-D matrix and 3-D volume data, controlling the view, lighting and transparency.

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

Surface and Mesh Plots

- Creating Surfaces and Meshes
- Domain Generation
- Color Operations
- Colormaps

Creating Surfaces and Meshes

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

Domain Generation

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

Color Operations

`brighten`	Brighten or darken colormap
`caxis`	Pseudocolor axis scaling
`colormapeditor`	Start colormap editor
`colorbar`	Display color bar (color scale)
`colordef`	Set up color defaults
`colormap`	Set the color look-up table (list of colormaps)
`ColorSpec`	Ways to specify color
`graymon`	Graphics figure defaults set for grayscale monitor
`hsv2rgb`	Hue-saturation-value to red-green-blue conversion
`rgb2hsv`	RGB to HSVconversion
`rgbplot`	Plot colormap
`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 colormap
`bone`	Gray-scale with a tinge of blue colormap
`contrast`	Gray colormap to enhance image contrast
`cool`	Shades of cyan and magenta colormap
`copper`	Linear copper-tone colormap
`flag`	Alternating red, white, blue, and black colormap
`gray`	Linear gray-scale colormap
`hot`	Black-red-yellow-white colormap
`hsv`	Hue-saturation-value (HSV) colormap
`jet`	Variant of HSV
`lines`	Line color colormap
`prism`	Colormap of prism colors
`spring`	Shades of magenta and yellow colormap
`summer`	Shades of green and yellow colormap
`winter`	Shades of blue and green colormap

View Control

- Controlling the Camera Viewpoint
- Setting the Aspect Ratio and Axis Limits
- Object Manipulation
- Selecting Region of Interest

Controlling the Camera Viewpoint

`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
`cameratoolbar`	Control camera toolbar programmatically
`camup`	Set or get camera up-vector
`camva`	Set or get camera view angle
`camzoom`	Zoom camera in or out
`view`	3-D graph viewpoint specification.
`viewmtx`	Generate view transformation matrices
`makehgtform`	Create a transform matrix

Setting the Aspect Ratio and Axis Limits

`daspect`	Set or get data aspect ratio
`pbaspect`	Set or get plot box aspect ratio
`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

Object Manipulation

`pan`	Turns panning on or off
`reset`	Reset axis or figure
`rotate`	Rotate objects about specified origin and direction
`rotate3d`	Interactively rotate the view of a 3-D plot
`selectmoveresize`	Interactively select, move, or resize objects
`zoom`	Zoom in and out on a 2-D plot

Selecting Region of Interest

`dragrect`	Drag XOR rectangles with mouse
`rbbox`	Rubberband box

Lighting

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

Volume Visualization

`coneplot`	Plot velocity vectors as cones in 3-D vector field
`contourslice`	Draw contours in volume slice plane
`curl`	Compute curl and angular velocity of vector field
`divergence`	Compute divergence of vector field
`flow`	Generate scalar volume data
`interpstreamspeed`	Interpolate streamline vertices from vector-field magnitudes
`isocaps`	Compute isosurface end-cap geometry
`isocolors`	Compute colors of isosurface vertices
`isonormals`	Compute normals of isosurface vertices
`isosurface`	Extract isosurface data from volume data
`reducepatch`	Reduce number of patch faces
`reducevolume`	Reduce number of elements in volume data set
`shrinkfaces`	Reduce size of patch faces
`slice`	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 surface data to patch data
`subvolume`	Extract subset of volume data set
`volumebounds`	Return coordinate and color limits for volume (scalar and vector)

Creating Graphical User Interfaces

Predefined dialog boxes and functions to control GUI programs.

Predefined Dialog Boxes	Dialog boxes for error, user input, waiting, etc.
Deploying User Interfaces	Launching GUIs, creating the handles structure
Developing User Interfaces	Starting GUIDE, managing application data, getting user input
User Interface Objects	Creating GUI components
Finding Objects from Callbacks	Finding object handles from within callbacks functions
GUI Utility Functions	Moving objects, text wrapping
Controlling Program Execution	Wait and resume based on user input

Predefined Dialog Boxes

`dialog`	Create 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
`pagesetupdlg`	Page setup dialog box
`printdlg`	Display print dialog box
`questdlg`	Create question dialog box
`uigetdir`	Display dialog box to retrieve name of directory
`uigetfile`	Display dialog box to retrieve name of file for reading
`uiputfile`	Display dialog box to retrieve name of file for writing
`uisetcolor`	Set `ColorSpec` using dialog box
`uisetfont`	Set font using dialog box
`waitbar`	Display wait bar
`warndlg`	Create warning dialog box

Deploying User Interfaces

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

Developing User Interfaces

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

Working with Application Data

`getappdata`	Get value of application data
`isappdata`	True if application data exists
`rmappdata`	Remove application data
`setappdata`	Specify application data

Interactive User Input

`ginput`	Graphical input from a mouse or cursor
`waitfor`	Wait for conditions before resuming execution
`waitforbuttonpress`	Wait for key/buttonpress over figure

User Interface Objects

`menu`	Generate menu of choices for user input
`uibuttongroup`	Create component to exclusively manage radiobuttons and togglebuttons
`uicontextmenu`	Create context menu
`uicontrol`	Create user interface control
`uimenu`	Create user interface menu
`uipanel`	Create panel container object
`uipushtool`	Create toolbar push button
`uitoggletool`	Create toolbar toggle button
`uitoolbar`	Create toolbar

Finding Objects from Callbacks

`findall`	Find all graphics objects
`findfigs`	Display off-screen visible figure windows
`findobj`	Find specific graphics object
`gcbf`	Return handle of figure containing callback object
`gcbo`	Return handle of object whose callback is executing

Functions — Alphabetical List

2factor

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`

Purpose Factorial function

Syntax

```
factorial(N)
```

Description

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

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

See Also `prod`

false

Purpose

False array

Syntax

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

Description

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

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

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

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

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

Remarks

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

See Also

`true`, `logical`

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`

feather

Purpose

Plot velocity vectors

Syntax

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

Description

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

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

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

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

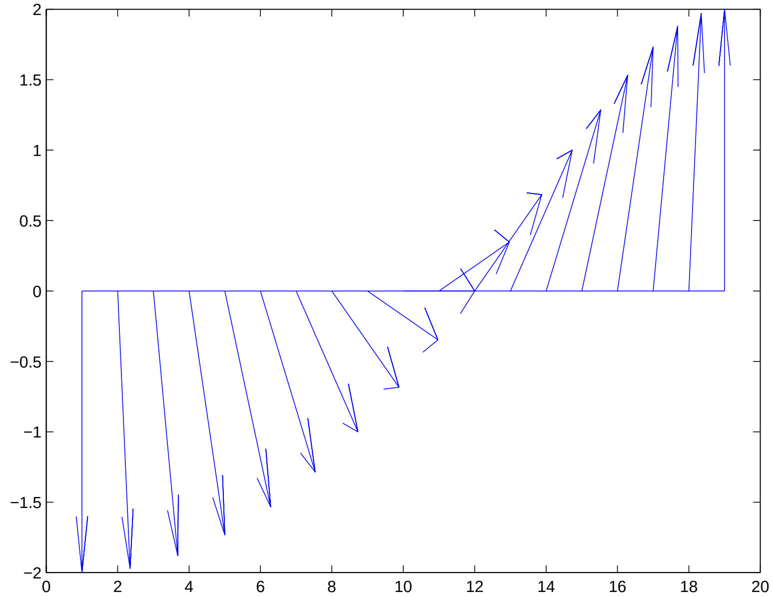
`feather(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

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

Examples

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

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



See Also

`compass`, `LineSpec`, `rose`

"Direction and Velocity Plots" for related functions

feof

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`

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

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

`[message,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 can contain information about the nature of the error. If the message is not helpful, consult the C run-time library manual for your host operating system for further details.

See Also

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

feval

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.

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

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

Remarks The following two statements are equivalent.

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

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

Discrete 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.

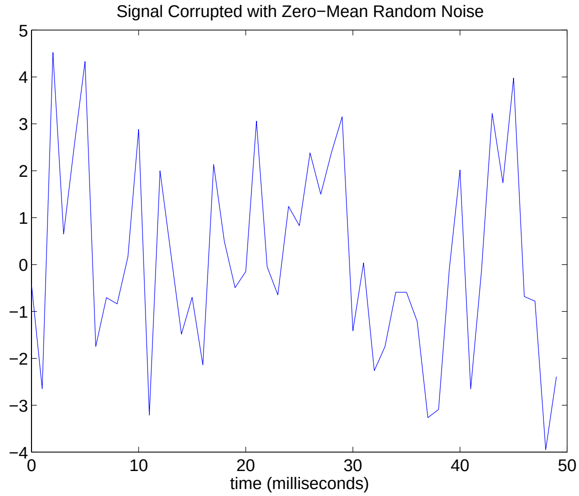
`Y = fft(X,n)` returns the `n`-point DFT. If the length of `X` is less than `n`, `X` is padded with trailing zeros to length `n`. If the length of `X` is greater than `n`, the sequence `X` is truncated. When `X` is a matrix, the length of the columns are adjusted in the same manner.

`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(1000*t(1:50),y(1:50))
title('Signal Corrupted with Zero-Mean Random Noise')
xlabel('time (milliseconds)')
```



It is difficult to identify the frequency components by looking at the original signal. Converting to the frequency domain, the discrete Fourier transform of the noisy signal `y` is found by taking the 512-point fast Fourier transform (FFT):

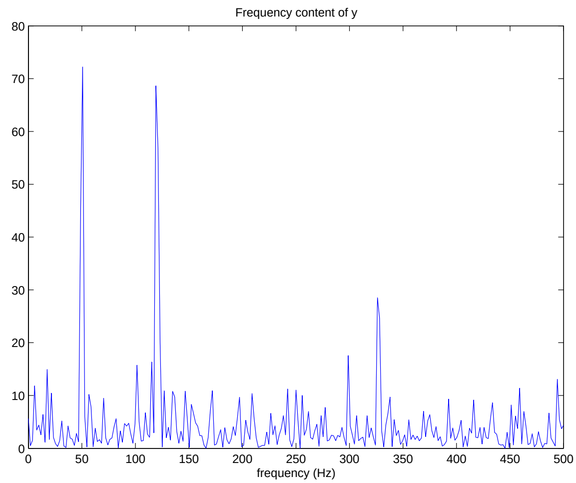
```
Y = fft(y,512);
```

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 $N = N_1 N_2$), the FFTW library decomposes the problem using the Cooley-Tukey algorithm [1], which first computes N_1 transforms of size N_2, and then computes N_2 transforms of size N_1. The decomposition is applied recursively to both the N_1- and N_2-point DFTs until the problem can be solved using one of several machine-generated fixed-size "codelets." The codelets in turn use several algorithms in combination, including a variation of Cooley-Tukey [5], a prime factor algorithm [6], and a split-radix algorithm [2]. The particular factorization of N is chosen heuristically.

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

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

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

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

Data Type Support

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

See Also

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

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

References

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

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

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

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

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

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

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

Purpose Two-dimensional discrete Fourier transform

Syntax

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

Description

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

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

Algorithm

`fft2(X)` can be simply computed as

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

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

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

Data Type Support

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

See Also

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

Purpose Multidimensional discrete Fourier transform

Syntax

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

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

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

Algorithm `fftn(X)` is equivalent to

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

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

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

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

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

Purpose

Shift zero-frequency component of discrete Fourier transform to center of spectrum

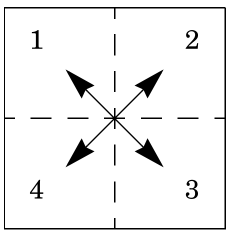
Syntax

```
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.

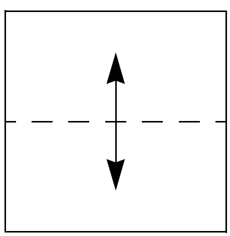
For vectors, `fftshift(X)` swaps the left and right halves of `X`. For matrices, `fftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth.



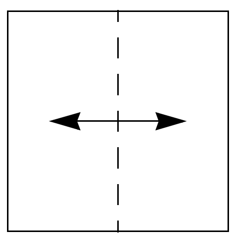
For higher-dimensional arrays, `fftshift(X)` swaps “half-spaces” of `X` along each dimension.

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

For `dim = 1`:



For `dim = 2`:



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

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

Purpose

Interface to the FFTW library run-time algorithm for tuning fast Fourier transform (FFT) computations

Syntax

```
fftw('planner', method)
method = fftw('planner')
str = fftw('wisdom')
fftw('wisdom', str)
fftw('wisdom', '')
fftw('wisdom', [])
```

Description

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

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

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

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

Note The FFT functions only uses the optimal FFT algorithm during the current MATLAB session. "Reusing Optimal FFT Algorithms" on page 2-760 explains how to ruse the optimal algorithm in a future MATLAB session.

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

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

If you set `'planner'` to `'hybrid'`, the default method, MATLAB

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

The following table compares the run times off the FFT functions for the different methods

Method	First Run of FFT Function	Subsequent Runs of FFT Function
`'estimate'`	Fastest	Slowest
`'measure'`	Faster	Slower
`'patient'`	Slower	Faster
`'exhaustive'`	Slowest	Fastest

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

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

`fftw('wisdom', str)` loads the string `str`, containing FFTW wisdom, into the FFTW library's internal wisdom database.

`fftw('wisdom','')` or `fftw('wisdom',[])` clears the internal wisdom database.

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

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

Example

Comparison of Speed for Different Planner Methods

The following example illustrates the run times for different settings of `'planner'`. The example first creates some data and applies `fft` to it using the default method `'hybrid'`. Since the dimension of the FFT is 1458, which is less than 8192, `'hybrid'` uses the same method as `'measure'`.

```
t=0:.001:5;
x = sin(2*pi*50*t)+sin(2*pi*120*t);
y = x + 2*randn(size(t));
tic; Y = fft(y,1458); toc
Elapsed time is 0.030000 seconds.
```

If you execute the commands

```
tic; Y = fft(y,1458); toc
```

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

```
tic; for k=1:1000
Y = fft(y,1458);
end; toc
Elapsed time is 0.911000 seconds.
```

This tells you that it takes approximately 1/1000 of a second to execute `fft(y, 1458)` a single time.

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

```
fftw('planner','patient')
tic;Y = fft(y,1458);toc
Elapsed time is 0.130000 seconds.
```

However, the next time you call `fft`, it runs approximately 10 times faster than it when you use the method `'measure'`.

```
tic;for k=1:1000
Y=fft(y,1458);
end;toc
Elapsed time is 0.080000 seconds.
```

Reusing Optimal FFT Algorithms

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

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

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

```
save str
```

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

```
load str
```

and then reload the "wisdom" into the FFTW database using the command

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

See Also

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

Purpose Read line from file, 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`.

Examples

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

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

See Also

`fgets`

fgets

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`

Purpose Return field names of a structure, or property names of an 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 a MATLAB, COM, or 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 a MATLAB, COM, or 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 `f` is an object of Java class `java.awt.Frame`, the command `fieldnames(f)` lists the properties of `f`.

```
f = java.awt.Frame;

fieldnames(f)
ans =
    'WIDTH'
    'HEIGHT'
    'PROPERTIES'
    'SOMEBITS'
```

```
'FRAMEBITS'
'ALLBITS'
    .
    .
```

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

figflag

Purpose This function is OBSOLETE.

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 sends 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 sends 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 Windows" for related functions

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.

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

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])
```

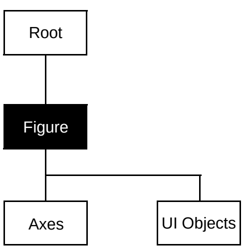
See Also

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

"Object Creation Functions" for related functions

Figure Properties for additional information on figure properties

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 take you to 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`)
`DockControls`	Can figure be docked in the desktop	Values: `on`, `off` Default: `on`
`MenuBar`	Toggles the figure menu bar on and off	Values: `none`, `figure` Default: `figure`
`Name`	Figure window title	Values: string Default: `''` (empty string)
`NumberTitle`	Displays “Figure No. n”, where n is the figure number	Values: `on`, `off` Default: `on`
`Resize`	Specifies whether the figure window can be resized using the mouse	Values: `on`, `off` Default: `on`
`SelectionHighlight`	Highlights figure when selected (`Selected` property is set to `on`)	Values: `on`, `off` Default: `on`
`Toolbar`	Control display of figure toolbar	Values: `none`, `auto`, `figure` Default: `auto`
`Visible`	Makes the figure visible or invisible	Values: `on`, `off` Default: `on`

Property Name	Property Description	Property Value
`WindowStyle`	Selects 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
`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`	Allows 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
Properties That Affect Rendering		
`BackingStore`	Enables off-screen pixel buffering	Values: `on`, `off` Default: `on`
`DoubleBuffer`	Flash-free rendering for simple animations	Values: `on`, `off` Default: `on`
`Renderer`	Rendering method used for screen and printing	Values: `painters`, `zbuffer`, `OpenGL` Default: automatic selection by MATLAB
`RendererMode`	Automatic or user-selected renderer	Values: `auto`, `manual` Default: `auto`
`WVisual`	Specifies the pixel format MATLAB uses for figures. (Windows only)	Value: identifier string Default: automatically selected by MATLAB

Property Name	Property Description	Property Value
`XDisplay`	Specifies display for MATLAB (UNIX only)	Value: display identifier Default: `:0.0`
`XVisual`	Selects visual used by MATLAB (UNIX only)	Value: visual ID
`XVisualMode`	Auto or manual selection of visual (UNIX only)	Values: `auto`, `manual` Default: `auto`
General Information About the Figure		
`Children`	Handles of any ui objects or axes contained in the figure	Value: vector of handles
`FileName`	Used by `guide`	String
`Parent`	The root object is the parent of all figures.	Value: always `0`
`Selected`	Indicates 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	Value: any matrix Default: `[]` (empty matrix)
Information About Current State		
`CurrentAxes`	Handle of the current axes in this figure	Value: axes handle
`Curren tCharacter`	The last key pressed in this figure	Value: single character
`CurrentObject`	Handle of the current object in this figure	Value: graphics object handle

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

Property Name	Property Description	Property Value
UIContextMenu	Associates a context menu with the figure	Value: handle of a Uicontrextmenu
WindowButtonDownFcn	Defines a callback routine that executes when you press the mouse button down in the figure	Values: string or function handle Default: empty string
WindowButtonMotionFcn	Defines a callback routine that executes when you move the pointer in the figure	Values: string or function handle Default: empty string
WindowButtonUpFcn	Defines a callback routine that executes when you release the mouse button	Values: string or function handle Default: empty string
Controlling Access to Objects		
IntegerHandle	Specifies integer or noninteger figure handle	Values: on, off Default: on (integer handle)
HandleVisibility	Determines if figure handle is visible to users or not	Values: on, callback, off Default: on
HitTest	Determines if the figure can become the current object (see the figure CurrentObject property)	Values: on, off Default: on
NextPlot	Determines how to display additional graphics to this figure	Values: add, replace, replacechildren Default: add
Defining the Pointer		
Pointer	Selects 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	Value: 16-by-16 matrix Default: set Pointer to custom and see
PointerShapeHotSpot	Specifies the pointer active spot	Value: 2-element vector [row, column] Default: [1,1]
Properties That Affect Printing		
InvertHardcopy	Changes figure colors for printing	Values: on, off Default: on
PaperOrientation	Horizontal or vertical paper orientation	Values: portrait, landscape Default: portrait
PaperPosition	Controls positioning figure on printed page	Value: 4-element vector [left, bottom, width, height]
PaperPositionMode	Enables WYSIWYG printing of figure	Values: auto, manual Default: manual
PaperSize	Size of the current PaperType specified in PaperUnits	Values: [width, height]
PaperType	Selects 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

Modifying Properties

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

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

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

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`

Offscreen pixel buffer. When `BackingStore` is `on`, MATLAB stores a copy of the figure window in an offscreen 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 backing store, 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.

Note that when the `Renderer` is set to `opengl`, MATLAB sets `BackingStore` to `off`.

`BeingDeleted` `on | {off}` Read Only

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

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

`BusyAction` `cancel | {queue}`

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

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

`ButtonDownFcn` string or function handle

Button press callback 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.

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

`Children` vector of handles

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

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

`Clipping` `{on} | off`

This property has no effect on figures.

`CloseRequestFcn` string or function handle

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

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

The basic mechanism is

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

```
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.

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

Color ColorSpec

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

Colormap m-by-3 matrix of RGB values

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

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 view. You can use this property to determine which object a user has selected. The function `gco` provides a convenient way to retrieve the `CurrentObject` of the `CurrentFigure`.

`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 a uicontrol object.

`DeleteFcn` string or function handle

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`.

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

`Dithermap` Obsolete

This property is not useful with TrueColor displays and will be removed in a future release.

`DithermapMode` Obsolete

This property is not useful with TrueColor displays and will be removed in a future release.

`DockControls` `{on} | off`

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

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

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

`DoubleBuffer` `{on} | off`

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

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

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

`FileName` String

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

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

Noncolormap colors. Fixed colors define all colors appearing in a figure window that are not obtained from the figure colormap. These colors include axis lines and labels, the colors 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`

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 the figure sets the `CurrentObject` to the empty matrix.

`IntegerHandle` `{on} | off`

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

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

`Interruptible` `{on} | off`

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

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

The callback 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 can query the figure's `SelectionType` property to determine whether modifier keys were also pressed.

The callback 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 to by the `gcf` command).

KeyPressFcn Event Structure

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

Field	Contents
`Character`	The character displayed as a result of the key(s) pressed.
`Modifier`	This field is a cell array that contains the names of one or more modifier keys that the user pressed (i.e., **Control**, **Alt**, **Shift**).
`Key`	The key pressed (lower case label on key)

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

Using the KeyPressFcn

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

```
function figure_keypress
figure('KeyPressFcn',@printfig);

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

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

`MenuBar` `none | {figure}`

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

This property affects only built-in menus. 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 that MATLAB uses exactly the colors you define in the figure colormap, set `MinColorMap` equal to the length of the colormap.

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

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

`Name` string

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

`NextPlot` `{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 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
`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

Property Value	Size (Width x Height)
`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 and Specifying the Figure Pointer for more information.

PointerShapeCData 16-by-16 matrix

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

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

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

PointerShapeHotSpot two-element vector

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

Position four-element vector

Figure position. This property specifies the size and location on the screen of the figure window. 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.

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

Renderer `painters | zbuffer | OpenGL`

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

- `painters` — The original rendering method used by MATLAB is faster when the figure contains only simple or small graphics objects.
- `zbuffer` — MATLAB draws graphics objects faster and more accurately because 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. Note that when the `Renderer` is set to `opengl`, MATLAB sets `BackingStore` to `off`.

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, OpenGL is available as part of the operating system. If you experience problems with OpenGL, contact your graphics driver vendor to obtain the latest qualified version of OpenGL.

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

OpenGL Renderer Feature – Microsoft Windows

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

```
feature('UseGenericOpenGL',1)
```

This command forces MATLAB to use generic OpenGL on Microsoft Windows computers. Generic OpenGL is useful if your hardware version of OpenGL does not function correctly and you want to use image, patch, or surface transparency, which requires the OpenGL renderer. To reenable hardware OpenGL, use the command

```
feature('UseGenericOpenGL',0)
```

Note that the default setting is to use hardware OpenGL. To query the current state of the generic OpenGL feature, use the command

```
feature('UseGenericOpenGL')
```

See the `opengl` reference page for additional information

Determining What Version You Are Using

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

```
opengl info
```

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 differences 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 interpolates the colors through the RGB color cube instead of through the colormap.
- OpenGL does not support the `phong` value for the `FaceLighting` and `EdgeLighting` properties of surfaces and patches.
- OpenGL does not support logarithmic-scale axes.

If You Are Having Problems

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

`RendererMode` `{auto} | manual`

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

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

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

Criteria for Autoselection of OpenGL Renderer

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

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

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

Or

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

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

`Resize` `{on} | off`

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

`ResizeFcn` string or function handle

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 the MATLAB `Position`/`Units` paradigm.

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

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

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

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

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

See Resize Behavior for information on creating resize functions using GUIDE.

`Selected` `on | off`

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 a graphics object places it in move or resize mode; double-clicking 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 sets 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 a uicontrol object.

ShareColors `{on} | off` **`Obsolete`**

Share slots in system color table with like colors. This property is obsolete because MATLAB now requires true color systems.

Tag string

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

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

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

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

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

`Toolbar` `none | {auto} | figure`

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

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

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

`Type` `string (read only)`

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

`UIContextMenu` handle of a uicontextmenu object

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

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

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

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

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

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

UserData matrix

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

Visible {on} | off

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

WindowButtonDownFcnstring or functional handle

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

`WindowButtonMotionFcn`string or functional handle

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.

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

`WindowButtonUpFcn` string or function handle

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`.

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

`WindowStyle` `{normal} | modal | docked`

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

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.

Docked WindowStyle

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

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

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

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

WVisual identifier string (MS Windows only)

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

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

Querying Available Pixel Formats on Window Systems

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

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

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

```
01 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated,
Opengl, GDI, Window)
02 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated,
Opengl, Double Buffered, Window)
03 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated,
Opengl, Double Buffered, Window)
```

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

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

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

Understanding the WVisual String

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

- `RGB 16 bits(05 06 05 00)` – indicates true color with 16-bit resolution (5 bits for red, 6 bits for green, 5 bits for blue, and 0 for alpha (transparency). MATLAB requires true color.

- `zdepth 24` – indicates 24-bit resolution for sorting object's front to back position on the screen. Selecting pixel formats with higher (24 or 32) zdepth might solve sorting problems.
- `Hardware Accelerated` – some graphics functions may be performed by hardware for increased speed. If there are incompatibilities between your particular graphic hardware and MATLAB, select a pixel format in which the term `Generic` appears instead of `Hardware Accelerated`.
- `Opengl` – supports OpenGL. See "Pixel Formats and OpenGL" for more information.
- `GDI` – supports for Windows 2-D graphics interface.
- `Double Buffered` – support for double buffering with the OpenGL renderer. Note that the figure `DoubleBuffer` property applies only to the painters renderer.
- `Bitmap` – support for rendering into a bitmap (as opposed to drawing in the window)
- `Window` – support for rendering into a window

Pixel Formats and OpenGL

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

```
feature('UseGenericOpenGL',1)
```

Then allow MATLAB to select best pixel format to use.

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

`WVisualMode` `auto | manual` (MS Windows only)

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

XDisplay display identifier (UNIX only)

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 available on your system, use the UNIX `xdpyinfo` command. From MATLAB, type

```
!xdpyinfo
```

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

```
visual id:    0x23
```

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

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

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

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

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

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

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

```
!glxinfo
```

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

```
visual  x  bf lv rg d st colorbuffer ax dp st accumbuffer  ms  cav
 id dep cl sp sz l  ci b ro  r  g  b  a bf th cl  r  g  b  a ns b eat
----------------------------------------------------------------------
0x23 24 tc  0 24  0 r  y  .  8  8  8  8  0  0  0  0  0  0  0  0 0 None
0x24 24 tc  0 24  0 r  .  .  8  8  8  8  0  0  0  0  0  0  0  0 0 None
0x25 24 tc  0 24  0 r  y  .  8  8  8  8  0 24  8  0  0  0  0  0 0 None
0x26 24 tc  0 24  0 r  .  .  8  8  8  8  0 24  8  0  0  0  0  0 0 None
0x27 24 tc  0 24  0 r  y  .  8  8  8  8  0  0  0 16 16 16  0  0 0 Slow
```

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

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

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

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

XVisualMode `auto | manual`

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`.

figurepalette

Purpose

Show or hide figure palette

Syntax

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

Description

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

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

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

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

See Also

`plotbrowser`, `propertyeditor`

Purpose Set or get attributes of file or directory

Syntax

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

Description The `fileattrib` function is like the DOS `attrib` command or the UNIX `chmod` command.

`fileattrib` displays the attributes for the current directory. Values are

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

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

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

Value for attrib	Description
`a`	Archive (Windows only)
`h`	Hidden file (Windows only)

Value for attrib	Description
`s`	System file (Windows only)
`w`	Write access (Windows and UNIX)
`x`	Executable (UNIX only)

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

`fileattrib('name','`*`attrib`*`','`*`users`*`')` sets the attribute for `name`, where `name` is the absolute or relative pathname for a directory or file, and defines which users are affected by *`attrib`*, where *`users`* is applicable only for UNIX systems. For more information about these attributes, see UNIX reference information for `chmod`. The default value for *`users`* is `u`. Values for *`users`* are

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

`fileattrib('name','`*`attrib`*`','`*`users`*`','`**`s`**`')` sets the attribute for `name`, where `name` is the absolute or relative pathname for a file or a directory and its contents, and defines which users are affected by *`attrib`*. Here the **`s`** specifies that *`attrib`* be applied to all contents of `name`, where `name` is a directory.

`[status,message,messageid] =`
`fileattrib('name','`*`attrib`*`','`*`users`*`','`**`s`**`')` sets the attribute for `name`, returning the status, a message, and the MATLAB error message ID (see `error` and `lasterr`). Here, `status` is `1` for success and is `0` for error. If `attrib`, *`users`*, and **`s`** are not specified, and `status` is `1`, `message` is a structure containing the file attributes and `messageid` is blank. If `status` is `0`, `messageid` contains the error. If you use a wildcard **`*`** at the end of `name`, `mess` will be a structure.

Examples

Get Attributes of File

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

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

MATLAB returns

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

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

Set File Attribute

To make `myfile.m` become writable, type

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

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

Set Attributes for Specified Users

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

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

The `-` preceding the write attribute, `w`, specifies that write status is removed.

Set Multiple Attributes for Directory and Its Contents

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

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

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

Return Status and Structure of Attributes

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

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

MATLAB returns

```
stat =
     1

mess =
            Name: 'd:\work\results'
         archive: 0
          system: 0
          hidden: 0
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
       GroupRead: NaN
      GroupWrite: NaN
    GroupExecute: NaN
       OtherRead: NaN
      OtherWrite: NaN
    OtherExecute: NaN
```

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

```
mess.Name
```

returns the path for `results`

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

Return Attributes with Wildcard for name

Return the attributes for all files in the current directory whose names begin with `new`.

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

MATLAB returns

```
stat =
     1

mess =
1x3 struct array with fields:
    Name
    archive
    system
    hidden
    directory
    UserRead
    UserWrite
    UserExecute
    GroupRead
    GroupWrite
    GroupExecute
    OtherRead
    OtherWrite
    OtherExecute
```

The results indicate there are three matching files. To view the filenames, type

```
mess.Name
```

MATLAB returns

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

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

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

To view just the first filename, type

```
mess(1).Name

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

See Also

`copyfile`, `cd`, `dir`, `filebrowser`, `fileparts`, `ls`, `mfilename`, `mkdir`, `movefile`, `rmdir`

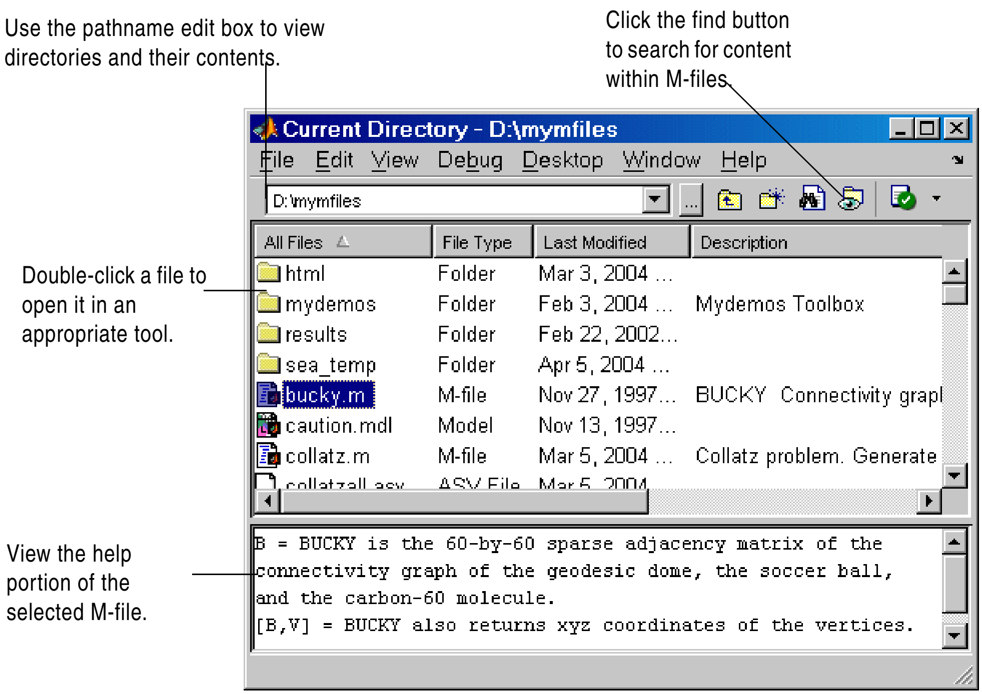
Purpose Display Current Directory browser, a tool for viewing files in current directory

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

Syntax

```
filebrowser
```

Description `filebrowser` displays the Current Directory browser.



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

file formats

Purpose

Readable file formats

Description

This table shows the file formats that MATLAB is capable of reading.

File Format	Extension	File Content	Read Command	Returns
Text	MAT	Saved MATLAB workspace	`load`	Variables in the file
	CSV	Comma-separated numbers	`csvread`	Double array
	DLM	Delimited text	`dlmread`	Double array
	TAB	Tab-separated text	`dlmread`	Double array
Scientific Data	CDF	Data in Common Data Format	`cdfread`	Cell array of CDF records
	FITS	Flexible Image Transport System data	`fitsread`	Primary or extension table data
	HDF	Data in Hierarchical Data Format	`hdfread`	HDF or HDF-EOS data set
Spread-sheet	XLS	Excel worksheet	`xlsread`	Double or cell array
	WK1	Lotus 123 worksheet	`wk1read`	Double or cell array

File Format	Extension	File Content	Read Command	Returns
Image	TIFF	TIFF image	`imread`	True color, grayscale, or indexed image(s)
	PNG	PNG image	`imread`	True color, grayscale, or indexed image
	HDF	HDF image	`imread`	True color, grayscale, or indexed image(s)
	BMP	BMP image	`imread`	True color or indexed image
	JPEG	JPEG image	`imread`	True color or grayscale image
	GIF	GIF image	`imread`	Indexed image
	PCX	PCX image	`imread`	Indexed image
	XWD	XWD image	`imread`	Indexed image
	CUR	Cursor image	`imread`	Indexed image
	ICO	Icon image	`imread`	Indexed image

File Format	Extension	File Content	Read Command	Returns
Audio file	AU	NeXT/SUN sound	`auread`	Sound data and sample rate
	WAV	Microsoft WAVE sound	`wavread`	Sound data and sample rate
Movie	AVI	Audio/video	`aviread`	MATLAB movie

See Also

`fscanf`, `fread`, `textread`, `importdata`

Purpose Return filename parts

Syntax

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

Description

`[pathstr,name,ext,versn] = fileparts('filename')` returns the path, filename, extension, and version for the specified file. The returned `ext` field contains a dot (`.`) before the file extension.

The `fileparts` function is platform dependent.

You can reconstruct the file from the parts using

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

Examples

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

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

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

pathstr =
\home\user4\matlab

name =
classpath

ext =
.txt

versn =
     ''
```

See Also

`fullfile`

Purpose Return the directory separator for this platform

Syntax

```
f = filesep
```

Description `f = filesep` returns the platform-specific file separator character. The file separator is the character that separates individual directory names in a path string.

Examples On the PC,

```
iofun_dir = ['toolbox' filesep 'matlab' filesep 'iofun']

iofun_dir =

toolbox\matlab\iofun
```

On a UNIX system,

```
iodir = ['toolbox' filesep 'matlab' filesep 'iofun']

iodir =

toolbox/matlab/iofun
```

See Also `fullfile`, `fileparts`, `pathsep`

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.

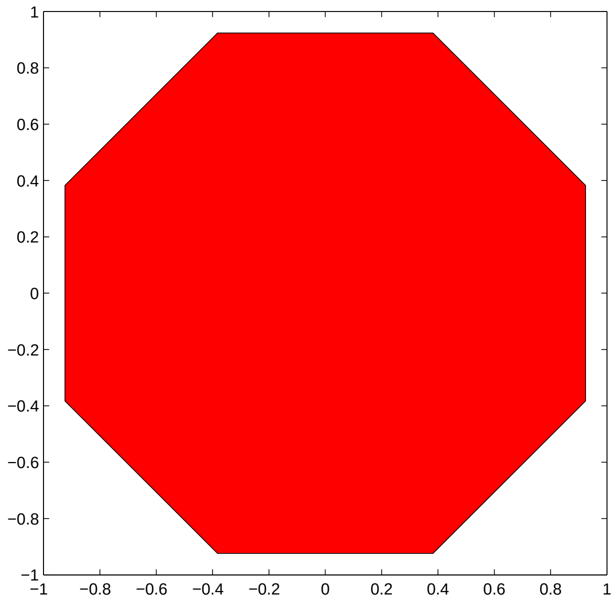
If `C` is a row vector, `fill` generates flat-shaded polygons where each element determines the color of the polygon defined by the respective column of the `X` and `Y` matrices. Each patch object's `FaceColor` property is set to `'flat'`. Each row element becomes the `CData` property value for the nth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

"Polygons and Surfaces" for related functions

fill3

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.

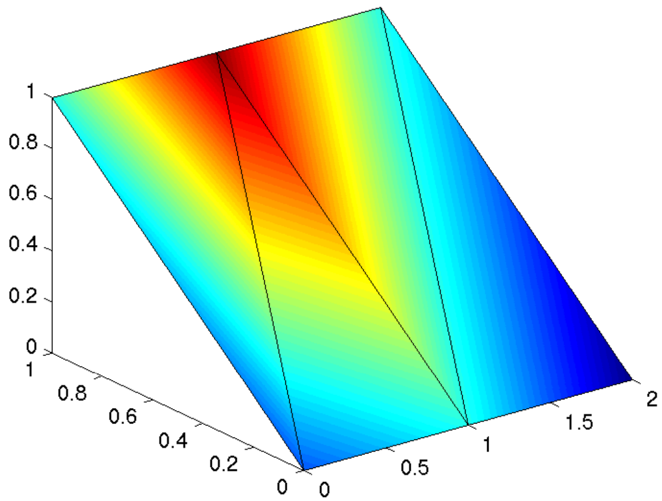
If C is a row vector, fill3 generates flat-shaded polygons and sets the FaceColor property of the patch objects to 'flat'. Each element becomes the CData property value for the respective patch object.

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`

"Polygons and Surfaces" for related functions

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. If `X` is a row or column vector, output `zf` is a column vector of `max(length(a),length(b))-1`. If `X` is a matrix, `zf` is an array of such vectors, one for each column of `X`, and similarly for multidimensional arrays.

`[y,zf] = filter(b,a,X,zi)` accepts initial conditions, `zi`, and returns the final conditions, `zf`, of the filter delays. Input `zi` is a vector of length `max(length(a),length(b))-1`, or an array with the leading dimension of size `max(length(a),length(b))-1` and with remaining dimensions matching those of `X`.

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

Example

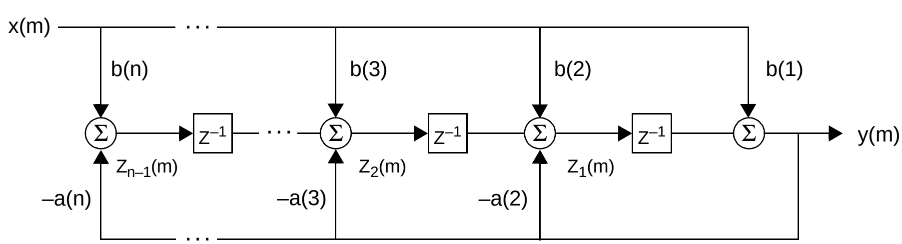
You can use `filter` to find a running average without using a `for` loop. This example finds the running average of a 16-element vector, using a window size of 5.

```
data = [1:0.2:4]';
```

```
windowSize = 5;
filter(ones(1,windowSize)/windowSize,1,data)

ans =
    0.2000
    0.4400
    0.7200
    1.0400
    1.4000
    1.6000
    1.8000
    2.0000
    2.2000
    2.4000
    2.6000
    2.8000
    3.0000
    3.2000
    3.4000
    3.6000
```

Algorithm

The `filter` function is implemented as a direct form II transposed structure,



or

```
y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb)
                 - a(2)*y(n-1) - ... - a(na+1)*y(n-na)
```

where `n-1` is the filter order, and which handles both FIR and IIR filters [1].

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

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

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`, `filtic` in the Signal Processing Toolbox

References

[1] Oppenheim, A. V. and R.W. Schafer. *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989, pp. 311-312.

Purpose 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'`	(default) Returns the central part of the correlation. In this case, `Y` is the same size as `X`.
`'valid'`	Returns only those parts of the correlation that are computed without zero-padded edges. In this case, `Y` is smaller than `X`.

Remarks Two-dimensional correlation is equivalent to two-dimensional convolution with the filter matrix rotated 180 degrees. See the Algorithm section for more information about how `filter2` performs linear filtering.

Algorithm Given a matrix `X` and a two-dimensional FIR filter `h`, `filter2` rotates your filter matrix 180 degrees to create a convolution kernel. It then calls `conv2`, the two-dimensional convolution function, to implement the filtering operation.

`filter2` uses `conv2` to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, `filter2` then extracts the central part of the convolution that is the same size as the input matrix, and returns this as the result. If the `shape` parameter specifies an alternate part of the convolution for the result, `filter2` returns the appropriate part.

See Also `conv2`, `filter`

Purpose Find indices and values of nonzero elements

Syntax

```
indices = find(X)
[i,j] = find(X)
[i,j,v] = find(X)
[...] = find(X, k)
find(X, k, 'first')
[...] = find(X, k, 'last')
```

Description

`indices = find(X)` returns the linear indices corresponding to the nonzero entries of the array `X`. If none are found, `find` returns an empty matrix. In general, `find(X)` regards `X` as `X(:)`, which is the long column vector formed by concatenating the columns of `X`.

`[i,j] = find(X)` returns the row and column indices of the nonzero entries in the matrix `X`. This syntax is especially useful when working with sparse matrices. If `X` is an N-dimensional array with N > 2, `j` contains linear indices for the dimensions of `X` other than the first.

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

`[...] = find(X, k)` or `[...] = find(X, k, 'first')` returns at most the first `k` indices corresponding to the nonzero entries of `X`. `k` must be a positive integer, but it can be of any numeric data type.

`[...] = find(X, k, 'last')` returns at most the last `k` indices corresponding to the nonzero entries of `X`.

Examples

```
X = [1 0 4 -3 0 0 0 8 6];
indices = find(X)
```

returns linear indices for the nonzero entries of `X`.

```
indices =

     1     3     4     8     9
```

You can use a logical expression to define `X`. For example,

```
find(X > 2)
```

returns linear indices corresponding to the entries of X that are greater than 2.

```
ans =

     3     8     9
```

The following commands

```
X = [3 2 0; -5 0 7; 0 0 1];
[i,j,v] = find(X)
```

return

```
i =

     1
     2
     1
     2
     3
```

a vector of row indices of the nonzero entries of X,

```
j =

     1
     1
     2
     3
     3
```

a vector of column indices of the nonzero entries of X, and

```
v =

     3
    -5
     2
     7
     1
```

a vector containing the nonzero entries of X.

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
```

For the matrix

```
M = magic(3)

M =

     8     1     6
     3     5     7
     4     9     2

find(M > 3, 4)
```

returns the indices of the first four entries of `M` that are greater than 3.

```
ans =

     1
     3
     5
```

```
6
```

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`

findfigs

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

See Also

`figflag`

"Finding and Identifying Graphics Objects" for related functions

Purpose

Locate graphics objects with specific properties

Syntax

```
h = findobj
h = findobj('PropertyName',PropertyValue,...)
h = findobj('PropertyName',PropertyValue,'-logicaloperator',
                        'PropertyName',PropertyValue,...)
h = findobj('-regexp','PropertyName','regexp',...)
h = findobj(objhandles,...)
h = findobj(objhandles,'-depth',d,...)
h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)
```

Description

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

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

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

`h = findobj('`*`PropertyName`*`',PropertyValue,'-`*`logicaloperator`*`',` *`PropertyName`*`',PropertyValue,...)` applies the logical operator to the property value matching. Possible values for *`-logicaloperator`* are:

- `-and`
- `-or`
- `-xor`
- `-not`

See the Examples section for examples of how to use these operators. See Logical Operators for an explanation of logical operators.

`h = findobj('-regexp','`*`PropertyName`*`','`*`regexp`*`',...)` matches objects using regular expressions as if the value of the property *`PropertyName`* was passed to the `regexp` function as

```
regexp(PropertyValue,'regexp')
```

If a match occurs, `findobj` returns the object's handle. See the `regexp` function for information on how MATLAB uses regular expressions.

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

`h = findobj(objhandles,'-depth',d,...)` specified the depth of the search. The depth argument `d` controls how many levels under the handles in `objhandles` are traversed. Specifying `d` as `inf` to get the default behavior of all levels. Specify `d` as `0` to get the same behavior as using the `flat` argument.

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

Remarks

`findobj` returns an error if a handle refers to a nonexistent graphics object.

`findobj` correctly matches any legal property value. For example,

```
findobj('Color','r')
```

finds all objects having a `Color` property set to `red`, `r`, or `[1 0 0]`.

When a graphics object is a descendant of more than one object identified in `objhandles`, MATLAB searches the object each time `findobj` encounters its handle. Therefore, implicit references to a graphics object can result in its handle being returned multiple times.

Examples

Find all line objects in the current axes:

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

Find all objects having a `Label` set to `'foo'` and a `String` set to `'bar'`:

```
h = findobj('Label','foo','-and','String','bar');
```

Find all objects whose `String` is not `'foo'` and is not `'bar'`:

```
h = findobj('-not','String','foo','-not','String','bar');
```

Find all objects having a `String` set to `'foo'` and a `Tag` set to `'button one'` and whose `Color` is not `'red'` or `'blue'`:

```
h = findobj('String','foo','-and','Tag','button one',...
```

```
'-and','-not',{'Color','red','-or','Color','blue'})
```

Find all objects for which you have assigned a value to the `Tag` property (that is, the value is not the empty string `''`):

```
h = findobj('-regexp','Tag','[^'']')
```

Find all children of the current figure that have their `BackgroundColor` property set to a certain shade of gray (`[.7 .7 .7]`). Note that this statement also searches the current figure for the matching property value pair.

```
h = findobj(gcf,'-depth',1,'BackgroundColor',[.7 .7 .7])
```

See Also

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

See Example — Using Logical Operators and Regular Expressions for more examples.

"Finding and Identifying Graphics Objects" for related functions

findstr

Purpose Find a string within another, longer string

Syntax

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

Description `k = findstr(str1,str2)` searches the longer of the two input strings for any occurrences of the shorter string, returning the starting index of each such occurrence in the double array `k`. If no occurrences are found, then `findstr` returns the empty array, `[]`.

The search performed by `findstr` is case sensitive. Any leading and trailing blanks in either input string are explicitly included in the comparison.

Unlike the `strfind` function, the order of the input arguments to `findstr` is not important. This can be useful if you are not certain which of the two input strings is the longer one.

Examples

```
s = 'Find the starting indices of the shorter string.';

findstr(s,'the')
ans =
     6    30

findstr('the',s)
ans =
     6    30
```

See Also `strfind`, `strmatch`, `strtok`, `strcmp`, `strncmp`, `strcmpi`, `strncmpi`, `regexp`, `regexpi`, `regexprep`

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 or in the current directory. This is a file that you create yourself in order to have MATLAB perform any final tasks just prior to terminating. For example, you might want to save the data in your workspace to a MAT-file before MATLAB exits.

`finish.m` is invoked whenever you do one of the following:

- Click the close box  in the MATLAB desktop on Windows or the UNIX equivalent
- 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 `$matlabroot/toolbox/local`. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` and add it to the path to use it.

- `finishsav.m`—Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m`—Displays a dialog allowing you to cancel quitting and saves the workspace. 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`

Purpose Return information about a FITS file

Syntax

```
S = fitsinfo(filename)
```

Description

`S = fitsinfo(filename)` returns a structure whose fields contain information about the contents of a Flexible Image Transport System (FITS) file. `filename` is a string that specifies the name of the FITS file.

The structure `S` contains the following fields.

Information Returned from a Basic FITS File

Field Name	Description	Return Type
`Contents`	List of extensions in the file in the order that they occur	Cell array of strings
`FileModDate`	File modification date	String
`Filename`	Name of the file	String
`FileSize`	Size of the file in bytes	Double
`PrimaryData`	Information about the primary data in the FITS file	Structure array

A FITS file can also include any number of optional components, called *extensions*, in FITS terminology. To provide information about these extensions, the structure `S` can also include one or more of the following structure arrays.

Additional Information Returned from FITS Extensions

Field Name	Description	Return Type
`AsciiTable`	ASCII Table extensions	Structure array
`BinaryTable`	Binary Table extensions	Structure array
`Image`	Image extensions	Structure array
`Unknown`	Nonstandard extensions	Structure array

The tables that follow show the fields of each of the structure arrays that can be returned by `fitsinfo`.

Note For all `Intercept` and `Slope` field names below, the equation used to calculate actual values is `actual_value = (Slope * array_value) + Intercept`.

Fields of the PrimaryData Structure Array

Field Name	Description	Return Type
`DataSize`	Size of the primary data in bytes	Double
`DataType`	Precision of the data	String
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values	Double
`Keywords`	Keywords, values, and comments of the header in each column	Cell array of strings
`MissingDataValue`	Value used to represent undefined data	Double
`Offset`	Number of bytes from beginning of the file to the first data value	Double
`Size`	Sizes of each dimension	Double array
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values	Double

Fields of the AsciiTable Structure Array

Field Name	Description	Return Type
`DataSize`	Size of the data in the ASCII Table in bytes	Double
`FieldFormat`	Formats in which each field is encoded, using FORTRAN-77 format codes	Cell array of strings
`FieldPos`	Starting column for each field	Double array
`FieldPrecision`	Precision in which the values in each field are stored	Cell array of strings
`FieldWidth`	Number of characters in each field	Double array
`Intercept`	Values, used with `Slope`, to calculate actual data values from the array data values	Double array
`Keywords`	Keywords, values, and comments in the ASCII table header	Cell array of strings
`MissingDataValue`	Representation of undefined data in each field	Cell array of strings
`NFields`	Number of fields in each row	Double array
`Offset`	Number of bytes from beginning of the file to the first data value	Double
`Rows`	Number of rows in the table	Double
`RowSize`	Number of characters in each row	Double
`Slope`	Values, used with `Intercept`, to calculate actual data values from the array data values	Double array

Fields of the BinaryTable Structure Array

Field Name	Description	Return Type
`DataSize`	Size of the data in the Binary Table, in bytes. Includes any data past the main part of the Binary Table.	Double
`ExtensionOffset`	Number of bytes from the beginning of the file to any data past the main part of the Binary Table	Double
`ExtensionSize`	Size of any data past the main part of the Binary Table, in bytes	Double
`FieldFormat`	Data type for each field, using FITS binary table format codes	Cell array of strings
`FieldPrecision`	Precisions in which the values in each field are stored	Cell array of strings
`FieldSize`	Number of values in each field	Double array
`Intercept`	Values, used with `Slope`, to calculate actual data values from the array data values	Double array
`Keywords`	Keywords, values, and comments in the Binary Table header	Cell array of strings
`MissingDataValue`	Representation of undefined data in each field	Cell array of double
`NFields`	Number of fields in each row	Double
`Offset`	Number of bytes from beginning of the file to the first data value	Double
`Rows`	Number of rows in the table	Double

Fields of the BinaryTable Structure Array

Field Name	Description	Return Type
`RowSize`	Number of bytes in each row	Double
`Slope`	Values, used with `Intercept`, to calculate actual data values from the array data values	Double array

Fields of the Image Structure Array

Field Name	Description	Return Type
`DataSize`	Size of the data in the Image extension in bytes	Double
`DataType`	Precision of the data	String
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values	Double
`Keywords`	Keywords, values, and comments in the Image header	Cell array of strings
`MissingDataValue`	Representation of undefined data	Double
`Offset`	Number of bytes from the beginning of the file to the first data value	Double
`Size`	Sizes of each dimension	Double array
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values	Double

Fields of the Unknown Structure Array

Field Name	Description	Return Type
DataSize	Size of the data in nonstandard extensions, in bytes	Double
DataType	Precision of the data	String
Intercept	Value, used with Slope, to calculate actual data values from the array data values	Double
Keywords	Keywords, values, and comments in the extension header	Cell array of strings
MissingDataValue	Representation of undefined data	Double
Offset	Number of bytes from beginning of the file to the first data value	Double
Size	Sizes of each dimension	Double array
Slope	Value, used with Intercept, to calculate actual data values from the array data values	Double

Example

Use `fitsinfo` to obtain information about FITS file `tst0012.fits`. In addition to its primary data, the file also contains three extensions: Binary Table, Image, and ASCII Table.

```
S = fitsinfo('tst0012.fits');
S =
       Filename: 'tst0012.fits'
    FileModDate: '27-Nov-2000 13:25:55'
       FileSize: 109440
       Contents: {'Primary'  'Binary Table'  'Image'  'ASCII'}
    PrimaryData: [1x1 struct]
    BinaryTable: [1x1 struct]
          Image: [1x1 struct]
     AsciiTable: [1x1 struct]
```

The `PrimaryData` substructure shows that the data resides in a 102-by-109 matrix of single-precision values. There are 44,472 bytes of primary data starting at an offset of 2,880 bytes from the start of the file.

```
S.PrimaryData
ans =
            DataType: 'single'
                Size: [102 109]
            DataSize: 44472
    MissingDataValue: []
           Intercept: 0
               Slope: 1
              Offset: 2880
            Keywords: {25x3 cell}
```

Examining the ASCII Table substructure, you can see that this table has 53 rows, 59 columns, and contains 8 fields per row. The last field in each row, for example, begins in the 55th column and contains a 4-digit integer.

```
S.AsciiTable
ans =
                Rows: 53
             RowSize: 59
             NFields: 8
         FieldFormat: {1x8 cell}
      FieldPrecision: {1x8 cell}
          FieldWidth: [9 6.2000 3 10.4000 20.1500 5 1 4]
            FieldPos: [1 11 18 22 33 54 54 55]
            DataSize: 3127
    MissingDataValue: {'*'  '---.--'  '*'  []  '*'  '*'  '*'  ''}
           Intercept: [0 0 -70.2000 0 0 0 0 0]
               Slope: [1 1 2.1000 1 1 1 1 1]
              Offset: 103680
            Keywords: {65x3 cell}

S.AsciiTable.FieldFormat
ans =
    'A9'   'F6.2'   'I3'   'E10.4'   'D20.15'   'A5'   'A1'   'I4'
```

The ASCII Table includes 65 keyword entries arranged in a 65-by-3 cell array.

```
key = S.AsciiTable.Keywords
```

```
key =
S.AsciiTable.Keywords
ans =
    'XTENSION'    'TABLE'         [1x48 char]
    'BITPIX'      [         8]    [1x48 char]
    'NAXIS'       [         2]    [1x48 char]
    'NAXIS1'      [        59]    [1x48 char]
        .             .                .
        .             .                .
        .             .                .
```

One of the entries in this cell array is shown here. Each row of the array contains a keyword, its value, and comment.

```
key{2,:}

ans =
BITPIX                             % Keyword

ans =
     8                             % Keyword value

ans =
 Character data 8 bits per pixel    % Keyword comment
```

See Also

`fitsread`

fitsread

Purpose Extract data from a FITS file

Syntax

```
data = fitsread(filename)
data = fitsread(filename, 'raw')
data = fitsread(filename, extname)
data = fitsread(filename, extname, index)
```

Description

`data = fitsread(filename)` reads the primary data of the Flexible Image Transport System (FITS) file specified by `filename`. Undefined data values are replaced by `NaN`. Numeric data are scaled by the slope and intercept values and are always returned in double precision.

`data = fitsread(filename, extname)` reads data from a FITS file according to the data array or extension specified in `extname`. You can specify only one `extname`. The valid choices for `extname` are shown in the following table.

Data Arrays or Extensions

extname	Description
'primary'	Read data from the primary data array.
'table'	Read data from the ASCII Table extension.
'bintable'	Read data from the Binary Table extension.
'image'	Read data from the Image extension.
'unknown'	Read data from the Unknown extension.

`data = fitsread(filename, extname, index)` is the same as the above syntax, except that if there is more than one of the specified extension type `extname` in the file, then only the one at the specified `index` is read.

`data = fitsread(filename, 'raw', ...)` reads the primary or extension data of the FITS file, but, unlike the above syntaxes, does not replace undefined data values with `NaN` and does not scale the data. The `data` returned has the same class as the data stored in the file.

Example

Read FITS file `tst0012.fits` into a 109-by-102 matrix called `data`.

```
data = fitsread('tst0012.fits');

whos data
  Name       Size           Bytes  Class

  data     109x102          88944  double array
```

Here is the beginning of the data read from the file.

```
data(1:5,1:6)
ans =
  135.2000  134.9436  134.1752  132.8980  131.1165  128.8378
  137.1568  134.9436  134.1752  132.8989  131.1167  126.3343
  135.9946  134.9437  134.1752  132.8989  131.1185  128.1711
  134.0093  134.9440  134.1749  132.8983  131.1201  126.3349
  131.5855  134.9439  134.1749  132.8989  131.1204  126.3356
```

Read only the Binary Table extension from the file.

```
data = fitsread('tst0012.fits', 'bintable')

data =
  Columns 1 through 4
    {11x1 cell}  [11x1 int16]  [11x3 uint8]  [11x2 double]
  Columns 5 through 9
   [11x3 cell]  {11x1 cell}  [11x1 int8]  {11x1 cell}  [11x3 int32]
  Columns 10 through 13
    [11x2 int32]  [11x2 single]  [11x1 double]  [11x1 uint8]
```

See Also

`fitsinfo`

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`

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`

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, generates fluid-flow data that is useful for demonstrating `slice`, `interp3`, and other functions that visualize scalar volume data.

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

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

See Also

`slice`, `interp3`

"Volume Visualization" for related functions

See Example — Slicing Fluid Flow Data for an example that uses `flow`.

Purpose Minimize a function of one variable on a fixed interval

Syntax

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
[x,fval] = fminbnd(...)
[x,fval,exitflag] = fminbnd(...)
[x,fval,exitflag,output] = fminbnd(...)
```

Description

`fminbnd` finds the minimum of a function of one variable within a fixed interval.

`x = fminbnd(fun,x1,x2)` returns a value `x` that is a local minimizer of the function that is described in `fun` in the interval `x1 <= x <= x2`. `fun` is a function handle for either an M-file function or an anonymous function.

Parameterizing Functions Called by Function Functions, in the online MATLAB documentation, explains how to provide addition parameters to the function `fun`, if necessary.

`x = fminbnd(fun,x1,x2,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminbnd` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`MaxFunEvals`	Maximum number of function evaluations allowed
`MaxIter`	Maximum number of iterations allowed
`TolX`	Termination tolerance on `x`

`[x,fval] = fminbnd(...)` returns the value of the objective function computed in `fun` at `x`.

`[x,fval,exitflag] = fminbnd(...)` returns a value `exitflag` that describes the exit condition of `fminbnd`:

1	`fminbnd` converged to a solution `x` based on `options.TolX`.
0	Maximum number of function evaluations or iterations was reached.
-1	Algorithm was terminated by the output function.
-2	Bounds are inconsistent (`ax > bx`).

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

`output.algorithm`	Algorithm used
`output.funcCount`	Number of function evaluations
`output.iterations`	Number of iterations
`output.message`	Exit message

Arguments

`fun` is the function to be minimized. `fun` accepts a scalar `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fminbnd(@myfun,x1,x2);
```

where `myfun.m` is an M-file function such as

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

or as a function handle for an anonymous function:

```
x = fminbnd(@(x) sin(x*x),x1,x2);
```

Other arguments are described in the syntax descriptions above.

Examples

`x = fminbnd(@cos,3,4)` computes π to a few decimal places and gives a message on termination.

```
[x,fval,exitflag] = ...
   fminbnd(@cos,3,4,optimset('TolX',1e-12,'Display','off'))
```

computes π to about 12 decimal places, suppresses output, returns the function value at `x`, and returns an `exitflag` of 1.

The argument fun can also be a function handle for an anonymous function. For example, to find the minimum of the function $f(x) = x^3 - 2x - 5$ on the interval (0,2), create an anonymous function f

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

Then invoke fminbnd with

```
x = fminbnd(f, 0, 2)
```

The result is

```
x =
    0.8165
```

The value of the function at the minimum is

```
y = f(x)

y =
    -6.0887
```

If fun is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function myfun defined by the following M-file function.

```
function f = myfun(x,a)
f = (x - a)^2;
```

Note that myfun has an extra parameter a, so you cannot pass it directly to fminbind. To optimize for a specific value of a, such as a = 1.5.

1 Assign the value to a.

   ```
   a = 1.5; % define parameter first
   ```

2 Call fminbnd with a one-argument anonymous function that captures that value of a and calls myfun with two arguments:

   ```
   x = fminbnd(@(x) myfun(x,a),0,1)
   ```

Algorithm

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` (`@`), anonymous functions

References

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

fminsearch

Purpose Minimize a function of several variables

Syntax

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

Description

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

`x = fminsearch(fun,x0)` starts at the point `x0` and finds a local minimum `x` of the function described in `fun`. `x0` can be a scalar, vector, or matrix. `fun` is a function handle for either an M-file function or an anonymous function.

Parameterizing Functions Called by Function Functions, in the online MATLAB documentation, explains how to provide addition parameters to the function `fun`, if necessary.

`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.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays a warning when the objective function returns a value that is `complex` or `NaN`. `'off'` (the default) displays no warning.
`MaxFunEvals`	Maximum number of function evaluations allowed
`MaxIter`	Maximum number of iterations allowed
`OutputFcn`	Specify a user-defined function that the optimization function calls at each iteration.

`TolFun`	Termination tolerance on the function value
`TolX`	Termination tolerance on `x`

`[x,fval] = fminsearch(...)` returns in `fval` the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fminsearch(...)` returns a value `exitflag` that describes the exit condition of `fminsearch`:

`1`	`fminsearch` converged to a solution `x`.
`0`	Maximum number of function evaluations or iterations was reached.
`-1`	Algorithm was terminated by the output function.

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

`output.algorithm`	Algorithm used
`output.funcCount`	Number of function evaluations
`output.iterations`	Number of iterations
`output.message`	Exit message

Arguments

`fun` is the function to be minimized. It accepts an input `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

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

where `myfun` is an M-file function such as

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

or as a function handle for an anonymous function:

```
x = fminsearch(@(x)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 anonymous function shown here defines the function and returns a function handle called `banana`:

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

Pass the function handle to `fminsearch`:

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

This produces

```
x =

    1.0000    1.0000

fval =

    8.1777e-010
```

This indicates that the minimizer was found to at least four decimal places with a value near zero.

Move the location of the minimum to the point `[a,a^2]` by adding a second parameter to the anonymous function:

```
banana = @(x,a)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`.

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = x(1)^2 + a*x(2)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminsearch`. To optimize for a specific value of `a`, such as `a = 1.5`.

1. Assign the value to `a`.

   ```
   a = 1.5; % define parameter first
   ```

2. Call `fminsearch` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

   ```
   x = fminbnd(@(x) myfun(x,a),0,1)
   ```

Algorithm

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

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

Limitations

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

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

See Also

`fminbnd`, `optimset`, `function_handle` (@), anonymous functions

References

[1] Lagarias, J.C., J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions," *SIAM Journal of Optimization*, Vol. 9 Number 1, pp. 112-147, 1998.

fopen

Purpose Open a file or obtain information about open files

Syntax

```
fid = fopen(filename)
fid = fopen(filename, mode)
[fid,message] = fopen(filename, mode, machineformat)
fids = fopen('all')
[filename, mode, machineformat] = 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, mode)` opens the file `filename` in the specified `mode`. The `mode` argument can be any of the following:

`'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 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 end of the `mode` 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.)

Note If the file is opened in update mode (`'+'`), an input command like `fread`, `fscanf`, `fgets`, or `fgetl` cannot be immediately followed by an output command like `fwrite` or `fprintf` without an intervening `fseek` or `frewind`. The reverse is also true: that is, an output command like `fwrite` or `fprintf` cannot be immediately followed by an input command like `fread`, `fscanf`, `fgets`, or `fgetl` without an intervening `fseek` or `frewind`.

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

The `'W'` and `'A'` modes 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 autoflush:

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

Examples

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

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

See Also

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

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

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

Step `s` with increments of `-0.1`

```
for s = 1.0: -0.1: 0.0,..., end
```

Successively set `e` to the unit `n`-vectors:

```
for e = eye(n),..., end
```

The line

```
for V = A,..., end
```

has the same effect as

```
for k = 1:n, V = A(:,k);..., end
```

except `k` is also set here.

See Also

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

Purpose

Control 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

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, instead 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`* and provides an example for pi, unless otherwise noted. 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 (hexadecimal representation of a binary double-precision number)	`400921fb54442d18`

Value for type	Result	Example
`long`	Scaled fixed point format, with 15 digits for double; 8 digits for single.	`3.14159265358979`
`long e`	Floating point format, with 15 digits for double; 8 digits for single.	`3.141592653589793e+00`
`long g`	Best of fixed or floating point, with 15 digits for double; 8 digits for single.	`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`	Scaled fixed point format, with 5 digits	`3.1416`
`short e`	Floating point format, with 5 digits.	`3.1416e+00`
`short g`	Best of fixed or floating point, with 5 digits.	`3.1416`

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

Examples

Example 1

Change the format to `long` by typing

```
format long
```

View the result for the value of `pi` by typing

```
pi
ans =
   3.14159265358979
```

View the current format by typing

```
get(0,'Format')
ans =
   long
```

Set the format to `short e` by typing

```
format short e
```

or use the function form of the syntax

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

Example 2

When the format is set to `short`, both `pi` and `single(pi)` display as 5-digit values:

```
format short

pi
ans =
    3.1416

single(pi)
ans =
    3.1416
```

Now set format to `long`, and `pi` displays a 15-digit value while `single(pi)` display an 8-digit value:

```
format long

pi
ans =
    3.14159265358979

single(pi)
ans =
    3.1415927
```

Example 3

Set the format to its default, and display the maximum values for integers and real numbers in MATLAB:

```
format

intmax('uint64')
ans =
   18446744073709551615

realmax
ans =
  1.7977e+308
```

Now change the format to hexadecimal, and display these same values:

```
format hex

intmax('uint64')
ans =
   ffffffffffffffff

realmax
ans =
   7fefffffffffffff
```

The hexadecimal display corresponds to the internal representation of the value. It is not the same as the hexadecimal notation in the C programming language.

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

`display`, `floor`, `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)
fplot(axes_handle,...)
[X,Y] = fplot(function,limits,...)
[...] = fplot(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
- A string with variable `x` that may be passed to `eval`, such as `'sin(x)'`, `'diric(x,10)'`, or `'[sin(x),cos(x)]'`
- A function handle for an M-file function or an anonymous function (see Function Handles and Anonymous Functions for more information)

The function *f(x)* must return a row vector for each element of vector `x`. For example, if *f(x)* returns `[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.

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

`[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)`.

`[...] = fplot(`*`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 (`[]`).

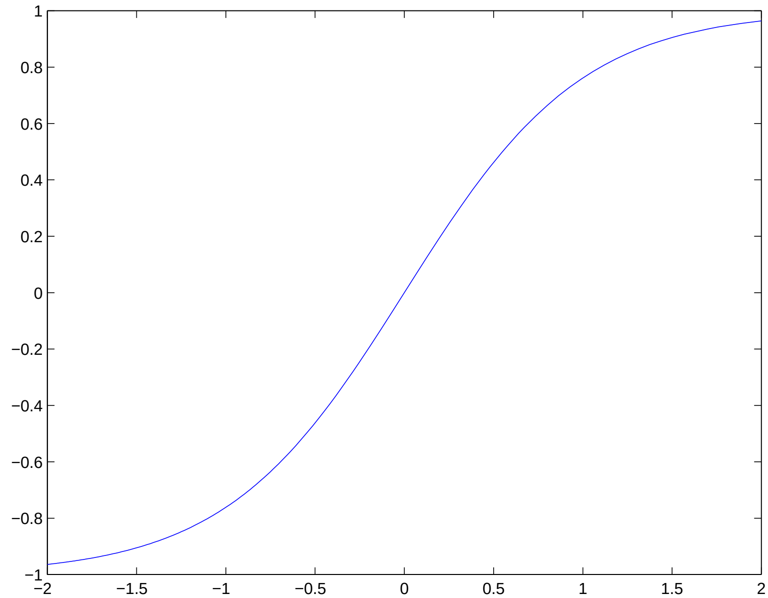
Remarks

`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:

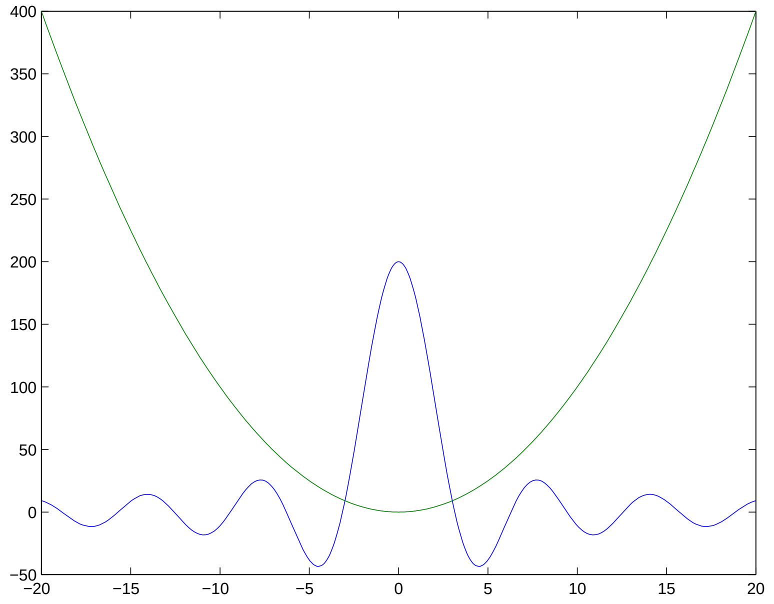
```
function Y = myfun(x)
Y(:,1) = 200*sin(x(:))./x(:);
Y(:,2) = x(:).^2;
```

Create a function handle pointing to `myfun`:

```
fh = @myfun;
```

Plot the function with the statement

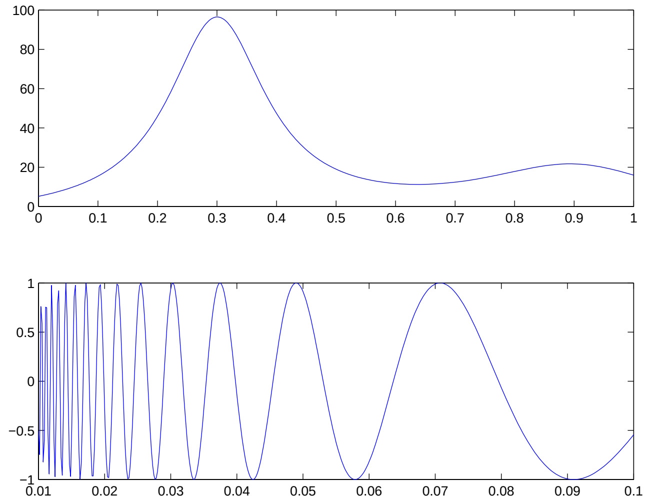
```
fplot(fh,[ 20 20])
```



Addition Examples

This example passes function handles to `fplot`, one created from a MATLAB function and the other created from an anonymous function.

```
hmp = @humps;
subplot(2,1,1);fplot(hmp,[0 1])
sn = @(x) sin(1./x);
subplot(2,1,2);fplot(sn,[.01 .1])
```



See Also

`eval`, `ezplot`, `feval`, `LineSpec`, `plot`

"Function Plots" for related functions

Plotting Mathematical Functions for more examples

Purpose

Write formatted data to file

Syntax

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

Description

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

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

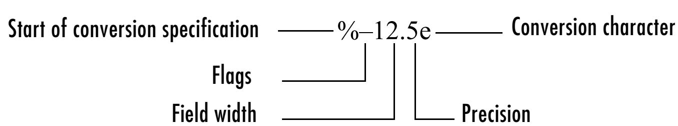
Format String

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 nonprinting characters such as newline characters and tabs.

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

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

You specify these elements in the following order:



Flags

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

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

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`
`%i`	Decimal notation (signed)
`%o`	Octal notation (unsigned)
`%s`	String of characters
`%u`	Decimal notation (unsigned)
`%x`	Hexadecimal notation (using lowercase letters `a`–`f`)
`%X`	Hexadecimal notation (using uppercase letters `A`–`F`)

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

Escape Characters

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

Character	Description
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\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 nonstandard subtype specifiers are supported for the conversion characters `%o`, `%u`, `%x`, and `%X`.

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

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

Note `fprintf` displays negative zero (`-0`) differently on some platforms, as shown in the following table.

Platform	Conversion Character %e or %E	%f	%g or %G
PC	0.000000e+000	0.000000	0
Others	-0.000000e+00	-0.000000	-0

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 radians.\n',2*pi)
```

displays a line on the screen:

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

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 integer values for use with an integer conversion specifier. For instance, to convert signed 32-bit data to hexadecimal format,

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

See Also

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.

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 true-color data, then `Map` is empty.

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

"Bit-Mapped Images" for related functions

Purpose Create and edit print frames for Simulink and Stateflow block diagrams

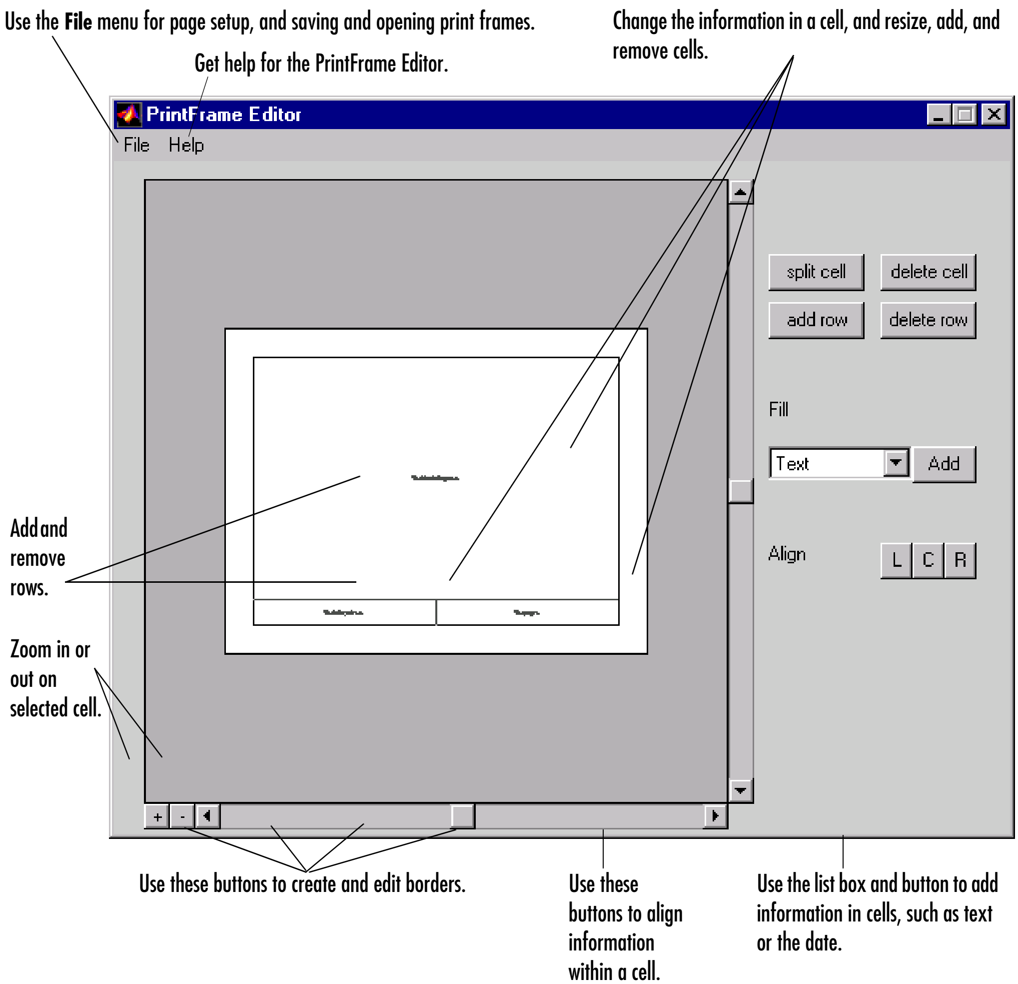
Syntax

```
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.

Purpose

Read binary data from file

Syntax

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

Description

`A = fread(fid)` reads data in binary format from the file specified by `fid` into matrix `A`. Open the file using `fopen` before calling `fread`. The `fid` argument is the integer file identifier obtained from the `fopen` operation. MATLAB reads the file from beginning to end, and then positions the file pointer at the end of the file (see `feof` for details).

`A = fread(fid, count)` reads the number of elements specified by `count`. At the end of the `fread`, MATLAB sets the file pointer to the next byte to be read. A subsequent `fread` will begin at the location of the file pointer. See "Specifying the Number of Elements", below.

Note In the following syntaxes, the `count` and `skip` arguments are optional. For example, `fread(fid, precision)` is a valid syntax.

`A = fread(fid, count, precision)` reads the file according to the data format specified by the string `precision`. This argument commonly contains a data type specifier such as `int` or `float`, followed by an integer giving the size in bits. See "Specifying Precision" and "Specifying Output Precision", below.

`A = fread(fid, count, precision, skip)` includes an optional `skip` argument that specifies the number of bytes to skip after each `precision` value is read. If `precision` specifies a bit format like `'bitN'` or `'ubitN'`, the `skip` argument is interpreted as the number of bits to skip. See "Specifying a Skip Value", below.

`A = fread(fid, count, precision, skip, machineformat)` treats the data read as having a format given by `machineformat`. You can obtain the `machineformat` argument from the output of the `fopen` function. See "Specifying Machine Format", below.

`[A, count] = fread(...)` returns the data read from the file in `A`, and the number of elements successfully read in `count`.

Specifying the Number of Elements

Valid options for `count` are

`n`	Reads `n` elements into a column vector.
`inf`	Reads to the end of the file, resulting in a column vector containing the same number of elements as are in the file. If using `inf` results in an "out of memory" error, specify a numeric count value.
`[m,n]`	Reads enough elements to fill an `m`-by-`n` matrix, filling in elements in column order, padding with zeros if the file is too small to fill the matrix. `n` can be specified as `inf`, but `m` cannot.

Specifying Precision

Any of the strings in the following table, either the MATLAB version or their C or Fortran equivalent, can be used for `precision`. If `precision` is not specified, MATLAB uses the default, which is `'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
'int64'	'integer*8'	Integer; 64 bits
'uint8'	'integer*1'	Unsigned integer; 8 bits
'uint16'	'integer*2'	Unsigned integer; 16 bits

MATLAB	C or Fortran	Interpretation
'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 ($1 \le N \le 64$)

Specifying Output Precision

By default, numeric values are returned in class `double` arrays. To return numeric values stored in classes other than `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 required to use the exact name of a MATLAB class type for destination. (See `class` for details). `fread` translates the name to the most appropriate MATLAB class type. If the source and destination formats are the same, the following shorthand notation can be used.

```
*source
```

which means

```
source=>source
```

For example, `'*uint16'` is the same as `'uint16=>uint16'`.

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.

Specifying a Skip Value

When `skip` is used, the `precision` string can contain a positive integer repetition factor of the form `'N*'`, which prefixes the source format specification, such as `'40*uchar'`.

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

When `skip` is specified, `fread` reads in, at most, a repetition factor number of values (default is 1), skips the amount of input specified by the `skip` argument, reads in another block of values, again skips input, and so on, until `count` number of values have been read. If a `skip` argument is not specified, the repetition factor is ignored. Use the repetition factor with the `skip` argument to extract data in noncontiguous fields from fixed-length records.

Specifying Machine Format

`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

Remarks

If the input stream is bytes and `fread` reaches the end of file (see `feof`) in the middle of reading the number of bytes required for an element, the partial result is ignored. However, if the input stream is bits, then the partial result is returned as the last value. If an error occurs before reaching the end of file, only full elements read up to that point are used.

Examples

Example 1

The file `alphabet.txt` contains the 26 letters of the English alphabet, all capitalized. Open the file for read access with `fopen`, and read the first five elements into output `c`. Because a precision has not been specified, MATLAB uses the default precision of `uchar`, and the output is numeric:

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, 5)
c =
    65
    66
    67
    68
    69
fclose(fid);
```

This time, specify that you want each element read as an unsigned 8-bit integer and output as a character. (Using a precision of `'char=>char'` or `'*char'` will produce the same result):

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, 5, 'uint8=>char')
c =
   A
   B
   C
   D
   E
fclose(fid);
```

When you leave out the optional `count` argument, MATLAB reads the file to the end, `A` through `Z`:

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, '*char');
```

```
fclose(fid);

sprintf(c)
ans =
   ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The `fopen` function positions the file pointer at the start of the file. So the first `fread` in this example reads the first five elements in the file, and then repositions the file pointer at the beginninig of the next element. For this reason, the next `fread` picks up where the previous `fread` left off, at the character `F`.

```
fid = fopen('alphabet.txt', 'r');
c1 = fread(fid, 5, '*char');
c2 = fread(fid, 8, '*char');
c3 = fread(fid, 5, '*char');
fclose(fid);

sprintf('%c', c1, ' * ', c2, ' * ', c3)
ans =
   ABCDE * FGHIJKLM * NOPQR
```

Skip two elements between each read by specifying a `skip` argument of `2`:

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, 'char', 2);    % Skip 2 bytes per read
fclose(fid);

sprintf('%c', c)
ans =
   ADGJMPSVY
```

Example 2

This command displays the complete M-file containing this `fread` help entry:

```
type fread.m
```

To simulate this command using `fread`, enter the following:

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

In the example, the fread command assumes the default size, 'inf', and precision '*uchar' (the same as 'char=>char'). fread reads the entire file. To display the result as readable text, the column vector is transposed to a row vector.

Example 3

As another example,

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

reads in 120 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 is done before the command is finished. If the last block read is not a full block, then fread does not finish with a skip.

See fopen for information about reading big and little-endian files.

Example 4

Invoke the fopen function with just an fid input argument to obtain the machine format for the file. You can see that this file was written in IEEE floating point with little-endian byte ordering ('ieee-le') format:

```
fid = fopen('A1.dat', 'r');

[fname, mode, mformat] = fopen(fid);
mformat
mformat =
   ieee-le
```

Use the MATLAB format function (not related to the machine format type) to have MATLAB display output using hexadecimal:

```
format hex
```

Now use the machineformat input with fread to read the data from the file using the same format:

```
x = fread(fid, 6, 'uint64', 'ieee-le')
x =
   4260800000002000
   0000000000000000
   4282000000180000
   0000000000000000
   42ca5e0000258000
   42f0000464d45200
fclose(fid);
```

Change the machine format to IEEE floating point with big-endian byte ordering ('ieee-be') and verify that you get different results:

```
fid = fopen('A1.dat', 'r');
x = fread(fid, 6, 'uint64', 'ieee-be')
x =
   4370000008400000
   0000000000000000
   4308000200100000
   0000000000000000
   4352c0002f0d0000
   43c022a6a3000000
fclose(fid);
```

See Also

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

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 might 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

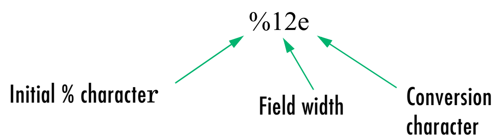
`n`	Read `n` elements into a column vector.
`inf`	Read to the end of the file, resulting in a column vector containing the same number of elements as are in the file.
`[m,n]`	Read enough elements to fill an `m`-by-`n` matrix, filling the matrix in column order. `n` can be specified as `inf`, but `m`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

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 `%*d`, then the value that matches `d` is ignored and is not stored.
A digit string	Maximum field width. For example, `%10d`.
A letter	The size of the receiving object, for example, `h` for short, as in `%hd` for a short integer, or `l` for long, as in `%ld` for a long integer, or `%lg` for a double floating-point number.

Valid conversion characters are

`%c`	Sequence of characters; number specified by field width
`%d`	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 can 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 causes 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`

fseek

Purpose

Set file position indicator

Syntax

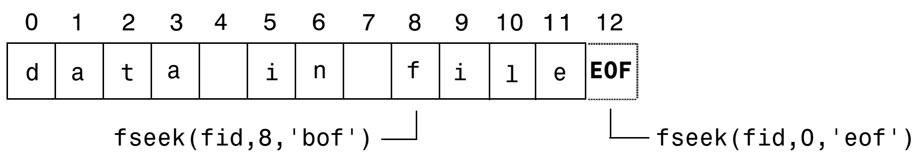
```
status = fseek(fid,offset,origin)
```

Description

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

For a file having `n` bytes, the bytes are numbered from `0` to `n-1`. The position immediately following the last byte is the end-of-file, or `eof`, position. You would seek to the `eof` position if you wanted to add data to the end of a file.

This figure represents a file having 12 bytes, numbered 0 through 11. The first command shown seeks to the ninth byte of data in the file. The second command seeks just past the end of the file data, to the `eof` position.



`fseek` does not seek beyond the end of file `eof` position. If you attempt to seek beyond `eof`, MATLAB returns an error status.

Arguments

`fid` — An integer file identifier obtained from `fopen`

`offset` — A value that is interpreted as follows,

- `offset > 0` — Move position indicator `offset` bytes toward the end of the file.
- `offset = 0` — Do not change position.
- `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.

Examples

This example opens the file `test1.dat`, seeks to the 20th byte, reads fifty 32-bit unsigned integers into variable `A`, and closes the file. It then opens a second file, `test2.dat`, seeks to the end-of-file position, appends the data in `A` to the end of this file, and closes the file.

```
fid = fopen('test1.dat', 'r');
fseek(fid, 19, 'bof');
A = fread(fid, 50, 'uint32');
fclose(fid);

fid = fopen('test2.dat', 'r+');
fseek(fid, 0, 'eof');
fwrite(fid, A, 'uint32');
fclose(fid);
```

See Also

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

ftell

Purpose

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

Connect to FTP server, creating an FTP object

Syntax

```
f = ftp('host','username','password')
```

Description

`f = ftp('host','username','password')` connects to the FTP server, `host`, creating the FTP object, `f`. If a username and password are not required for an anonymous connection, only use the `host` argument. Specify an alternate port by separating it from `host` using a colon (:). After running `ftp`, perform file operation functions on the FTP object, `f`, using methods such as `cd` and others listed under "See Also." When you're finished using the server, run `close (ftp)` to close the connection.

The `ftp` function is based on code from the Apache Jakarta Project.

Examples

Connect Without Username

Connect to `ftp.mathworks.com`, which does not require a username or password. Assign the resulting FTP object to `tmw`. You can access this FTP site to experiment with the FTP functions.

```
tmw=ftp('ftp.mathworks.com')
```

MATLAB returns

```
tmw =
 FTP Object
  host: ftp.mathworks.com
  user: anonymous
   dir: /
  mode: binary
```

Connect To Specified Port

To connect to port 34, type

```
tmw=ftp('ftp.mathworks.com:34')
```

Connect With Username

Connect to `ftp.testsite.com` and assign the resulting FTP object to `test`.

```
test=ftp('ftp.testsite.com','myname','mypassword')
```

MATLAB returns

```
test =
 FTP Object
  host: ftp.testsite.com
  user: myname
   dir: /
  mode: binary
  myname@ftp.testsite.com
  /
```

See Also

ascii (ftp), binary (ftp), cd (ftp), delete (ftp), dir (ftp), close (ftp), mget (ftp), mkdir (ftp), mput (ftp), rename (ftp), rmdir (ftp)

Purpose Convert sparse matrix to full matrix

Syntax

```
A = full(S)
```

Description `A = full(S)` converts a sparse matrix `S` to full storage organization. If `S` is a full matrix, it is left unchanged. If `A` is full, `issparse(A)` is `0`.

Remarks Let `X` be an `m`-by-`n` matrix with `nz = nnz(X)` nonzero entries. Then `full(X)` requires space to store `m*n` real numbers while `sparse(X)` requires space to store `nz` real numbers and `(nz+n)` integers.

On most computers, a real number requires twice as much storage as an integer. On such computers, `sparse(X)` requires less storage than `full(X)` if the density, `nnz/prod(size(X))`, is less than one third. Operations on sparse matrices, however, require more execution time per element than those on full matrices, so density should be considerably less than two-thirds before sparse storage is used.

Examples Here is an example of a sparse matrix with a density of about two-thirds. `sparse(S)` and `full(S)` require about the same number of bytes of storage.

```
S = sparse(+(rand(200,200) < 2/3));
A = full(S);
whos
Name     Size      Bytes   Class
  A    200X200   320000   double array
  S    200X200   318432   double array (sparse)
```

See Also `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')

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

See Also `fileparts`, `genpath`

Purpose Construct 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 does not operate on nonscalar function handles. Passing a nonscalar function handle to `func2str` results in an error.

Examples

Example 1

Convert a `sin` function handle to a string:

```
fhandle = @sin;

func2str(fhandle)
ans =
    sin
```

Example 2

The `catcherr` function shown here accepts function handle and data arguments and attempts to evaluate the function through its handle. If the function fails to execute, `catcherr` uses `sprintf` to display an error message giving the name of the failing function. The function name must be a string for `sprintf` to display it. The code derives the function name from the function handle using `func2str`:

```
function catcherr(func, data)
try
    ans = func(data);
    disp('Answer is:');
    ans
catch
    disp(sprintf('Error executing function ''%s''\n', ...
        func2str(func)))
end
```

The first call to catcherr passes a handle to the round function and a valid data argument. This call succeeds and returns the expected answer. The second call passes the same function handle and an improper data type (a MATLAB structure). This time, round fails, causing catcherr to display an error message that includes the failing function name:

```
catcherr(@round, 5.432)
ans =
Answer is 5

xstruct.value = 5.432;
catcherr(@round, xstruct)
Error executing function "round"
```

See Also

function_handle, str2func, functions

Purpose Function M-files

Description You add new functions to the MATLAB vocabulary by expressing them in terms of existing functions. The existing commands and functions that compose the new function reside in a text file called an *M-file*.

M-files can be either *scripts* or *functions*. Scripts are simply files containing a sequence of MATLAB statements. Functions make use of their own local variables and accept input arguments.

The name of an M-file begins with an alphabetic character and has a filename extension of `.m`. The M-file name, less its extension, is what MATLAB searches for when you try to use the script or function.

A line at the top of a function M-file contains the syntax definition. The name of a function, as defined in the first line of the M-file, should be the same as the name of the file without the `.m` extension. 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. The section "Determining Which Function Is Called" in the MATLAB Programming documentation explains how MATLAB interprets variable and function names that you enter, and also covers the precedence used in function dispatching.

When you call an M-file function from the command line or from within another M-file, MATLAB parses the function and stores it in memory. The parsed function remains in memory until cleared with the `clear` command or you `quit` MATLAB. The `pcode` command performs the parsing step and stores the result on the disk as a P-file to be loaded later.

See Also

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

Purpose

MATLAB data type that is a handle to a function

Syntax

```
handle = @functionname
handle = @(arglist)anonymous_function
```

Description

`handle = @functionname` returns a handle to the specified MATLAB function.

A function handle is a MATLAB value that provides a means of calling a function indirectly. You can pass function handles in calls to other functions (often called *function functions*). You can also store function handles in data structures for later use (for example, as Handle Graphics callbacks). A function handle is one of the standard MATLAB data types.

At the time you create a function handle, the function you specify must be on the MATLAB path and in the current scope. This condition does not apply when you evaluate the function handle. You can, for example, execute a subfunction from a separate (out-of-scope) M-file using a function handle as long as the handle was created within the subfunction's M-file (in-scope).

`handle = @(arglist)anonymous_function` constructs an anonymous function and returns a `handle` to that function. The body of the function, to the right of the parentheses, is a single MATLAB statement or command. `arglist` is a comma-separated list of input arguments. Execute the function by calling it by means of the function handle, `handle`.

Remarks

The function handle is a standard MATLAB data type. As such, you can manipulate and operate on function handles in the same manner as on other MATLAB data types. This includes using function handles in structures and cell arrays:

```
S.a = @sin;  S.b = @cos;  S.c = @tan;
C = {@sin, @cos, @tan};
```

However, standard matrices or arrays of function handles are not supported:

```
A = [@sin, @cos, @tan];        % This is not supported
```

For nonoverloaded functions, subfunctions, and private functions, a function handle references just the one function specified in the `@functionname` syntax. When you evaluate an overloaded function by means of its handle, the arguments the handle is evaluated with determine the actual function that MATLAB dispatches to.

Examples

Example 1 — Constructing a Handle to a Named Function

The following example creates a function handle for the `humps` function and assigns it to the variable `fhandle`.

```
fhandle = @humps;
```

Pass the handle to another function in the same way you would pass any argument. This example passes the function handle just created to `fminbnd`, which then minimizes over the interval `[0.3, 1]`.

```
x = fminbnd(fhandle, 0.3, 1)
x =
    0.6370
```

The `fminbnd` function evaluates the `@humps` function handle. A small portion of the `fminbnd` M-file is shown below. In line 1, the `funfcn` input parameter receives the function handle `@humps` that was passed in. The statement, in line 113, evaluates the handle.

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

Example 2 — Constructing a Handle to an Anonymous Function

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `sqr`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the sqr function defined above, type

```
a = sqr(5)
a =
   25
```

Because sqr is a function handle, you can pass it in an argument list to other functions. The code shown here passes the sqr anonymous function to the MATLAB quad function to compute its integral from zero to one:

```
quad(sqr, 0, 1)
ans =
   0.3333
```

See Also

str2func, func2str, functions

functions

Purpose Return information about a function handle

Syntax

```
S = functions(funhandle)
```

Description

`S = functions(funhandle)` returns, in MATLAB structure `S`, the function name, type, filename, and other information for the function handle stored in the variable `funhandle`.

Caution The `functions` function is provided for querying and debugging purposes. Because its behavior may change in subsequent releases, you should not rely upon it for programming purposes.

Note `functions` does not operate on nonscalar function handles. Passing a nonscalar function handle to `functions` results in an error.

Other Stuff

The fields of the return structure are listed in the following table.

Field Name	Field Description
`function`	Function name.
`type`	Function type. See the table in "Function Type" on page 2-919.
`file`	The file to be executed when the function handle is evaluated with a nonoverloaded data type.

For handles to functions that overload one of the standard MATLAB data types, like `double` or `char`, the structure returned by `functions` contains an additional field named `methods`. The `methods` field is a substructure containing one field name for each MATLAB class that overloads the function. The value of each field is the path and name of the file that defines the method.

For example, to obtain information on a function handle for the `floor` function, use

```
f = functions(@floor)
f =
    function: 'floor'
        type: 'simple'
        file: 'matlabroot\toolbox\matlab\elfun\floor.m
```

Individual fields of the structure are accessible using the dot selection notation:

```
f.type
ans =
   simple
```

Fields Returned by the Functions Command

The `functions` function returns a MATLAB structure with the fields `function`, `type`, `file`, and for some overloaded functions, `methods`. This section describes each of those fields.

Function Name. The `function` field is a character array that holds the name of the function corresponding to the function handle.

Function Type. The `type` field is a one-word character array indicating what type of function the handle represents.

The contents of the next two fields, `file` and `methods`, depend upon the function `type`.

Function File. The `file` field is a character array that specifies the path and name of the file that implements the default function. The *default* function is the one function implementation that is not specialized to operate on any particular data type. Unless the arguments in the function call specify a class that has a specialized version of the function defined, it is the default function that gets called.

Function Methods. The `methods` field exists only for functions of type `overloaded`. This field is a separate MATLAB structure that identifies all M-files that overload the function for any of the standard MATLAB data types.

The structure contains one field for each M-file that overloads the function. The field names are the MATLAB classes that overload the function. Each field value is a character array holding the path and name of the source file that defines the method.

Remarks

For handles to functions that overload one of the MATLAB classes, like `double` or `char`, the structure returned by `functions` contains an additional field named `methods`. The `methods` field is a substructure containing one field name for each MATLAB class that overloads the function. The value of each field is the path and name of the file that defines the method.

Examples

To obtain information on a function handle for the `deblank` function,

```
f = functions(@poly)
f =
    function: 'poly'
        type: 'simple'
        file: 'matlabroot\toolbox\matlab\polyfun\poly.m'
```

See Also

`function_handle`

Purpose

Evaluate general matrix function

Syntax

```
F = funm(A,fun)
F = funm(A, fun, options)
[F, exitflag] = funm(...)
[F, exitflag, output] = funm(...)
```

Description

`F = funm(A,fun)` evaluates the user-defined function `fun` at the square matrix argument `A`. f = `fun(x, k)` must accept a vector `x` and an integer `k`, and return a vector `f` of the same size of `x`, where `f(i)` is the `k`th derivative of the function `fun` evaluated at `x(i)`. The function represented by fun must have a Taylor series with an infinite radius of convergence, except for `fun = @log`, which is treated as a special case.

You can also use `funm` to evaluate the special functions listed in the following table at the matrix `A`.

Function	Syntax for Evaluating Function at Matrix A
`exp`	`funm(A, @exp)`
`log`	`funm(A, @log)`
`sin`	`funm(A, @sin)`
`cos`	`funm(A, @cos)`
`sinh`	`funm(A, @sinh)`
`cosh`	`funm(A, @cosh)`

For matrix square roots, use `sqrtm(A)` instead. For matrix exponentials, which of `expm(A)` or `funm(A, @exp)` is the more accurate depends on the matrix `A`.

Parameterizing Functions Called by Function Functions, in the online MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`F = funm(A, fun, options)` sets the algorithm's parameters to the values in the structure `options`. The following table lists the fields of `options`.

Field	Description	Values
`options.TolBlk`	Level of display	`'off'` (default), `'on'`, `'verbose'`
`options.TolTay`	Tolerance for blocking Schur form	Positive scalar. The default is `eps`.
`options.MaxTerms`	Maximum number of Tayor series terms	Positive integer. The default is `250`.
`options.MaxSqrt`	When computing a logarithm, maximum number of square roots computed in inverse scaling and squaring method.	Positive integer. The default is `100`.
`options.Ord`	Specifies the ordering of the Schur form `T`.	A vector of length `length(A).options.Ord(i)` is the index of the block into which `T(i,i)` is placed. The default is `[]`.

`[F, exitflag] = funm(...)` returns a scalar `exitflag` that describes the exit condition of `funm`. `exitflag` can have the following values:

- `0` — The algorithm was successful.
- `1` — One or more Taylor series evaluations did not converge. However, the computed value of `F` might still be accurate.

[F, exitflag, output] = funm(...) returns a structure output with the following fields:

Field	Description
output.terms	Vector for which output.terms(i) is the number of Taylor series terms used when evaluating the ith block, or, in the case of the logarithm, the number of square roots.
output.ind	Cell array for which the (i,j) block of the reordered Schur factor T is T(output.ind{i}, output.ind{j}).
output.ord	Ordering of the Schur form, as passed to ordschur
output.T	Reordered Schur form

If the Schur form is diagonal then
output = struct('terms',ones(n,1),'ind',{1:n}).

Examples

Example 1. The following command computes the matrix sine of the 3-by-3 magic matrix.

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

F =

   -0.3850    1.0191    0.0162
    0.6179    0.2168   -0.1844
    0.4173   -0.5856    0.8185
```

Example 2. The statements

```
S = funm(X,@sin);
C = funm(X,@cos);
```

produce the same results to within roundoff error as

```
E = expm(i*X);
C = real(E);
S = imag(E);
```

In either case, the results satisfy `S*S+C*C = I`, where `I = eye(size(X))`.

Example 3.

To compute the function `exp(x) + cos(x)` at `A` with one call to `funm`, use

```
F = funm(A,@fun_expcos)
```

where `fun_expcos` is the following M-file function.

```
function f = fun_expcos(x, k)
% Return kth derivative of exp + cos at X.
        g = mod(ceil(k/2),2);
        if mod(k,2)
           f = exp(x) + sin(x)*(-1)^g;
        else
           f = exp(x) + cos(x)*(-1)^g;
        end
```

Algorithm

The algorithm `funm` uses is described in [1].

See Also

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

References

[1] Davies, P. I. and N. J. Higham, "A Schur-Parlett algorithm for computing matrix functions," *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Third Edition, Johns Hopkins University Press, 1996, p. 384.

[3] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later" *SIAM Review 20*, Vol. 45, Number 1, pp. 1-47, 2003.

Purpose Write 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`

Purpose Find zero of a function of one variable

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)
[x,fval] = fzero(...)
[x,fval,exitflag] = fzero(...)
[x,fval,exitflag,output] = fzero(...)
```

Description

`x = fzero(fun,x0)` tries to find a zero of `fun` near `x0`, if `x0` is a scalar. `fun` is a function handle for either an M-file function or an anonymous function.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.

Parameterizing Functions Called by Function Functions, in the online MATLAB documentation, explains how to provide addition parameters to the function `fun`, if necessary.

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,fval] = fzero(...)` returns the value of the objective function `fun` at the solution `x`.

[x,fval,exitflag] = fzero(...) returns a value exitflag that describes the exit condition of fzero:

1	Function converged to a solution x.
-1	Algorithm was terminated by the output function.
-3	NaN or Inf function value was encountered during search for an interval containing a sign change.
-4	Complex function value was encountered during search for an interval containing a sign change.
-5	fzero might have converged to a singular point.

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

output.algorithm	Algorithm used
output.funcCount	Number of function evaluations
output.intervaliterations	Number of iterations taken to find an interval
output.iterations	Number of zero-finding iterations
output.message	Exit message

Note For the purposes of this command, zeros are considered to be points where the function actually crosses, not just touches, the *x*-axis.

Arguments

fun is the function whose zero is to be computed. It accepts a vector x and returns a scalar f, the objective function evaluated at x. The function fun can be specified as a function handle for an M-file function

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

where myfun is an M-file function such as

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

or as a function handle for an anonymous function:

```
x = fzero(@(x)sin(x*x),x0);
```

Other arguments are described in the syntax descriptions above.

Examples

Example 1. Calculate π by finding the zero of the sine function near 3.

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

Example 2. To find the zero of cosine between 1 and 2

```
x = fzero(@cos,[1 2])
x =
     1.5708
```

Note that `cos(1)` and `cos(2)` differ in sign.

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

write an anonymous function `f`:

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

Then find the zero near 2:

```
z = fzero(f,2)
z =
    2.0946
```

Because this function is a polynomial, the statement `roots([1 0 -2 -5])` finds the same real zero, and a complex conjugate pair of zeros.

```
    2.0946
   -1.0473 + 1.1359i
   -1.0473 - 1.1359i
```

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = cos(a*x);
```

2gallery

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	leslie
lesp	lotkin	minij	moler
neumann	orthog	parter	pei
poisson	prolate	randcolu	randcorr
rando	randhess	randsvd	redheff
riemann	ris	rosser	smoke

Test Matrices (Continued)			
`toeppd`	`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 of the Chebyshev spectral differentiation matrix is ill-conditioned.

chebvand—Vandermonde-like matrix for the Chebyshev polynomials

`C = gallery('chebvand',p)` produces the (primal) Chebyshev Vandermonde matrix based on the vector of points `p`, which define where the Chebyshev polynomial is calculated.

`C = gallery('chebvand',m,p)` where `m` is scalar, produces a rectangular version of the above, with `m` rows.

If `p` is a vector, then $C(i,j) = T_{i-1}(p(j))$ where T_{i-1} is the Chebyshev polynomial of degree i-1. If `p` is a scalar, then `p` equally spaced points on the interval `[0,1]` are used to calculate `C`.

chow—Singular Toeplitz lower Hessenberg matrix

`A = gallery('chow',n,alpha,delta)` returns `A` such that `A = H(alpha) + delta*eye(n)`, where $H_{i,j}(\alpha) = \alpha^{(i-j+1)}$ and argument `n` is the order of the Chow matrix. Default value for scalars `alpha` and `delta` are `1` and `0`, respectively.

`H(alpha)` has `p = floor(n/2)` eigenvalues that are equal to zero. The rest of the eigenvalues are equal to `4*alpha*cos(k*pi/(n+2))^2, k=1:n-p`.

circul—Circulant matrix

`C = gallery('circul',v)` returns the circulant matrix whose first row is the vector `v`.

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. It is a special Toeplitz matrix in which the diagonals “wrap around.”

If `v` is a scalar, then `C = gallery('circul',1:v)`.

The eigensystem of `C` (`n`-by-`n`) is known explicitly: If `t` is an `n`th root of unity, then the inner product of `v` and $w = [1\ t\ t^2 \ldots t^{(n-1)}]$ is an eigenvalue of `C` and `w(n:-1:1)` is an eigenvector.

clement—Tridiagonal matrix with zero diagonal entries

`A = gallery('clement',n,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 `gallery('compar',A)`.

condex—Counter-examples to matrix condition number estimators

`A = gallery('condex',n,k,theta)` returns a "counter-example" matrix to a condition estimator. It has order `n` and scalar parameter `theta` (default 100).

The matrix, its natural size, and the estimator to which it applies are specified by `k`:

`k = 1`	4-by-4	LINPACK
`k = 2`	3-by-3	LINPACK
`k = 3`	arbitrary	LINPACK (`rcond`) (independent of `theta`)
`k = 4`	`n >= 4`	LAPACK (RCOND) (default). It is the inverse of this matrix that is a counter-example.

If `n` is not equal to the natural size of the matrix, then the matrix is padded out with an identity matrix to order `n`.

cycol—Matrix whose columns repeat cyclically

`A = gallery('cycol',[m n],k)` returns an `m`-by-`n` matrix with cyclically repeating columns, where one "cycle" consists of `randn(m,k)`. Thus, the rank of matrix `A` cannot exceed `k`, and `k` must be a scalar.

Argument `k` defaults to `round(n/4)`, and need not evenly divide `n`.

`A = gallery('cycol',n,k)`, where `n` is a scalar, is the same as `gallery('cycol',[n n],k)`.

dorr—Diagonally dominant, ill-conditioned, tridiagonal matrix

`[c,d,e] = gallery('dorr',n,theta)` returns the vectors defining an `n`-by-`n`, row diagonally dominant, tridiagonal matrix that is ill-conditioned for small nonnegative values of `theta`. The default value of `theta` is `0.01`. The Dorr matrix itself is the same as `gallery('tridiag',c,d,e)`.

`A = gallery('dorr',n,theta)` returns the matrix itself, rather than the defining vectors.

dramadah—Matrix of zeros and ones whose inverse has large integer entries

`A = gallery('dramadah',n,k)` returns an `n`-by-`n` matrix of `0`'s and `1`'s for which `mu(A) = norm(inv(A),'fro')` is relatively large, although not necessarily maximal. An anti-Hadamard matrix `A` is a matrix with elements `0` or `1` for which `mu(A)` is maximal.

`n` and `k` must both be scalars. Argument `k` determines the character of the output matrix:

`k = 1`	Default. `A` is Toeplitz, with `abs(det(A)) = 1`, and `mu(A) > c(1.75)^n`, where `c` is a constant. The inverse of `A` has integer entries.
`k = 2`	`A` is upper triangular and Toeplitz. The inverse of `A` has integer entries.
`k = 3`	`A` has maximal determinant among lower Hessenberg (0,1) matrices. `det(A)` = the `n`th Fibonacci number. `A` is Toeplitz. The eigenvalues have an interesting distribution in the complex plane.

fiedler—Symmetric matrix

`A = gallery('fiedler',c)`, where `c` is a length `n` vector, returns the `n`-by-`n` symmetric matrix with elements `abs(n(i)-n(j))`. For scalar `c`, `A = gallery('fiedler',1:c)`.

Matrix `A` has a dominant positive eigenvalue and all the other eigenvalues are negative.

Explicit formulas for `inv(A)` and `det(A)` are given in [Todd, J., *Basic Numerical Mathematics*, Vol. 2: Numerical Algebra, Birkhauser, Basel, and Academic Press, New York, 1977, p. 159] and attributed to Fiedler. These indicate that `inv(A)` is tridiagonal except for nonzero `(1,n)` and `(n,1)` elements.

forsythe—Perturbed Jordan block

`A = gallery('forsythe',n,alpha,lambda)` returns the `n`-by-`n` matrix equal to the Jordan block with eigenvalue `lambda`, excepting that `A(n,1) = alpha`. The default values of scalars `alpha` and `lambda` are `sqrt(eps)` and `0`, respectively.

The characteristic polynomial of `A` is given by:

```
det(A-t*I) = (lambda-t)^N - alpha*(-1)^n.
```

frank—Matrix with ill-conditioned eigenvalues

`F = gallery('frank',n,k)` returns the Frank matrix of order `n`. It is upper Hessenberg with determinant `1`. If `k = 1`, the elements are reflected about the anti-diagonal `(1,n)`—`(n,1)`. The eigenvalues of `F` may be obtained in terms of the zeros of the Hermite polynomials. They are positive and occur in reciprocal pairs; thus if `n` is odd, `1` is an eigenvalue. `F` has `floor(n/2)` ill-conditioned eigenvalues—the smaller ones.

gearmat—Gear matrix

`A = gallery('gearmat',n,i,j)` returns the `n`-by-`n` matrix with ones on the sub- and super-diagonals, `sign(i)` in the `(1,abs(i))` position, `sign(j)` in the (n,n+1-abs(j)) position, and zeros everywhere else. Arguments i and j default to n and -n, respectively.

Matrix A is singular, can have double and triple eigenvalues, and can be defective.

All eigenvalues are of the form 2*cos(a) and the eigenvectors are of the form [sin(w+a), sin(w+2*a), ..., sin(w+n*a)], where a and w are given in Gear, C. W., "A Simple Set of Test Matrices for Eigenvalue Programs", *Math. Comp.*, Vol. 23 (1969), pp. 119-125.

grcar—Toeplitz matrix with sensitive eigenvalues

A = gallery('grcar',n,k) returns an n-by-n Toeplitz matrix with -1s on the subdiagonal, 1s on the diagonal, and k superdiagonals of 1s. The default is k = 3. The eigenvalues are sensitive.

hanowa—Matrix whose eigenvalues lie on a vertical line in the complex plane

A = gallery('hanowa',n,d) returns an n-by-n block 2-by-2 matrix of the form:

```
[d*eye(m)   -diag(1:m)
diag(1:m)   d*eye(m)]
```

Argument n is an even integer n=2*m. Matrix A has complex eigenvalues of the form d ± k*i, for 1 <= k <= m. The default value of d is -1.

house—Householder matrix

[v,beta,s] = gallery('house',x,k) takes x, an n-element column vector, and returns V and beta such that H*x = s*e1. In this expression, e1 is the first column of eye(n), abs(s) = norm(x), and H = eye(n) - beta*V*V' is a Householder matrix.

k determines the sign of s:

k = 0	sign(s) = -sign(x(1)) (default)
k = 1	sign(s) = sign(x(1))
k = 2	sign(s) = 1 (x must be real)

If x is complex, then sign(x) = x./abs(x) when x is nonzero.

If x = 0, or if x = alpha*e1 (alpha >= 0) and either k = 1 or k = 2, then V = 0, beta = 1, and s = x(1). In this case, H is the identity matrix, which is not strictly a Householder matrix.

invhess—Inverse of an upper Hessenberg matrix

A = gallery('invhess',x,y), where x is a length n vector and y is a length n-1 vector, returns the matrix whose lower triangle agrees with that of ones(n,1)*x' and whose strict upper triangle agrees with that of [1 y]*ones(1,n).

The matrix is nonsingular if x(1) ~= 0 and x(i+1) ~= y(i) for all i, and its inverse is an upper Hessenberg matrix. Argument y defaults to -x(1:n-1).

If x is a scalar, invhess(x) is the same as invhess(1:x).

invol—Involutory matrix

A = gallery('invol',n) returns an n-by-n involutory (A*A = eye(n)) and ill-conditioned matrix. It is a diagonally scaled version of hilb(n).

B = (eye(n)-A)/2 and B = (eye(n)+A)/2 are idempotent (B*B = B).

ipjfact—Hankel matrix with factorial elements

[A,d] = gallery('ipjfact',n,k) returns A, an n-by-n Hankel matrix, and d, the determinant of A, which is known explicitly. If k = 0 (the default), then the elements of A are A(i,j) = (i+j)! If k = 1, then the elements of A are A(i,j) = 1/(i+j).

Note that the inverse of A is also known explicitly.

jordbloc—Jordan block

A = gallery('jordbloc',n,lambda) returns the n-by-n Jordan block with eigenvalue lambda. The default value for lambda is 1.

kahan—Upper trapezoidal matrix

`A = gallery('kahan',n,theta,pert)` returns an upper trapezoidal matrix that has interesting properties regarding estimation of condition and rank.

If `n` is a two-element vector, then `A` is `n(1)`-by-`n(2)`; otherwise, `A` is `n`-by-`n`. The useful range of `theta` is `0 < theta < pi`, with a default value of `1.2`.

To ensure that the QR factorization with column pivoting does not interchange columns in the presence of rounding errors, the diagonal is perturbed by `pert*eps*diag([n:-1:1])`. The default `pert` is `25`, which ensures no interchanges for `gallery('kahan',n)` up to at least `n = 90` in IEEE arithmetic.

kms—Kac-Murdock-Szego Toeplitz matrix

`A = gallery('kms',n,rho)` returns the `n`-by-`n` Kac-Murdock-Szego Toeplitz matrix such that `A(i,j) = rho^(abs(i-j))`, for real `rho`.

For complex `rho`, the same formula holds except that elements below the diagonal are conjugated. `rho` defaults to 0.5.

The KMS matrix `A` has these properties:

- An LDL' factorization with `L = inv(gallery('triw',n,-rho,1))'`, and `D(i,i) = (1-abs(rho)^2)*eye(n)`, except `D(1,1) = 1`.
- Positive definite if and only if `0 < abs(rho) < 1`.
- The inverse `inv(A)` is tridiagonal.

krylov—Krylov matrix

`B = gallery('krylov',A,x,j)` returns the Krylov matrix

```
[x, Ax, A^2x, ..., A^(j-1)x]
```

where `A` is an `n`-by-`n` matrix and `x` is a length `n` vector. The defaults are `x = ones(n,1)`, and `j = n`.

`B = gallery('krylov',n)` is the same as `gallery('krylov',(randn(n))`.

lauchli—Rectangular matrix

`A = gallery('lauchli',n,mu)` returns the `(n+1)`-by-`n` matrix

```
[ones(1,n); mu*eye(n)]
```

The Lauchli matrix is a well-known example in least squares and other problems that indicates the dangers of forming `A'*A`. Argument `mu` defaults to `sqrt(eps)`.

lehmer—Symmetric positive definite matrix

`A = gallery('lehmer',n)` returns the symmetric positive definite `n`-by-`n` matrix such that `A(i,j) = i/j` for `j >= i`.

The Lehmer matrix `A` has these properties:

- `A` is totally nonnegative.
- The inverse `inv(A)` is tridiagonal and explicitly known.
- The order `n <= cond(A) <= 4*n*n`.

leslie—

`L = gallery('leslie',a,b)` is the `n`-by-`n` matrix from the Leslie population model with average birth numbers `a(1:n)` and survival rates `b(1:n-1)`. It is zero, apart from the first row (which contains the `a(i)`) and the first subdiagonal (which contains the `b(i)`). For a valid model, the `a(i)` are nonnegative and the `b(i)` are positive and bounded by 1, i.e., `0 < b(i) <= 1`.

`L = gallery('leslie',n)` generates the Leslie matrix with `a = ones(n,1)`, `b = ones(n-1,1)`.

lesp—Tridiagonal matrix with real, sensitive eigenvalues

`A = gallery('lesp',n)` returns an `n`-by-`n` matrix whose eigenvalues are real and smoothly distributed in the interval approximately `[-2*N-3.5, -4.5]`.

The sensitivities of the eigenvalues increase exponentially as the eigenvalues grow more negative. The matrix is similar to the symmetric tridiagonal matrix with the same diagonal entries and with off-diagonal entries 1, via a similarity transformation with `D = diag(1!,2!,...,n!)`.

lotkin—Lotkin matrix

`A = gallery('lotkin',n)` returns the Hilbert matrix with its first row altered to all ones. The Lotkin matrix `A` is nonsymmetric, ill-conditioned, and has many negative eigenvalues of small magnitude. Its inverse has integer entries and is known explicitly.

minij—Symmetric positive definite matrix

`A = gallery('minij',n)` returns the `n`-by-`n` symmetric positive definite matrix with `A(i,j) = min(i,j)`.

The `minij` matrix has these properties:

- The inverse `inv(A)` is tridiagonal and equal to `-1` times the second difference matrix, except its `(n,n)` element is `1`.
- Givens' matrix, `2*A-ones(size(A))`, has tridiagonal inverse and eigenvalues `0.5*sec((2*r-1)*pi/(4*n))^2`, where `r=1:n`.
- `(n+1)*ones(size(A))-A` has elements that are `max(i,j)` and a tridiagonal inverse.

moler—Symmetric positive definite matrix

`A = gallery('moler',n,alpha)` returns the symmetric positive definite `n`-by-`n` matrix `U'*U`, where `U = gallery('triw',n,alpha)`.

For the default `alpha = -1`, `A(i,j) = min(i,j)-2`, and `A(i,i) = i`. One of the eigenvalues of `A` is small.

neumann—Singular matrix from the discrete Neumann problem (sparse)

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

orthog—Orthogonal and nearly orthogonal matrices

`Q = gallery('orthog',n,k)` returns the kth type of matrix of order `n`, where `k > 0` selects exactly orthogonal matrices, and `k < 0` selects diagonal scalings of orthogonal matrices. Available types are:

`k = 1` `Q(i,j) = sqrt(2/(n+1)) * sin(i*j*pi/(n+1))`
Symmetric eigenvector matrix for second difference matrix. This is the default.

`k = 2` `Q(i,j) = 2/(sqrt(2*n+1)) * sin(2*i*j*pi/(2*n+1))`
Symmetric.

`k = 3` `Q(r,s) = exp(2*pi*i*(r-1)*(s-1)/n) / sqrt(n)`
Unitary, the Fourier matrix. Q^4 is the identity. This is essentially the same matrix as `fft(eye(n))/sqrt(n)`!

`k = 4` Helmert matrix: a permutation of a lower Hessenberg matrix, whose first row is `ones(1:n)/sqrt(n)`.

`k = 5` `Q(i,j) = sin(2*pi*(i-1)*(j-1)/n) + cos(2*pi*(i-1)*(j-1)/n)`
Symmetric matrix arising in the Hartley transform.

`K = 6` `Q(i,j) = sqrt(2/n)*cos((i-1/2)*(j-1/2)*pi/n)`
Symmetric matrix arising as a discrete cosine transform.

`k = -1` `Q(i,j) = cos((i-1)*(j-1)*pi/(n-1))`
Chebyshev Vandermonde-like matrix, based on extrema of `T(n-1)`.

`k = -2` `Q(i,j) = cos((i-1)*(j-1/2)*pi/n))`
Chebyshev Vandermonde-like matrix, based on zeros of `T(n)`.

parter—Toeplitz matrix with singular values near pi

`C = gallery('parter',n)` returns the matrix `C` such that
`C(i,j) = 1/(i-j+0.5)`.

`C` is a Cauchy matrix and a Toeplitz matrix. Most of the singular values of `C` are very close to `pi`.

pei—Pei matrix

`A = gallery('pei',n,alpha)`, where `alpha` is a scalar, returns the symmetric matrix `alpha*eye(n) + ones(n)`. The default for `alpha` is 1. The matrix is singular for `alpha` equal to either `0` or `-n`.

poisson—Block tridiagonal matrix from Poisson's equation (sparse)

`A = gallery('poisson',n)` returns the block tridiagonal (sparse) matrix of order `n^2` resulting from discretizing Poisson's equation with the 5-point operator on an `n`-by-`n` mesh.

prolate—Symmetric, ill-conditioned Toeplitz matrix

`A = gallery('prolate',n,w)` returns the `n`-by-`n` prolate matrix with parameter `w`. It is a symmetric Toeplitz matrix.

If `0 < w < 0.5` then `A` is positive definite

- The eigenvalues of `A` are distinct, lie in `(0,1)`, and tend to cluster around `0` and `1`.
- The default value of `w` is 0.25.

randcolu — Random matrix with normalized cols and specified singular values

`A = gallery('randcolu',n)` is a random `n`-by-`n` matrix with columns of unit 2-norm, with random singular values whose squares are from a uniform distribution.

`A'*A` is a correlation matrix of the form produced by `gallery('randcorr',n)`.

`gallery('randcolu',x)` where `x` is an `n`-vector ($n > 1$), produces a random `n`-by-`n` matrix having singular values given by the vector `x`. The vector `x` must have nonnegative elements whose sum of squares is `n`.

`gallery('randcolu',x,m)` where `m >= n`, produces an `m`-by-`n` matrix.

`gallery('randcolu',x,m,k)` provides a further option:

`k = 0`	`diag(x)` is initially subjected to a random two-sided orthogonal transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have zero entries.

For more information, see:

[1] Davies, P. I. and N. J. Higham, “Numerically Stable Generation of Correlation Matrices and Their Factors,” *BIT*, Vol. 40, 2000, pp. 640-651.

randcorr — Random correlation matrix with specified eigenvalues

`gallery('randcorr',n)` is a random `n`-by-`n` correlation matrix with random eigenvalues from a uniform distribution. A correlation matrix is a symmetric positive semidefinite matrix with `1`s on the diagonal (see `corrcoef`).

`gallery('randcorr',x)` produces a random correlation matrix having eigenvalues given by the vector `x`, where `length(x) > 1`. The vector `x` must have nonnegative elements summing to `length(x)`.

`gallery('randcorr',x,k)` provides a further option:

`k = 0`	The diagonal matrix of eigenvalues is initially subjected to a random orthogonal similarity transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have some zero entries.

For more information, see:

[1] Bendel, R. B. and M. R. Mickey, “Population Correlation Matrices for Sampling Experiments,” *Commun. Statist. Simulation Comput.*, B7, 1978, pp. 163-182.

[2] Davies, P. I. and N. J. Higham, “Numerically Stable Generation of Correlation Matrices and Their Factors,” *BIT*, Vol. 40, 2000, pp. 640-651.

randhess—Random, orthogonal upper Hessenberg matrix

`H = gallery('randhess',n)` returns an `n`-by-`n` real, random, orthogonal upper Hessenberg matrix.

`H = gallery('randhess',x)` if `x` is an arbitrary, real, length `n` vector with `n > 1`, constructs `H` nonrandomly using the elements of `x` as parameters.

Matrix `H` is constructed via a product of `n-1` Givens rotations.

rando—Random matrix composed of elements -1, 0 or 1

`A = gallery('rando',n,k)` returns a random `n`-by-`n` matrix with elements from one of the following discrete distributions:

`k = 1`	`A(i,j)` = `0` or `1` with equal probability (default).
`k = 2`	`A(i,j)` = `-1` or `1` with equal probability.
`k = 3`	`A(i,j)` = `-1`, `0` or `1` with equal probability.

Argument `n` may be a two-element vector, in which case the matrix is `n(1)`-by-`n(2)`.

randsvd—Random matrix with preassigned singular values

`A = gallery('randsvd',n,kappa,mode,kl,ku)` returns a banded (multidiagonal) random matrix of order `n` with `cond(A) = kappa` and singular values from the distribution mode. If `n` is a two-element vector, `A` is `n(1)`-by-`n(2)`.

Arguments `kl` and `ku` specify the number of lower and upper off-diagonals, respectively, in `A`. If they are omitted, a full matrix is produced. If only `kl` is present, `ku` defaults to `kl`.

Distribution `mode` can be:

`1`	One large singular value.
`2`	One small singular value.
`3`	Geometrically distributed singular values (default).

1	One large singular value.
4	Arithmetically distributed singular values.
5	Random singular values with uniformly distributed logarithm.
< 0	If `mode` is `-1`, `-2`, `-3`, `-4`, or `-5`, then `randsvd` treats `mode` as `abs(mode)`, except that in the original matrix of singular values the order of the diagonal entries is reversed: small to large instead of large to small.

Condition number `kappa` defaults to `sqrt(1/eps)`. In the special case where `kappa < 0`, `A` is a random, full, symmetric, positive definite matrix with `cond(A) = -kappa` and eigenvalues distributed according to `mode`. Arguments `kl` and `ku`, if present, are ignored.

`A = gallery('randsvd',n,kappa,mode,kl,ku,method)` specifies how the computations are carried out. `method = 0` is the default, while `method = 1` uses an alternative method that is much faster for large dimensions, even though it uses more flops.

redheff—Redheffer's matrix of 1s and 0s

`A = gallery('redheff',n)` returns an `n`-by-`n` matrix of `0`'s and `1`'s defined by `A(i,j) = 1`, if `j = 1` or if `i` divides `j`, and `A(i,j) = 0` otherwise.

The Redheffer matrix has these properties:

- `(n-floor(log2(n)))-1` eigenvalues equal to `1`
- A real eigenvalue (the spectral radius) approximately `sqrt(n)`
- A negative eigenvalue approximately `-sqrt(n)`
- The remaining eigenvalues are provably "small."
- The Riemann hypothesis is true if and only if $\det(A) = O(n^{\frac{1}{2}+\varepsilon})$ for every epsilon `> 0`.

Barrett and Jarvis conjecture that "the small eigenvalues all lie inside the unit circle `abs(Z) = 1`," and a proof of this conjecture, together with a proof that some eigenvalue tends to zero as `n` tends to infinity, would yield a new proof of the prime number theorem.

riemann—Matrix associated with the Riemann hypothesis

`A = gallery('riemann',n)` returns an `n`-by-`n` matrix for which the Riemann hypothesis is true if and only if

$$\det(A) = O(n!n^{-\frac{1}{2}+\varepsilon})$$

for every $\varepsilon > 0$.

The Riemann matrix is defined by:

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

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

The Riemann matrix has these properties:

- Each eigenvalue `e(i)` satisfies `abs(e(i)) <= m-1/m`, where `m = n+1`.
- `i <= e(i) <= i+1` with at most `m-sqrt(m)` exceptions.
- All integers in the interval `(m/3, m/2]` are eigenvalues.

ris—Symmetric Hankel matrix

`A = gallery('ris',n)` returns a symmetric `n`-by-`n` Hankel matrix with elements

```
A(i,j) = 0.5/(n-i-j+1.5)
```

The eigenvalues of `A` cluster around $\pi/2$ and $-\pi/2$. This matrix was invented by F.N. Ris.

rosser—Classic symmetric eigenvalue test matrix

`A = rosser` returns the Rosser matrix. This matrix was a challenge for many matrix eigenvalue algorithms. But the 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 `gallery('smoke',n,1)` are the `n`th roots of unity; those of `gallery('smoke',n)` are the `n`th roots of unity times `2^(1/n)`.

toeppd—Symmetric positive definite Toeplitz matrix

`A = gallery('toeppd',n,m,w,theta)` returns an `n`-by-`n` symmetric, positive semi-definite (SPD) Toeplitz matrix composed of the sum of `m` rank 2 (or, for certain `theta`, rank 1) SPD Toeplitz matrices. Specifically,

```
T = w(1)*T(theta(1)) + ... + w(m)*T(theta(m))
```

where `T(theta(k))` has `(i,j)` element `cos(2*pi*theta(k)*(i-j))`.

By default: `m = n`, `w = rand(m,1)`, and `theta = rand(m,1)`.

toeppen—Pentadiagonal Toeplitz matrix (sparse)

`P = gallery('toeppen',n,a,b,c,d,e)` returns the `n`-by-`n` sparse, pentadiagonal Toeplitz matrix with the diagonals: `P(3,1) = a`, `P(2,1) = b`, `P(1,1) = c`, `P(1,2) = d`, and `P(1,3) = e`, where `a`, `b`, `c`, `d`, and `e` are scalars.

By default, `(a,b,c,d,e) = (1,-10,0,10,1)`, yielding a matrix of Rutishauser. This matrix has eigenvalues lying approximately on the line segment `2*cos(2*t) + 20*i*sin(t)`.

tridiag—Tridiagonal matrix (sparse)

`A = gallery('tridiag',c,d,e)` returns the tridiagonal matrix with subdiagonal `c`, diagonal `d`, and superdiagonal `e`. Vectors `c` and `e` must have `length(d)-1`.

`A = gallery('tridiag',n,c,d,e)`, where `c`, `d`, and `e` are all scalars, yields the Toeplitz tridiagonal matrix of order `n` with subdiagonal elements `c`, diagonal elements `d`, and superdiagonal elements `e`. This matrix has eigenvalues

```
d + 2*sqrt(c*e)*cos(k*pi/(n+1))
```

where `k = 1:n`. (see [1].)

`A = gallery('tridiag',n)` is the same as `A = gallery('tridiag',n,-1,2,-1)`, which is a symmetric positive definite M-matrix (the negative of the second difference matrix).

triw—Upper triangular matrix discussed by Wilkinson and others

`A = gallery('triw',n,alpha,k)` returns the upper triangular matrix with ones on the diagonal and `alpha`s on the first `k >= 0` superdiagonals.

Order `n` may be a 2-element vector, in which case the matrix is `n(1)`-by-`n(2)` and upper trapezoidal.

Ostrowski ["On the Spectrum of a One-parametric Family of Matrices, *J. Reine Angew. Math.,* 1954] shows that

```
cond(gallery('triw',n,2)) = cot(pi/(4*n))^2,
```

and, for large `abs(alpha)`, `cond(gallery('triw',n,alpha))` is approximately `abs(alpha)^n*sin(pi/(4*n-2))`.

Adding `-2^(2-n)` to the `(n,1)` element makes `triw(n)` singular, as does adding `-2^(1-n)` to all the elements in the first column.

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 = 3`	Upper triangular system `Ux=b` illustrating inaccurate solution.
`n = 4`	Lower triangular system `Lx=b`, ill-conditioned.

`n = 5`	`hilb(6)(1:5,2:6)*1.8144`. A symmetric positive definite matrix.
`n = 21`	`W21+`, a tridiagonal matrix. Eigenvalue problem. For more detail, see [2].

See Also

`hadamard`, `hilb`, `invhilb`, `magic`, `wilkinson`

References

[1] The MATLAB gallery of test matrices is based upon the work of Nicholas J. Higham at the Department of Mathematics, University of Manchester, Manchester, England. 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.

[2] Wilkinson, J. H., *The Algebraic Eigenvalue Problem*, Oxford University Press, London, 1965, p.308.

gamma, gammainc, gammaln

Purpose Gamma functions

Syntax

`Y = gamma(A)`	Gamma function
`Y = gammainc(X,A)`	Incomplete gamma function
`Y = gammainc(X,A,tail)`	Tail of the 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:

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

For any `a>=0`, `gammainc(x,a)` approaches 1 as `x` approaches `infinity`. For small `x` and `a`, `gammainc(x,a)` is approximately equal to `x^a`, so `gammainc(0,0) = 1`.

Description

`Y = gamma(A)` returns the gamma function at the elements of `A`. `A` must be real.

`Y = gammainc(X,A)` returns the incomplete gamma function of corresponding elements of `X` and `A`. Arguments `X` and `A` must be real and the same size (or either can be scalar).

`Y = gammainc(X,A,tail)` specifies the tail of the incomplete gamma function when `X` is non-negative. The choices are for `tail` are `'lower'` (the default) and `'upper'`. The upper incomplete gamma function is defined as

```
1 - gammainc(x,a)
```

Note When `X` is negative, `Y` can be inaccurate for `abs(X)>A+1`.

`Y = gammaln(A)` returns the logarithm of the gamma function, `gammaln(A) = log(gamma(A))`. The `gammaln` command avoids the underflow and overflow that may occur if it is computed directly using `log(gamma(A))`.

Algorithm

The computations of `gamma` and `gammaln` are based on algorithms outlined in [1]. Several different minimax rational approximations are used depending upon the value of `A`. Computation of the incomplete gamma function is based on the algorithm in [2].

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

Purpose 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.

Current Axes

The current axes is the target for graphics output when you create axes children. The current axes is typically the last axes used for plotting or the last axes clicked on by the mouse. Graphics commands such as `plot`, `text`, and `surf` draw their results in the current axes. Changing the current figure also changes the current axes.

See Also

`axes`, `cla`, `gcf`, `findobj`

figure `CurrentAxes` property

"Finding and Identifying Graphics Objects" for related functions

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 `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`

"Finding and Identifying Graphics Objects" for related functions

Purpose Greatest common divisor

Syntax

```
G = gcd(A,B)
[G,C,D] = gcd(A,B)
```

Description `G = gcd(A,B)` returns an array containing the greatest common divisors of the corresponding elements of integer arrays `A` and `B`. By convention, `gcd(0,0)` returns a value of `0`; all other inputs return positive integers for `G`.

`[G,C,D] = gcd(A,B)` returns both the greatest common divisor array `G`, and the arrays `C` and `D`, which satisfy the equation: `A(i).*C(i) + B(i).*D(i) = G(i)`. These are useful for solving Diophantine equations and computing elementary Hermite transformations.

Examples The first example involves elementary Hermite transformations.

For any two integers `a` and `b` there is a 2-by-2 matrix `E` with integer entries and determinant = `1` (a *unimodular* matrix) such that:

```
E * [a;b] = [g,0],
```

where `g` is the greatest common divisor of `a` and `b` as returned by the command `[g,c,d] = gcd(a,b)`.

The matrix `E` equals:

```
c       d
-b/g    a/g
```

In the case where `a = 2` and `b = 4`:

```
[g,c,d] = gcd(2,4)
g =
     2
c =
     1
d =
     0
```

So that

```
E =
     1     0
    -2     1
```

In the next example, we solve for `x` and `y` in the Diophantine equation `30x + 56y = 8`.

```
[g,c,d] = gcd(30,56)
g =
      2
c =
    -13
d =
      7
```

By the definition, for scalars `c` and `d`:

```
30(-13) + 56(7) = 2,
```

Multiplying through by 8/2:

```
30(-13*4) + 56(7*4) = 8
```

Comparing this to the original equation, a solution can be read by inspection:

```
x = (-13*4) = -52; y = (7*4) = 28
```

See Also

`lcm`

References

[1] Knuth, Donald, *The Art of Computer Programming*, Vol. 2, Addison-Wesley: Reading MA, 1973. Section 4.5.2, Algorithm X.

Purpose 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

`clf`, `figure`, `gca`

Root `CurrentFigure` property

"Finding and Identifying Graphics Objects" for related functions

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

"Finding and Identifying Graphics Objects" for related functions

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;
```

You can also use genpath in conjunction with addpath to add subdirectories to the path from the command line. The following example adds the /control directory and its subdirectories to the current path.

```
%  Display the current path
path

              MATLABPATH

        K:\toolbox\matlab\general
        K:\toolbox\matlab\ops
        K:\toolbox\matlab\lang
        K:\toolbox\matlab\elmat
        K:\toolbox\matlab\elfun
                  :
                  :
                  :

%  Use GENPATH to add /control and its subdirectories
addpath(genpath('K:/toolbox/control'))


%  Display the new path
path

              MATLABPATH

        K:\toolbox\control
        K:\toolbox\control\ctrlutil
        K:\toolbox\control\control
        K:\toolbox\control\ctrlguis
        K:\toolbox\control\ctrldemos
        K:\toolbox\matlab\general
        K:\toolbox\matlab\ops
        K:\toolbox\matlab\lang
        K:\toolbox\matlab\elmat
        K:\toolbox\matlab\elfun
                  :
                  :
                  :
```

See Also

`addpath`, `path`, `pathdef`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `rmpath`, `savepath`

Search Path

Purpose Construct valid variable name from string

Syntax

```
varname = genvarname(str)
varname = genvarname(str, exclusions)
```

Description

`varname = genvarname(str)` constructs a string `varname` that is similar to or the same as the `str` input, and can be used as a valid variable name. `str` can be a single character array or a cell array of strings. If `str` is a cell array of strings, `genvarname` returns a cell array of strings in `varname`. The strings in a cell array returned by `genvarname` are guaranteed to be different from each other.

`varname = genvarname(str, exclusions)` returns a valid variable name that is different from any name listed in the `exclusions` input. The `exclusions` input can be a single character array or a cell array of strings. Specify the string `'who'` for exclusions to create a variable name that will be unique in the current MATLAB workapace (see “Example 4”, below).

Note `genvarname` returns a string that can be used as a variable name. It does not create a variable in the MATLAB workspace. You cannot, therefore, assign a value to the output of `genvarname`.

Remarks

A valid MATLAB variable name is a character string of letters, digits, and underscores, such that the first character is a letter, and the length of the string is less than or equal to the value returned by the `namelengthmax` function. Any string that excedes `namelengthmax` is truncated in the `varname` output. See “Example 6”, below.

The variable name returned by `genvarname` is not guaranteed to be different from other variable names currently in the MATLAB workspace unless you use the `exclusions` input in the manner shown in “Example 4”, below.

If you use `genvarname` to generate a field name for a structure, MATLAB does create a variable for the structure and field in the MATLAB workspace. See “Example 3”, below.

If the `str` input contains any whitespace characters, `genvarname` removes then and capitalizes the next alphabetic character in `str`. If `str` contains any nonalphanumeric characters, `genvarname` translates these characters into their hexadecimal value.

Examples

Example 1

Create four similar variable name strings that do not conflict with each other:

```
v = genvarname({'A', 'A', 'A', 'A'})
v =
    'A'    'A1'    'A2'    'A3'
```

Example 2

Read a column header `hdr` from worksheet `trial2` in Excel spreadsheet `myproj_apr23`:

```
[data hdr] = xlsread('myproj_apr23.xls', 'trial2');
```

Make a variable name from the text of the column header that will not conflict with other names:

```
v = genvarname(['Column ' hdr{1,3}]);
```

Assign data taken from the spreadsheet to the variable in the MATLAB workspace:

```
eval([v '= data(1:7, 3);']);
```

Example 3

Collect readings from an instrument once every minute over the period of an hour into different fields of a structure. `genvarname` not only generates unique fieldname strings, but also creates the structure and fields in the MATLAB workspace:

```
for k = 1:60
record.(genvarname(['reading' datestr(clock, 'HHMMSS')])) ...
   = takeReading;
pause(60)
end
```

After the program ends, display the recorded data from the workspace:

```
record
record =
    reading090446: 27.3960
    reading090546: 23.4890
    reading090646: 21.1140
    reading090746: 23.0730
    reading090846: 28.5650
          .
          .
          .
```

Example 4

Generate variable names that are unique in the MATLAB workspace by putting the output from the `who` function in the `exclusions` list.

```
for k = 1:5
   t = clock;
   pause(uint8(rand * 10));
   v = genvarname('time_elapsed', who);
   eval([v ' = etime(clock,t)'])
   end
```

As this code runs, you can see that the variables created by `genvarname` are unique in the workspace:

```
time_elapsed =
    5.0070
time_elapsed1 =
    2.0030
time_elapsed2 =
    7.0010
time_elapsed3 =
    8.0010
time_elapsed4 =
    3.0040
```

After the program completes, use the `who` function to view the workspace variables:

```
who
```

```
k           time_elapsed   time_elapsed2  time_elapsed4
t           time_elapsed1  time_elapsed3  v
```

Example 5

If you try to make a variable name from a MATLAB keyword, `genvarname` creates a variable name string that capitalizes the keyword and precedes it with the letter `x`:

```
v = genvarname('global')
v =
   xGlobal
```

Example 6

If you enter a string that is longer than the value returned by the `namelengthmax` function, genvarname truncates the resulting variable name string:

```
namelengthmax
ans =
    63

vstr = genvarname(sprintf('%s%s', ...
   'This name truncates because it contains ', ...
   'more than the maximum number of characters'))
vstr =
ThisNameTruncatesBecauseItContainsMoreThanTheMaximumNumberOfCha
```

See Also

`isvarname`, `iskeyword`, `isletter`, `namelengthmax`, `who`, `regexp`

Purpose 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 of the graphics object identified by the handle `h` and their current values.

`get(h,'PropertyName')` returns the value of the property `'PropertyName'` of the graphics object identified by `h`.

`<m-by-n value cell array> = get(H,pn)` returns n property values for m graphics objects in the m-by-n cell array, where `m = 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,'FactoryObjectTypePropertyName')` returns the factory-defined value of the named property for the specified object type. The argument *`FactoryObjectTypePropertyName`* is the word `Factory` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`).

`FactoryFigureColor a = get(h,'Default')` returns all default values currently defined on object `h`. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(h,'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

“Finding and Identifying Graphics Objects” for related functions

Purpose

Display or get timer object properties

Syntax

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

Description

`get(obj)` displays all property names and their current values for the timer object `obj`. `obj` must be a single timer object.

`V = get(obj)` returns a structure, `V`, where each field name is the name of a property of `obj` and each field contains the value of that property. If `obj` is an M-by-1 vector of timer objects, V is an M-by-1 array of structures.

`V = get(obj,'`*`PropertyName`*`')` returns the value, `V`, of the timer object property specified in *`PropertyName`*.

If *`PropertyName`* is a 1-by-N or N-by-1 cell array of strings containing property names, `V` is a 1-by-N cell array of values. If `obj` is a vector of timer objects, `V` is an M-by-N cell array of property values where M is equal to the length of `obj` and N is equal to the number of properties specified.

Examples

```
t = timer;
get(t)
       AveragePeriod: NaN
            BusyMode: 'drop'
            ErrorFcn: ''
       ExecutionMode: 'singleShot'
       InstantPeriod: NaN
                Name: 'timer-1'
    ObjectVisibility: 'on'
              Period: 1
             Running: 'off'
          StartDelay: 1
            StartFcn: ''
             StopFcn: ''
                 Tag: ''
       TasksExecuted: 0
      TasksToExecute: Inf
            TimerFcn: ''
                Type: 'timer'
```

```
             UserData: []
get(t, {'StartDelay','Period'})
ans =

    [0]    [1]
```

See Also

`timer`, `set`

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`

Purpose

Get environment variable

Syntax

```
getenv 'name'
N = getenv('name')
```

Description

`getenv 'name'` searches the underlying operating system's environment list for a string of the form `name=value`, where `name` is the input string. If found, 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.

Remarks In many cases, you can use dynamic field names in place of the `getfield` and `setfield` functions. Dynamic field names express structure fields as variable expressions that MATLAB evaluates at run-time. See Technical Note 32236 for information about using dynamic field names versus the `getfield` and `setfield` functions.

Examples Given the structure

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

Then the command `f = getfield(mystr,{2,1},'name')` yields

```
f =
   gertrude
```

To list the contents of all name (or other) fields, embed `getfield` in a loop.

```
for k = 1:2
    name{k} = getfield(mystr,{k,1},'name');
end
name
```

```
name =

    'alice'    'gertrude'
```

The following example starts out by creating a structure using the standard structure syntax. It then reads the fields of the structure, using `getfield` with variable and quoted field names and additional subscripting arguments.

```
class = 5;     student = 'John_Doe';
grades(class).John_Doe.Math(10,21:30) = ...
             [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];
```

Use `getfield` to access the structure fields.

```
getfield(grades,{class}, student, 'Math', {10,21:30})

ans =

    85    89    76    93    85    91    68    84    95    73
```

See Also

`setfield`, `fieldnames`, `isfield`, `orderfields`, `rmfield`, dynamic field names

Purpose Get movie frame

Syntax

```
F = getframe
F = getframe(h)
F = getframe(h,rect)
```

Description

`getframe` returns a movie frame. The frame is a snapshot (pixmap) of the current axes or figure.

`F = getframe` gets a frame from the current axes.

`F = getframe(h)` gets a frame from the figure or axes identified by 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)
```

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)
```

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

`frame2im`, `image`, `im2frame`, `movie`

"Bit-Mapped Images" for related functions

Purpose Utility function for creating and obtaining the figure components used for plot editing.

Syntax

```
c = getplottool(figure_handle,'figurepalette')
c = getplottool(figure_handle,'plotbrowser')
c = getplottool(figure_handle,'propertyeditor')
```

Description

`c = getplottool(figure_handle,'figurepalette')` returns the Java figure palette for the specified figure.

`c = getplottool(figure_handle,'plotbrowser')` returns the Java plot browser for the specified figure.

`c = getplottool(figure_handle,'propertyeditor')` returns the Java property editor for the specified figure.

In each case, `getplottool` creates the component if it does not already exist. The component is not automatically shown. If you want to both create it and show it, use `showplottool`.

See Also `showplottool`

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 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' coordinate systems.

Examples

Pick 10 two-dimensional points from the figure window.

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

Position the cursor with the mouse. Enter data points by pressing a mouse button or a key on the keyboard. To terminate input before entering 10 points, press the **Return** key.

See Also

`gtext`

Interactive Plotting for an example

"Interactive User Input" for related functions

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 should be large and sparse. The column vector `b` must have length `n`. `A` can be a function `afun` such that `afun(x)` returns `A*x`. For this syntax, `gmres` does not restart; the maximum number of iterations is `min(n,10)`.

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

`gmres(A,b,restart)` restarts the method every `restart` inner iterations. The maximum number of outer iterations is `min(n/restart,10)`. The maximum number of total iterations is `restart*min(n/restart,10)`. If `restart` is `n` or `[]`, then `gmres` does not restart and the maximum number of total iterations is `min(n,10)`.

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

`gmres(A,b,restart,tol,maxit)` specifies the maximum number of outer iterations, i.e., the total number of iterations does not exceed `restart*maxit`. If `maxit` is `[]` then `gmres` uses the default, `min(n/restart,10)`. If `restart` is `n` or [], then the maximum number of total iterations is maxit (instead of restart*maxit).

gmres(A,b,restart,tol,maxit,M) and gmres(A,b,restart,tol,maxit,M1,M2) use preconditioner M or M = M1*M2 and effectively solve the system inv(M)*A*x = inv(M)*b for x. If M is [] then gmres applies no preconditioner. M can be a function 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 = 0	gmres converged to the desired tolerance tol within maxit outer iterations.
flag = 1	gmres iterated maxit times but did not converge.
flag = 2	Preconditioner M was ill-conditioned.
flag = 3	gmres stagnated. (Two consecutive iterates were the same.)

Whenever flag is not 0, the solution x returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the flag output is specified.

[x,flag,relres] = gmres(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = gmres(A,b,...) also returns both the outer and inner iteration numbers at which x was computed, where 0 <= iter(1) <= maxit and 0 <= iter(2) <= restart.

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

Examples

Example 1.

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = gmres(A,b,10,tol,maxit,M1,[],[]);
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 the `gmres` extra input `n=21`.

Example 2.

```
load west0479
A = west0479
b = sum(A,2)
[x,flag] = gmres(A,b,5)
```

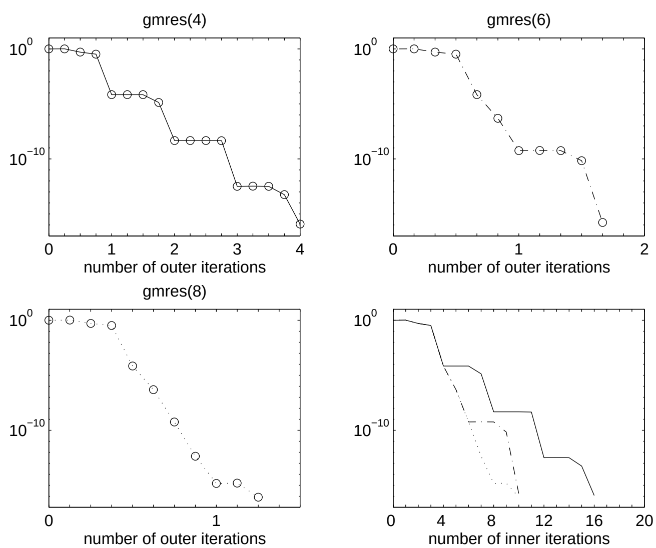
`flag` is `1` because `gmres` does not converge to the default tolerance `1e-6` within the default 10 outer iterations.

```
[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

Syntax

```
gplot(A,Coordinates)
gplot(A,Coordinates,LineSpec)
```

Description

The `gplot` function graphs a set of coordinates using an adjacency matrix.

`gplot(A,Coordinates)` plots a graph of the nodes defined in `Coordinates` according to the n-by-n adjacency matrix `A`, where n is the number of nodes. `Coordinates` is an n-by-2 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`.

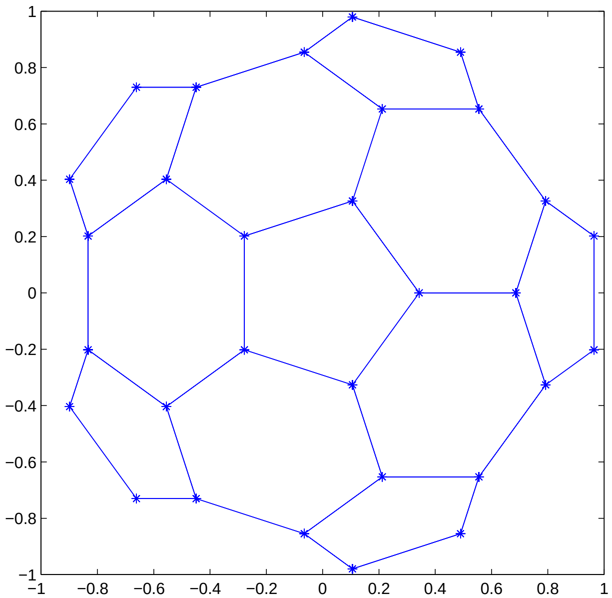
Remarks

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)` is nonzero; otherwise, `A(i,j)` and `A(j,i)` are zero.

Examples

To draw half of a Bucky ball with asterisks at each node,

```
k = 1:30;
[B,XY] = bucky;
gplot(B(k,k),XY(k,:),'-*')
axis square
```



See Also

`LineSpec`, `sparse`, `spy`

"Tree Operations" for related functions

gradient

Purpose

Numerical gradient

Syntax

```
FX = gradient(F)
[FX,FY] = gradient(F)
[Fx,Fy,Fz,...] = gradient(F)
[...] = gradient(F,h)
[...] = gradient(F,h1,h2,...)
```

Definition

The *gradient* of a function of two variables, $F(x, y)$, is defined as

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

and can be thought of as a collection of vectors pointing in the direction of increasing values of F. In MATLAB, numerical gradients (differences) can be computed for functions with any number of variables. For a function of N variables, $F(x, y, z, \ldots)$,

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

Description

`FX = gradient(F)` where `F` is a vector returns the one-dimensional numerical gradient of `F`. `FX` corresponds to $\partial F/\partial x$, the differences in 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.

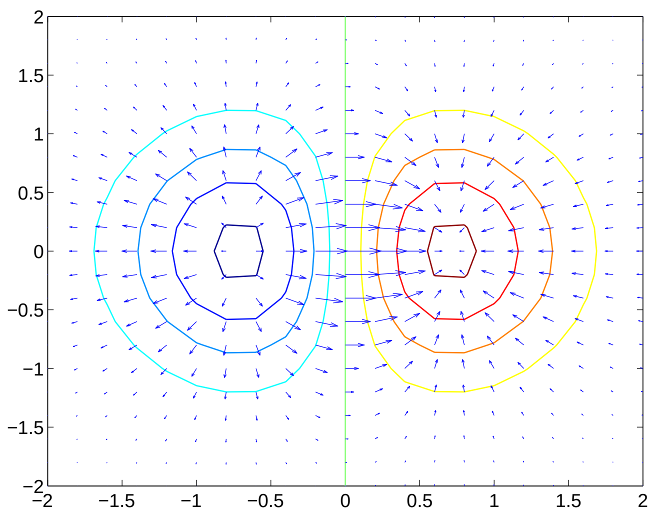
`[...] = gradient(F,h1,h2,...)` with `N` spacing parameters specifies the spacing for each dimension of `F`.

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(v,v,px,py), hold off
```

produce



Given,

```
F(:,:,1) = magic(3); F(:,:,2) = pascal(3);
```

```
gradient(F)
```

takes dx = dy = dz = 1.

```
[PX,PY,PZ] = gradient(F,0.2,0.1,0.2)
```

takes dx = 0.2, dy = 0.1, and dz = 0.2.

See Also del2, diff

Purpose Set default figure properties for grayscale monitors

Syntax

```
graymon
```

Description `graymon` sets defaults for graphics properties to produce more legible displays for grayscale monitors.

See Also `axes`, `figure`

"Color Operations" for related functions

grid

Purpose

Grid lines for two- and three-dimensional plots

Syntax

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

Description

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

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

`grid off` removes major and minor grid lines from the current axes.

`grid` toggles the major grid visibility state.

`grid(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

Algorithm

`grid` sets the `XGrid`, `YGrid`, and `ZGrid` properties of the axes.

`grid minor` sets the `XGridMinor`, `YGridMinor`, and `ZGridMinor` properties of the axes.

You can set the grid lines for just one axis using the `set` command and the individual property. For example,

```
set(axes_handle,'XGrid','on')
```

turns on only x-axis grid lines.

Note that the grid line width is not affected by the axes `LineWidth` property.

See Also

`axes`, `set`

The properties of axes objects

"Axes Operations" for related functions

Purpose Data gridding

Syntax

```
ZI = griddata(x,y,z,XI,YI)
[XI,YI,ZI] = griddata(x,y,z,XI,YI)
[...] = griddata(...,method)
[...] = griddata(...,method,options)
```

Description

`ZI = griddata(x,y,z,XI,YI)` fits a surface of the form `z = f(x,y)` to the data in the (usually) nonuniformly spaced vectors `(x,y,z)`. `griddata` interpolates this surface at the points specified by `(XI,YI)` to produce `ZI`. The surface always passes through the data points. `XI` and `YI` usually form a uniform grid (as produced by `meshgrid`).

`XI` can be a row vector, in which case it specifies a matrix with constant columns. Similarly, `YI` can be a column vector, and it specifies a matrix with constant rows.

`[XI,YI,ZI] = griddata(x,y,z,XI,YI)` returns the interpolated matrix `ZI` as above, and also returns the matrices `XI` and `YI` formed from row vector `XI` and column vector `yi`. These latter are the same as the matrices returned by `meshgrid`.

`[...] = griddata(...,method)` uses the specified interpolation method:

`'linear'`	Triangle-based linear interpolation (default)
`'cubic'`	Triangle-based cubic interpolation
`'nearest'`	Nearest neighbor interpolation
`'v4'`	MATLAB 4 `griddata` method

The `method` defines the type of surface fit to the data. The `'cubic'` and `'v4'` methods produce smooth surfaces while `'linear'` and `'nearest'` have discontinuities in the first and zero'th derivatives, respectively. All the methods except `'v4'` are based on a Delaunay triangulation of the data. If `method` is `[]`, then the default `'linear'` method is used.

`[...] = griddata(...,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`. If `options` is `[]`, the default delaunayn options are used. If options is {''}, no options are used, not even the default.

Occasionally, griddata might return points on or very near the convex hull of the data as NaNs. This is because roundoff in the computations sometimes makes it difficult to determine if a point near the boundary is in the convex hull.

Remarks

XI and YI can be matrices, in which case griddata returns the values for the corresponding points (XI(i,j),YI(i,j)). Alternatively, you can pass in the row and column vectors xi and yi, respectively. In this case, griddata interprets these vectors as if they were matrices produced by the command meshgrid(xi,yi).

Algorithm

The griddata(...,'v4') command uses the method documented in [3]. The other griddata methods are based on a Delaunay triangulation of the data that uses Qhull [2]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.html.

Examples

Sample a function at 100 random points between ±2.0:

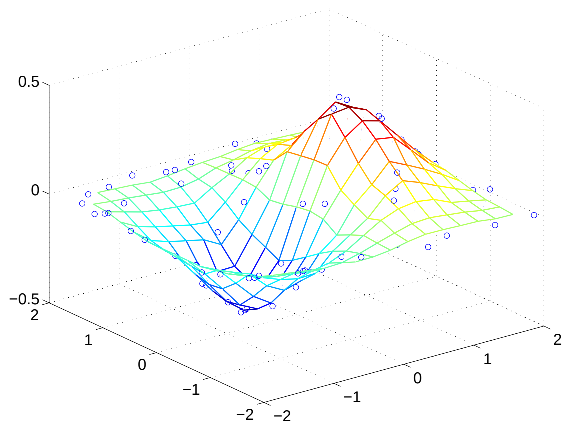
```
rand('seed',0)
x = rand(100,1)*4-2; y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

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] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in HTML format at `http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/` and in PostScript format at `ftp://geom.umn.edu/pub/software/qhull-96.ps`.

[2] National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center), University of Minnesota. 1993.

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

[4] Watson, David E., *Contouring: A Guide to the Analysis and Display of Spatial Data*, Tarrytown, NY: Pergamon (Elsevier Science, Inc.): 1992.

griddata3

Purpose Data gridding and hypersurface fitting for 3-D data

Syntax

```
w = griddata3(x,y,z,v,xi,yi,zi)
w = griddata3(x,y,z,v,xi,yi,zi,method)
w = griddata3(x,y,z,v,xi,yi,zi,method,options)
```

Description `w = griddata3(x, y, z, v, xi, yi, zi)` fits a hypersurface of the form $w = f(x, y, z)$ to the data in the (usually) nonuniformly spaced vectors (`x`, `y`, `z`, `v`). `griddata3` interpolates this hypersurface at the points specified by (`xi`,`yi`,`zi`) to produce `w`. `w` is the same size as `xi`, `yi`, and `zi`.

(`xi`,`yi`,`zi`) is usually a uniform grid (as produced by `meshgrid`) and is where `griddata3` gets its name.

`w = griddata3(x,y,z,v,xi,yi,zi,method)` defines the type of surface that is fit to the data, where `method` is either:

`'linear'`	Tesselation-based linear interpolation (default)
`'nearest'`	Nearest neighbor interpolation

If `method` is `[]`, the default `'linear'` method is used.

`w = griddata3(x,y,z,v,xi,yi,zi,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default.

Algorithm The `griddata3` methods are based on a Delaunay triangulation of the data that uses Qhull [2]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.html`.

See Also `delaunayn`, `griddata`, `griddatan`, `meshgrid`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in HTML format at `http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/` and in PostScript format at `ftp://geom.umn.edu/pub/software/qhull-96.ps`.

[2] 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(x,y,z,v,xi,yi,zi,method)
yi = griddatan(x,y,z,v,xi,yi,zi,method,options)
```

Description

`yi = griddatan(X, y, xi)` fits a hyper-surface of the form $y = f(X)$ to the data in the (usually) nonuniformly-spaced vectors (`X`, `y`). `griddatan` interpolates this hyper-surface at the points specified by `xi` to produce `yi`. `xi` can be nonuniform.

`X` is of dimension `m`-by-`n`, representing `m` points in n-dimensional space. `y` is of dimension `m`-by-`1`, representing `m` values of the hyper-surface f(`X`). `xi` is a vector of size `p`-by-`n`, representing `p` points in the n-dimensional space whose surface value is to be fitted. `yi` is a vector of length `p` approximating the values f(`xi`). The hypersurface always goes through the data points (`X`,`y`). `xi` is usually a uniform grid (as produced by `meshgrid`).

`yi = griddatan(x,y,z,v,xi,yi,zi,method)` defines the type of surface fit to the data, where `'method'` is one of:

`'linear'`	Tessellation-based linear interpolation (default)
`'nearest'`	Nearest neighbor interpolation

All the methods are based on a Delaunay tessellation of the data.

If `method` is `[]`, the default `'linear'` method is used.

`yi = griddatan(x,y,z,v,xi,yi,zi,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default.

Algorithm The `griddatan` methods are based on a Delaunay triangulation of the data that uses Qhull [2]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also `delaunayn`, `griddata`, `griddata3`, `meshgrid`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in HTML format at `http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/` and in PostScript format at `ftp://geom.umn.edu/pub/software/qhull-96.ps`.

[2] 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.

See Also

`qr`, `svd`

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

Purpose Mouse placement of text in two-dimensional view

Syntax

```
gtext('string')
gtext({'string1','string2','string3',...})
gtext({'string1';'string2';'string3';...})
h = gtext(...)
```

Description `gtext` displays a text string in the current figure window after you select a location with the mouse.

`gtext('`*`string`*`')` waits for you to press a mouse button or keyboard key while the pointer is within a figure window. Pressing a mouse button or any key places `'`*`string`*`'` on the plot at the selected location.

`gtext({'string1','string2','string3',...})` places all strings with one click, each on a separate line.

`gtext({'string1';'string2';'string3';...})` places one string per click, in the sequence specified.

`h = gtext(...)` returns the handle to a text graphics object that is placed on the plot at the location you select.

Remarks As you move the pointer into a figure window, the pointer becomes crosshairs to indicate that `gtext` is waiting for you to select a location. `gtext` uses the functions `ginput` and `text`.

Examples Place a label on the current plot:

```
gtext('Note this divergence!')
```

See Also `ginput`, `text`

"Annotating Plots" for related functions

guidata

Purpose Store or retrieve application data

Syntax

```
guidata(object_handle, data)
data = guidata(object_handle)
```

Description

`guidata(object_handle,data)` stores the variable `data` in the figure's application data. If `object_handle` is not a figure handle, then the object's parent figure is used. `data` can be any MATLAB variable, but is typically a structure, which enables you to add new fields as required.

Note that there can be only one variable stored in a figure's application data at any time. Subsequent calls to `guidata(object_handle,data)` overwrite the previously created version of `data`. See the Examples section for information on how to use this function.

`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.

```
% create structure of handles
handles = guihandles(figure_handle);
% add some additional data
handles.numberOfErrors = 0;
```

```
% save the structure
guidata(figure_handle,handles)
```

You can recall the data from within a subfunction callback routine and then save the structure again:

```
% get the structure in the subfunction
handles = guidata(gcbo);
handles.numberOfErrors = handles.numberOfErrors + 1;
% save the changes to the structure
guidata(gcbo,handles)
```

See Also

`guide`, `guihandles`, `getappdata`, `setappdata`

guide

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

2hadamard

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 low-level functions that enable you to access the HDF4 library developed and supported by the National Center for Supercomputing Applications (NCSA). For information about HDF, see the NCSA HDF Web page at `http://hdf.ncsa.uiuc.edu`.

The following table lists all the HDF4 application programming interfaces (APIs) supported by MATLAB with the name of the MATLAB function used to access the API. To use these functions, you must be familiar with the HDF library.

Application Programming Interface	Description	MATLAB Function
Annotations	Stores, manages, and retrieves text used to describe an HDF file or any of the data structures contained in the file.	`hdfan`
General Raster Images	Stores, manages, and retrieves raster images, their dimensions and palettes. It can also manipulate unattached palettes. Note: Use the MATLAB functions `imread` and `imwrite` with HDF raster image formats.	`hdfdf24` `hdfdfr8`

HDF-EOS	Provides functions to read HDF-EOS grid (GD), point (PT), and swath (SW) data.	`hdfgd`
		`hdfpt`
		`hdfsw`

HDF Utilities	Provides functions to open and close HDF files and handle errors.	`hdfh`
		`hdfhd`
		`hdfhe`

Application Programming Interface	Description	MATLAB Function
MATLAB HDF Utilities	Provides utility functions that help you work with HDF files in the MATLAB environment.	`hdfml`
Scientific Data	Stores, manages, and retrieves multidimensional arrays of character or numeric data, along with their dimensions and attributes.	`hdfsd`
V Groups	Creates and retrieves groups of other HDF data objects, such as raster images or V data.	`hdfv`

V Data	Stores, manages, and retrieves multivariate data stored as records in a table.	`hdfvf`
		`hdfvh`
		`hdfvs`

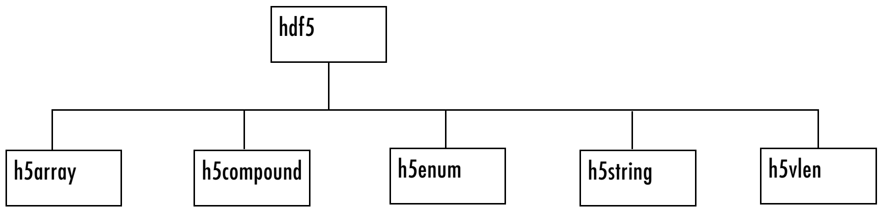
See Also

`hdf5read`, `hdfread`, `hdfinfo`, `imread`

Purpose HDF5 data type classes

Syntax `hdf5*(...)`

Description MATLAB provides a set of classes to represent HDF5 data types. MATLAB defines a general HDF5 data type class, with subclasses for individual HDF5 data types. The following figure illustrates these classes and subclasses. For more information about a specific class, see the sections that follow. To learn more about the HDF5 data types in general, see the NCSA HDF Web page at `http://hdf.ncsa.uiuc.edu`. For information about using these classes, see "Remarks" on page 2-1019.



h5array

The HDF5 `h5array` class associates a name with an array. The following are the data members of the `h5array` class.

Data Members	
`Data`	Multidimensional array
`Name`	Text string specifying the name of the object

The following are the methods of the `h5array` class. This table shows the function calling syntax. You can also access methods using subscripted reference (dot notation). For an example of the syntax, see “HDF5 Enumerated Object Example” on page 2-1021.

Methods	Description	Syntax

`hdf5.h5array`	Constructs object of class `h5array`.	`arr = hdf5.h5array;`
		`arr = hdf5.h5array(data)`
		
		where `arr` is an `h5array` object and `data` can be numeric, a cell array, or an HDF5 data type.

`setData`	Sets the value of the object’s `Data` member.	`setData(arr, data)` where `arr` is an `h5array` object and `data` can be numeric, a cell array, or an HDF5 data type.
`setName`	Sets the value of the object’s `Name` member.	`setName(arr, name)` where `arr` is an `h5array` object and `name` is a string or cell array.

h5compound

The HDF5 `h5compound` class associates a name with a structure, where you can define the field names in the structure and their values. The following are the data members of the `h5compound` class.

Data Members	
`Data`	Multidimensional array.
`MemberNames`	Text string specifying the names of fields in the structure
`Name`	Text string specifying the name of the object

The following are the methods of the `h5compound` class.

Methods	Description	Syntax

`hdf5.h5compound`	Constructs object of class `h5compound`.	`C = hdf5.h5compound;`
		`C = hdf5.h5compound(mName1,mName2,...)`
		
		where `C` is an `h5compound` object and `mName1` and `mName2` are text strings that specify field names. The constructor creates a corresponding data field for every member name.

`addMember`	Creates a new field in the structure.	`addMember(C, mName)` where `mName` is a text string that specifies the name of the field. This method automatically creates a corresponding data field for the new member name.
`setMember`	Sets the value of the `Data` element associated with a particular field.	`setData(C, mName, mData)` where `C` is an `h5compound` object, `mName` is the name of a field in the object, and `mdata` is the value you want to assign to the field. `mData` can be numeric or an HDF5 data type.
`setMemberNames`	Specifies the names of fields in the structure.	`setData(C, mName1, mName2,...)` where `C` is an `h5compound` object and `mName1` and `mName2` are text strings that specify field names. The constructor creates a corresponding data field for every member name.
`setName`	Sets the value of the object's `Name` member.	`setName(C, name)` where `arr` is an `h5compound` object and `name` is a string or cell array.

h5enum

The HDF5 `h5enum` class defines an enumerated types, where you can specify the enumerations (text strings) and the values the represent. The following are the data members of the `h5enum` class.

Data Members	
`Data`	Multidimensional array
`EnumNames`	Text string specifying the enumerations, that is, the text strings that represent values.
`EnumValues`	The values associated with enumerations
`Name`	Text string specifying the name of the object

The following are the methods of the `h5enum` class.

Methods	Description	Syntax

`hdf5.h5enum`	Constructs object of class `h5enum`.	`E = hdf5.h5enum;`
		`E = hdf5.h5enum(eNames, eVals)`
		
		where `E` is an `h5enum` object, `eNames` is a cell array of strings, and `eVals` is vector of integers. `eNames` and `eVals` must have the same number of elements.

`defineEnum`	Defines the set of enumerations with the integer values they represent.	`detineEnum(E, eNames, eVals)` where `E` is an `h5enum` object, `eNames` is a cell array of strings, and `eVals` is vector of integers. `eNames` and `eVals` must have the same number of elements.
`getString`	Returns data as enumeration's values, not integer values	`enumdata = getString(E)` where `enumdata` is a cell array of strings and `E` is an `h5enum` object.

Methods	Description	Syntax
setData	Sets the value of the object's Data member.	setData(E, eData) where E is an h5enum object and eData is a vector of integers.
setEnumNames	Specifies the enumerations.	setEnumNames(E, eNames) where E is an h5enum object and eNames is a cell array of strings.
setEnumValues	Specifies the value associated with each enumeration.	setEnumValues(E, eVals) where E is an h5enum object and eVals is a vector of integers.
setName	Sets the value of the object's Name member.	setName(E, name) where E is an h5enum object and name is a string or cell array.

h5string

The HDF5 h5string class associates a name with an text string and provides optional padding behavior. The following are the data members of the h5string class.

Data Members	
Data	Text string
Length	Scalar value
Name	Text string specifying the name of the object
Padding	Type of padding to use: 'spacepad','nullterm', or 'nullpad'

The following are the methods of the `h5string` class.

Methods	Description	Syntax

`hdf5.h5string`	Constructs object of class `h5string`.	`str = hdf5.h5string;`
		`str = hdf5.h5string(data)`
		`str = hdf5.h5string(data, padType)`
		
		where `str` is an `h5string` object, `data` is a text string, and `padType` is a text string specifying one of the supported pad types.

`setData`	Sets the value of the object's `Data` member.	`setData(str, data)` where `str` is an `h5string` object and `data` is a text string.
`setLength`	Sets the value of the object's `Length` member.	`setLength(str, lenVal)` where `str` is an `h5string` object and `lenVal` is a scalar.
`setName`	Sets the value of the object's `Name` member.	`setName(str, name)` where `str` is an `h5string` object and `name` is a string or cell array.
`setPadding`	Specifies the value of the object's `Padding` member.	`setData(str, padType)` where `str` is an `h5string` object and `padType` is a text string specifying one of the supported pad types.

h5vlen

The HDF5 `h5vlen` class associates a name with an array. The following are the data members of the `h5vlen` class.

Data Members	
`Data`	Multidimensional array
`Name`	Text string specifying the name of the object

The following are the methods of the `h5vlen` class.

Methods	Description	Syntax

`hdf5.h5vlen`	Constructs object of class `h5vlen`.	`V = hdf5.h5vlen;`
		`V = hdf5.h5vlen(data)`
		
		where `V` is `h5vlen` object and `data` can be a scalar, vector, text string, cell array, or an HDF5 data type.

`setData`	Sets the value of the object's `Data` member.	`setData(V, data)` where `V` is `h5vlen` object and `data` can be a scalar, vector, text string, cell array, or an HDF5 data type.
`setName`	Sets the value of the object's `Name` member.	`setName(V, name)` where `name` is a string or cell array.

Remarks

The `hdf5read` function uses the HDF5 data type classes when the data it is reading from the HDF5 file cannot be represented in the workspace using a native MATLAB data type. For example, if an HDF5 file contains a data set made up of an enumerated data type which cannot be represented in MATLAB, `hdf5read` uses the HDF5 `h5enum` class to represent the data. An `h5enum` object has data members that store the enumerations (text strings), their corresponding values, and the enumerated data.

You might also need to use these HDF5 data type classes when using the `hdf5write` function to write data from the MATLAB workspace to an HDF5 file. By default, `hdf5write` can convert most MATLAB data to appropriate HDF5 data types. However, if this default data type mapping is not suitable, you can create HDF5 data types directly.

Examples

HDF5 Array Object Example

1. Create an array in the MATLAB workspace.

```
data = magic(5);
```

2. Create an HDF5 `h5array` object, passing the MATLAB array as the only argument to the constructor.

```
dset = hdf5.h5array(data)

hdf5.h5array:

    Name: ''
    Data: [5x5 double]
```

3. Assign a name to the object.

```
dset.setName('my numeric array data set')
```

HDF5 Compound Object Example

1. Create several variables in the MATLAB workspace.

```
data = magic(5);
str = 'a text string';
```

2. Create an HDF5 `h5compound` object, specifying member names. The method creates corresponding `Data` fields for each member name.

```
dset2 = hdf5.h5compound('temp1','temp2','temp3')

Adding member "temp1"
Adding member "temp2"
Adding member "temp3"

hdf5.h5compound:

           Name: ''
```

```
           Data: {[]  []  []}
    MemberNames: {'temp1'  'temp2'  'temp3'}
```

3 Set the values of the members.

```
setMember(dset2,'temp1',89)
setMember(dset2,'temp2',95)
setMember(dset2,'temp3',108)

dset2

hdf5.h5compound:

           Name: ''
           Data: {[89]  [95]  [108]}
    MemberNames: {'temp1'  'temp2'  'temp3'}
```

HDF5 Enumerated Object Example

1 Create an HDF5 `h5enum` object.

```
enum_obj = hdf5.h5enum;
```

2 Define the enumerations and their corresponding values. The values must be integers.

```
enum_obj.defineEnum({'RED' 'GREEN' 'BLUE'}, uint8([1 2 3]));
```

`enum_obj` now contains the definition of the enumeration that associates the names `RED`, `GREEN`, and `BLUE` with the numbers 1, 2, and 3.

3 Add enumerated data to the object.

```
enum_obj.setData(uint8([2 1 3 3 2 3 2 1]));
```

4 Use the `h5enum` `getString` method to read the data as enumerated values, rather than integers.

```
vals = enum_obj.getString

vals =

  Columns 1 through 7

    'GREEN'    'RED'    'BLUE'    'BLUE'    'GREEN'    'BLUE'    'GREEN'
```

```
  Column 8

    'RED'
```

HDF5 h5string Object Example

Create an HDF5 string object.

```
hdf5.h5vlen({0 [0 1] [0 2] [0:10]})

hdf5.h5vlen:

    Name: ''
    Data: [0 0 1 0 2 0 1 2 3 4 5 6 7 8 9 10]
```

HDF5 h5string Object Example

Create an HDF5 `h5vlen` object.

```
hdf5.h5vlen({0 [0 1] [0 2] [0:10]})

hdf5.h5vlen:

    Name: ''
    Data: [0 0 1 0 2 0 1 2 3 4 5 6 7 8 9 10]
```

See Also

`hdf5read`, `hdf5write`

Purpose

Return information about an HDF5 file

Syntax

```
fileinfo = hdf5info(filename)
fileinfo = hdf5info(filename,'ReadAttributes',BOOL)
```

Description

`S = hdf5info(filename)` returns a structure `fileinfo` whose fields contain information about the contents of the HDF5 file `filename`. `filename` is a string that specifies the name of the HDF5 file.

`S = hdf5info(...,'ReadAttributes',BOOL)` specifies whether `hdf5info` returns the values of the attributes or just information describing the attributes. By default, `hdf5info` reads in attribute values (`BOOL` = `true`).

Examples

To find out about the contents of the HDF5 file, look at the `GroupHierarchy` field returned by `hdf5info`.

```
fileinfo = hdf5info('example.h5')

fileinfo =

          Filename: 'example.h5'
        LibVersion: '1.4.5'
            Offset: 0
          FileSize: 8172
    GroupHierarchy: [1x1 struct]
```

To probe further into the hierarchy, keep examining the `Groups` field.

```
toplevel = fileinfo.GroupHierarchy

toplevel =

      Filename: [1x64 char]
          Name: '/'
        Groups: [1x2 struct]
      Datasets: []
     Datatypes: []
         Links: []
    Attributes: [1x2 struct]
```

See also

hdf5read, hdf5write, hdfinfo

Purpose

Read data from an HDF5 file

Syntax

```
data = hdf5read(filename,datasetname)
attr = hdf5read(filename,attributename)
[data, attr] = hdf5read(...,'ReadAttributes',BOOL)
data = hdf5read(hinfo)
```

Description

`data = hdf5read(filename,datasetname)` reads all the data in the data set `datasetname` that is stored in the HDF5 file `filename` and returns it in the variable `data`. To determine the names of data sets in an HDF5 file, use the `hdf5info` function.

The return value, `data`, is a multidimensional array. `hdf5read` maps HDF5 data types to native MATLAB data types, whenever possible. If it cannot represent the data using MATLAB data types, `hdf5read` uses one of the HDF5 data type objects. For example, if an HDF5 file contains a data set made up of an enumerated data type, `hdf5read` uses the `hdf5.h5enum` object to represent the data in the MATLAB workspace. The `hdf5.h5enum` object has data members that store the enumerations (names), their corresponding values, and the enumerated data. For more information about the HDF5 data type objects, see the `hdf5` reference page.

`attr = hdf5read(filename,attributename)` reads all the metadata in the attribute `attributename`, stored in the HDF5 file `filename`, and returns it in the variable `attr`. To determine the names of attributes in an HDF5 file, use the `hdf5info` function.

`[data,attr] = hdf5read(...,'ReadAttributes',BOOL)` reads all the data as well as all of the associated attribute information contained within that data set. By default, `BOOL` is false.

`data = hdf5read(hinfo)` reads all of the data in the data set specified in the structure `hinfo` and returns it in the variable `data`. The `hinfo` structure is extracted from the output returned by `hdf5info` which specifies an HDF5 file and a specific data set.

Examples

Read a data set specified by an `hinfo` structure. Use `hdf5info` to get information about the HDF5 file.

```
hinfo = hdf5info('example.h5');
```

Use `hdf5read` to read the data set specified by the `info` structure.

```
dset = hdf5read(hinfo.GroupHierarchy.Groups(2).Datasets(1));
```

See Also

`hdf5`, `hdf5info`, `hdf5write`

Purpose

Write a Hierarchical Data Format (HDF) Version 5 file

Syntax

```
hdf5write(filename,location,dataset)
hdf5write(filename,details,dataset)
hdf5write(filename,details1,dataset1,details2,dataset2,...)
hdf5write(filename,...,'WriteMode',mode,...)
```

Description

`hdf5write(filename,location,dataset)` writes the data `dataset` to the HDF5 file named `filename`. If `filename` does not exist, `hdf5write` creates it. If `filename` exists, `hdf5write` overwrites the existing file, by default, but you can also append data to an existing file using an optional syntax.

`location` defines where to write the data set in the file. HDF5 files are organized in a hierarchical structure similar to a UNIX directory structure. `location` is a string that resembles a UNIX path.

`hdf5write` maps the data in `dataset` to HDF5 data types according to rules outlined below.

`hdf5write(filename,details,dataset)` writes `dataset` to `filename` using the values in the `details` structure. For a data set, the `details` structure can contain the following fields.

Field Name	Description	Data Type
`Location`	Location of the data set in the file	Character array
`Name`	Name to attach to the data set	String

`hdf5write(filename,details,attribute)` writes the metadata `attribute` to `filename` using the values in the `details` structure. For an attribute, the `details` structure can contain following fields.

Field Name	Description	Data Type
`AttachedTo`	Location of the object this attribute modifies	Structure array
`AttachType`	String that identifies what kind of object this attribute modifies; possible values are `'group'` and `'dataset'`	String
`Name`	Name to attach to the data set	Character array

`hdf5write(filename, details1, dataset1, details2, dataset2,...)` writes multiple data sets and associated attributes to `filename` in one operation. Each data set and attribute must have an associated `details` structure.

`hdf5write(filename,...,'WriteMode',mode,...)` specifies whether `hdf5write` overwrites the existing file (the default) or appends data sets and attributes to the file. Possible values for mode are `'overwrite'` and `'append'`.

Data Type Mappings

If the data being written to the file is composed of HDF5 objects, `hdf5write` uses the same data type when writing to the file. For `HDF5.h5enum` objects, the size and dimensions of the data set in the HDF5 file, called the *dataspace* in HDF5 terminology, is the same as the object's Data field.

Field Name	Description	Data Type
`AttachedTo`	Location of the object this attribute modifies	Structure array
`AttachType`	String that identifies what kind of object this attribute modifies. Possible values are `'group'` and `'dataset'`	String
`Name`	Name to attach to the data set	Character array

If the data in the workspace that is being written to the file is a MATLAB data type, `hdf5write` uses the following rules when translating MATLAB data into HDF5 data objects.

MATLAB Data Type	HDF5 Data Set or Attribute
Numeric	Corresponding HDF5 native datatype. For example, if the workspace data type is `uint8`, the `hdf5write` function writes the data to the file as 8-bit integers. The size of the HDF5 dataspace is the same size as the MATLAB array.
String	Single, null-terminated string
Cell array of strings	Multiple, null-terminated strings, each the same length. Length is determined by the length of the longest string in the cell array. The size of the HDF5 dataspace is the same size as the cell array.
Cell array of numeric data	Numeric array, the same dimensions as the cell array. The elements of the array must have all have the same size and type. The data type is determined by the first element in the cell array.
Structure array	HDF5 compound type. Individual fields in the structure employ the same data translation rules for individual data types. For example, a cell array of strings becomes a multiple, null-terminated strings.

Examples

Write a 5-by-5 data set of `uint8` values to the root group.

```
hdf5write('myfile.h5', '/dataset1', uint8(magic(5)))
```

Write a 2-by-2 string data set in a subgroup.

```
dataset = {'north', 'south'; 'east', 'west'};
hdf5write('myfile2.h5', '/group1/dataset1.1', dataset);
```

Write a data set and attribute to an existing group.

```
dset = single(rand(10,10));
dset_details.Location = '/group1/dataset1.2';
```

```
dset_details.Name = 'Random';

attr = 'Some random data';
attr_details.AttachedTo = '/group1/dataset1.2';
attr_details.AttachType = 'dataset';

hdf5write('myfile2.h5', dset_details, dset, ...
            attr_details, attr, 'WriteMode', 'append');
```

Write a data set using objects.

```
dset = hdf5.h5array(magic(5));
hdf5write('myfile3.h5', '/g1/objects', dset);
```

See Also

hdf5, hdf5read, hdf5info

Purpose Return information about an HDF or HDF-EOS file

Syntax

```
S = hdfinfo(filename)
S = hdfinfo(filename,mode)
```

Description `S = hdfinfo(filename)` returns a structure `S` whose fields contain information about the contents of an HDF or HDF-EOS file. `filename` is a string that specifies the name of the HDF file.

`S = hdfinfo(filename,mode)` reads the file as an HDF file, if `mode` is `'hdf'`, or as an HDF-EOS file, if `mode` is `'eos'`. If `mode` is `'eos'`, only HDF-EOS data objects are queried. To retrieve information on the entire contents of a file containing both HDF and HDF-EOS objects, `mode` must be `'hdf'`.

Note `hdfinfo` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files.

The set of fields in the returned structure `S` depends on the individual file. Fields that can be present in the `S` structure are shown in the following table.

HDF Object Fields

Mode	Field Name	Description	Return Type
`HDF`	`Attributes`	Attributes of the data set	Structure array
	`Description`	Annotation description	Cell array
	`Filename`	Name of the file	String
	`Label`	Annotation label	Cell array
	`Raster8`	Description of 8-bit raster images	Structure array
	`Raster24`	Description of 24-bit raster images	Structure array
	`SDS`	Description of scientific data sets	Structure array
	`Vdata`	Description of Vdata sets	Structure array
	`Vgroup`	Description of Vgroups	Structure array
`EOS`	`Filename`	Name of the file	String
	`Grid`	Grid data	Structure array
	`Point`	Point data	Structure array
	`Swath`	Swath data	Structure array

Those fields in the table above that contain structure arrays are further described in the tables shown below.

Fields Common to Returned Structure Arrays

Structure arrays returned by `hdfinfo` contain some common fields. These are shown in the table below. Not all structure arrays will contain all of these fields.

Common Fields

Field Name	Description	Data Type
`Attributes`	Data set attributes. Contains fields `Name` and `Value`.	Structure array
`Description`	Annotation description	Cell array
`Filename`	Name of the file	String
`Label`	Annotation label	Cell array
`Name`	Name of the data set	String
`Rank`	Number of dimensions of the data set	Double
`Ref`	Data set reference number	Double
`Type`	Type of HDF or HDF-EOS object	String

Fields Specific to Certain Structures

Structure arrays returned by `hdfinfo` also contain fields that are unique to each structure. These are shown in the tables below.

Fields of the Attribute Structure

Field Name	Description	Data Type
`Name`	Attribute name	String
`Value`	Attribute value or description	Numeric or string

Fields of the Raster8 and Raster24 Structures

Field Name	Description	Data Type
`HasPalette`	`1` (`true`) if the image has an associated palette, otherwise `0` (`false`) (8-bit only)	Logical
`Height`	Height of the image, in pixels	Number
`Interlace`	Interlace mode of the image (24-bit only)	String
`Name`	Name of the image	String
`Width`	Width of the image, in pixels	Number

Fields of the SDS Structure

Field Name	Description	Data Type
`DataType`	Data precision	String
`Dims`	Dimensions of the data set. Contains fields `Name`, `DataType`, `Size`, `Scale`, and `Attributes`. `Scale` is an array of numbers to place along the dimension and demarcate intervals in the data set.	Structure array
`Index`	Index of the SDS	Number

Fields of the Vdata Structure

Field Name	Description	Data Type
`DataAttributes`	Attributes of the entire data set. Contains fields `Name` and `Value`.	Structure array
`Class`	Class name of the data set	String
`Fields`	Fields of the Vdata. Contains fields `Name` and `Attributes`.	Structure array

Fields of the Vdata Structure

Field Name	Description	Data Type
`NumRecords`	Number of data set records	Double
`IsAttribute`	`1` (`true`) if Vdata is an attribute, otherwise `0` (`false`)	Logical

Fields of the Vgroup Structure

Field Name	Description	Data Type
`Class`	Class name of the data set	String
`Raster8`	Description of the 8-bit raster image	Structure array
`Raster24`	Description of the 24-bit raster image	Structure array
`SDS`	Description of the Scientific Data sets	Structure array
`Tag`	Tag of this Vgroup	Number
`Vdata`	Description of the Vdata sets	Structure array
`Vgroup`	Description of the Vgroups	Structure array

Fields of the Grid Structure

Field Name	Description	Data Type
`Columns`	Number of columns in the grid	Number
`DataFields`	Description of the data fields in each `Grid` field of the grid. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, `FillValue`, and `TileDims`.	Structure array
`LowerRight`	Lower right corner location, in meters	Number
`Origin Code`	Origin code for the grid	Number
`PixRegCode`	Pixel registration code	Number

Fields of the Grid Structure

Field Name	Description	Data Type
`Projection`	Projection code, zone code, sphere code, and projection parameters of the grid. Contains fields `ProjCode`, `ZoneCode`, `SphereCode`, and `ProjParam`.	Structure
`Rows`	Number of rows in the grid	Number
`UpperLeft`	Upper left corner location, in meters	Number

Fields of the Point Structure

Field Name	Description	Data Type
`Level`	Description of each level of the point. Contains fields `Name`, `NumRecords`, `FieldNames`, `DataType`, and `Index`.	Structure

Fields of the Swath Structure

Field Name	Description	Data Type
`DataFields`	Data fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array
`GeolocationFields`	Geolocation fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array
`IdxMapInfo`	Relationship between indexed elements of the geolocation mapping. Contains fields `Map` and `Size`.	Structure
`MapInfo`	Relationship between data and geolocation fields. Contains fields `Map`, `Offset`, and `Increment`.	Structure

Examples

To retrieve information about the file `example.hdf`,

```
fileinfo = hdfinfo('example.hdf')

fileinfo =
    Filename: 'example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

And to retrieve information from this about the scientific data set in `example.hdf`,

```
sds_info = fileinfo.SDS

sds_info =
       Filename: 'example.hdf'
           Type: 'Scientific Data Set'
           Name: 'Example SDS'
           Rank: 2
       DataType: 'int16'
     Attributes: []
           Dims: [2x1 struct]
          Label: {}
    Description: {}
          Index: 0
```

See Also

`hdfread`, `hdf`

hdfread

Purpose

Extract data from an HDF or HDF-EOS file

Syntax

```
data = hdfread(filename, dataset)
data = hdfread(hinfo)
data = hdfread(...,param1,value1,param2,value2,...)
[data,map] = hdfread(...)
```

Description

`data = hdfread(filename, dataset)` returns all the data in the specified data set `dataset` from the HDF or HDF-EOS file `filename`. To determine the names of the data sets in an HDF file, use the `hdfinfo` function. The information returned by `hdfinfo` contains structures describing the data sets contained in the file. You can extract one of these structures and pass it directly to `hdfread`.

Note `hdfread` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files.

`data = hdfread(hinfo)` returns all the data in the data set specified in the structure `hinfo`. The `hinfo` structure can be extracted from the data returned by the `hdfinfo` function.

`data = hdfread(...,param1,value1,param2,value2,...)` returns subsets of the data according to the specified parameter and value pairs. See the tables below to find the valid parameters and values for different types of data sets.

`[data,map] = hdfread(...)` returns the image `data` and the colormap `map` for an 8-bit raster image.

Subsetting Parameters

The following tables show the subsetting parameters that can be used with the `hdfread` function for certain types of HDF data. These data types are

- HDF Scientific Data (SD)
- HDF Vdata (V)
- HDF-EOS Grid Data
- HDF-EOS Point Data
- HDF-EOS Swath Data

Note the following:

- If a parameter requires multiple values, the values must be stored in a cell array. For example, the 'Index' parameter requires three values: start, stride, and edge. Enclose these values in curly braces as a cell array.

```
hdfread(dataset_name, 'Index', {start,stride,edge})
```

- All values that are indices are 1-based.

Subsetting Parameters for HDF Scientific Data (SD) Data Sets

When you are working with HDF SD files, hdfread supports the parameters listed in this table.

Parameter	Description
'Index'	Three-element cell array, {start,stride,edge}, specifying the location, range, and values to be read from the data set • start — A 1-based array specifying the position in the file to begin reading Default: 1, start at the first element of each dimension. The values specified must not exceed the size of any dimension of the data set. • stride — A 1-based array specifying the interval between the values to read Default: 1, read every element of the data set. • edge — A 1-based array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions

For example, this code reads the data set Example SDS from the HDF file example.hdf. The 'Index' parameter specifies that hdfread start reading data at the beginning of each dimension, read until the end of each dimension, but only read every other data value in the first dimension.

```
hdfread('example.hdf','Example SDS', ...
                     'Index', {[], [2 1], []})
```

Subsetting Parameters for HDF Vdata Sets

When you are working with HDF Vdata files, `hdfread` supports these parameters.

Parameter	Description
`'Fields'`	Text string specifying the name of the data set field to be read from. When specifying multiple field names, use a comma-separated list.
`'FirstRecord'`	1-based number specifying the record from which to begin reading
`'NumRecords'`	Number specifying the total number of records to read

For example, this code reads the Vdata set `Example Vdata` from the HDF file `example.hdf`.

```
hdfread('example.hdf', 'Example Vdata', 'FirstRecord', 400,
'NumRecords', 50)
```

Subsetting Parameters for HDF-EOS Grid Data

When you are working with HDF-EOS grid data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive parameters — You can only specify one of these parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
'Fields'	String naming the data set field to be read. You can specify only one field name for a Grid data set.
Mutually Exclusive Optional Parameters	
'Index'	Three-element cell array, {start,stride,edge}, specifying the location, range, and values to be read from the data set • start — An array specifying the position in the file to begin reading Default: 1, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. • stride — An array specifying the interval between the values to read Default: 1, read every element of the data set. • edge — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions
'Interpolate'	Two-element cell array, {longitude,latitude}, specifying the longitude and latitude points that define a region for bilinear interpolation. Each element is an N-length vector specifying longitude and latitude coordinates.
'Pixels'	Two-element cell array, {longitude,latitude}, specifying the longitude and latitude coordinates that define a region. Each element is an N-length vector specifying longitude and latitude coordinates. This region is converted into pixel rows and columns with the origin in the upper left corner of the grid. Note: This is the pixel equivalent of reading a 'Box' region.

Parameter	Description
'Tile'	Vector specifying the coordinates of the tile to read, for HDF-EOS Grid files that support tiles
Optional Parameters	
'Box'	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are each two-element vectors specifying longitude and latitude coordinates.
'Time'	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and end-point for a period of time
'Vertical'	Two-element cell array, `{dimension, range}` • `dimension` — String specifying the name of the data set field to be read from. You can specify only one field name for a Grid data set. • `range` — Two-element array specifying the minimum and maximum range for the subset. If `dimension` is a dimension name, then `range` specifies the range of elements to extract. If `dimension` is a field name, then `range` specifies the range of values to extract. `'Vertical'` subsetting can be used alone or in conjunction with `'Box'` or `'Time'`. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to `hdfread`.

For example,

```
hdfread(grid_dataset, 'Fields', fieldname, ...
                'Vertical', {dimension, [min, max]})
```

Subsetting Parameters for HDF-EOS Point Data

When you are working with HDF-EOS Point data, `hdfread` has two required parameters and three optional parameters.

Parameter	Description
Required Parameters	
`'Fields'`	String naming the data set field to be read. For multiple field names, use a comma-separated list.
`'Level'`	1-based number specifying which level to read from in an HDF-EOS Point data set
Optional Parameters	
`'Box'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are each two-element vectors specifying longitude and latitude coordinates.
`'RecordNumbers'`	Vector specifying the record numbers to read
`'Time'`	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and endpoint for a period of time

For example,

```
hdfread(point_dataset, 'Fields', {field1, field2}, ...
'Level', level, 'RecordNumbers', [1:50, 200:250])
```

Subsetting Parameters for HDF-EOS Swath Data

When you are working with HDF-EOS Swath data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive

You can only use one of the mutually exclusive parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
`'Fields'`	String naming the data set field to be read. You can specify only one field name for a Swath data set.
Mutually Exclusive Optional Parameters	
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set • `start` — An array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. • `stride` — An array specifying the interval between the values to read Default: `1`, read every element of the data set. • `edge` — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions
`'Time'`	Three-element cell array, `{start, stop, mode}`, where `start` and `stop` specify the beginning and the endpoint for a period of time, and `mode` is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (`mode='midpoint'`). • Either endpoint is within the box (`mode='endpoint'`). • Any point is within the box (`mode='anypoint'`).

Parameter	Description
Optional Parameters	
'Box'	Three-element cell array, {longitude, latitude, mode} specifying the longitude and latitude coordinates that define a region. longitude and latitude are two-element vectors that specify longitude and latitude coordinates. mode is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (mode='midpoint'). • Either endpoint is within the box (mode='endpoint'). • Any point is within the box (mode='anypoint').
'ExtMode'	String specifying whether geolocation fields and data fields must be in the same swath (mode='internal'), or can be in different swaths (mode='external') Note: mode is only used when extracting a time period or a region.
'Vertical'	Two-element cell array, {dimension, range} • dimension is a string specifying either a dimension name or field name to subset the data by. • range is a two-element vector specifying the minimum and maximum range for the subset. If dimension is a dimension name, then range specifies the range of elements to extract. If dimension is a field name, then range specifies the range of values to extract. 'Vertical' subsetting can be used alone or in conjunction with 'Box' or 'Time'. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to hdfread.

For example,

```
hdfread('example.hdf',swath_dataset, 'Fields', fieldname, ...
'Time', {start, stop, 'midpoint'})
```

Examples

Importing a Data Set by Name

When you know the name of the data set, you can refer to the data set by name in the hdfread command. To read a data set named 'Example SDS', use

```
data = hdfread('example.hdf', 'Example SDS')
```

Importing a Data Set Using the Hinfo Structure

When you don't know the name of the data set, follow this procedure.

1 Use hdfinfo first to retrieve information on the data set.

```
fileinfo = hdfinfo('example.hdf')
fileinfo =

    Filename: 'N:\toolbox\matlab\demos\example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

2 Extract the structure containing information about the particular data set you want to import from fileinfo.

```
sds_info = fileinfo.SDS
sds_info =

       Filename: 'N:\toolbox\matlab\demos\example.hdf'
           Type: 'Scientific Data Set'
           Name: 'Example SDS'
           Rank: 2
       DataType: 'int16'
     Attributes: []
           Dims: [2x1 struct]
          Label: {}
    Description: {}
          Index: 0
```

3 Pass this structure to hdfread to import the data in the data set.

```
data = hdfread(sds_info)
```

Importing a Subset of a Data Set

You can check the size of the information returned as follows.

```
sds_info.Dims.Size
```

```
ans =
    16
ans =
     5
```

Using `hdfread` parameter/value pairs, you can read a subset of the data in the data set. This example specifies a starting index of `[3 3]`, an interval of 1 between values (`[]` meaning the default value of 1), and a length of 10 rows and 2 columns.

```
data = hdfread(sds_info, 'Index', {[3 3],[],[10 2]});

data(:,1)
ans =
     7
     8
     9
    10
    11
    12
    13
    14
    15
    16

data(:,2)
ans =
     8
     9
    10
    11
    12
    13
    14
    15
    16
    17
```

Importing Fields from a Vdata Set

This example retrieves information from `example.hdf` first, and then reads two fields of the data, `Idx` and `Temp`.

```
info = hdfinfo('example.hdf');

data = hdfread(info.Vdata,...
   'Fields',{'Idx','Temp'})

data =
    [1x10 int16]
    [1x10 int16]

index = data{1,1};
temp = data{2,1};

temp(1:6)
ans =
     0    12     3     5    10    -1
```

See Also

`hdfinfo`, `hdf`

Purpose

Browse and import data from HDF or HDF-EOS files

Syntax

```
hdftool
hdftool(filename)
h = hdfinfo(...)
```

Description

`hdftool` starts the HDF Import Tool, a graphical user interface used to browse the contents of HDF and HDF-EOS files and import data and data subsets from these files. When you use `hdftool` without an argument, the tool displays the **Choose an HDF file** dialog box. Select an HDF or HDF-EOS file to start the HDF Import Tool.

`hdftool(filename)` opens the HDF or HDF-EOS file `filename` in the HDF Import Tool.

`h = hdftool(...)` returns a handle `h` to the HDF Import Tool. To close the tool from the command line, use `dispose(h)`.

You can run only one instance of the HDF Import Tool during a MATLAB session; however, you can open multiple files.

Using the HDF Import Tool, HDF-EOS files can be viewed as either HDF-EOS files or as HDF files. HDF files can only be viewed as HDF files.

Example

```
hdftool('example.hdf');
```

See Also

`hdf`, `hdfinfo`, `hdfread`, `uiimport`

help

Purpose Display help for MATLAB functions in Command Window

Syntax

```
help
help /
help functionname
help toolboxname
help toolboxname/functionname
help classname.methodname
help classname
help syntax
t = help('topic')
```

Description

`help` lists all primary help topics in the Command Window. Each main help topic corresponds to a directory name on the MATLAB search path.

`help /` lists all operators and special characters, along with their descriptions.

`help functionname` displays M-file help, which is a brief description and the syntax for `functionname`, in the Command Window. The output includes a link to `doc functionname`, which displays the reference page in the Help browser, often providing additional information. Output also includes see also links, which display help in the Command Window for related functions. If `functionname` is overloaded, that is, appears in multiple directories on the search path, `help` displays the M-file help for the first `functionname` found on the search path, and displays a hyperlinked list of the overloaded functions and their directories. If `functionname` is also the name of a toolbox, help also displays the list of subdirectories and functions in the toolbox.

`help toolboxname` displays the contents file for the specified directory named `toolboxname`. It is not necessary to give the full pathname of the directory; the last component, or the last several components, are sufficient. If `toolboxname` is also a function name, help also displays the M-file help for the function `toolboxname`.

`help toolboxname/functionname` displays the M-file help for `functionname`, which resides in the `toolboxname` directory. Use this form to get direct help for an overloaded function.

`help classname.methodname` displays help for the method, `methodname`, of the fully qualified class, `classname`. If you do not know the fully qualified class for the method, use `class(obj)`, where `methodname` is of the same class as the object `obj`.

`help classname` displays help for the fully qualified class, `classname`.

`help` **`syntax`** displays M-file help describing the syntax used in MATLAB commands and functions.

`t = help('topic')` returns the help text for `topic` as a string, with each line separated by `/n`, where `topic` is any allowable argument for `help`.

Note M-file help displayed in the Command Window uses all uppercase characters for the function and variable names to make them stand out from the rest of the text. When typing function names, however, use lowercase characters. Some functions for interfacing to Java do use mixed case; the M-file help accurately reflects that and you should use mixed case when typing them. For example, the `javaObject` function uses mixed case.

Remarks

To prevent long descriptions from scrolling off the screen before you have time to read them, enter `more on`, and then enter the `help` statement.

Creating Online Help for Your Own M-Files

The MATLAB 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 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 the MATLAB 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, the function `soundspeed.m`, begins with

```
function c=soundspeed(s,t,p)
% soundspeed computes the speed of sound in water
% where c is the speed of sound in water in m/s

t = 0:.1:35;
```

When you execute `help soundspeed`, MATLAB displays

```
soundspeed computes the speed of sound in water
where c is the speed of sound in water in m/s
```

These lines are the first block of contiguous comment lines. After the first contiguous comment lines, enter an executable statement or blank line, which effectively ends the help section. Any later comments in the M-file do not appear when you type `help` for the function.

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. For the `soundspeed` example, the H1 line is

```
% soundspeed computes speed of sound in water
```

Use the Help Report to help you create and manage M-file help for your own files.

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, it is a good practice to create `Contents.m` files for them too. Use the Contents Report to help you create and maintain your own `Contents.m` files.

Examples

`help close` displays help for the `close` function.

`help database/close` displays help for the `close` function in the Database Toolbox.

`help datafeed` displays help for the Datafeed Toolbox

`help database` lists the functions in the Database Toolbox and displays help for the `database` function, because there is a function and a toolbox called `database`.

`help general` lists all functions in the directory `$matlabroot/toolbox/matlab/general`. This illustrates how to specify a relative partial pathname, rather than a full pathname.

`help embedded.fi.lsb` displays help for the `lsb` method of the `fi` class in the Fixed-Point Toolbox. Running `a = fi(pi); class(a)`, for example, returns `embedded.fi`, which is the fully qualified class for the `lsb` method.

`help embedded.fi` displays help for the `fi` class in the Fixed-Point Toolbox. This is actually the help for the class's object constructor, in this case, `fi`.

`t = help( close')` gets help for the function `close` and stores it as a string in `t`.

See Also

`class`, `doc`, `docsearch`, `helpbrowser`, `helpwin`, `lookfor`, `more`, `partialpath`, `path`, `what`, `which`, `whos`

helpbrowser

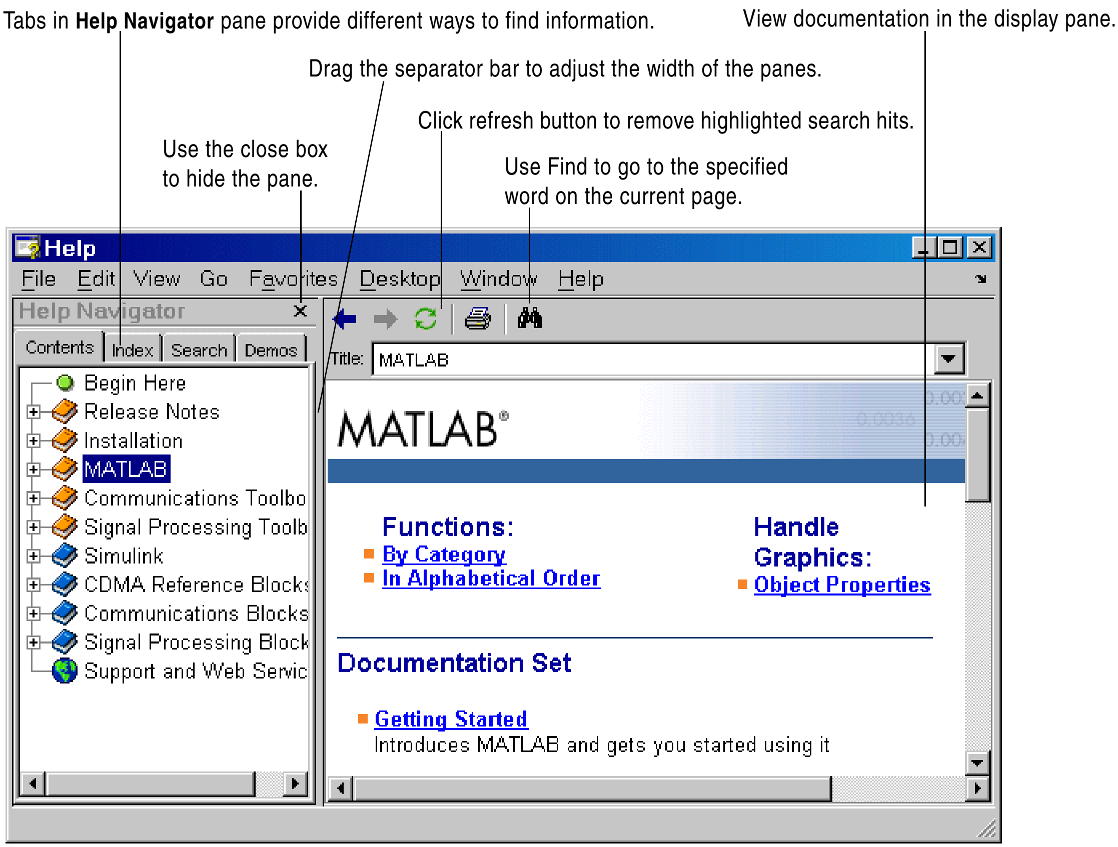
Purpose Display Help browser for access to full online documentation and demos

Graphical Interface As an alternative to the `helpbrowser` function, select **Help** from the **Desktop** 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 documentation, including reference pages and user guides. If the Help browser was previously opened in the current session, `helpbrowser` shows the last page viewed; otherwise it shows the **Begin Here** page. For details, see the Help Browser documentation.



See Also

`doc`, `docopt`, `docsearch`, `help`, `helpdesk`, `helpwin`, `lookfor`, `web`

helpdesk

Purpose Display 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—use the `doc` or `helpbrowser` function instead.

See Also `doc`, `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'`.

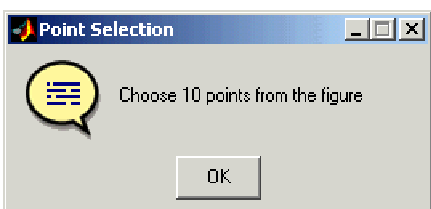
`h = helpdlg(...)` returns the handle of the dialog box.

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 you press 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`

"Predefined Dialog Boxes" for related functions

Purpose

Provide access to and display 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, displayed in the Help browser. You cannot follow links in the `helpwin` list of functions if MATLAB is busy (for example, running a program).

`helpwin topic` displays help information for the topic in the Help browser. If `topic` is a directory, it displays all functions in the directory. The directory name cannot include spaces. If `topic` is a function, `helpwin` displays M-file help for that function in the Help browser. From the page, you can access a list of directories (**Default Topics** link) as well as the reference page help for the function (**Go to online doc** link). You cannot follow links in the `helpwin` list of functions if MATLAB is busy (for example, running a program).

Examples

Typing

```
helpwin datafun
```

displays the functions in the `datafun` directory and a brief description of each.

Typing

```
helpwin fft
```

displays the M-file help for the `fft` function in the Help browser.

See Also

`doc`, `docopt`, `help`, `helpbrowser`, `lookfor`, `web`

Purpose

Hessenberg form of a matrix

Syntax

```
[P,H] = hess(A)
H = hess(A)
[AA,BB,Q,Z] = HESS(A,B)
```

Description

`H = hess(A)` finds `H`, the Hessenberg form of matrix `A`.

`[P,H] = hess(A)` produces a Hessenberg matrix `H` and a unitary matrix `P` so that `A = P*H*P'` and `P'*P = eye(size(A))`.

`[AA,BB,Q,Z] = HESS(A,B)` for square matrices `A` and `B`, produces an upper Hessenberg matrix `AA`, an upper triangular matrix `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA` and `Q*B*Z = BB`.

Definition

A Hessenberg matrix is zero below the first subdiagonal. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original, but less computation is needed to reveal them.

Examples

`H` is a 3-by-3 eigenvalue test matrix:

```
H =
    -149     -50    -154
     537     180     546
     -27      -9     -25
```

Its Hessenberg form introduces a single zero in the (3,1) position:

```
hess(H) =
    -149.0000     42.2037    -156.3165
    -537.6783    152.5511    -554.9272
            0      0.0728       2.4489
```

Algorithm

Inputs of Type Double

For inputs of type double, `hess` uses the following LAPACK routines to compute the Hessenberg form of a matrix:

Matrix A	Routine
Real symmetric	DSYTRD DSYTRD, DORGTR, (with output P)
Real nonsymmetric	DGEHRD DGEHRD, DORGHR (with output P)
Complex Hermitian	ZHETRD ZHETRD, ZUNGTR (with output P)
Complex non-Hermitian	ZGEHRD ZGEHRD, ZUNGHR (with output P)

Inputs of Type Single

For inputs of type `single`, `hess` uses the following LAPACK routines to compute the Hessenberg form of a matrix:

Matrix A	Routine
Real symmetric	SSYTRD SSYTRD, DORGTR, (with output P)
Real nonsymmetric	SGEHRD SGEHRD, SORGHR (with output P)
Complex Hermitian	CHETRD CHETRD, CUNGTR (with output P)
Complex non-Hermitian	CGEHRD CGEHRD, CUNGHR (with output P)

See Also

`eig`, `qz`, `schur`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

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 Convert IEEE hexadecimal string to double precision number

Syntax

```
n = hex2num(S)
```

Description

n = hex2num(S), where S is a 16 character string representing a hexadecimal number, returns the IEEE double-precision floating-point number n that it represents. Fewer than 16 characters are padded on the right with zeros. If S is a character array, each row is interpreted as a double-precision number.

NaNs, infinities and denorms are handled correctly.

Example

```
hex2num('400921fb54442d18')
```

returns Pi.

```
hex2num('bff')
```

returns

```
ans =

    -1
```

See Also num2hex, hex2dec, sprintf, format

hgexport

Purpose Export figure

Syntax

```
hgexport(fig, 'filename')
hgexport(fig, '-clipboard')
```

Description

`hgexport(h,filename)` writes figure `h` to the file `filename`.

`hgexport(fig,'-clipboard')` writes figure `h` to the Windows clipboard.

The format in which the figure is exported is determined by which renderer you use. The Painters renderer generates a metafile. The ZBuffer and OpenGL renderers generate a bitmap.

See Also `print`

Purpose

Create hggroup object

Syntax

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

Description

An hggroup object can be the parent of any axes children, including other hggroup objects. You can use hggroup objects to form a group of objects that can be treated as a single object with respect to the following cases:

- Visible — Setting the hggroup object's `Visible` property also sets each child object's `Visible` property to the same value.
- Selectable — Setting each hggroup child object's `HitTest` property to `off` enables you to select all children by clicking any child object.
- Current object — Setting each hggroup child object's `HitTest` property to `off` enables the hggroup object to become the current object when any child object is picked. See the next section for an example.

Examples

This example defines a callback for the `ButtonDownFcn` property of an hggroup object. In order for the hggroup to receive the mouse button down event that executes the `ButtonDownFcn` callback, the `HitTest` properties of all the line objects must be set to `off`. The event is then passed up the hierarchy to the hggroup.

The following function creates a random set of lines that are parented to an hggroup object. The subfunction `set_lines` defines a callback that executes when the mouse button is pressed over any of the lines. The callback simply increases the widths of all the lines by 1 with each button press.

Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

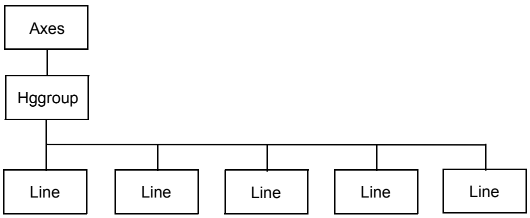
```
function doc_hggroup
hg = hggroup('ButtonDownFcn',@set_lines);
hl  = line(randn(5),randn(5),'HitTest','off','Parent',hg);

function set_lines(cb,eventdata)
hl = get(cb,'Children');% cb is handle of hggroup object
```

```
lw = get(hl,'LineWidth');% get current line widths
set(hl,{'LineWidth'},num2cell([lw{:}]+1,[5,1])')
```

Note that selecting any one of the lines selects all the lines. (To select an object, enable plot edit mode by selecting **Plot Edit** from the **Tools** menu.)

Instance Diagram for This Example

The following diagram shows in object hierarchy created by this example.



See Also

`hgtransform`

See Group Objects for more information and examples.

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

Hggroup Properties

Setting Default Properties

You can set default hggroup properties on the axes, figure, and root levels.

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

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

Property Name	Property Description	Property Value
Controlling the Appearance		
`Clipping`	Clipping to axes rectangle	Values: `on`, `off` Default: `on`
`EraseMode`	Method of drawing and erasing the hggroup object children (useful for animation)	Values: `normal`, `none`, `xor`, `background` Default: `normal`
`SelectionHighlight`	Hggroup object children are highlighted when selected (`Selected` property set to `on`).	Values: `on`, `off` Default: `on`
`Visible`	Makes the hggroup children visible or invisible	Values: `on`, `off` Default: `on`
Controlling Access to Objects		
`HandleVisibility`	Determines if and when the hggroup object's handle is visible to other functions	Values: `on`, `callback`, `off` Default: `on`
`HitTest`	Determines whether the hggroup object can become the current object (see the figure `CurrentObject` property)	Values: `on`, `off` Default: `on`
General Information About the Hggroup Object		
`Children`	Any axes child can be the child of an hggroup object.	Values: handles of objects
`Parent`	The parent of an hggroup object can be an axes, hggroup, or hgtransform object.	Value: object handle
`Selected`	Indicates whether the hggroup object is in a selected state	Values: `on`, `off` Default: `on`
`Tag`	User-specified label	Value: any string Default: `''` (empty string)

Property Name	Property Description	Property Value
Type	The type of graphics object (read only)	Value: the string 'hggroup'
UserData	User-specified data	Value: any matrix Default: [] (empty matrix)
Properties Related to Callback Routine Execution		
BeingDeleted	Query this property to see if object is being deleted.	Values: on | off Read only
BusyAction	Specifies how to handle callback routine interruption	Values: cancel, queue Default: queue
ButtonDownFcn	Defines callback routine that executes when mouse button is pressed over the hggroup object's children	Value: string or function handle Default: '' (empty string)
CreateFcn	Defines callback routine that executes when hggroup object is created	Value: string or function handle Default: '' (empty string)
DeleteFcn	Defines callback routine that executes when hggroup object is deleted (via close or delete)	Value: string or function handle Default: '' (empty string)
Interruptible	Determines whether callback routine can be interrupted	Value: on, off Default: on (can be interrupted)
UIContextMenu	Associates a context menu with the hggroup object	Value: handle of a Uicontextmenu

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands.

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

See Group Objects for general information on this type of object.

Hggroup Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`BeingDeleted` `on | {off}` Read Only

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

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

`BusyAction` `cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

ButtonDownFcn string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over the children of the hggroup object.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

The expression executes in the MATLAB workspace.

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

Children array of graphics object handles

Children of the hggroup object. An array containing the handles of all objects parented to the hggroup object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not appear in the hggroup `Children` property unless you set the Root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

Clipping `{on} | off`

Clipping mode. MATLAB clips stairs plots to the axes plot box by default. If you set `Clipping` to `off`, lines might be displayed outside the axes plot box.

CreateFcn string or function handle

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

```
set(0,'DefaultStairsCreateFcn',@myCreateFcn)
```

defines a default value on the Root level that applies to every hggroup object created in a MATLAB session. Whenever you create an hggroup object, the function associated with the function handle `@myCreateFcn` executes.

MATLAB executes the callback after setting all the hggroup object's properties. Setting the `CreateFcn` property on an existing hggroup object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

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

`DeleteFcn` string or function handle

Callback executed during object deletion. A callback that executes when the hggroup object is deleted (e.g., this might happen when you issue a `delete` command on the hggroup object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

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

See the `BeingDeleted` property for related information.

`EraseMode` `{normal} | none | xor | background`

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

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor`— Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage

the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.

- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

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

`HandleVisibility` `{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the hggroup object.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might

potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

`HitTest` `{on} | off`

Pickable by mouse click. `HitTest` determines whether the hggroup object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the hggroup child objects. Note that to pick the hggroup object, its children must have their `HitTest` property set to `off`.

If the hggroup object's `HitTest` is `off`, clicking it picks the object behind it.

`Interruptible` `{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an hggroup object callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the ButtonDownFcn property are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback only when it encounters a drawnow, figure, getframe, or pause command in the routine. See the BusyAction property for related information.

Setting Interruptible to on allows any graphics object's callback to interrupt callback routines originating from an hggroup property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the gca or gcf command) when an interruption occurs.

Parent axes handle

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

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

Selected on | {off}

Is object selected? When you set this property to on, MATLAB displays selection handles at the corners and midpoints of hggroup child objects if the SelectionHighlight property is also on (the default).

SelectionHighlight {on} | off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing selection handles on the hggroup child objects. When SelectionHighlight is off, MATLAB does not draw the handles.

Tag string

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

For example, you might create an hggroup object and set the Tag property:

```
t = hggroup('Tag','group1')
```

When you want to access the object, you can use `findobj` to find its handle. For example,

```
h = findobj('Tag','group1');
```

`Type` string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For hggroup objects, `Type` is `'hggroup'`. The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu` handle of a uicontextmenu object

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

`UserData` array

User-specified data. This property can be any data you want to associate with the hggroup object (including cell arrays and structures). The hggroup object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible` `{on} | off`

Visibility of hggroup object and its children. By default, hggroup object visibility is `on`. This means all children of the hggroup are visible unless the child object's `Visible` property is set to `off`. Setting an hggroup object's `Visible` property to `off` also makes its children invisible.

hgload

Purpose

Load Handle Graphics object hierarchy from a file

Syntax

```
h = hgload('filename')
[h,old_props] = hgload(...,property_structure)
h = hgload(...,'all')
```

Description

`h = hgload('filename')` loads Handle Graphics objects and its children if any from the FIG-file specified by `filename` and returns handles to the top-level objects. If `filename` contains no extension, then MATLAB adds the `.fig` extension.

`[h,old_prop_values] = hgload(...,property_structure)` overrides the properties on the top-level objects stored in the FIG-file with the values in `property_structure`, and returns their previous values in `old_prop_values`.

`property_structure` must be a structure having field names that correspond to property names and values that are the new property values.

`old_prop_values` is a cell array equal in length to `h`, containing the old values of the overridden properties for each object. Each cell contains a structure having field names that are property names, each of which contains the original value of each property that has been changed. Any property specified in `property_structure` that is not a property of a top-level object in the FIG-file is not included in `old_prop_values`.

`hgload(...,'all')` overrides the default behavior, which does not reload nonserializable objects saved in the file. These objects include the default toolbars and default menus.

Nonserializable objects (such as the default toolbars and the default menus) are normally not reloaded because they are loaded from different files at figure creation time. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files. Passing the string `all` to `hgload` ensures that any nonserializable objects contained in the file are also reloaded.

Note that, by default, `hgsave` excludes nonserializable objects from the FIG-file unless you use the `all` flag.

See Also

`hgsave`, `open`

"Figure Windows" for related functions

Purpose Saves a Handle Graphics object hierarchy to a file

Syntax

```
hgsave('filename')
hgsave(h,'filename')
hgsave(...,'all')
hgsave(...,'-v6')
```

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(...,'all')` overrides the default behavior, which does not save nonserializable objects. Nonserializable objects include the default toolbars and default menus. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files and also reduces the size of FIG-files. Passing the string `all` to `hgsave` ensures that nonserializable objects are also saved.

Note: the default behavior of `hgload` is to ignore nonserializable objects in the file at load time. This behavior can be overwritten using the `all` argument with `hgload`.

`hgsave(...,'-v6')` saves the FIG-file in a format that can be loaded by versions prior to MATLAB 7.

Full Backward Compatibility

When creating a figure you want to save and use in a MATLAB version prior to MATLAB 7, use the `'v6'` option with the plotting function and the `'-v6'` option for `hgsave`. Check the reference page for the plotting function you are using for more information.

See Plot Objects and Backward Compatibility for more information.

See Also `hgload`, `open`, `save`

"Figure Windows" for related functions

hgtransform

Purpose

Create an hgtransform graphics object

Syntax

```
h = hgtransform
h = hgtransform('PropertyName',PropertyValue,...)
```

Description

`h = hgtransform` creates an hgtransform object and returns its handle.

`h = hgtransform('`*`PropertyName`*`',PropertyValue,...)` creates an hgtransform object with the property value settings specified in the argument list.

Hgtransform objects can contain other objects and thereby enable you to treat the hgtransform and its children as a single entity with respect to visibility, size, orientation, etc. You can group objects together by parenting them to a single hgtransform object (i.e., setting the object's `Parent` property to the hgtransform object's handle). For example,

```
h = hgtransform;
surface('Parent',h,...)
```

The primary advantage of parenting objects to an hgtransform object is that it provides the ability to perform *transformations* (e.g., translation, scaling, rotation, etc.) on the child objects in unison.

An hgtransform object can be the parent of any number of axes children including other hgtransform objects.

The parent of an hgtransform object is either an axes object or another hgtransform.

Although you cannot see an hgtransform object, setting its `Visible` property to `off` makes all its children invisible as well.

Note Many plotting functions clear the axes (i.e., remove axes children) before drawing the graph. Clearing the axes also deletes any hgtransform objects in the axes.

More Information

- The references in the “See Also” section for information on types of transforms
- The “Examples” section provides examples that illustrate the use of transforms.

Examples

Transforming a Group of Objects

This example shows how to create a 3-D star with a group of surface objects parented to a single hgtransform object. The hgtransform object is then rotated about the z-axis while its size is scaled.

Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

---

1. Create an axes and adjust the view. Set the axes limits to prevent auto limit selection during scaling.

```
ax = axes('XLim',[-1.5 1.5],'YLim',[-1.5 1.5],...
   'ZLim',[-1.5 1.5]);
view(3); grid on; axis equal
```

2. Create the objects you want to parent to the hgtransform object.

```
[x y z] = cylinder([.2 0]);
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```

3. Create an hgtransform object and parent the surface objects to it.

```
t = hgtransform('Parent',ax);
set(h,'Parent',t)
```

4. Select a renderer and show the objects.

```
set(gcf,'Renderer','opengl')
drawnow
```

5 Initialize the rotation and scaling matrix to the identity matrix (eye).

```
Rz = eye(4);
Sxy = Rz;
```

6 Form the *z*-axis rotation matrix and the scaling matrix. Rotate 360 degrees (2*pi radians) and scale by using the increasing values of r.

```
for r = 1:.1:2*pi
   % Z-axis rotation matrix
   Rz = makehgtform('zrotate',r);
   % Scaling matrix
   Sxy = makehgtform('scale',r/4);
   % Concatenate the transforms and
   % set the hgtransform Matrix property
    set(t,'Matrix',Rz*Sxy)
    drawnow
end
pause(1)
```

7 Reset to the original orientation and size using the identity matrix.

```
set(t,'Matrix',eye(4))
```

Transforming Objects Independently

This example creates two hgtransform objects to illustrate how each can be transformed independently within the same axes. One of the hgtransform objects has been moved (by translation) away from the origin.

Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

---

1 Create and set up the axes object that will be the parent of both hgtransform objects. Set the limits to accommodate the translated object.

```
ax = axes('XLim',[-2 1],'YLim',[-2 1],'ZLim',[-1 1]);
view(3); grid on; axis equal
```

2 Create the surface objects to group.

```
[x y z] = cylinder([.3 0]);
```

```
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```

3 Create the hgtransform objects and parent them to the same axes.

```
t1 = hgtransform('Parent',ax);
t2 = hgtransform('Parent',ax);
```

4 Set the renderer to use OpenGL.

```
set(gcf,'Renderer','opengl')
```

5 Parent the surfaces to hgtransform `t1`, then copy the surface objects and parent the copies to hgtransform `t2`.

```
set(h,'Parent',t1)
h2 = copyobj(h,t2);
```

6 Translate the second hgtransform object away from the first hgtransform object and display the result.

```
Txy = makehgtform('translate',[-1.5 -1.5 0]);
set(t2,'Matrix',Txy)
drawnow
```

7 Rotate both hgtransform objects in opposite directions. Hgtransform `t2` has already been translated away from the origin, so to rotate it about its z-axis you must first translate it to its original position. You can do this with the identity matrix (`eye`).

```
% rotate 5 times (2pi radians = 1 rotation)
for r = 1:.1:20*pi
   % Form z-axis rotation matrix
   Rz = makehgtform('zrotate',r);
   % Set transforms for both hgtransform objects
   set(t1,'Matrix',Rz)
   set(t2,'Matrix',Txy*inv(Rz)*I)
   drawnow
end
```

See Also

`hggroup`, `makehgtform`

For more information about transforms, see Tomas Moller and Eric Haines, *Real-Time Rendering*, A K Peters, Ltd., 1999.

See Group Objects for more information and examples.

Setting Default Properties

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

```
set(0,'DefaultHgtransformPropertyName',propertyvalue,...)
set(gcf,'DefaultHgtransformPropertyName',propertyvalue,...)
set(gca,'DefaultHgtransformPropertyName',propertyvalue,...)
```

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

Property List

The following table lists all hgtransform properties and provides a brief description of each. The property names link to expanded descriptions of the properties.

Property Name	Property Description	Property Value
Specifying a Transformation Matrix		
`Matrix`	Applies the transformation matrix to the hgtransform object and objects parented to it	Value: 4-by-4 transform matrix Default: identity matrix
General Information About Hgtransform Object		
`Children`	Handles of the axes children objects that are parented to the hgtransform object	Value: vector of handles
`Parent`	Handle of the axes, hggroup, or hgtransform object containing the hgtransform object	Value: scalar handle
`Selected`	Currently not implemented	Values: `on`, `off` Default: `on`

Property Name	Property Description	Property Value
`Tag`	User-specified label	Value: any string Default: `''` (empty string)
`Type`	Type of graphics object (read only)	Value: the string `'hgtransform'`
`UserData`	User-specified data	Value: any array Default: `[]` (empty matrix)
`Visible`	Makes hgtransform (and all its `Children`) visible or invisible	Values: `on`, `off` Default: `on`
Controlling Callback Routine Execution		
`BeingDeleted`	Query to see whether object is being deleted.	Values: on | off Read only
`BusyAction`	Specifies how to handle events that interrupt executing callback routines	Values: `cancel`, `queue` Default: `queue`
`ButtonDownFcn`	Defines a callback that executes when a button is pressed over the hgtransform object	Value: string or function handle Default: an empty string
`CreateFcn`	Defines a callback that executes when an hgtransform object is created	Value: string or function handle Default: an empty string
`DeleteFcn`	Defines a callback that executes when an hgtransform object is deleted	Value: string or function handle Default: an empty string
`Interruptible`	Controls whether an executing callback can be interrupted	Values: `on`, `off` Default: `on`
`UIContextMenu`	Associates a context menu with the hgtransform object	Value: handle of a uicontextmenu
`HandleVisibility`	Controls access to hgtransform object's handle	Values: `on`, `callback`, `off` Default: `on`

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands.

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

See Group Objects for general information on this type of object.

Hgtransform Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

BeingDeleted `on | {off}` Read Only

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

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

BusyAction `cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback functions. If there is a callback executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

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

ButtonDownFcn string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is within the extent of the hgtransform object, but not over another graphics object. The extent of an hgtransform object is the smallest rectangle that encloses all the children. Note that you cannot execute the hgtransform object's button down function if it has no children.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

The expression executes in the MATLAB workspace.

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

Children array of graphics object handles

Children of the hgtransform object. An array containing the handles of all graphics objects parented to the hgtransform object (whether visible or not).

The graphics objects that can be children of an hgtransform are images, lights, lines, patches, rectangles, surfaces, and text. You can change the order of the handles and thereby change the stacking of the objects on the display.

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in the hgtransform `Children` property unless you set the Root `ShowHiddenHandles` property to `on`.

Clipping `{on} | off`

This property has no effect on hgtransform objects.

CreateFcn string or function handle

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

```
set(0,'DefaultHgtransformCreateFcn',@myCreateFcn)
```

defines a default value on the root level that applies to every hgtransform object created in a MATLAB session. Whenever you create an hgtransform object, the function associated with the function handle `@myCreateFcn` executes.

MATLAB executes the callback after setting all the hgtransform object's properties. Setting the `CreateFcn` property on an existing hgtransform object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

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

`DeleteFcn` string or function handle

Callback executed during object deletion. A callback that executes when the hgtransform object is deleted (e.g., this might happen when you issue a `delete` command on the hgtransform object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

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

See the `BeingDeleted` property for related information.

`EraseMode` `{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase hgtransform child objects (light objects have no erase mode). Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor`— Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR operation on a pixel color and the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

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

`HandleVisibility` `{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the hgtransform object.

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

- callback — Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- off — Setting HandleVisibility to off makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

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

Properties Affected by Handle Visibility

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

Overriding Handle Visibility

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

Handle Validity

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

HitTest {on} | off

Pickable by mouse click. HitTest determines whether the hgtransform object can become the current object (as returned by the gco command and the figure CurrentObject property) as a result of a mouse click within the limits of the hgtransform object. If HitTest is off, clicking the hgtransform picks the object behind it.

Interruptible {on} | off

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

Setting Interruptible to on allows any graphics object's callback to interrupt callback routines originating from an hgtransform property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the gca or gcf command) when an interruption occurs.

Matrix 4-by-4 matrix

Transformation matrix applied to hgtransform object and its children. The hgtransform object applies the transformation matrix to all its children.

See Group Objects for more information and examples.

Parent figure handle

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

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

Selected on | {off}

Is object selected? When you set this property to on, MATLAB displays selection handles on all child objects of the hgtransform if the SelectionHighlight property is also on (the default).

SelectionHighlight {on} | off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing selection handles on the objects parented to the hgtransform. When SelectionHighlight is off, MATLAB does not draw the handles.

Tag string

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

For example, you might create an hgtransform object and set the Tag property:

```
t = hgtransform('Tag','subgroup1')
```

When you want to access the hgtransform object to add another object, you can use findobj to find the hgtransform object's handle. The following statement adds a line to subgroup1 (assuming x and y are defined).

```
line('XData',x,'YData',y,'Parent',findobj('Tag','subgroup1'))
```

Type string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For hgtransform objects, Type is set to 'hgtransform'. The following statement finds all the hgtransform objects in the current axes.

```
t = findobj(gca,'Type','hgtransform');
```

UIContextMenu handle of a uicontextmenu object

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

UserData array

User-specified data. This property can be any data you want to associate with the hgtransform object (including cell arrays and structures). The hgtransform object does not set values for this property, but you can access it using the set and get functions.

Visible {on} | off

Visibility of hgtransform object and its children. By default, hgtransform object visibility is on. This means all children of the hgtransform are visible unless the child object's `Visible` property is set to `off`. Setting an hgtransform object's `Visible` property to `off` also makes its children invisible.

hidden

Purpose Remove hidden lines from 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`

"Creating Surfaces and Meshes" for related functions

Purpose Hilbert matrix

Syntax

```
H = hilb(n)
```

Description `H = hilb(n)` returns the Hilbert matrix of order `n`.

Definition The Hilbert matrix is a notable example of a poorly conditioned matrix [1]. The elements of the Hilbert matrices are $H(i,j) = 1/(i+j-1)$.

Examples Even the fourth-order Hilbert matrix shows signs of poor conditioning.

```
cond(hilb(4)) =
    1.5514e+04
```

Note 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.

hist

Purpose

Histogram plot

Syntax

```
n = hist(Y)
n = hist(Y,x)
n = hist(Y,nbins)
[n,xout] = hist(...)
hist(...)
hist(axes_handle,...)
```

Description

A histogram shows the distribution of data values.

`n = hist(Y)` bins the elements in vector `Y` into 10 equally spaced containers and returns the number of elements in each container as a row vector. If `Y` is an `m`-by-`p` matrix, `hist` treats the columns of `Y` as vectors and returns a 10-by-`p` matrix `n`. Each column of `n` contains the results for the corresponding column of `Y`.

`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 produces a histogram plot of the output described above. `hist` distributes the bins along the *x*-axis between the minimum and maximum values of `Y`.

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

Remarks

All elements in vector `Y` or in one column of matrix `Y` are grouped according to their numeric range. Each group is shown as one bin.

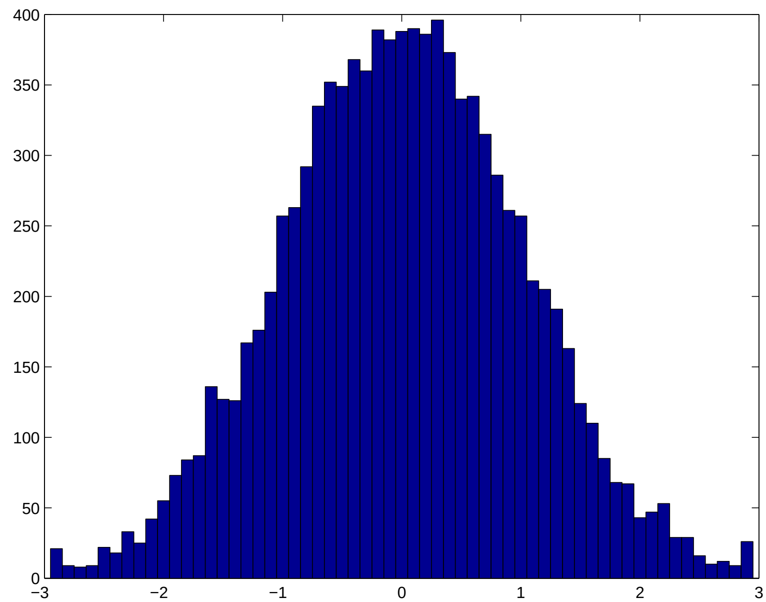
The histogram's *x*-axis reflects the range of values in `Y`. The histogram's *y*-axis shows the number of elements that fall within the groups; therefore, the *y*-axis ranges from `0` to the greatest number of elements deposited in any bin.

The 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

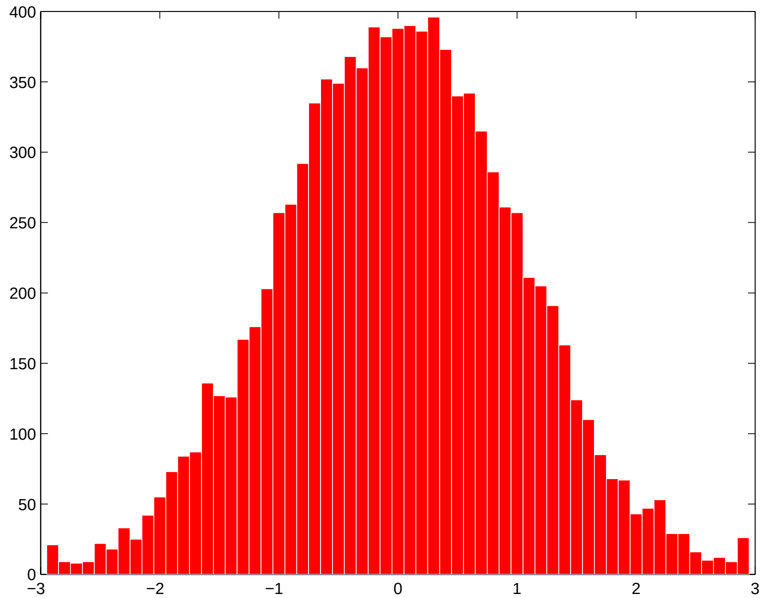
Generate a bell-curve histogram from Gaussian data.

```
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`, `rose`, `stairs`

"Specialized Plotting" for related functions

Histograms for examples

Purpose Histogram count

Syntax

```
n = histc(x,edges)
n = histc(x,edges,dim)
[n,bin] = histc(...)
```

Description

`n = histc(x,edges)` counts the number of values in vector `x` that fall between the elements in the `edges` vector (which must contain monotonically nondecreasing values). `n` is a `length(edges)` vector containing these counts.

`n(k)` counts the value `x(i)` if `edges(k) <= x(i) < edges(k+1)`. The last bin counts any values of `x` that match `edges(end)`. Values outside the values in `edges` are not counted. Use `-inf` and `inf` in `edges` to include all non-`NaN` values.

For matrices, `histc(x,edges)` returns a matrix of column histogram counts. For N-D arrays, `histc(x,edges)` operates along the first nonsingleton dimension.

`n = histc(x,edges,dim)` operates along the dimension `dim`.

`[n,bin] = histc(...)` also returns an index matrix `bin`. If `x` is a vector, `n(k) = sum(bin==k)`. `bin` is zero for out of range values. If `x` is an `M`-by-`N` matrix, then

```
for j=1:N, n(k,j) = sum(bin(:,j)==k); end
```

To plot the histogram, use the `bar` command.

See Also

`hist`

"Specialized Plotting" for related functions

hold

Purpose

Hold current graph in the figure

Syntax

```
hold on
hold off
hold all
hold
hold(axes_handle,...)
```

Description

The `hold` function determines whether new graphics objects are added to the graph or replace objects in the graph.

`hold on` retains the current plot and certain axes properties so that subsequent graphing commands add to the existing graph.

`hold off` resets axes properties to their defaults before drawing new plots. `hold off` is the default.

`hold all` holds the plot and the current line color and line style so that subsequent plotting commands do not reset the `ColorOrder` and `LineStyleOrder` property values to the beginning of the list. Plotting commands continue cyclicing through the predefined colors and linestyles from where the last plot stopped in the list.

`hold` toggles the hold state between adding to the graph and replacing the graph.

`hold(axes_handle,...)` applies the hold to the axes identified by the handle `axes_handle`.

Remarks

Test the hold state using the `ishold` function.

Although the hold state is `on`, some axes properties change to accommodate additional graphics objects. For example, the axes' limits increase when the data requires them to do so.

The `hold` function sets the `NextPlot` property of the current figure and the current axes. If several axes objects exist in a figure window, each axes has its own hold state. `hold` also creates an axes if one does not exist.

`hold on` sets the `NextPlot` property of the current figure and axes to `add`.

hold off sets the NextPlot property of the current axes to replace.

hold toggles the NextPlot property between the add and replace states.

See Also

axis, cla, ishold, newplot

The NextPlot property of axes and figure graphics objects.

"Basic Plots and Graphs" for related functions

home

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. 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`

Purpose Horizontal concatenation

Syntax

```
C = horzcat(A1,A2,...)
```

Description

`C = horzcat(A1,A2,...)` horizontally concatenates matrices `A1`, `A2`, and so on. All matrices in the argument list must have the same number of rows.

`horzcat` concatenates N-dimensional arrays along the second dimension. The first and remaining dimensions must match.

MATLAB calls `C = horzcat(A1,A2,...)` for the syntax `C = [A1 A2 ...]` when any of `A1`, `A2`, etc., is an object.

Examples

Create a 3-by-5 matrix, `A`, and a 3-by-3 matrix, `B`. Then horizontally concatenate `A` and `B`.

```
A = magic(5);            % Create 3-by-5 matrix, A
A(4:5,:) = []

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22


B = magic(3)*100         % Create 3-by-3 matrix, B

B =

   800   100   600
   300   500   700
   400   900   200


C = horzcat(A,B)         % Horizontally concatenate A and B

C =

    17    24     1     8    15   800   100   600
```

```
    23     5     7    14    16   300   500   700
     4     6    13    20    22   400   900   200
```

See Also

vertcat, cat

Purpose Return MATLAB server host identification number

Syntax

```
id = hostid
```

Description

`id = hostid` usually returns a single element cell array containing the identifier as a string. UNIX systems may have more than one identifier. In this case, `hostid` returns a cell array with an identifier in each cell.

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.

`rgb_image = hsv2rgb(hsv_image)` converts the HSV image to the equivalent RGB image. HSV is an m-by-n-by-3 image array whose three planes contain the hue, saturation, and value components for the image. RGB is returned as an m-by-n-by-3 image array whose three planes contain the red, green, and blue components for the image.

Remarks

As `H(:,1)` varies from 0 to 1, the resulting color varies from red through yellow, green, cyan, blue, and magenta, and returns to red. When `H(:,2)` is 0, the colors are unsaturated (i.e., shades of gray). When `H(:,2)` is 1, the colors are fully saturated (i.e., they contain no white component). As `H(:,3)` varies from 0 to 1, the brightness increases.

The MATLAB `hsv` colormap uses `hsv2rgb([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`

"Color Operations" for related functions

2i

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`

if

Purpose

Conditionally execute statements

Syntax

```
if expression
   statements
end
```

Description

MATLAB evaluates the *expression* and, if the evaluation yields a logical true or nonzero result, executes one or more MATLAB commands denoted here as *statements*.

When you are nesting `if`s, each `if` must be paired with a matching `end`.

When using `elseif` and/or `else` within an `if` statement, the general form of the statement is

```
if expression1
    statements1
elseif expression2
    statements2
else
    statements3
end
```

Arguments

expression

expression is a MATLAB expression, usually consisting of variables or smaller expressions joined by relational operators (e.g., `count < limit`), or logical functions (e.g., `isreal(A)`).

Simple expressions can be combined by logical operators (`&`,`|`,`~`) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

```
(count < limit) & ((height - offset) >= 0)
```

statements

statements is one or more MATLAB statements to be executed only if the *expression* is `true` or nonzero.

Remarks

Nonscalar Expressions

If the evaluated `expression` yields a nonscalar value, then every element of this value must be `true` or nonzero for the entire expression to be considered `true`. For example, the statement `if (A < B)` is `true` only if each element of matrix `A` is less than its corresponding element in matrix `B`. See Example 2, below.

Partial Evaluation of the expression Argument

Within the context of an `if` or `while` expression, MATLAB does not necessarily evaluate all parts of a logical expression. In some cases it is possible, and often advantageous, to determine whether an expression is true or false through only partial evaluation.

For example, if `A` equals zero in statement 1 below, then the expression evaluates to `false`, regardless of the value of `B`. In this case, there is no need to evaluate `B` and MATLAB does not do so. In statement 2, if `A` is nonzero, then the expression is `true`, regardless of `B`. Again, MATLAB does not evaluate the latter part of the expression.

```
1)   if (A & B)               2)   if (A | B)
```

You can use this property to your advantage to cause MATLAB to evaluate a part of an expression only if a preceding part evaluates to the desired state. Here are some examples.

```
while (b ~= 0) & (a/b > 18.5)

if exist('myfun.m') & (myfun(x) >= y)

if iscell(A) & all(cellfun('isreal', A))
```

Examples

Example 1 - Simple if Statement

In this example, if both of the conditions are satisfied, then the student passes the course.

```
if ((attendance >= 0.90) & (grade_average >= 60))
   pass = 1;
end;
```

Example 2 - Nonscalar Expression

Given matrices `A` and `B`,

```
A =                  B =
     1     0              1     1
     2     3              3     4
```

Expression	Evaluates As	Because
`A < B`	`false`	`A(1,1)` is not less than `B(1,1)`.
`A < (B + 1)`	`true`	Every element of `A` is less than that same element of `B` with `1` added.
`A & B`	`false`	`A(1,2) & B(1,2)` is false.
`B < 5`	`true`	Every element of `B` is less than `5`.

See Also

`else`, `elseif`, `end`, `for`, `while`, `switch`, `break`, `return`, relational operators, logical operators (elementwise and short-circuit)

Purpose

Inverse discrete Fourier transform

Syntax

```
y = ifft(X)
y = ifft(X,n)
y = ifft(X,[],dim)
y = ifft(X,n,dim)
y = ifft(..., 'symmetric')
y = ifft(..., 'nonsymmetric')
```

Description

`y = ifft(X)` returns the inverse discrete Fourier transform (DFT) of vector `X`, computed with a fast Fourier transform (FFT) algorithm. If `X` is a matrix, `ifft` returns the inverse DFT of each column of the matrix.

`ifft` tests `X` to see whether vectors in `X` along the active dimension are *conjugate symmetric*. If so, the computation is faster and the output is real. An `N`-element vector `x` is conjugate symmetric if `x(i) = conj(x(mod(N-i+1,N)+1))` for each element of `x`.

If `X` is a multidimensional array, `ifft` operates on the first non-singleton dimension.

`y = ifft(X,n)` returns the `n`-point inverse DFT of vector `X`.

`y = ifft(X,[],dim)` and `y = ifft(X,n,dim)` return the inverse DFT of `X` across the dimension `dim`.

`y = ifft(..., 'symmetric')` causes `ifft` to treat `X` as conjugate symmetric along the active dimension. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft(..., 'nonsymmetric')` is the same as calling `ifft(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft(fft(X))` equals `X` to within roundoff error.

Algorithm

The algorithm for `ifft(X)` is the same as the algorithm for `fft(X)`, except for a sign change and a scale factor of `n = length(X)`. As for `fft`, the execution time for `ifft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

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

Data Type Support

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

See Also

`fft`, `fft2`, `ifft2`, `ifftn`, `ifftshift`, `fftw`, `ifft2`, `ifftn`

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

Purpose

Two-dimensional inverse discrete Fourier transform

Syntax

```
Y = ifft2(X)
Y = ifft2(X,m,n)
y = ifft2(..., 'nonsymmetric')
y = ifft2(..., 'nonsymmetric')
```

Description

`Y = ifft2(X)` returns the two-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifft2` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `M`-by-`N` matrix `X` is conjugate symmetric if `X(i,j) = conj(X(mod(M-i+1, M) + 1, mod(N-j+1, N) + 1))` for each element of `X`.

`Y = ifft2(X,m,n)` returns the `m`-by-`n` inverse fast Fourier transform of matrix `X`.

`y = ifft2(..., 'symmetric')` causes `ifft2` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft2(..., 'nonsymmetric')` is the same as calling `ifft2(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft2(fft2(X))` equals `X` to within roundoff error.

Algorithm

The algorithm for `ifft2(X)` is the same as the algorithm for `fft2(X)`, except for a sign change and scale factors of `[m,n] = size(X)`. The execution time for `ifft2` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifft2` using the utility function `fftw`, which controls how MATLAB optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`ifft2` supports inputs of data types `double` and `single`. If you call `ifft2` with the syntax `y = ifft2(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

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

`fft2`, `fftw`, `fftshift`, `ifft`, `ifftn`, `ifftshift`

Purpose

Multidimensional inverse discrete Fourier transform

Syntax

```
Y = ifftn(X)
Y = ifftn(X,siz)
y = ifftn(..., 'nonsymmetric')
y = ifftn(..., 'nonsymmetric')
```

Description

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

`ifftn` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `N1`-by-`N2`-by- ... `Nk` array `X` is conjugate symmetric if

```
X(i1,i2, ...,ik) = conj(X(mod(N1-i1+1,N1)+1, mod(N2-i2+1,N2)+1,
... mod(Nk-ik+1,Nk)+1))
```

for each element of `X`.

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

`y = ifftn(..., 'symmetric')` causes `ifftn` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifftn(..., 'nonsymmetric')` is the same as calling `ifftn(...)` without the argument `'nonsymmetric'`.

Remarks

For any `X`, `ifftn(fftn(X))` equals `X` within roundoff error.

Algorithm

`ifftn(X)` is equivalent to

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

This computes in-place the one-dimensional inverse DFT along each dimension of `X`.

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

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

Data Type Support

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

See Also

`fftn`, `fftw`, `ifft`, `ifft2`, `ifftshift`

Purpose Inverse FFT shift

Syntax

```
ifftshift(X)
ifftshift(X,dim)
```

Description `ifftshift(X)` 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 image into movie format

Syntax

```
f = im2frame(X,map)
f = im2frame(X)
```

Description `f = im2frame(X,map)` converts the indexed image `X` and associated colormap `map` into a movie frame `f`. If `X` is a true color (m-by-n-by-3) image, then `map` is optional and has no effect.

Typical usage:

```
M(1) = im2frame(X1,map);
M(2) = im2frame(X2,map);
    ...
M(n) = im2frame(Xn,map);
movie(M)
```

`f = im2frame(X)` converts the indexed image `X` into a movie frame `f` using the current colormap if `X` contains an indexed image.

See Also `frame2im`, `movie`

"Bit-Mapped Images" for related functions

Purpose Convert image to Java image

Syntax

```
jimage = im2java(I)
jimage = im2java(X,MAP)
jimage = im2java(RGB)
```

Description To work with a MATLAB image in the Java environment, you must convert the image from its MATLAB representation into an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(I)` converts the intensity image `I` to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(X,MAP)` converts the indexed image `X`, with colormap `MAP`, to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(RGB)` converts the RGB image `RGB` to an instance of the Java image class, `java.awt.Image`.

Class Support The input image can be of class `uint8`, `uint16`, or `double`.

Note Java requires `uint8` data to create an instance of the Java image class, `java.awt.Image`. If the input image is of class `uint8`, `jimage` contains the same `uint8` data. If the input image is of class `double` or `uint16`, `im2java` makes an equivalent image of class `uint8`, rescaling or offsetting the data as necessary, and then converts this `uint8` representation to an instance of the Java image class, `java.awt.Image`.

Example This example reads an image into the MATLAB workspace and then uses `im2java` to convert it into an instance of the Java image class.

```
I = imread('your_image.tif');
javaImage = im2java(I);
frame = javax.swing.JFrame;
icon = javax.swing.ImageIcon(javaImage);
label = javax.swing.JLabel(icon);
frame.getContentPane.add(label);
frame.pack
```

```
frame.show
```

See Also

"Bit-Mapped Images" for related functions

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`

Purpose Display image object

Syntax

```
image(C)
image(x,y,C)
image(...,'PropertyName',PropertyValue,...)
image('PropertyName',PropertyValue,...)  Formal syntax – 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 true color 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 methods 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] (uint8) or [0, 65535] (uint16); colormap is an m-by-3 array of floating-point values in the range [0, 1].
true color (RGB)	Image is stored as a three-dimensional (m-by-n-by-3) array of floating-point values in the range [0, 1].	Image is stored as a three-dimensional (m-by-n-by-3) array of integers in the range [0, 255] (uint8) or [0, 65535] (uint16).

Indexed Images

In an indexed image of class double, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. In a uint8 or uint16 indexed image, there is an offset; the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on.

If you want to convert a uint8 or uint16 indexed image to double, you need to add 1 to the result. For example,

```
X64 = double(X8) + 1;
```

or

```
X64 = double(X16) + 1;
```

To convert from double to uint8 or 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 always `m`-by-3 arrays of double-precision floating-point numbers in the range [0, 1]. In most graphics file formats, colormaps are stored as integers, but MATLAB does not support colormaps with integer values. `imread` and `imwrite` automatically convert colormap values when reading and writing files.

True Color Images

In a true color image of class `double`, the data values are floating-point numbers in the range [0, 1]. In a true color image of class `uint8`, the data values are integers in the range [0, 255], and for true color images of class `uint16` the data values are integers in the range [0, 65535].

If you want to convert a true color image from one data type to the other, you must rescale the data. For example, this statement converts a `uint8` true color image to `double`.

```
RGB64 = double(RGB8)/255;
```

or for `uint16` images,

```
RGB64 = double(RGB16)/65535;
```

This statement converts a `double` true color image to `uint8`.

```
RGB8 = uint8(round(RGB64*255));
```

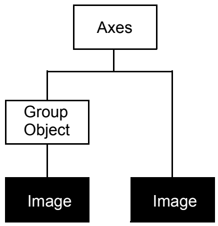
or for `uint16` images,

```
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 true color image using `imwrite`, MATLAB automatically converts the values if necessary.

Object Hierarchy



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	Value: matrix or m-by-n-by-3 array Default: enter `image;axis image ij` and see
`CDataMapping`	Specifies the mapping of data to colormap	Values: `scaled`, `direct` Default: `direct`
`XData`	Controls placement of image along *x*-axis	Values: `[min max]` Default: `[1 size(CData,2)]`
`YData`	Controls 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

Property Name	Property Description	Property Value
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`	Highlights image when selected (`Selected` property set to `on`)	Values: `on`, `off` Default: `on`
`Visible`	Makes the image visible or invisible	Values: `on`, `off` Default: `on`
Controlling Access to Objects		
`HandleVisibility`	Determines if and when the line's handle is visible to other functions	Values: `on`, `callback`, `off` Default: `on`
`HitTest`	Determines 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)
`Parent`	The parent of an image object is the axes, hggroup, or hgtransform object containing it.	Value: scalar handle
`Selected`	Indicates 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'`

Property Name	Property Description	Property Value
UserData	User-specified data	Value: any matrix Default: [] (empty matrix)
Properties Related to Callback Routine Execution		
BeingDeleted	Query to see if object is being deleted.	Values: on | off Read only
BusyAction	Specifies how to handle callback routine interruption	Values: cancel, queue Default: queue
ButtonDownFcn	Defines a callback routine that executes when a mouse button is pressed over the image	Values: string or function handle Default: empty string
CreateFcn	Defines a callback routine that executes when an image is created	Values: string or function handle Default: empty string
DeleteFcn	Defines a callback routine that executes when the image is deleted (via close or delete)	Values: string or function handle Default: empty string
Interruptible	Determines if callback routine can be interrupted	Values: on, off Default: on (can be interrupted)
UIContextMenu	Associates a context menu with the image	Values: handle of a uicontextmenu

See Also

colormap, imfinfo, imread, imwrite, pcolor, newplot, surface

Displaying Bit-Mapped Images chapter

"Bit-Mapped Images" for related functions

Modifying Properties

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

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

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

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

Image Properties

This section lists property names along with the types of values each property accepts.

`AlphaData` m-by-n matrix of `double` or `uint8`

The transparency data. A matrix of non-NaN values specifying the transparency of each 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 of `uint8` integers, then the indexing begins at `0` (i.e., MATLAB maps a value of `0` to the first alpha value in the alphamap).

`BeingDeleted` `on | {off}` Read Only

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

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

`BusyAction` `cancel | {queue}`

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

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

ButtonDownFcn string or function handle

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

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

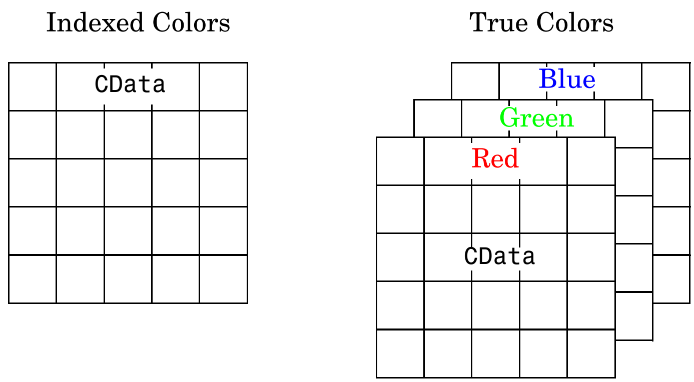
CData matrix or m-by-n-by-3 array

The image data. A matrix or 3-D array of values specifying the color of each rectangular area defining the image. `image(C)` assigns the values of `C` to `CData`. MATLAB determines the coloring of the image in one of three ways:

- Using the elements of `CData` as indices into the current colormap (the default) (`CDataMapping` set to `direct`)
- Scaling the elements of `CData` to range between the values `min(get(gca,'CLim'))` and `max(get(gca,'CLim'))` (`CDataMapping` set to `scaled`)
- Interpreting the elements of `CData` directly as RGB values (true color specification)

Note that the behavior of `NaN`s in image `CData` is not defined. See the image `AlphaData` property for information on using transparency with images.

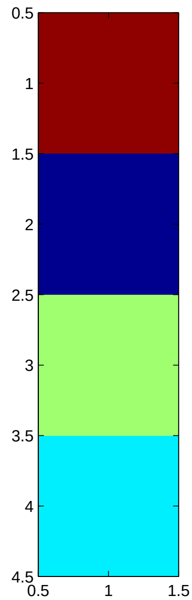
A true color specification for `CData` requires an m-by-n-by-3 array of RGB values. The first page contains the red component, the second page the green component, and the third page the blue component of each element in the image. RGB values range from 0 to 1. The following picture illustrates the relative dimensions of `CData` for the two color models.



If CData has only one row or column, the height or width respectively is always one data unit and is centered about the first YData or XData element respectively. For example, using a 4-by-1 matrix of random data,

```
C = rand(4,1);
image(C,'CDataMapping','scaled')
axis image
```

produces



CDataMapping `scaled | {direct}`

Direct or scaled indexed colors. This property determines whether MATLAB interprets the values in `CData` as indices into the figure colormap (the default) or scales the values according to the values of the axes `CLim` property.

When `CDataMapping` is `direct`, the values of `CData` should be in the range 1 to `length(get(gcf,'Colormap'))`. If you use true color specification for `CData`, this property has no effect.

Children handles

The empty matrix; image objects have no children.

Clipping `on | off`

Clipping mode. By default, MATLAB clips images to the axes rectangle. If you set `Clipping` to `off`, the image can be displayed outside the axes rectangle. For example, if you create an image, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger image, it extends beyond the axis limits.

CreateFcn string or function handle

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates an image object. You must define this property as a default value for images or in a call to the image function to create a new image object. For example, the statement

```
set(0,'DefaultImageCreateFcn','axis image')
```

defines a default value on the root level that sets the aspect ratio and the axis limits so the image has square pixels. MATLAB executes this routine after setting all image properties. Setting this property on an existing image object has no effect.

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

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

DeleteFcn string or function handle

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.

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

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 are redrawn is necessary to improve performance and obtain the desired effect.

- normal (the default) — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- `none` — Do not erase the 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 Nonnormal Erase Modes. MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

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

`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 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 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 the image selects the object below it (which may be the axes containing it).

`Interruptible` `{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an image callback routine can be interrupted by callback routines invoked subsequently. 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, hggroup, or hgtransform

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

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

Selected on | {off}

Is object selected? When this property is on, MATLAB displays selection handles if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

SelectionHighlight {on} | off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing 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 you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define Tag as any string.

Type string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For 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 element one unit apart.

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 Examples for an illustration.)

`imagesc(C)` displays `C` as an image. Each element of `C` corresponds to a rectangular area in the image. The values of the elements of `C` are indices into the current colormap that determine the color of each patch.

`imagesc(x,y,C)` displays `C` as an image and specifies the bounds of the *x*- and *y*-axis with vectors `x` and `y`.

`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.

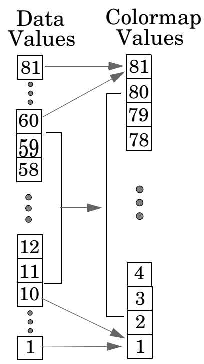
`imagesc` creates an image with `CDataMapping` set to `scaled`, and sets the axes `CLim` property to the value passed in `clims`.

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 in this illustration.

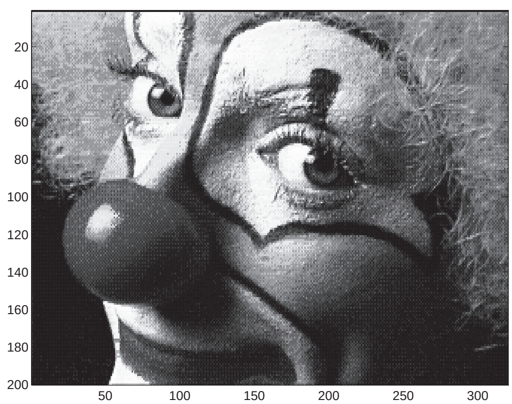


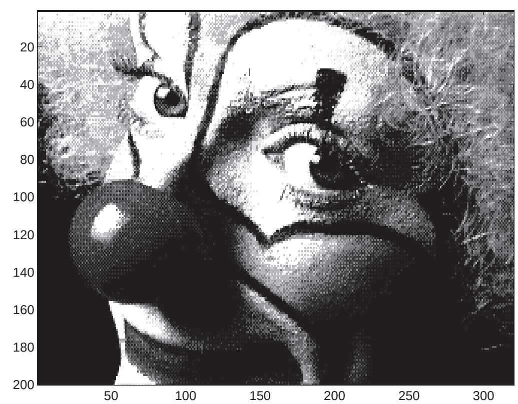
In this example, the left image maps to the `gray` colormap using the statements

```
load clown
imagesc(X)
colormap(gray)
```

The right image has values between `10` and `60` scaled to the full range of the `gray` colormap using the statements

```
load clown
clims = [10 60];
imagesc(X,clims)
colormap(gray)
```





See Also

`image`

"Bit-Mapped Images" for related functions

imfinfo

Purpose Information about graphics file

Syntax

```
info = imfinfo(filename,fmt)
info = imfinfo(filename)
```

Description

`info = imfinfo(filename,fmt)` returns a structure, `info`, 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 all the possible values for `fmt`.

Format	File Type
'bmp'	Windows Bitmap (BMP)
'cur'	Windows Cursor resources (CUR)
'gif'	Graphics Interchange Format (GIF)
'hdf'	Hierarchical Data Format (HDF)
'ico'	Windows Icon resources (ICO)
'jpg' or 'jpeg'	Joint Photographic Experts Group (JPEG)
'pbm'	Portable Bitmap (PBM)
'pcx'	Windows Paintbrush (PCX)
'pgm'	Portable Graymap (PGM)
'png'	Portable Network Graphics (PNG)
'pnm'	Portable Anymap (PNM)
'ppm'	Portable Pixmap (PPM)
'ras'	Sun Raster (RAS)

Format	File Type
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF)
`'xwd'`	X Windows Dump (XWD)

If `filename` is a TIFF, HDF, ICO, GIF, or CUR 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.

`info = imfinfo(filename)` attempts to infer the format of the file from its contents.

Information Returned

The set of fields in `info` depends on the individual file and its format. However, the first nine fields are always the same. This table lists these common fields, in the order they appear in the structure, and describes their values.

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

Field	Value
`BitDepth`	An integer indicating the number of bits per pixel
`ColorType`	A string indicating the type of image; either `'truecolor'` for a true color RGB image, `'grayscale'` for a grayscale intensity image, or `'indexed'` for an indexed image

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 `imformats`, `imread`, `imwrite`

"Bit-Mapped Images" for related functions

imformats

Purpose

Manage file format registry

Syntax

```
imformats
formats = imformats
formats = imformats('fmt')
formats = imformats(format_struct)
formats = imformats('factory')
```

Description

`imformats` displays a table of information listing all the values in the MATLAB file format registry. This registry determines which file formats are supported by the `imfinfo`, `imread`, and `imwrite` functions.

`formats = imformats` returns a structure containing all the values in the MATLAB file format registry. The following tables lists the fields in the order they appear in the structure.

Field	Value
`ext`	A cell array of strings that specify filename extensions that are valid for this format
`isa`	A string specifying the name of the function that determines if a file is a certain format. This can also be a function handle.
`info`	A string specifying the name of the function that reads information about a file. This can also be a function handle.
`read`	A string specifying the name of the function that reads image data in a file. This can also be a function handle.
`write`	A string specifying the name of the function that writes MATLAB data to a file. This can also be a function handle.

Field	Value
`alpha`	Returns `1` if the format has an alpha channel, `0` otherwise
`description`	A text description of the file format

Note The values for the `isa`, `info`, `read`, and `write` fields must be functions on the MATLAB search path or function handles.

`formats = imformats('fmt')` searches the known formats in the MATLAB file format registry for the format associated with the filename extension `'fmt'`. If found, `imformats` returns a structure containing the characteristics and function names associated with the format. Otherwise, it returns an empty structure.

`formats = imformats(format_struct)` sets the MATLAB file format registry to the values in `format_struct`. The output structure, `formats`, contains the new registry settings.

Caution Using `imformats` to specify values in the MATLAB file format registry can result in the inability to load any image files. To return the file format registry to a working state, use `imformats` with the `'factory'` setting.

`formats = imformats('factory')` resets the MATLAB file format registry to the default format registry values. This removes any user-specified settings.

Changes to the format registry do not persist between MATLAB sessions. To have a format always available when you start MATLAB, add the appropriate `imformats` command to the MATLAB startup file, `startup.m`, located in `$MATLAB/toolbox/local` on UNIX systems, or `$MATLAB\toolbox\local` on Windows systems.

Example

```
formats = imformats;
formats(1)
```

```
ans =

            ext: {'bmp'}
            isa: @isbmp
           info: @imbmpinfo
           read: @readbmp
          write: @writebmp
          alpha: 0
    description: 'Windows Bitmap (BMP)'
```

See Also

`fileformats`, `imfinfo`, `imread`, `imwrite`, `path`

"Bit-Mapped Images" for related functions

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`

imread

Purpose Read image from graphics file

Syntax

```
A = imread(filename,fmt)
[X,map] = imread(filename,fmt)
[...] = imread(filename)
[...] = imread(URL,...)
[...] = imread(...,idx)     (CUR, GIF, ICO, and TIFF only)
[...] = imread(...,'PixelRegion',{ROWS, COLS})   (TIFF only)
[...] = imread(...,'frames',idx)  (GIF only)
[...] = imread(...,ref)     (HDF only)
[...] = imread(...,'BackgroundColor',BG)  (PNG only)
[A,map,alpha] = imread(...)  (ICO, CUR, and PNG only)
```

Description The `imread` function supports four general syntaxes, described below. The `imread` function also supports several other format-specific syntaxes. See “Special Case Syntax” on page 2-1150 for information about these syntaxes.

`A = imread(filename,`*`fmt`*`)` reads a greyscale or color image from the file specified by the string `filename`, where the string *`fmt`* 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 of the location on your system. For a list of all the possible values for *`fmt`*, see “Supported Formats” on page 2-1149. If `imread` cannot find a file named `filename`, it looks for a file named `filename.`*`fmt`*.

`imread` returns the image data in the array `A`. If the file contains a grayscale image, `A` is a two-dimensional (M-by-N) array. If the file contains a color image, `A` is a three-dimensional (M-by-N-by-3) array. The class of the returned array depends on the data type used by the file format. See “Class Support” on page 2-1154 for more information.

For most file formats, the color image data returned uses the RGB color space. For TIFF files, however, `imread` can return color data that uses the RGB, CIELAB, ICCLAB, or CMYK color spaces. If the color image uses the CMYK color space, `A` is an M-by-N-by-4 array. See the “TIFF-Specific Syntax” on page 2-1153 for more information.

`[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].

`[...] = imread(filename)` attempts to infer the format of the file from its content.

`[...] = imread(URL,...)` reads the image from an Internet URL. The URL must include the protocol type (e.g., `http://`).

Supported Formats

This table lists all the types of images that `imread` can read, in alphabetical order by the `fmt` abbreviation. You can also get a list of all supported formats by using the `imformats` function. Note that, for certain formats, `imread` may take additional parameters, described in Special Case Syntax.

Format	Full Name	Variants
`'bmp'`	Windows Bitmap (BMP)	1-bit, 4-bit, 8-bit, 16-bit, 24-bit, and 32-bit uncompressed images and 4-bit and 8-bit run-length encoded (RLE) images
`'cur'`	Windows Cursor resources (CUR)	1-bit, 4-bit, and 8-bit uncompressed images
`'gif'`	Graphics Interchange Format (GIF)	1-bit to 8-bit images
`'hdf'`	Hierarchical Data Format (HDF)	8-bit raster image data sets, with or without an associated colormap, and 24-bit raster image data sets
`'ico'`	Windows Icon resources (ICO)	1-bit, 4-bit, and 8-bit uncompressed images
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)	Any baseline JPEG image or JPEG image with some commonly used extensions, including: **Image Type** / **Bitdepth** / **Compression** grayscale / 8- or 12-bit / lossy grayscale / 8-, 12-, or 16-bit / lossless RGB / 24- and 36-bit / lossy or lossless
`'pbm'`	Portable Bitmap (PBM)	1-bit images using either raw (binary) or ASCII (plain) encoding
`'pcx'`	Windows Paintbrush (PCX)	1-bit, 8-bit, and 24-bit images

Format	Full Name	Variants
'pgm'	Portable Graymap (PGM)	ASCII (plain) encoding with arbitrary color depth, or raw (binary) encoding with up to 16 bits per gray value
'png'	Portable Network Graphics (PNG)	1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit indexed images; and 24-bit and 48-bit RGB images
'pnm'	Portable Anymap (PNM)	PNM is not a file format itself. It is a common name for any of the other three members of the Portable Bitmap family of image formats: Portable Bitmap (PBM), Portable Graymap (PGM) and Portable Pixel Map (PPM).
'ppm'	Portable Pixmap (PPM)	ASCII (plain) encoding with arbitrary color depth or raw (binary) encoding with up to 16 bits per color component
'ras'	Sun Raster (RAS)	1-bit bitmap, 8-bit indexed, 24-bit true color and 32-bit true color with alpha data
'tif' or 'tiff'	Tagged Image File Format (TIFF)	Any baseline image, 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 compression; and 16-bit grayscale, 16-bit indexed, and 48-bit RGB images
'xwd'	X Windows Dump (XWD)	1-bit and 8-bit ZPixmaps, XYBitmaps, and 1-bit XYPixmaps

Special Case Syntax

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.

GIF-Specific Syntaxes

`[...] = imread(...,idx)` reads in one or more frames from a multiframe (i.e., animated) GIF file. `idx` must be an integer scalar or vector of integer values. For example, if `idx` is `3`, `imread` reads the third image in the file. If `idx` is `1:5`, `imread` returns only the first five frames.

`[...] = imread(...,'frames',idx)` is the same as the syntax above except that `idx` can be `'all'`. In this case, all the frames are read and returned in the order that they appear in the file.

Note Because of the way that GIF files are structured, all the frames must be read when a particular frame is requested. Consequently, it is much faster to specify a vector of frames or `'all'` for `idx` than to call `imread` in a loop when reading multiple frames from the same GIF file.

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 image order with reference number.) 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, are 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 are treated as transparent. For example, 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 are composited against the specified background color.

If the PNG file does not contain a background color chunk, the transparent pixels are 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 is 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 is 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 true color, `map` may be empty.

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.

For TIFF files, `imread` can read color data represented in the RGB, CIELAB or ICCLAB color spaces. To determine which color space is used, look at the value of the `PhotometricInterpretation` field returned by `imfinfo`. Note, however, that if a file contains CIELAB color data, `imread` converts it to ICCLAB before bringing it into the MATLAB workspace. 8- or 16-bit TIFF CIELAB-encoded values use a mixture of signed and unsigned data types that cannot be represented as a single MATLAB array.

`[...] = imread(...,'PixelRegion',{ROWS, COLS})` returns the sub-image specified by the boundaries in `ROWS` and `COLS`. For tiled TIFF images, `imread` reads only the tiles that encompass the region specified by `ROWS` and `COLS`, improving memory efficiency and performance. `ROWS` and `COLS` must be either two or three element vectors. If two elements are provided, they denote the 1-based indices `[START STOP]`. If three elements are provided, the indices `[START INCREMENT STOP]` allow image downsampling.

Class Support

For most file formats, `imread` uses 8 or fewer bits per color plane to store pixels. The following table lists the class of the returned array for all data types used by the file formats.

Data Type Used in File	Class of Array Returned by imread
1-bit	`logical`
8-bits (or fewer) per color plane	`uint8`
12-bits	`uint16`
16-bits (JPEG, PNG, and TIFF)	`uint16`
16-bits (BMP only)	`uint8`

Note For indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself may be of class `uint8` or `uint16`.

Examples

This example reads the sixth image in a TIFF file.

```
[X,map] = imread('your_image.tif',6);
```

This example reads the fourth image in an HDF file.

```
info = imfinfo('your_hdf_file.hdf');
[X,map] = imread('your_hdf_file.hdf',info(4).Reference);
```

This example reads a 24-bit PNG image and sets any of its fully transparent (alpha channel) pixels to red.

```
bg = [255 0 0];
A = imread('your_image.png','BackgroundColor',bg);
```

This example returns the alpha channel (if any) of a PNG image.

```
[A,map,alpha] = imread('your_image.png');
```

This example reads an ICO image, applies a transparency mask, and then displays the image.

```
[a,b,c] = imread('your_icon.ico');
% Augment colormap for background color (white).
b2 = [b; 1 1 1];
% Create new image for display.
d = ones(size(a)) * (length(b2) - 1);
% Use the AND mask to mix the background and
% foreground data on the new image
d(c == 0) = a(c == 0);
% Display new image
image(uint8(d)), colormap(b2)
```

See Also

`double`, `fread`, `image`, `imfinfo`, `imformats`, `imwrite`, `uint8`, `uint16`

"Bit-Mapped Images" for related functions

Purpose Write image to graphics file

Syntax

```
imwrite(A,filename,fmt)
imwrite(X,map,filename,fmt)
imwrite(...,filename)
imwrite(...,Param1,Val1,Param2,Val2...)
```

Description

`imwrite(A,filename,fmt)` writes the image `A` to the file specified by `filename` in the format specified by `fmt`.

`A` can be an M-by-N (greyscale image) or M-by-N-by-3 (color image) array. `A` cannot be an empty array. If the format specified is TIFF, `imwrite` can also accept an M-by-N-by-4 arrray containing color data that uses the CMYK color space. For information about the class of the input array and the output image, see "Class Support" on page 2-1164.

`filename` is a string that specifies the name of the output file.

`fmt` can be any of the text strings listed in the table in "Supported Formats" on page 2-1157. This list of supported formats is determined by the MATLAB image file format registry. See `imformats` for more information about this registry.

`imwrite(X,map,filename,fmt)` writes the indexed image in `X` and its associated colormap `map` to `filename` in the format specified by `fmt`. If `X` is of class `uint8` or `uint16`, `imwrite` writes the actual values in the array to the file. If `X` is of class `double`, the `imwrite` function offsets the values in the array before writing, using `uint8(X 1)`. The `map` parameter must be a valid MATLAB colormap. Note that most image file formats do not support colormaps with more than 256 entries.

`imwrite(...,filename)` writes the image to `filename`, inferring the format to use from the filename's extension. The extension must be one of the values for `fmt`, listed in "Supported Formats" on page 2-1157.

`imwrite(...,Param1,Val1,Param2,Val2...)` specifies parameters that control various characteristics of the output file for HDF, JPEG, PBM, PGM, PNG, PPM, and TIFF files. For example, if you are writing a JPEG file, you can specify the quality of the output image. For the lists of parameters available for each format, see "Format-Specific Parameters" on page 2-1158.

Supported Formats

This table summarizes the types of images that `imwrite` can write. The MATLAB file format registry determines which file formats are supported. See `imformats` for more information about this registry. Note that, for certain formats, `imwrite` may take additional parameters, described in “Format-Specific Parameters” on page 2-1158.

Format	Full Name	Variants
`'bmp'`	Windows Bitmap (BMP	1-bit, 8-bit, and 24-bit uncompressed images
`'hdf'`	Hierarchical Data Format (HDF)	8-bit raster image data sets, with or without associated colormap, 24-bit raster image data sets; uncompressed or with RLE or JPEG compression
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)	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.
`'pbm'`	Portable Bitmap (PBM)	Any 1-bit PBM image, ASCII (plain) or raw (binary) encoding
`'pcx'`	Windows Paintbrush (PCX)	8-bit images
`'pgm'`	Portable Graymap (PGM)	Any standard PGM image; ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per gray value
`'png'`	Portable Network Graphics (PNG)	1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit grayscale images with alpha channels; 1-bit, 2-bit, 4-bit, and 8-bit indexed images; 24-bit and 48-bit true color images with or without alpha channels
`'pnm'`	Portable Anymap (PNM)	Any of the PPM/PGM/PBM formats, chosen automatically

Format	Full Name	Variants
`'ppm'`	Portable Pixmap (PPM)	Any standard PPM image. ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per color component
`'ras'`	Sun Raster (RAS)	Any RAS image, including 1-bit bitmap, 8-bit indexed, 24-bit true color and 32-bit true color with alpha
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF)	Baseline TIFF images, including 1-bit, 8-bit, 16-bit, and 24-bit uncompressed images; 1-bit, 8-bit, 16-bit, and 24-bit images with packbits compression; 1-bit images with CCITT 1D, Group 3, and Group 4 compression
`'xwd'`	X Windows Dump (XWD)	8-bit ZPixmaps

Format-Specific Parameters

The following tables list parameters that can be used with specific file formats.

HDF-Specific Parameters

This table describes the available parameters for HDF files.

Parameter	Values	Default
`'Compression'`	One of these strings: `'none'` `'jpeg'` (valid only for grayscale and RGB images) `'rle'` (valid only for grayscale and indexed 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'`
	`'append'`
		`'overwrite'`

JPEG-Specific Parameters

This table describes the available parameters for JPEG files.

Parameter	Values	Default
'Bitdepth'	A scalar value indicating desired bitdepth; for grayscale images this can be 8, 12, or 16; for color images this can be 8 or 12.	8 (grayscale) and 8 bit per plane for color images
'Comment'	A column vector cell array of strings or a character matrix. Each row of input is written out as a comment in the JPEG file.	Empty
'Mode'	Specifies the type of compression used; value can be either of these strings: 'lossy' or 'lossless'	'lossy'
'Quality'	A number between 0 and 100; higher numbers mean higher quality (less image degradation due to compression), but the resulting file size is larger.	75

PBM-, PGM-, and PPM-Specific Parameters

This table describes the available parameters for PBM, PGM, and PPM files.

Parameter	Values	Default
'Encoding'	One of these strings: 'ASCII' for plain encoding 'rawbits' for binary encoding	'rawbits'
'MaxValue'	A scalar indicating the maximum gray or color value. Available only for PGM and PPM files. For PBM files, this value is always 1.	Default is 65535 if image array is 'uint16'; 255 otherwise.

PNG-Specific Parameters

The following table describes the available parameters for PNG files. In addition to these PNG parameters, you can use any parameter name that satisfies the PNG specification for keywords; that is, uses only printable characters, contains 80 or fewer characters, and no contains no leading or trailing spaces. The value corresponding to these user-specified parameters must be a string that contains no control characters other than linefeed.

Parameter	Values	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
`'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 true color 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`

Parameter	Values	Default
'Transparency'	This value is used to indicate transparency information only when no alpha channel is used. Set to the value that indicates which pixels should be considered transparent. (If the image uses a colormap, this value represents an index number to the colormap.) For indexed images: a Q-element vector in the range [0,1], where Q is no larger than the colormap length and each value indicates the transparency associated with the corresponding colormap entry. In most cases, Q = 1. For grayscale images: a scalar in the range [0,1]. The value indicates the grayscale color to be considered transparent. For true color images: a three-element vector in the range [0,1]. The value indicates the true-color color to be considered transparent. **Note**: You cannot specify 'Transparency' and 'Alpha' at the same time.	Empty
'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 true color 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

Parameter	Values	Default
'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 true color images: a three-element vector. For true color images with an alpha channel: a four-element vector.	Empty

RAS-Specific Parameters

This table describes the available parameters for RAS files.

Parameter	Values	Default
'Alpha'	A matrix specifying the transparency of each pixel individually; the row and column dimensions must be the same as the data array; can be uint8, uint16, or double. Can only be used with true color images.	Empty matrix ([])
'Type'	One of these strings: 'standard' (uncompressed, b-g-r color order with true color images) 'rgb' (like 'standard', but uses r-g-b color order for true color images) 'rle' (run-length encoding of 1-bit and 8-bit images)	'standard'

TIFF-Specific Parameters

This table describes the available parameters for TIFF files.

Parameter	Values	Default
'ColorSpace'	Specifies one of the following color spaces used to represent the color data. 'rgb' 'cielab' 'icclab' See "L*a*b* Color Data" on page 2-1165 for more information about this parameter.	'rgb'
'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

Parameter	Values	Default
'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' 'append'	'overwrite'

Class Support

The input array A can be of class logical, uint8, uint16, or double. Indexed images (X) can be of class uint8, uint16, or double; the associated colormap, map, must be of class double.

The class of the image written to the file depends on the format specified. For most formats, if the input array is of class uint8, imwrite outputs the data as 8-bit values. If the input array is of class uint16 and the format supports 16-bit data (JPEG, PNG, and TIFF), imwrite outputs the data as 16-bit values. If the format does not support 16-bit values, imwrite issues an error. Several formats, such as JPEG and PNG, support a parameter that lets you specify the bitdepth of the output data.

If the input array is of class double, and the image is a grayscale or RGB color image, imwrite assumes the dynamic range is [0,1] and automatically scales the data by 255 before writing it to the file as 8-bit values.

If the input array is of class double, and the image is an indexed image, imwrite converts the indices to zero-based indices by subtracting 1 from each element, and then writes the data as uint8.

If the input array is of class logical, imwrite assumes the data is a binary image and writes it to the file with a bit depth of 1, if the format allows it. BMP, PNG, or TIFF formats accept binary images as input arrays.

L*a*b* Color Data

For TIFF files only, `imwrite` can write a color image that uses the *L*a*b** color space. The 1976 CIE *L*a*b** specification defines numeric values that represent luminance (*L**) and chrominance (*a** and *b**) information.

To store *L*a*b** color data in a TIFF file, the values must be encoded to fit into either 8-bit or 16-bit storage. `imwrite` can store *L*a*b** color data in a TIFF file using these encodings:

- 8-bit and 16-bit encodings defined by the TIFF specification, called the CIELAB encodings
- 8-bit and 16-bit encodings defined by the International Color Consortium , called ICCLAB encodings

The output class and encoding used by `imwrite` to store color data depends on the class of the input array and the value you specify for the TIFF-specific `ColorSpace` parameter. The following table explains these options. (The 8-bit and 16-bit CIELAB encodings cannot be input arrays because they use a mixture of signed and unsigned values and cannot be represented as a single MATLAB array.)

Input Class and Encoding	**ColorSpace Parameter Value**	**Output Class and Encoding**
8-bit ICCLAB [1]	`'icclab'`	8-bit ICCLAB
	`'cielab'`	8-bit CIELAB
16-bit ICCLAB [2]	`'icclab'`	16-bit ICCLAB
	`'cielab'`	16-bit CIELAB
double precision 1976 CIE *L*a*b** values [3]	`'icclab'`	8-bit ICCLAB
	`'cielab'`	8-bit CIELAB

[1] 8-bit ICCLAB represents values as integers in the range [0 255]. *L** values are multiplied by `255/100`; `128` is added to both the *a** and *b** values.

[2] 16-bit ICCLAB multiplies $L*$ values by 65280/100 and represents the values as integers in the range [0, 65280]. 32768 is added to both the $a*$ and $b*$ values, which are represented as integers in the range [0,65535].

[3] $L*$ is in the dynamic range [0, 100]. $a*$ and $b*$ can take any value. Setting a* and $b*$ to 0 produces a neutral color (gray).

Example

This example appends an indexed image X and its colormap map to an existing uncompressed multipage HDF file.

```
imwrite(X,map,'your_hdf_file.hdf','Compression','none',...
'WriteMode','append')
```

See Also

fwrite, imfinfo, imformats, imread

"Bit-Mapped Images" for related functions

Purpose Convert an indexed image to an RGB image

Syntax `RGB = ind2rgb(X,map)`

Description `RGB = ind2rgb(X,map)` converts the matrix `X` and corresponding colormap `map` to RGB (true color) 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`

"Bit-Mapped Images" for related functions

ind2sub

Purpose

Subscripts from linear index

Syntax

```
[I,J] = ind2sub(siz,IND)
[I1,I2,I3,...,In] = ind2sub(siz,IND)
```

Description

The `ind2sub` command determines the equivalent subscript values corresponding to a single index into an array.

`[I,J] = ind2sub(siz,IND)` returns the matrices `I` and `J` containing the equivalent row and column subscripts corresponding to each linear index in the matrix `IND` for a matrix of size `siz`. `siz` is a 2-element vector, where `siz(1)` is the number of rows and `siz(2)` is the number of columns.

Note For matrices, `[I,J] = ind2sub(size(A),find(A>5))` returns the same values as `[I,J] = find(A>5)`.

`[I1,I2,I3,...,In] = ind2sub(siz,IND)` returns `n` subscript arrays `I1,I2,..,In` containing the equivalent multidimensional array subscripts equivalent to `IND` for an array of size `siz`. `siz` is an `n`-element vector that specifies the size of each array dimension.

Examples

Example 1. The mapping from linear indexes to subscript equivalents for a 3-by-3 matrix is

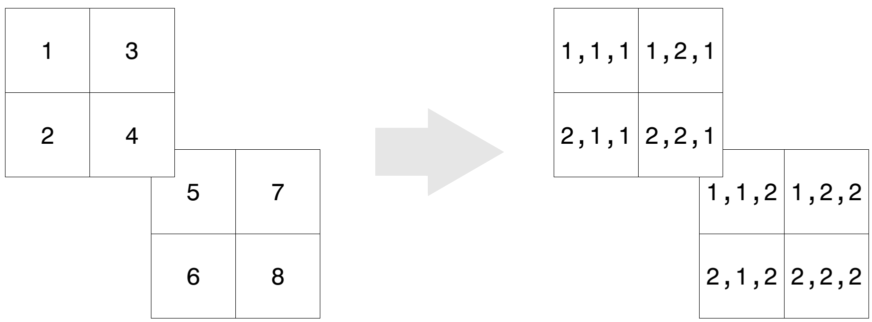
1	4	7
2	5	8
3	6	9

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

This code determines the row and column subscripts in a 3-by-3 matrix, of elements with linear indices 3, 4, 5, 6.

```
IND = [3 4 5 6]
s = [3,3];
[I,J] = ind2sub(s,IND)

I =
     3     1     2     3

J =
     1     2     2     2
```

Example 2. The mapping from linear indexes to subscript equivalents for a 2-by-2-by-2 array is



This code determines the subscript equivalents in a 2-by-2-by-2 array, of elements whose linear indices 3, 4, 5, 6 are specified in the `IND` matrix.

```
IND = [3 4;5 6];
s = [2,2,2];
[I,J,K] = ind2sub(s,IND)

I =
     1     2
     1     2

J =
     2     2
     1     1
```

```
K =
     1     1
     2     2
```

See Also `find`, `size`, `sub2ind`

Purpose Infinity

Syntax

```
Inf
Inf('double')
Inf('single')
Inf(n)
Inf(m,n)
Inf(m,n,p,...)
Inf(...,classname)
```

Description

`Inf` returns the IEEE arithmetic representation for positive infinity. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values.

`Inf('double')` is the same as `Inf` with no inputs.

`Inf('single')` is the single precision representation of `Inf`.

`Inf(n)` is an `n`-by-`n` matrix of `Inf`s.

`Inf(m,n)` or `inf([m,n])` is an `m`-by-`n` matrix of `Inf`s.

`Inf(m,n,p,...)` or `Inf([m,n,p,...])` is an `m`-by-`n`-by-`p`-by-... array of `Inf`s.

`Inf(...,classname)` is an array of `Inf`s of class specified by `classname`. `classname` must be either `'single'` or `'double'`.

Examples

`1/0`, `1.e1000`, `2^2000`, and `exp(1000)` all produce `Inf`.

`log(0)` produces `-Inf`.

`Inf-Inf` and `Inf/Inf` both produce `NaN` (Not-a-Number).

See Also

`isinf`, `NaN`

inferiorto

Purpose Inferior class relationship

Syntax

```
inferiorto('class1','class2',...)
```

Description The `inferiorto` function establishes a hierarchy that 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`

Purpose Display Release Notes for MathWorks products

Syntax

```
info
```

Description `info` displays the Release Notes in the Help browser, containing information about new features, problems from previous releases that have been fixed in the current release, and known problems, all organized by product.

See Also `help`, `lookfor`, `path`, `version`, `which`

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

Example 1. This example creates a simple inline function to square a number.

```
g = inline('t^2')
g =

     Inline function:
```

```
      g(t) = t^2
```

You can convert the result to a string using the `char` function.

```
char(g)

ans =

t^2
```

Example 2. This example creates an inline function to represent the formula $f = 3\sin(2x^2)$. The resulting inline function can be evaluated with the `argnames` and `formula` functions.

```
f = inline('3*sin(2*x.^2)')

f =
     Inline function:
     f(x) = 3*sin(2*x.^2)

argnames(f)

ans =
    'x'

formula(f)
ans =

3*sin(2*x.^2)ans =
```

Example 3. This call to `inline` defines the function `f` to be dependent on two variables, `alpha` and `x`:

```
f = inline('sin(alpha*x)')

f =
     Inline function:
     f(alpha,x) = sin(alpha*x)
```

If `inline` does not return the desired function variables or if the function variables are in the wrong order, you can specify the desired variables explicitly with the `inline` argument list.

```
g = inline('sin(alpha*x)','x','alpha')

g =

     Inline function:
     g(x,alpha) = sin(alpha*x)
```

Purpose

Return functions in memory

Syntax

```
M = inmem
[M, X] = inmem
[M, X, J] = inmem
[...] = inmem('-completenames')
```

Description

`M = inmem` returns a cell array of strings containing the names of the M-files that are currently loaded.

`[M, X] = inmem` returns an additional cell array `X` containing the names of the MEX-files that are currently loaded.

`[M, X, J] = inmem` also returns a cell array `J` containing the names of the Java classes that are currently loaded.

`[...] = inmem('-completenames')` returns not only the names of the currently loaded M- and MEX-files, but the path and filename extension for each as well. No additional information is returned for loaded Java classes.

Examples

Example 1

This example lists the M-files that are required to run `erf`.

```
clear all;        % Clear the workspace
erf(0.5);

M = inmem
M =
   'erf'
```

Example 2

Generate a plot, and then find the M- and MEX-files that had been loaded to perform this operation:

```
clear all
surf(peaks)

[m x] = inmem('-completenames');
```

```
m(1:5)
ans =
    'F:\matlab\toolbox\matlab\ops\ismember.m'
    'F:\matlab\toolbox\matlab\datatypes\@opaque\double.m'
    'F:\matlab\toolbox\matlab\datatypes\isfield.m'
    'F:\matlab\toolbox\matlab\graphics\gcf.m'
    'F:\matlab\toolbox\matlab\elmat\meshgrid.m'

x(1:end)
ans =
    'F:\matlab\toolbox\matlab\graph2d\private\lineseriesmex.dll'
```

See Also

`clear`

Purpose

Detect points inside a polygonal region

Syntax

```
IN = inpolygon(X,Y,xv,yv)
[IN ON] = inpolygon(X,Y,xv,yv)
```

Description

`IN = inpolygon(X,Y,xv,yv)` returns a matrix `IN` the same size as `X` and `Y`. Each element of `IN` is assigned the value 1 or 0 depending on whether the point `(X(p,q),Y(p,q))` is inside the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

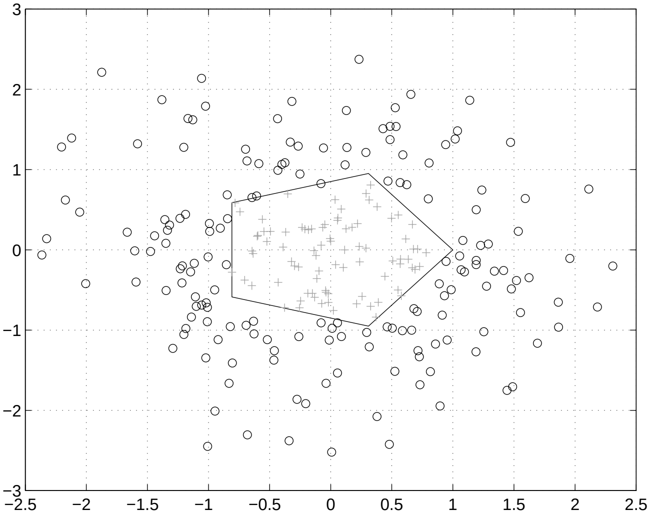
`IN(p,q) = 1`	If `(X(p,q),Y(p,q))` is inside the polygonal region or on the polygon boundary
`IN(p,q) = 0`	If `(X(p,q),Y(p,q))` is outside the polygonal region

`[IN ON] = inpolygon(X,Y,xv,yv)` returns a second matrix `ON` the same size as `X` and `Y`. Each element of `ON` is assigned the value 1 or 0 depending on whether the point `(X(p,q),Y(p,q))` is on the boundary of the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

`IN(p,q) = 1`	If `(X(p,q),Y(p,q))` is on the polygon boundary
`IN(p,q) = 0`	If `(X(p,q),Y(p,q))` is inside or outside the polygon boundary

Examples

```
L = linspace(0,2.*pi,6); xv = cos(L)';yv = sin(L)';
xv = [xv ; xv(1)]; yv = [yv ; yv(1)];
x = randn(250,1); y = randn(250,1);
in = inpolygon(x,y,xv,yv);
plot(xv,yv,x(in),y(in),'r+',x(~in),y(~in),'bo')
```


3
2
1
0
−1
−2
−3
−2.5
−2
−1.5
−1
−0.5
0
0.5
1
1.5
2
2.5

Purpose Request user input

Syntax

```
user_entry = input('prompt')
user_entry = input('prompt','s')
```

Description The response to the `input` prompt can be any MATLAB expression, which is evaluated using the variables in the current workspace.

`user_entry = input('`*`prompt`*`')` displays *`prompt`* as a prompt on the screen, waits for input from the keyboard, and returns the value entered in `user_entry`.

`user_entry = input('`*`prompt`*`','s')` returns the entered string as a text variable rather than as a variable name or numerical value.

Remarks If you press the **Return** key without entering anything, `input` returns an empty matrix.

The text string for the prompt can contain one or more `'\n'` characters. The `'\n'` means to skip to the next line. This allows the prompt string to span several lines. To display just a backslash, use `'\\'`.

Examples Press **Return** to select a default value by detecting an empty matrix:

```
reply = input('Do you want more? Y/N [Y]: ','s');
if isempty(reply)
    reply = 'Y';
end
```

See Also `keyboard`, `menu`, `ginput`, `uicontrol`

Purpose

Create input dialog box

Syntax

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,dlg_title)
answer = inputdlg(prompt,dlg_title,num_lines)
answer = inputdlg(prompt,dlg_title,num_lines,defAns)
answer = inputdlg(prompt,dlg_title,num_lines,defAns,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,dlg_title)` `dlg_title` specifies a title for the dialog box.

`answer = inputdlg(prompt,dlg_title,num_lines)` `num_lines` specifies the number of lines for each user-entered value. `num_lines` can be a scalar, column vector, or matrix.

- If `num_lines` is a scalar, it applies to all prompts.
- If `num_lines` is a column vector, each element specifies the number of lines of input for a prompt.
- If `num_lines` is a matrix, it should be size m-by-2, where m is the number of prompts on the dialog box. Each row refers to a prompt. The first column specifies the number of lines of input for a prompt. The second column specifies the width of the field in characters.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns)` `defAns` specifies the default value to display for each prompt. `defAns` must contain the same number of elements as `prompt` and all elements must be strings.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns,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.

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

Display graphical user interface to list and modify property values

Syntax

```
inspect
inspect(h)
inspect([h1,h2,...])
```

Description

`inspect` creates a separate Property Inspector window to enable the display and modification of the properties of any object you select in the figure window or Layout Editor. If no object is selected, the Property Inspector is blank.

`inspect(h)` creates a Property Inspector window for the object whose handle is `h`.

`inspect([h1,h2,...])` creates a Property Inspector window for the objects whose handles are elements of the vector `[h1,h2,...]`. If the objects are of different types, the inspector displays only those properties the objects have in common.

To change the value of any property, click on the property name shown at the left side of the window, and then enter the new value in the field at the right.

Notes `inspect h` displays a Property Inspector window that enables modification of the string `'h'`, not the object whose handle is `h`.

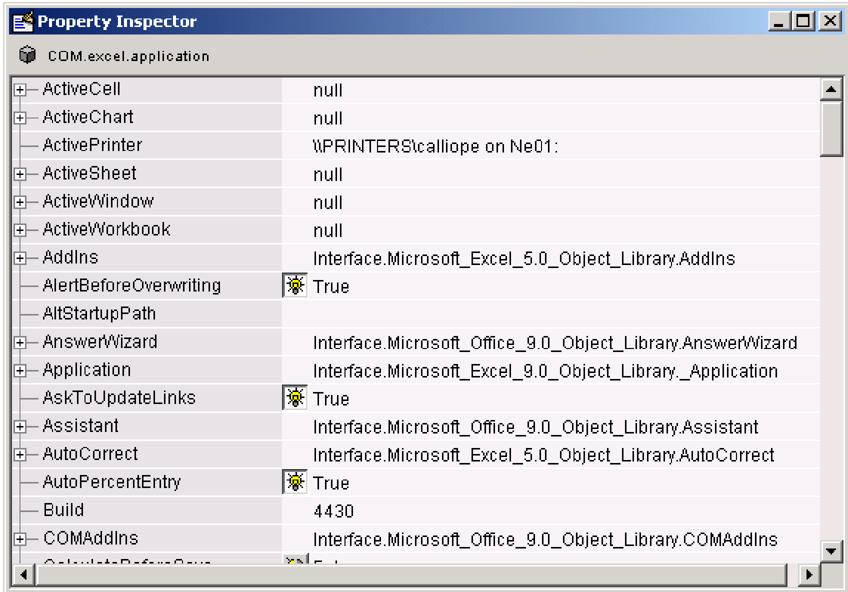
If you modify properties at the MATLAB command line, you must refresh the Property Inspector window to see the change reflected there. Refresh the Property Inspector by reinvoking `inspect` on the object.

Example

Create a COM Excel server and open a Property Inspector window with `inspect`:

```
h = actxserver('excel.application');
inspect(h)
```

Scroll down until you see the `DefaultFilePath` property. Click on the property name shown at the left. Then replace the text at the right with `C:\ExcelWork`.



Check this field in the MATLAB command window and confirm that it has changed:

```
get(h,'DefaultFilePath')
ans =
   C:\ExcelWork
```

See Also

`get`, `set`, `isprop`, `guide`, `addproperty`, `deleteproperty`

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)
I = int64(X)
```

Description `I = int*(X)` converts the elements of array `X` into signed integers. `X` can be any numeric object (such as a `double`). The results of an `int*` operation are shown in the next table.

Operation	Output Range	Output Type	Bytes per Element	Output Class
`int8`	-128 to 127	Signed 8-bit integer	1	`int8`
`int16`	-32,768 to 32,767	Signed 16-bit integer	2	`int16`
`int32`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	4	`int32`
`int64`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	8	`int64`

`double` and `single` values are rounded to the nearest `int*` value on conversion. A value of `X` that is above or below the range for an integer class is mapped to one of the endpoints of the range. For example,

```
int16(40000)
ans =
   32767
```

If `X` is already a signed integer of the same class, then `int*` has no effect.

You can define or overload your own methods for `int*` (as you can for any object) by placing the appropriately named method in an `@int*` directory within a directory on your path. Type `help datatypes` for the names of the methods you can overload.

Remarks

Most operations that manipulate arrays without changing their elements are defined for integer values. Examples are `reshape`, `size`, the logical and relational operators, subscripted assignment, and subscripted reference.

Some arithmetic operations are defined for integer arrays on interaction with other integer arrays of the same class (e.g., where both operands are `int16`). Examples of these operations are `+`, `-`, `.*`, `./`, `.\` and `.^`. If at least one operand is scalar, then `*`, `/`, `\`, and `^` are also defined. Integer arrays may also interact with scalar `double` variables, including constants, and the result of the operation is an integer array of the same class. Integer arrays saturate on overflow in arithmetic.

A particularly efficient way to initialize a large array is by specifying the data type (i.e., class name) for the array in the `zeros`, `ones`, or `eye` function. For example, to create a 100-by-100 `int64` array initialized to zero, type

```
I = zeros(100, 100, 'int64');
```

An easy way to find the range for any MATLAB integer type is to use the `intmin` and `intmax` functions as shown here for `int32`:

```
intmin('int32')                 intmax('int32')
ans =                           ans =
   -2147483648                     2147483647
```

See Also

`double`, `single`, `uint8`, `uint16`, `uint32`, `uint64`, `intmax`, `intmin`

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,'extrap')
yi = interp1(x,Y,xi,method,extrapval)
pp = interp1(x,Y,method,'pp')
```

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

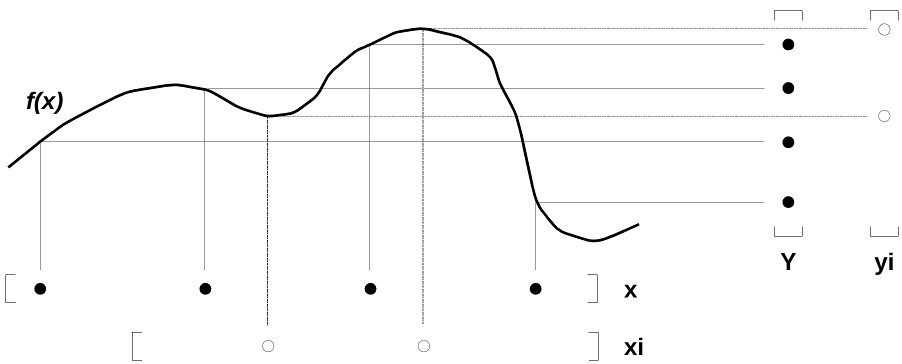
For the `'nearest'`, `'linear'`, and `'v5cubic'` methods, `interp1(x,Y,xi,method)` returns `NaN` for any element of `xi` that is outside the interval spanned by `x`. For all other methods, `interp1` performs extrapolation for out of range values.

`yi = interp1(x,Y,xi,method,'extrap')` uses the specified method to perform extrapolation for out of range values.

`yi = interp1(x,Y,xi,method,extrapval)` returns the scalar `extrapval` for out of range values. `NaN` and `0` are often used for `extrapval`.

`pp = interp1(x,Y,method,'pp')` uses the specified method to generate the piecewise polynomial form (ppform) of `Y`. You can use any of the methods in the preceding table, except for `'v5cubic'`.

The `interp1` command interpolates between data points. It finds values at intermediate points, of a one-dimensional function $f(x)$ that underlies the data. This function is shown below, along with the relationship between vectors `x`, `Y`, `xi`, and `yi`.



Interpolation is the same operation as *table lookup*. Described in table lookup terms, the *table* is `[x,Y]` and `interp1` *looks up* the elements of `xi` in `x`, and, based upon their locations, returns values `yi` interpolated within the elements of `Y`.

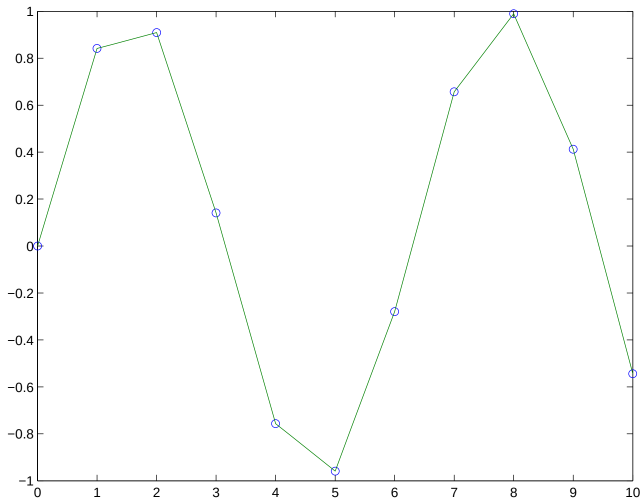
Note `interp1q` is quicker than `interp1` on non-uniformly spaced data because it does no input checking. For `interp1q` to work properly, `x` must be a monotonically increasing column vector and `Y` must be a column vector or matrix with `length(X)` rows. Type `help interp1q` at the command line for more information.

Examples

Example 1. Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = 0:10;
```

```
y = sin(x);
xi = 0:.25:10;
yi = interp1(x,y,xi);
plot(x,y,'o',xi,yi)
```



Example 2. 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];
```

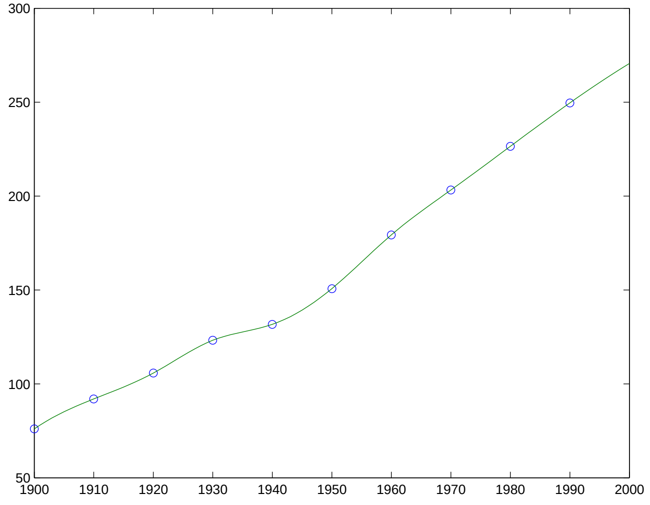
The expression `interp1(t,p,1975)` interpolates within the census data to estimate the population in 1975. The result is

```
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.

interp2

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)
ZI = interp2(...,method, extrapval)
```

Description `ZI = interp2(X,Y,Z,XI,YI)` returns matrix `ZI` containing elements corresponding to the elements of `XI` and `YI` and determined by interpolation within the two-dimensional function specified by matrices `X`, `Y`, and `Z`. `X` and `Y` must be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. Matrices `X` and `Y` specify the points at which the data `Z` is given. Out of range values are returned as `NaN`s.

`XI` and `YI` can be matrices, in which case `interp2` returns the values of `Z` corresponding to the points `(XI(i,j),YI(i,j))`. Alternatively, you can pass in the row and column vectors `xi` and `yi`, respectively. In this case, `interp2` interprets these vectors as if you issued the command `meshgrid(xi,yi)`.

`ZI = interp2(Z,XI,YI)` assumes that `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`.

`ZI = interp2(Z,ntimes)` expands `Z` by interleaving interpolates between every element, working recursively for `ntimes`. `interp2(Z)` is the same as `interp2(Z,1)`.

`ZI = interp2(X,Y,Z,XI,YI,method)` specifies an alternative interpolation method:

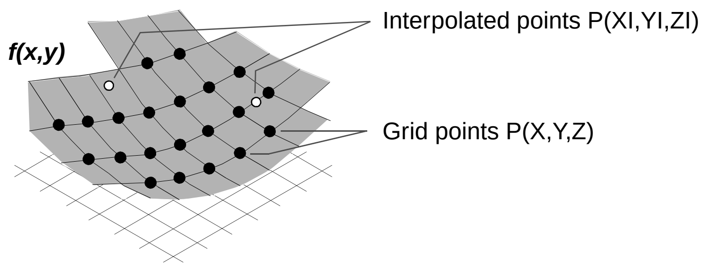
`'nearest'`	Nearest neighbor interpolation
`'linear'`	Bilinear interpolation (default)
`'spline'`	Cubic spline interpolation
`'cubic'`	Bicubuc interpolation

All interpolation methods require that `X` and `Y` be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. If you provide two monotonic vectors, `interp2` changes them to a plaid internally. Variable spacing is handled by mapping the given values in X, Y, XI, and YI to an equally spaced domain before interpolating. For faster interpolation when X and Y are equally spaced and monotonic, use the methods '*linear', '*cubic', '*spline', or '*nearest'.

ZI = interp2(...,method, extrapval) specificies a method and a scalar value for ZI outside of the domain created by X and Y. Thus, ZI equals extrapval for any value of YI or XI that is not spanned by Y or X respectively. A method must be specified to use extrapval. The default method is 'linear'.

Remarks

The interp2 command interpolates between data points. It finds values of a two-dimensional function $f(x, y)$ underlying the data at intermediate points.

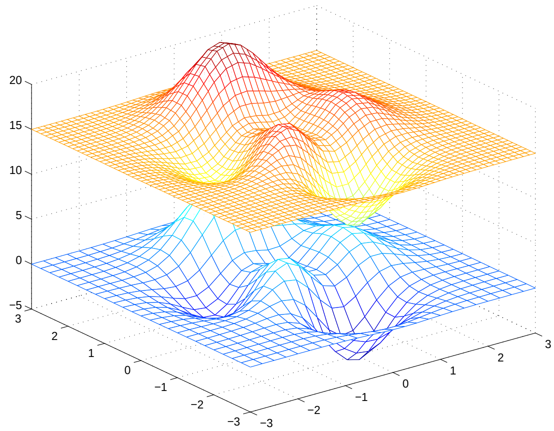


Interpolation is the same operation as table lookup. Described in table lookup terms, the table is tab = [NaN,Y; X,Z] and interp2 looks up the elements of XI in X, YI in Y, and, based upon their location, returns values ZI interpolated within the elements of Z.

Examples

Example 1. Interpolate the peaks function over a finer grid.

```
[X,Y] = meshgrid(-3:.25:3);
Z = peaks(X,Y);
[XI,YI] = meshgrid(-3:.125:3);
ZI = interp2(X,Y,Z,XI,YI);
mesh(X,Y,Z), hold, mesh(XI,YI,ZI+15)
hold off
axis([-3 3 -3 3 -5 20])
```



Example 2. Given this set of employee data,

```
years = 1950:10:1990;
service = 10:10:30;
wage = [150.697 199.592 187.625
        179.323 195.072 250.287
        203.212 179.092 322.767
        226.505 153.706 426.730
        249.633 120.281 598.243];
```

it is possible to interpolate to find the wage earned in 1975 by an employee with 15 years' service:

```
w = interp2(service,years,wage,15,1975)
w =
  190.6287
```

See Also

griddata, interp1, interp3, interpn, meshgrid

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)
VI = INTERP3(...,'method',extrapval)
```

Description `VI = interp3(X,Y,Z,V,XI,YI,ZI)` interpolates to find `VI`, the values of the underlying three-dimensional function `V` at the points in arrays `XI`,`YI` and `ZI`. `XI`,`YI`, `ZI` must be arrays of the same size, or vectors. Vector arguments that are not the same size, and have mixed orientations (i.e. with both row and column vectors) are passed through `meshgrid` to create the `Y1`, `Y2`, `Y3` arrays. Arrays `X`, `Y`, and `Z` specify the points at which the data `V` is given. Out of range values are returned as `NaN`.

`VI = interp3(V,XI,YI,ZI)` assumes `X=1:N`, `Y=1:M`, `Z=1:P` where `[M,N,P]=size(V)`.

`VI = interp3(V,ntimes)` expands `V` by interleaving interpolates between every element, working recursively for `ntimes` iterations. The command `interp3(V)` is the same as `interp3(V,1)`.

`VI = interp3(...,method)` specifies alternative methods:

`'linear'`	Linear interpolation (default)
`'cubic'`	Cubic interpolation
`'spline'`	Cubic spline interpolation
`'nearest'`	Nearest neighbor interpolation

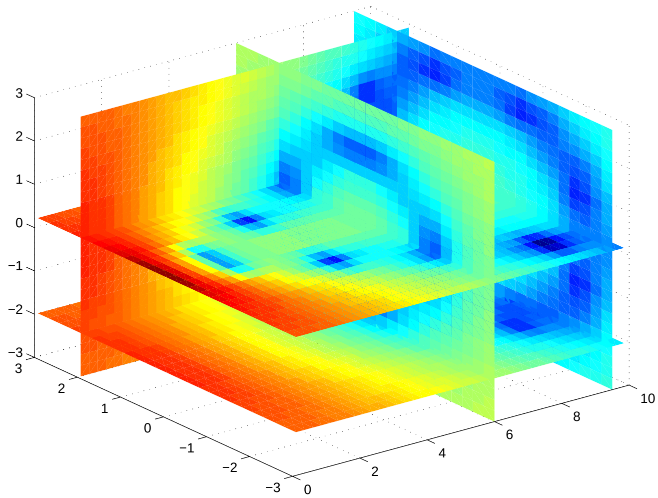
`VI = INTERP3(...,'method',extrapval)` specifies a method and a value for `VI` outside of the domain created by `X`,`Y` and `Z`. Thus, `VI` equals `extrapval` for any value of `XI`, `YI` or `ZI` that is not spanned by `X`, `Y`, and `Z`, respectively. You must specify a method to use `extrapval`. The default method is `'linear'`.

Discussion All the interpolation methods require that `X`,`Y` and `Z` be monotonic and have the same format ("plaid") as if they were created using `meshgrid`. `X`, `Y`, and `Z` can be non-uniformly spaced. For faster interpolation when `X`, `Y`, and `Z` are equally spaced and monotonic, use the methods `'*linear'`, `'*cubic'`, or `'*nearest'`.

Examples

To generate a coarse approximation of `flow` and interpolate over a finer mesh:

```
[x,y,z,v] = flow(10);
[xi,yi,zi] = meshgrid(.1:.25:10, -3:.25:3, -3:.25:3);
vi = interp3(x,y,z,v,xi,yi,zi); % vi is 25-by-40-by-25
slice(xi,yi,zi,vi,[6 9.5],2,[-2 .2]), shading flat
```



See Also

`interp1`, `interp2`, `interpn`, `meshgrid`

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`

interpn

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-dimensional array `V`, `interpn` is called with `2*N+1` arguments. Arrays `X1`, `X2`, `X3`, etc. specify the points at which the data `V` is given. Out of range values are returned as `NaN`s. `Y1`, `Y2`, `Y3`, etc. must be arrays of the same size, or vectors. Vector arguments that are not the same size, and have mixed orientations (i.e. with both row and column vectors) are passed through `ndgrid` to create the `Y1`, `Y2`, `Y3`, etc. arrays. `interpn` works for all n-dimensional arrays with 2 or more dimensions.

`VI = interpn(V,Y1,Y2,Y3,...)` interpolates as above, assuming `X1 = 1:size(V,1)`, `X2 = 1:size(V,2)`, `X3 = 1:size(V,3)`, etc.

`VI = interpn(V,ntimes)` expands `V` by interleaving interpolates between each element, working recursively for `ntimes` iterations. `interpn(V,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

`VI = INTERPN(...,'method',extrapval)` specifies a method and a value for `VI` outside of the domain created by `X1`, `X2`,... Thus, `VI` equals `extrapval` for any value of `Y1`, `Y2`,.. that is not spanned by `X1`, `X2`,... respectively. You must specify a method to use `extrapval`. The default method is `'linear'`.

`interpn` requires that `X1`, `X2`, `X3`, ... be monotonic and plaid (as if they were created using `ndgrid`). `X1`, `X2`, `X3`, and so on can be non-uniformly spaced.

Discussion

All the interpolation methods require that X1,X2, X3 ... be monotonic and have the same format ("plaid") as if they were created using ndgrid. X1,X2,X3,... and Y1, Y2, Y3, etc. can be non-uniformly spaced. For faster interpolation when X1, X2, X3, etc. are equally spaced and monotonic, use the methods '*linear', '*cubic', or '*nearest'.

See Also

interp1, interp2, interp3, ndgrid

interpstreamspeed

Purpose Interpolate stream line vertices from flow speed

Syntax

```
interpstreamspeed(X,Y,Z,U,V,W,vertices)
interpstreamspeed(U,V,W,vertices)
interpstreamspeed(X,Y,Z,speed,vertices)
interpstreamspeed(speed,vertices)

interpstreamspeed(X,Y,U,V,vertices)
interpstreamspeed(U,V,vertices)
interpstreamspeed(X,Y,speed,vertices)
interpstreamspeed(speed,vertices)

interpstreamspeed(...,sf)
vertsout = interpstreamspeed(...)
```

Description

`interpstreamspeed(X,Y,Z,U,V,W,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`).

`interpstreamspeed(U,V,W,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(U)`.

`interpstreamspeed(X,Y,Z,speed,vertices)` uses the 3-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p]=size(speed)`.

`interpstreamspeed(X,Y,U,V,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`. The arrays `X`, `Y` define the coordinates for U, V and must be monotonic and 2-D plaid (as if produced by meshgrid)

interpstreamspeed(U,V,vertices) assumes X and Y are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where [M N]=size(U).

interpstreamspeed(X,Y,speed,vertices) uses the 2-D array speed for the speed of the vector field.

interpstreamspeed(speed,vertices) assumes X and Y are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where [M,N]= size(speed).

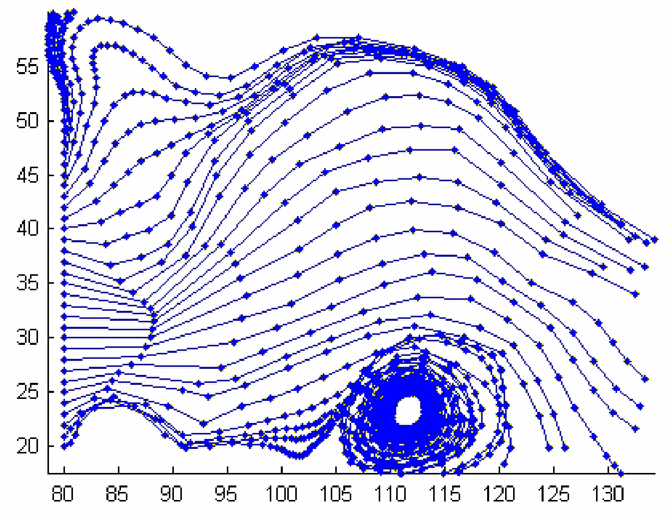
interpstreamspeed(...,sf) uses sf to scale the magnitude of the vector data and therefore controls the number of interpolated vertices. For example, if sf is 3, then interpstreamspeed creates only one-third of the vertices.

vertsout = interpstreamspeed(...) returns a cell array of vertex arrays.

Examples

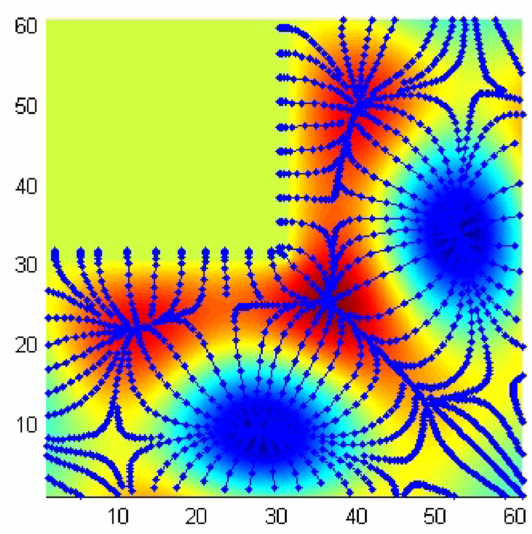
This example draws streamlines using the vertices returned by interpstreamspeed. Dot markers indicate the location of each vertex. This example enables you to visualize the relative speeds of the flow data. Streamlines having widely spaced vertices indicate faster flow; those with closely spaced vertices indicate slower flow.

```
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.2);
sl = streamline(iverts);
set(sl,'Marker','.')
axis tight; view(2); daspect([1 1 1])
```



This example plots streamlines whose vertex spacing indicates the value of the gradient along the streamline.

```
z = membrane(6,30);
[u v] = gradient(z);
[verts averts] = streamslice(u,v);
iverts = interpstreamspeed(u,v,verts,15);
sl = streamline(iverts);
set(sl,'Marker','.')
hold on; pcolor(z); shading interp
axis tight; view(2); daspect([1 1 1])
```



See Also

`stream2`, `stream3`, `streamline`, `streamslice`, `streamparticles`

"Volume Visualization" for related functions

intersect

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`, `issorted`, `setdiff`, `setxor`, `union`, `unique`

Purpose Return largest possible integer value

Syntax

```
v = intmax
v = intmax('classname')
```

Description

`v = intmax` is the largest positive value that can be represented in MATLAB with a 32-bit integer. Any value larger than the value returned by `intmax` saturates to the `intmax` value when cast to a 32-bit integer.

`v = intmax('classname')` is the largest positive value in the integer class `classname`. Valid values for the string `classname` are

'int8'	'int16'	'int32'	'int64'
'uint8'	'uint16'	'uint32'	'uint64'

`intmax('int32')` is the same as `intmax` with no arguments.

Examples

Find the maximum value for a 64-bit signed integer:

```
v = intmax('int64')
v =
  9223372036854775807
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
  2147483647
```

Compare the result with the default value returned by `intmax`:

```
isequal(x, intmax)
ans =
     1
```

See Also

`intmin`, `realmax`, `realmin`, `int8`, `uint8`, `isa`, `class`

intmin

Purpose Return smallest possible integer value

Syntax

```
v = intmin
v = intmin('classname')
```

Description

`v = intmin` is the smallest value that can be represented in MATLAB with a 32-bit integer. Any value smaller than the value returned by `intmin` saturates to the `intmin` value when cast to a 32-bit integer.

`v = intmin('classname')` is the smallest positive value in the integer class `classname`. Valid values for the string `classname` are

'int8'	'int16'	'int32'	'int64'
'uint8'	'uint16'	'uint32'	'uint64'

`intmin('int32')` is the same as `intmin` with no arguments.

Examples

Find the minimum value for a 64-bit signed integer:

```
v = intmin('int64')
v =
 -9223372036854775808
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
  2147483647
```

Compare the result with the default value returned by `intmin`:

```
isequal(x, intmin)
ans =
     1
```

See Also `intmax`, `realmin`, `realmax`, `int8`, `uint8`, `isa`, `class`

Purpose Control state of integer warnings

Syntax

```
intwarning('action')
s = intwarning('action')
intwarning(s)
sOld = intwarning(sNew)
```

Description MATLAB has four types of integer warnings. The `intwarning` function enables, disables, or returns information on these warnings:

- `MATLAB:intConvertNaN` — Warning on an attempt to convert NaN (Not a Number) to an integer. The result of the operation is zero.
- `MATLAB:intConvertNonIntVal` — Warning on an attempt to convert a non-integer value to an integer. The result is that the input value is rounded to the nearest integer for that class.
- `MATLAB:intConvertOverflow` — Warning on overflow when attempting to convert from a numeric class to an integer class. The result is the maximum value for the target class.
- `MATLAB:intMathOverflow` — Warning on overflow when attempting an integer arithmetic operation. The result is the maximum value for the class of the input value. MATLAB also issues this warning when NaN is computed (e.g., `int8(0)/0`).

`intwarning('`*`action`*`')` sets or displays the state of integer warnings in MATLAB according to the string, *`action`*. There are three possible actions, as shown here. The default state is `'off'`.

Action	**Description**
`off`	Disable the display of integer warnings
`on`	Enable the display of integer warnings
`query`	Display the state of all integer warnings

`s = intwarning('`*`action`*`')` sets the state of integer warnings in MATLAB according to the string *`action`*, and then returns the previous state in a 4-by-1 structure array, `s`. The return structure array has two fields: `identifier` and `state`.

`intwarning(s)` sets the state of integer warnings in MATLAB according to the `identifier` and `state` fields in structure array `s`.

`sOld = intwarning(sNew)` sets the state of integer warnings in MATLAB according to `sNew`, and then returns the previous state in `sOld`.

Remarks

Examples of the four types of integer warnings are shown here.

MATLAB:intConvertNaN

Attempt to convert NaN (Not a Number) to an unsigned integer:

```
uint8(NaN);
Warning: NaN converted to uint8(0).
```

MATLAB:intConvertNonIntVal

Attempt to convert a floating point number to an unsigned integer:

```
uint8(2.7);
Warning: Conversion rounded non-integer floating point
   value to nearest uint8 value.
```

MATLAB:intConvertOverflow

Attempt to convert a large unsigned integer to a signed integer, where the operation overflows:

```
int8(uint8(200));
Warning: Out of range value converted to intmin('int8')
   or intmax('int8').
```

MATLAB:intMathOverflow

Attempt an integer arithmetic operation that overflows:

```
intmax('uint8') + 5;
Warning: Out of range value or NaN computed in integer arithmetic.
```

Examples

Check the initial state of integer warnings:

```
intwarning('query')
The state of warning 'MATLAB:intConvertNaN' is 'off'.
The state of warning 'MATLAB:intConvertNonIntVal' is 'off'.
The state of warning 'MATLAB:intConvertOverflow' is 'off'.
The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

Convert a floating point value to an 8-bit unsigned integer. MATLAB does the conversion, but that requires rounding the resulting value. Because all integer warnings have been disabled, no warning is displayed:

```
uint8(2.7)
ans =
    3
```

Store this state in structure array `iwState`:

```
iwState = intwarning('query');
```

Change the state of the `ConvertNonIntVal` warning to `'on'` by first setting the state to `'on'` in the `iwState` structure array, and then loading `iwState` back into the internal integer warning settings for your MATLAB session:

```
maxintwarn = 4;

for k = 1:maxintwarn
   if strcmp(iwState(k).identifier, 'MATLAB:intConvertNonIntVal')
       iwState(k).state = 'on';
       intwarning(iwState);
    end
end
```

Verify that the state of `ConvertNonIntVal` has changed:

```
intwarning('query')
The state of warning 'MATLAB:intConvertNaN' is 'off'.
The state of warning 'MATLAB:intConvertNonIntVal' is 'on'.
The state of warning 'MATLAB:intConvertOverflow' is 'off'.
The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

Now repeat the conversion from floating point to integer. This time MATLAB displays the warning:

```
uint8(2.7)
Warning: Conversion rounded non-integer floating point value
   to nearest uint8 value.
ans =
    3
```

See Also

warning, lastwarn

Purpose Matrix inverse

Syntax

```
Y = inv(X)
```

Description `Y = inv(X)` returns the inverse of the square matrix `X`. A warning message is printed if `X` is badly scaled or nearly singular.

In practice, it is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises when solving the system of linear equations $Ax = b$. One way to solve this is with `x = inv(A)*b`. A better way, from both an execution time and numerical accuracy standpoint, is to use the matrix division operator `x = A\b`. This produces the solution using Gaussian elimination, without forming the inverse. See `\` and `/` for further information.

Examples Here is an example demonstrating the difference between solving a linear system by inverting the matrix with `inv(A)*b` and solving it directly with `A\b`. A random matrix `A` of order 500 is constructed so that its condition number, `cond(A)`, is `1.e10`, and its norm, `norm(A)`, is `1`. The exact solution `x` is a random vector of length 500 and the right-hand side is `b = A*x`. Thus the system of linear equations is badly conditioned, but consistent.

On a 300 MHz, laptop computer the statements

```
n = 500;
Q = orth(randn(n,n));
d = logspace(0,-10,n);
A = Q*diag(d)*Q';
x = randn(n,1);
b = A*x;
tic, y = inv(A)*b; toc
err = norm(y-x)
res = norm(A*y-b)
```

produce

```
elapsed_time =
    1.4320
err =
    7.3260e-006
res =
    4.7511e-007
```

while the statements

```
tic, z = A\b, toc
err = norm(z-x)
res = norm(A*z-b)
```

produce

```
elapsed_time =
    0.6410
err =
    7.1209e-006
res =
    4.4509e-015
```

It takes almost two and one half times as long to compute the solution with `y = inv(A)*b` as with `z = A\b`. Both produce computed solutions with about the same error, `1.e-6`, reflecting the condition number of the matrix. But the size of the residuals, obtained by plugging the computed solution back into the original equations, differs by several orders of magnitude. The direct solution produces residuals on the order of the machine accuracy, even though the system is badly conditioned.

The behavior of this example is typical. Using `A\b` instead of `inv(A)*b` is two to three times as fast and produces residuals on the order of machine accuracy, relative to the magnitude of the data.

Algorithm

Inputs of Type Double

For inputs of type `double`, `inv` uses the following LAPACK routines to compute the matrix inverse:

Matrix	Routine
Real	`DLANGE`, `DGETRF`, `DGECON`, `DGETRI`
Complex	`ZLANGE`, `ZGETRF`, `ZGECON`, `ZGETRI`

Inputs of Type Single

For inputs of type `single`, `inv` uses the following LAPACK routines to compute the matrix inverse:

Matrix	Routine
Real	`SLANGE`, `SGETRF`, `SGECON`, `SGETRI`
Complex	`CLANGE`, `CGETRF`, `CGECON`, `CGETRI`

See Also

`det`, `lu`, `rref`

The arithmetic operators `\`, `/`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

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] [1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

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

Description These functions detect the state of MATLAB entities:

`isappdata`	Determine if object has specific application-defined data
`iscell`	Determine if input is a cell array
`iscellstr`	Determine if input is a cell array of strings
`ischar`	Determine if input is a character array
`isdir`	Determine if input is a directory
`isempty`	Determine if input is an empty array
`isequal`	Determine if arrays are numerically equal
`isequalwithequalnans`	Determine if arrays are numerically equal, treating `NaN`s as equal
`isevent`	Determine if input is an event of an object
`isfield`	Determine if input is a MATLAB structure array field
`isfinite`	Detect finite elements of an array
`isfloat`	Determine if input is a floating-point array
`isglobal`	Determine if input is a global variable
`ishandle`	Detect valid graphics object handles
`ishold`	Determine if graphics hold state is on
`isinf`	Detect infinite elements of an array
`isinteger`	Determine if input is an integer array
`isjava`	Determine if input is a Java object
`iskeyword`	Determine if input is a MATLAB keyword

islogical	Determine if input is a logical array
ismember	Detect members of a specific set
ismethod	Determine if input is a method of an object
isnan	Detect elements of an array that are not a number (NaN)
isnumeric	Determine if input is a numeric array
isobject	Determine if input is a MATLAB OOPs object
ispc	Determine if PC (Windows) version of MATLAB
isprime	Detect prime elements of an array
isprop	Determine if input is a property of an object
isreal	Determine if all array elements are real numbers
isscalar	Determine if input is scalar
issorted	Determine if set elements are in sorted order
isspace	Detect space characters in an array
issparse	Determine if input is a sparse array
isstrprop	Determine if string is of specified category
isstruct	Determine if input is a MATLAB structure array
isstudent	Determine if student edition of MATLAB
isunix	Determine if UNIX version of MATLAB
isvalid	Determine if timer object is valid
isvarname	Determine if input is a valid variable name
isvector	Determine if input is a vector

See Also

`isa`

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

`logical`	Logical array of `true` and `false` values
`char`	Characters array
`numeric`	Integer or floating-point array
`integer`	Signed or unsigned integer array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64-bit unsigned integer array
`float`	Single- or double-precision floating-point array
`single`	Single-precision floating-point array
`double`	Double-precision floating-point array
`cell`	Cell array
`struct`	Structure array
`function_handle`	Function handle
`'class_name'`	Custom MATLAB object class or Java class

To check for a sparse array, use `issparse`. To check for a complex array, use `~isreal`.

Examples

```
isa(rand(3,4),'double')
ans =
    1
```

The following example creates an instance of the user-defined MATLAB class named `polynom`. The `isa` function identifies the object as being of the `polynom` class.

```
polynom_obj = polynom([1 0 -2 -5]);
isa(polynom_obj, 'polynom')
ans =
     1
```

See Also

`class`, is*

isappdata

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`

Purpose

Determine if input is a cell array

Syntax

```
tf = iscell(A)
```

Description

`tf = iscell(A)` returns logical true (`1`) if `A` is a cell array and logical false (`0`) otherwise.

Examples

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];
A{1,2} = 'Anne Smith';
A{2,1} = 3+7i;
A{2,2} = -pi:pi/10:pi;

iscell(A)

ans =

     1
```

See Also

`cell`, `iscellstr`, `isstruct`, `isnumeric`, `islogical`, `isobject`, `isa`, `is*`

iscellstr

Purpose

Determine if input is a cell array of strings

Syntax

```
tf = iscellstr(A)
```

Description

`tf = iscellstr(A)` returns logical true (`1`) if `A` 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.

Examples

```
A{1,1} = 'Thomas Lee';
A{1,2} = 'Marketing';
A{2,1} = 'Allison Jones';
A{2,2} = 'Development';

iscellstr(A)

ans =

     1
```

See Also

`cell`, `char`, `iscell`, `isstruct`, `isa`, `is*`

Purpose

Determine if input is a character array

Syntax

```
tf = ischar(A)
```

Description

`tf = ischar(A)` returns logical true (`1`) if `A` is a character array and logical false (`0`) otherwise.

Examples

Given the following cell array,

```
C{1,1} = magic(3);            % double array
C{1,2} = 'John Doe';          % char array
C{1,3} = 2 + 4i               % complex double

C =

    [3x3 double]    'John Doe'    [2.0000+ 4.0000i]
```

`ischar` shows that only `C{1,2}` is a character array.

```
for k = 1:3
x(k) = ischar(C{1,k});
end

x

x =

     0     1     0
```

See Also

`char`, `isnumeric`, `islogical`, `isobject`, `isstruct`, `iscell`, `isa`, `is*`

isdir

Purpose Determine if item is a directory

Syntax

```
tf = isdir('A')
```

Description `tf = isdir('A')` returns logical true (`1`) if `A` is a directory and `0` otherwise.

Examples Type

```
tf=isdir('mymfiles/results')
```

and MATLAB returns

```
tf =
    1
```

indicating that `mymfiles/results` is a directory.

See Also `dir`, `is*`

Purpose Test if array is empty

Syntax

```
tf = isempty(A)
```

Description

`tf = 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.

Examples

```
B = rand(2,2,2);
B(:,:,:) = [];

isempty(B)

ans =
     1
```

See Also

`is*`

isequal

Purpose Determine if arrays are numerically equal

Syntax

```
tf = isequal(A,B,...)
```

Description `tf = 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.

Remarks When comparing structures, the order in which the fields of the structures were created is not important. As long as the structures contain the same fields, with corresponding fields set to equal values, `isequal` considers the structures to be equal. See Example 2, below.

When comparing numeric values, `isequal` does not consider the data type used to store the values in determining whether they are equal. See Example 3, below.

`NaNs` (Not a Number), by definition, are not equal. Therefore, arrays that contain `NaN` elements are not equal, and `isequal` returns zero when comparing such arrays. See Example 4, below. Use the `isequalwithequalnans` function when you want to test for equality with `NaNs` treated as equal.

`isequal` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequal` returns logical `1`.

Examples

Example 1

Given

```
A =                    B =                    C =
     1     0                1     0                1     0
     0     1                0     1                0     0
```

`isequal(A,B,C)` returns `0`, and `isequal(A,B)` returns `1`.

Example 2

When comparing structures with `isequal`, the order in which the fields of the structures were created is not important:

```
A.f1 = 25;    A.f2 = 50
A =
    f1: 25
```

```
    f2: 50

B.f2 = 50;    B.f1 = 25
B =
    f2: 50
    f1: 25

isequal(A, B)
ans =
     1
```

Example 3

When comparing numeric values, the data types used to store the values are not important:

```
A = [25 50];   B = [int8(25) int8(50)];

isequal(A, B)
ans =
     1
```

Example 4

Arrays that contain `NaN` (Not a Number) elements cannot be equal, since `NaN`s, by definition, are not equal:

```
A = [32 8 -29 NaN 0 5.7];
B = A;

isequal(A, B)
ans =
     0
```

See Also

`isequalwithequalnans`, `strcmp`, `isa`, `is*`, `relational operators`

Purpose

Determine if arrays are numerically equal, treating NaNs as equal

Syntax

`tf = isequalwithequalnans(A,B,...)`

Description

`tf = isequalwithequalnans(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. NaN (Not a Number) values are considered to be equal to each other. Numeric data types and structure field order do not have to match.

Remarks

`isequalwithequalnans` is the same as `isequal`, except `isequalwithequalnans` considers NaN (Not a Number) values to be equal, and `isequal` does not.

`isequalwithequalnans` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequalwithequalnans` returns logical 1.

Examples

Arrays containing NaNs are handled differently by `isequal` and `isequalwithequalnans`. `isequal` does not consider NaNs to be equal, while `isequalwithequalnans` does.

```
A = [32 8 -29 NaN 0 5.7];
B = A;
isequal(A, B)
ans =
     0

isequalwithequalnans(A, B)
ans =
     1
```

The position of NaN elements in the array does matter. If they are not in the same position in the arrays being compared, then `isequalwithequalnans` returns zero.

```
A = [2 4 6 NaN 8];   B = [2 4 NaN 6 8];
isequalwithequalnans(A, B)
ans =
     0
```

See Also

`isequal`, `strcmp`, `isa`, `is*`, `relational operators`

Purpose

Determine if input is a MATLAB structure array field

Syntax

```
tf = isfield(A, 'field')
```

Description

`tf = isfield(A, 'field')` returns logical `1` (`true`) if `field` is the name of a field in the structure array `A`, and logical `0` (`false`) otherwise. If `A` is not a structure array, `isfield` returns logical `0` (`false`).

Examples

Given the following MATLAB structure,

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

`isfield` identifies `billing` as a field of that structure.

```
isfield(patient,'billing')

ans =

     1
```

See Also

`fieldnames`, `setfield`, `getfield`, `orderfields`, `rmfield`, `struct`, `isstruct`, `iscell`, `isa`, `is*`, dynamic field names

isfinite

Purpose Detect finite elements of an array

Syntax

```
TF = isfinite(A)
```

Description

`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 a complex number `z`, `isfinite(z)` returns `1` if both the real and imaginary parts of `z` are finite, and `0` if either the real or the imaginary part is infinite or `NaN`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2];

isfinite(1./a)
Warning: Divide by zero.

ans =
     1     1     0     1     1

isfinite(0./a)
Warning: Divide by zero.

ans =
     1     1     0     1     1
```

See Also

`isinf`, `isnan`, `is*`

Purpose Detect floating-point arrays

Syntax `isfloat(A)`

Description `isfloat(A)` returns a logical true (`1`) if A is a floating-point array and a logical false (`0`) otherwise. The only floating-point data types in MATLAB are `single` and `double`.

See Also `isa`, `isinteger`, `double`, `single`, `isnumeric`

isglobal

Purpose Determine if input is a global variable

Syntax

```
tf = isglobal(A)
```

Description `tf = isglobal(A)` returns logical true (`1`) if `A` has been declared to be a global variable and logical false (`0`) otherwise.

See Also `global`, `isvarname`, `isa`, `is*`

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`

"Finding and Identifying Graphics Objects" for related functions

ishold

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`

"Axes Operations" for related functions

Purpose Detect infinite elements of an array

Syntax `TF = isinf(A)`

Description `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. For a complex number `z`, `isinf(z)` returns `1` if either the real or imaginary part of `z` is infinite, and `0` if both the real and imaginary parts are finite or `NaN`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2]

isinf(1./a)
Warning: Divide by zero.

ans =
     0     0     1     0     0

isinf(0./a)
Warning: Divide by zero.

ans =
     0     0     0     0     0
```

See Also `isfinite`, `isnan`, `is*`

isinteger

Purpose Detect whether an array has integer data type

Syntax `isinteger(A)`

Description `isinteger(A)` returns a logical true (`1`) if the array `A` has integer data type and a logical false (`0`) otherwise. The integer data types in MATLAB are

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `int64`
- `uint64`

See Also `isa`, `isnumeric`, `isfloat`

Purpose Determine if input is a MATLAB keyword

Syntax

```
tf = iskeyword('str')
iskeyword str
iskeyword
```

Description

`tf = iskeyword('str')` returns logical true (`1`) if the string `str` is a keyword in the MATLAB language and logical false (`0`) otherwise.

`iskeyword str` uses the MATLAB command format.

`iskeyword` returns a list of all MATLAB keywords.

Examples

To test if the word `while` is a MATLAB keyword,

```
iskeyword while
ans =
    1
```

To obtain a list of all MATLAB keywords,

```
iskeyword
    'break'
    'case'
    'catch'
    'continue'
    'else'
    'elseif'
    'end'
    'for'
    'function'
    'global'
    'if'
    'otherwise'
    'persistent'
    'return'
    'switch'
    'try'
    'while'
```

See Also `isvarname`, `genvarname`, `is*`

Purpose Detect array elements that are letters of the alphabet

---

Note Use the `isstrprop` function in place of `isletter`. The `isletter` function will be removed in a future version of MATLAB.

Syntax

```
tf = isletter('str')
```

Description

`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.

Examples

Find the letters in character array `s`.

```
s = 'A1, B2, C3';

isletter(s)
ans =
     1     0     0     1     0     0     1     0
```

See Also

`isstrprop`, `isnumeric`, `ischar`, `char`, `isspace`, `isa`, `is*`

islogical

Purpose Determine if input is a logical array

Syntax

```
tf = islogical(A)
```

Description `tf = islogical(A)` returns logical true (`1`) if `A` is a logical array and logical false (`0`) otherwise.

Examples Given the following cell array,

```
C{1,1} = pi;                  % double
C{1,2} = 1;                   % double
C{1,3} = ispc;                % logical
C{1,4} = magic(3)             % double array

C =
    [3.1416]    [1]    [1]    [3x3 double]
```

`islogical` shows that only `C{1,3}` is a logical array.

```
for k = 1:4
x(k) = islogical(C{1,k});
end

x
x =
     0     0     1     0
```

See Also `logical`, `isnumeric`, `ischar`, `isreal`, logical operators (elementwise and short-circuit), `isa`, `is*`

Purpose Detect members of a specific set

Syntax

```
tf = ismember(A, S)
tf = ismember(A, S, 'rows')
[tf, loc] = ismember(A, S, ...)
```

Description

`tf = ismember(A, S)` returns a vector the same length as `A`, containing logical true (`1`) where the elements of `A` are in the set `S`, and logical false (`0`) elsewhere. In set theory terms, `k` is 1 where `A` ∈ `S`. `A` and `S` can be cell arrays of strings.

`tf = ismember(A, S, 'rows')`, when `A` and `S` are matrices with the same number of columns, returns a vector containing `1` where the rows of `A` are also rows of `S` and `0` otherwise. You cannot use this syntax if `A` or `S` is a cell array of strings.

`[tf, loc] = ismember(A, S, ...)` returns index vector `loc` containing the highest index in `S` for each element in `A` that is a member of `S`. For those elements of `A` that do not occur in `S`, `ismember` returns `0`.

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

set = [5 2 4 2 8 10 12 2 16 18 20 3];
[tf, index] = ismember(a, set);
```

```
index
index =
     0
     8
    12
     3
     1
```

See Also

issorted, intersect, setdiff, setxor, union, unique, is*

Purpose

Determine if input is a method of an object

Syntax

`ismethod(h, 'name')`

Description

`ismethod(h, 'name')` returns a logical `true` (`1`) if the specified `name` is a method that you can call on object `h`. Otherwise, `ismethod` returns logical `false` (`0`).

Examples

Create an Excel application and test to see if `SaveWorkspace` is a method of the object. `ismethod` returns `true`:

```
h = actxserver ('Excel.Application');

ismethod(h, 'SaveWorkspace')
ans =
     1
```

Try the same test on `UsableWidth`, which is a property. `isevent` returns `false`:

```
ismethod(h, 'UsableWidth')
ans =
     0
```

See Also

`methods`, `methodsview`, `isprop`, `isevent`, `isobject`, `class`, `invoke`

Purpose Detect `NaN` elements of an array

Syntax `TF = isnan(A)`

Description `TF = isnan(A)` returns an array the same size as `A` containing logical true (`1`) where the elements of `A` are `NaN`s and logical false (`0`) where they are not. For a complex number `z`, `isnan(z)` returns `1` if either the real or imaginary part of `z` is `NaN`, and `0` if both the real and imaginary parts are finite or `Inf`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2]

isnan(1./a)
Warning: Divide by zero.

ans =
     0     0     0     0     0

isnan(0./a)
Warning: Divide by zero.

ans =
     0     0     1     0     0
```

See Also `isfinite`, `isinf`, `is*`

Purpose Determine if input is a numeric array

Syntax

```
tf = isnumeric(A)
```

Description

`tf = 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 and logicals are not.

Examples

Given the following cell array,

```
C{1,1} = pi;                  % double
C{1,2} = 'John Doe';          % char array
C{1,3} = 2 + 4i;              % complex double
C{1,4} = ispc;                % logical
C{1,5} = magic(3)             % double array

C =
    [3.1416]   'John Doe'   [2.0000+ 4.0000i]   [1]   [3x3 double]
```

`isnumeric` shows that all but `C{1,2}` and `C{1,4}` are numeric arrays.

```
for k = 1:5
x(k) = isnumeric(C{1,k});
end

x
x =
     1     0     1     0     1
```

See Also

`isstrprop`, `isnan`, `isreal`, `isprime`, `isfinite`, `isinf`, `isa`, `is*`

isobject

Purpose Determine if input is a MATLAB OOPs object

Syntax

```
tf = isobject(A)
```

Description `tf = isobject(A)` returns logical true (`1`) if `A` is a MATLAB object and logical false (`0`) otherwise.

Examples Create an instance of the `polynom` class as defined in the section "Example - A Polynomial Class" in the MATLAB documentation.

```
p = polynom([1 0 -2 -5])
p =
   x^3 - 2*x - 5
```

`isobject` indicates that `p` is a MATLAB object.

```
isobject(p)
ans =
     1
```

Note that `isjava`, which tests for Java objects in MATLAB, returns false (`0`).

```
isjava(p)
ans =
     0
```

See Also `isjava`, `isstruct`, `iscell`, `ischar`, `isnumeric`, `islogical`, `ismethod`, `isprop`, `isevent`, `methods`, `class`, `isa`, `is*`

Purpose

Compute isosurface end cap geometry

Syntax

```
fvc = isocaps(X,Y,Z,V,isovalue)
fvc = isocaps(V,isovalue)
fvc = isocaps(...,'enclose')
fvc = isocaps(...,'whichplane')
[f,v,c] = isocaps(...)
isocaps(...)
```

Description

`fvc = isocaps(X,Y,Z,V,isovalue)` computes isosurface end cap geometry for the volume data `V` at isosurface value `isovalue`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`.

The struct `fvc` contains the face, vertex, and color data for the end caps and can be passed directly to the `patch` command.

`fvc = isocaps(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isocaps(...,'enclose')` specifies whether the end caps enclose data values above or below the value specified in `isovalue`. The string *`enclose`* can be either `above` (default) or `below`.

`fvc = isocaps(...,'whichplane')` specifies on which planes to draw the end caps. Possible values for *`whichplane`* are `all` (default), `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, or `zmax`.

`[f,v,c] = isocaps(...)` returns the face, vertex, and color data for the end caps in three arrays instead of the struct `fvc`.

`isocaps(...)` without output arguments draws a patch with the computed faces, vertices, and colors.

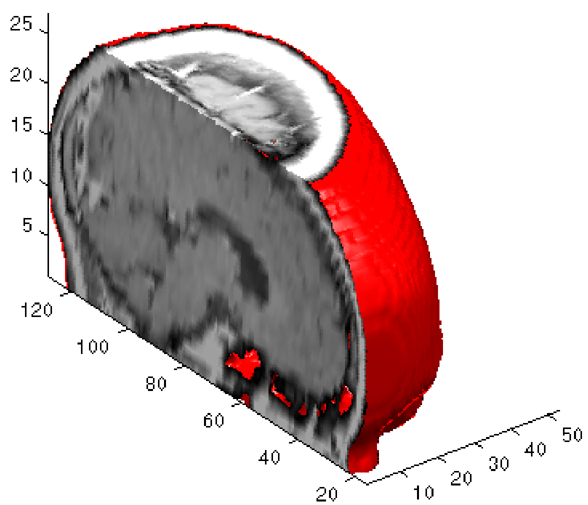
Examples

This example uses a data set that is a collection of MRI slices of a human skull. It illustrates the use of `isocaps` to draw the end caps on this cutaway volume.

The red `isosurface` shows the outline of the volume (skull) and the end caps show what is inside of the volume.

The patch created from the end cap data (p2) uses interpolated face coloring, which means the gray colormap and the light sources determine how it is colored. The isosurface patch (p1) used a flat red face color, which is affected by the lights, but does not use the colormap.

```
load mri
D = squeeze(D);
D(:,1:60,:) = [];
p1 = patch(isosurface(D, 5),'FaceColor','red',...
   'EdgeColor','none');
p2 = patch(isocaps(D, 5),'FaceColor','interp',...
   'EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight left; camlight; lighting gouraud
isonormals(D,p1)
```



See Also

isosurface, isonormals, smooth3, subvolume, reducevolume, reducepatch

Isocaps Add Context to Visualizations for more illustrations of isocaps

“Volume Visualization” for related functions

isocolors

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 (true color).

`nc = isocolors(C,vertices)`, and `nc = isocolors(R,G,B,vertices)` assume `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(C)`.

`nc = isocolors(...,PatchHandle)` uses the vertices from the patch identified by `PatchHandle`.

`isocolors(...,PatchHandle)` sets the `FaceVertexCData` property of the patch specified by `PatchHandle` to the computed colors.

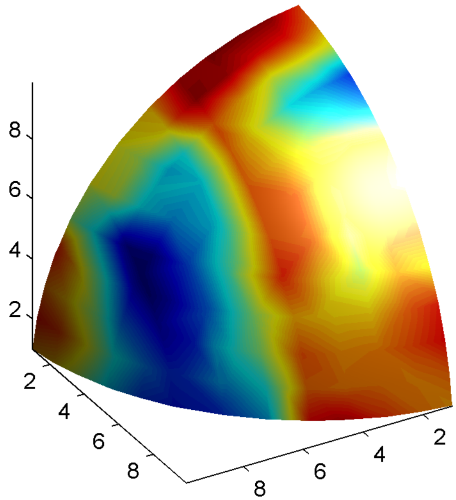
Examples

Indexed Color Data

This example displays an isosurface and colors it with random data using indexed color. (See "Interpolating in Indexed Color 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));
```

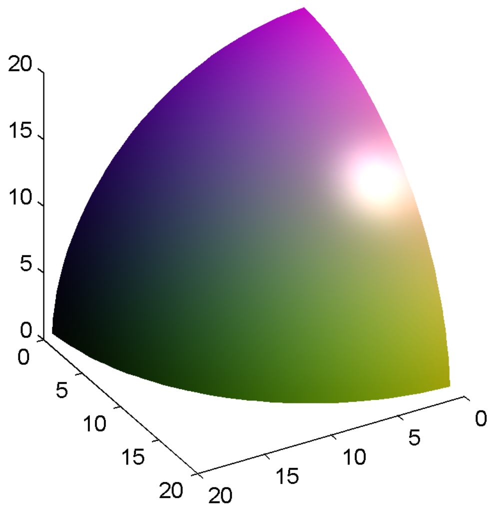
```
isonormals(x,y,z,data,p);
isocolors(x,y,z,cdata,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);axis tight
camlight; lighting phong;
```



True Color Data

This example displays an isosurface and colors it with true color (RGB) data.

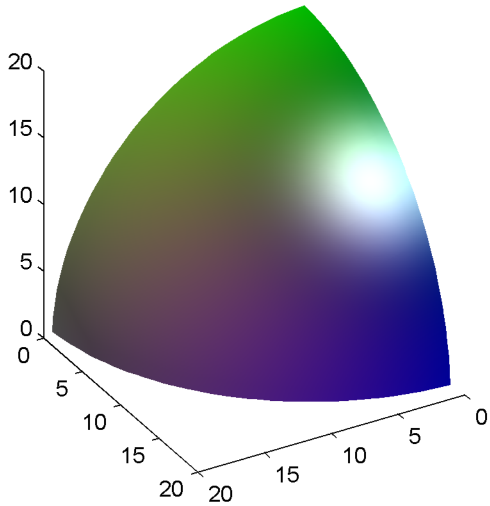
```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(x,y,z,data,20));
isonormals(x,y,z,data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
isocolors(x,y,z,r/20,g/20,b/20,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);
camlight; lighting phong;
```



Modified True Color Data

This example uses `isocolors` to calculate the true color data using the isosurface's (patch object's) vertices, but then returns the color data in a variable (`c`) in order to modify the values. It then explicitly sets the isosurface's `FaceVertexCData` to the new data (`1-c`).

```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(data,20));
isonormals(data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
c = isocolors(r/20,g/20,b/20,p);
set(p,'FaceVertexCData',1-c)
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);
camlight; lighting phong;
```



See Also isosurface, isocaps, smooth3, subvolume, reducevolume, reducepatch, isonormals

"Volume Visualization" for related functions

isonormals

Purpose

Compute normals of isosurface vertices

Syntax

```
n = isonormals(X,Y,Z,V,vertices)
n = isonormals(V,vertices)
n = isonormals(V,p), 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');
```

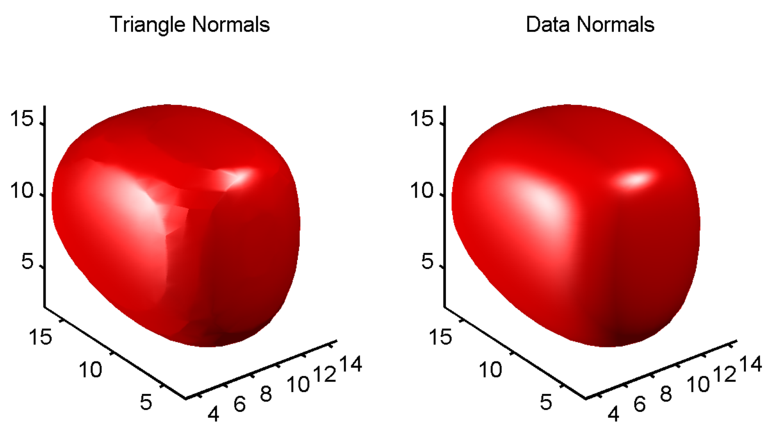
Draw an isosurface from the volume data and add lights. This isosurface uses triangle normals (`patch`, `isosurface`, `view`, `daspect`, `axis`, `camlight`, `lighting`, `title`):

```
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`

"Volume Visualization" for related functions

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`. That is, the isosurface connects points that have the specified value much the way contour lines connect points of equal elevation.

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The structure `fv` contains the faces and vertices of the isosurface, which you can pass directly to the `patch` command.

`fv = isosurface(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isosurface(...,colors)` interpolates the array `colors` onto the scalar field and returns the interpolated values in the `facevertexcdata` field of the `fvc` structure. The size of the `colors` array must be the same as `V`. The `colors` argument enables you to control the color mapping of the isosurface with data different from that used to calculate the isosurface (e.g., temperature data superimposed on a wind current isosurface).

`fv = isosurface(...,'noshare')` does not create shared vertices. This is faster, but produces a larger set of vertices.

`fv = isosurface(...,'verbose')` prints progress messages to the command window as the computation progresses.

`[f,v] = isosurface(...)` 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

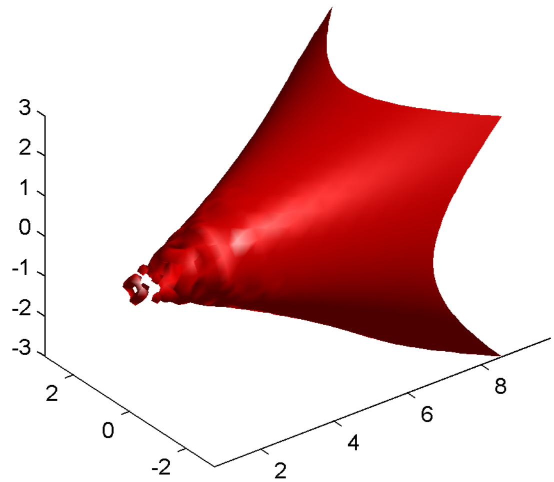
```
[f,v] = isosurface(X,Y,Z,V,isovalue);
patch('Faces',f,'Vertices',v)
```

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`, `shrinkfaces`, `smooth3`, `subvolume`

Connecting Equal Values with Isosurfaces for more examples

"Volume Visualization" for related functions

Purpose Determine if PC (Windows) version of MATLAB

Syntax

```
tf = ispc
```

Description

```
tf = ispc
```

returns logical true (`1`) for the PC version of MATLAB and logical false (`0`) otherwise.

See Also `isunix`, `isstudent`, `is*`

isprime

Purpose Detect prime elements of an array

Syntax

```
TF = isprime(A)
```

Description `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. `A` must contain only positive integers.

Examples

```
c = [2 3 0 6 10]


c =
     2     3     0     6    10

isprime(c)

ans =
     1     1     0     0     0
```

See Also `is*`

Purpose Determine if input is a property of an object

Syntax

```
isprop(h, 'name')
```

Description

```
isprop(h, 'name')
```

returns a logical `1` (`true`) if the specified `name` is a property you can use with object `h`. Otherwise, `isprop` returns logical `0` (`false`).

Examples Create an Excel application and test to see if `UsableWidth` is a property of the object. `isprop` returns `true`:

```
  h = actxserver ('Excel.Application');


  isprop(h, 'UsableWidth')
h.isprop('UsableWidth')
  ans =
       1
```

Try the same test on `SaveWorkspace`, which is a method, and `isprop` returns `false`:

```
  isprop(h, 'SaveWorkspace')
h.isprop('SaveWorkspace')
  ans =
       0
```

See Also `get(COM)`, `inspect`, `addproperty`, `deleteproperty`, `ismethod`, `isevent`, `isobject`, `methods`, `class`

isreal

Purpose

Determine if all array elements are real numbers

Syntax

```
tf = isreal(A)
```

Description

`tf = isreal(A)` returns logical false (`0`) if any element of array `A` has an imaginary component, even if the value of that component is 0. It returns logical true (`1`) otherwise.

`~isreal(x)` returns logical true for arrays that have at least one element with an imaginary component. The value of that component can be `0`.

Note If `a` is real, `complex(a)` returns a complex number whose imaginary component is `0`, and `isreal(complex(a))` returns false. In contrast, the addition `a + 0i` returns the real value `a`, and `isreal(a + 0i)` returns true.

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.

Examples

Example 1. These examples use `isreal` to detect the presence or absence of imaginary numbers in an array. Let

```
x = magic(3);
y = complex(x);
```

`isreal(x)` returns true because no element of `x` has an imaginary component.

```
isreal(x)
ans =
     1
```

`isreal(y)` returns false, because every element of `x` has an imaginary component, even though the value of the imaginary components is `0`.

```
isreal(y)
ans =
     0
```

This expression detects strictly real arrays, i.e., elements with 0-valued imaginary components are treated as real.

```
~any(imag(y(:)))
ans =
     1
```

Example 2. Given the following cell array,

```
C{1,1} = pi;                 % double
C{1,2} = 'John Doe';         % char array
C{1,3} = 2 + 4i;             % complex double
C{1,4} = ispc;               % logical
C{1,5} = magic(3)            % double array
C{1,6} = complex(5,0)        % complex double

C =
  [3.1416]  'John Doe'  [2.0000+ 4.0000i]  [1]  [3x3 double]  [5]
```

isreal shows that all but C{1,3} and C{1,6} are real arrays.

```
for k = 1:6
x(k) = isreal(C{1,k});
end

x
x =
     1     1     0     1     1    0
```

See Also

complex, isnumeric, isnan, isprime, isfinite, isinf, isa, is*

isscalar

Purpose

Determine if input is scalar

Syntax

```
tf = isscalar(A)
```

Description

`tf = isscalar(A)` returns logical `1` (`true`) if `A` is a 1-by-1 matrix, and logical `0` (`false`) otherwise.

The `A` argument can also be a MATLAB object, as described in MATLAB Classes and Objects, as long as that object overloads the `size` function.

Examples

Test matrix `A` and one element of the matrix:

```
A = rand(5);

isscalar(A)
ans =
     0

isscalar(A(3,2))
ans =
     1
```

See Also

`isvector`, `isempty`, `isnumeric`, `islogical`, `ischar`, `isa`, `is*`

Purpose

Determine if set elements are in sorted order

Syntax

```
tf = issorted(A)
tf = issorted(A, 'rows')
```

Description

`tf = issorted(A)` returns logical true (`1`) if the elements of vector `A` are in sorted order, and logical false (`0`) otherwise. Vector `A` is considered to be sorted if `A` and the output of `sort(A)` are equal.

`tf = issorted(A, 'rows')` returns logical true (`1`) if the rows of two-dimensional matrix `A` are in sorted order, and logical false (`0`) otherwise. Matrix `A` is considered to be sorted if `A` and the output of `sortrows(A)` are equal.

Remarks

For character arrays, `issorted` uses ASCII, rather than alphabetical, order.

You cannot use `issorted` on arrays of greater than two dimensions.

Examples

Using `issorted` on a vector,

```
A = [5 12 33 39 78 90 95 107 128 131];

issorted(A)
ans =
     1
```

Using `issorted` on a matrix,

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

issorted(A, 'rows')
ans =
     0

B = sortrows(A)
B =
```

```
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
    17    24     1     8    15
    23     5     7    14    16

issorted(B)
ans =
     1
```

See Also `sort`, `sortrows`, `ismember`, `unique`, `intersect`, `union`, `setdiff`, `setxor`, `is*`

Purpose Detect space characters in an array

Syntax

```
tf = isspace('str')
```

Description

`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.

Examples

```
isspace('  Find spa ces ')
  Columns 1 through 13
    1    1    0    0    0    0    1    0    0    0    1    0    0
  Columns 14 through 15
     0     1
```

See Also

`isstrprop`, `ischar`, `isa`, `is*`

issparse

Purpose Test if matrix is sparse

Syntax

```
tf = issparse(S)
```

Description `tf = issparse(S)` returns logical true (`1`) if the storage class of `S` is sparse and logical false (`0`) otherwise.

See Also `is*`

Purpose Determine if input is a character array

Description This MATLAB 4 function has been renamed `ischar` in MATLAB 5.

See Also `ischar`, `isa`, `is*`

isstrprop

Purpose Determine if string is of specified category

Syntax

```
tf = isstrprop('str', 'category')
```

Description

`tf = isstrprop('str', 'category')` returns a logical array the same size as `str` containing logical `true` (1) where the elements of `str` belong to the specified *`category`*, and logical `false` (0) where they do not.

The `str` input can be a character array, cell array, or any MATLAB numeric type. If `str` is a cell array, then the return value is a cell array of the same shape as `str`.

The *`category`* input can be any of the strings shown in the left column below:

Category	Description
`alpha`	`True` for those elements of `str` that are alphabetic
`alphanum`	`True` for those elements of `str` that are alphanumeric
`cntrl`	`True` for those elements of `str` that are control characters (for example, `char(0:20)`)
`digit`	`True` for those elements of `str` that are numeric digits

`graphic`	`True` for those elements of `str` that are graphic characters. These are all values that represent any characters except for the following:
	`unassigned, space, line separator,`
	`paragraph separator, control characters,`
	`Unicode format control characters,`
	`private user-defined characters,`
	`Unicode surrogate characters,`
	`Unicode other characters`

`lower`	`True` for those elements of `str` that are lowercase letters
`print`	`True` for those elements of `str` that are graphic characters, plus `char(32)`
`punct`	`True` for those elements of `str` that are punctuation characters

Category	Description
wspace	`True` for those elements of `str` that are white-space characters. This range includes the ANSI C definition of white space, `{' ','\t','\n','\r','\v','\f'}`.
upper	`True` for those elements of `str` that are uppercase letters
xdigit	`True` for those elements of `str` that are valid hexadecimal digits

Remarks

Numbers of type `double` are converted to `int32` according to MATLAB rules of double-to-integer conversion. Numbers of type `int64` and `uint64` bigger than `int32(inf)` saturate to `int32(inf)`.

MATLAB classifies the elements of the `str` input according to the Unicode definition of the specified category. If the numeric value of an element in the input array falls within the range that defines a Unicode character category, then this element is classified as being of that category. The set of Unicode character codes includes the set of ASCII character codes, but also covers a large number of languages beyond the scope of the ASCII set. The classification of characters is dependent on the global location of the platform on which MATLAB is installed.

Examples

Test for alphabetic characters in a string:

```
A = isstrprop('abc123def', 'alpha')
A =
   1  1  1  0  0  0  1  1  1
```

Test for numeric digits in a string:

```
A = isstrprop('abc123def', 'digit')
A =
   0  0  0  1  1  1  0  0  0
```

Test for hexadecimal digits in a string:

```
A = isstrprop('abcd1234efgh', 'xdigit')
A =
   1  1  1  1  1  1  1  1  1  1  0  0
```

Test for numeric digits in a character array:

```
A = isstrprop(char([97 98 99 49 50 51 101 102 103]), 'digit')
A =
   0 0 0 1 1 1 0 0 0
```

Test for alphabetic characters in a two-dimensional cell array:

```
A = isstrprop({'abc123def';'456ghi789'}, 'alpha')
A =
   [1x9 logical]
   [1x9 logical]

A{:,:}
ans =
   1 1 1 0 0 0 1 1 1
   0 0 0 1 1 1 0 0 0
```

Test for white-space characters in a string:

```
A = isstrprop(sprintf('a bc\n'), 'wspace')
A =
   0 1 0 0 1
```

See Also `ischar`, `isnumeric`, `isspace`, `iscellstr`, `isa`, `is*`

Purpose Determine if input is a MATLAB structure array

Syntax

```
tf = isstruct(A)
```

Description `tf = isstruct(A)` returns logical true (`1`) if `A` is a MATLAB structure and logical false (`0`) otherwise.

Examples

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];

isstruct(patient)

ans =

     1
```

See Also `struct`, `isfield`, `iscell`, `ischar`, `isobject`, `isnumeric`, `islogical`, `isa`, `is*`, dynamic field names

isstudent

Purpose Determine if student edition of MATLAB

Syntax

```
tf = isstudent
```

Description `tf = isstudent` returns logical true (`1`) for the student edition of MATLAB and logical false (`0`) for commercial editions.

See Also `ispc`, `isunix`, `is*`

Purpose Determine if UNIX version of MATLAB

Syntax

```
tf = isunix
```

Description `tf = isunix` returns logical true (`1`) for the UNIX version of MATLAB and logical false (`0`) otherwise.

See Also `ispc`, `isstudent`, `is*`

isvalid (timer)

Purpose Determine if timer object is valid

Syntax

```
out = isvalid(obj)
```

Description `out = isvalid(obj)` returns a logical array, `out`, that contains a `0` where the elements of `obj` are invalid timer objects and a `1` where the elements of `obj` are valid timer objects.

An invalid timer object is an object that has been deleted and cannot be reused. Use the `clear` command to remove an invalid timer object from the workspace.

Examples Create a valid timer object.

```
t = timer;
out = isvalid(t)
out =

     1
```

Delete the timer object, making it invalid.

```
delete(t)
out1 = isvalid(t)
out1 =

     0
```

See Also `timer`, `delete`

Purpose Determine if input is a valid variable name

Syntax

```
tf = isvarname('str')
isvarname str
```

Description `tf = isvarname 'str'` returns logical true (1) if the string `str` is a valid MATLAB variable name and logical false (0) otherwise. A valid variable name is a character string of letters, digits, and underscores, totaling not more than `namelengthmax` characters and beginning with a letter.

`isvarname str` uses the MATLAB command format.

Examples This variable name is valid:

```
isvarname foo
ans =
   1
```

This one is not because it starts with a number:

```
isvarname 8th_column
ans =
   0
```

If you are building strings from various pieces, place the construction in parentheses.

```
d = date;

isvarname(['Monday_', d(1:2)])
ans =
   1
```

See Also `genvarname`, `isglobal`, `iskeyword`, `namelengthmax`, is*

isvector

Purpose Determine if input is a vector

Syntax

```
tf = isvector(A)
```

Description

`tf = isvector(A)` returns logical `1` (`true`) if `A` is a 1-by-`N` or `N`-by-1 vector where `N >= 0`, and logical `0` (`false`) otherwise.

The `A` argument can also be a MATLAB object, as described in MATLAB Classes and Objects, as long as that object overloads the `size` function.

Examples

Test matrix `A` and its row and column vectors:

```
A = rand(5);

isvector(A)
ans =
     0

isvector(A(3, :))
ans =
     1

isvector(A(:, 2))
ans =
     1
```

See Also

`isscalar`, `isempty`, `isnumeric`, `islogical`, `ischar`, `isa`, `is*`

2j

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`

2keyboard

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`, `pause`, `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

2lasterr

Purpose Return last error message

Syntax

```
msgstr = lasterr
[msgstr, msgid] = lasterr
lasterr('new_msgstr')
lasterr('new_msgstr','new_msgid')
[msgstr,msgid] = lasterr('new_msgstr','new_msgid')
```

Description

`msgstr = lasterr` returns the last error message generated by MATLAB.

`[msgstr, msgid] = lasterr` returns the last error in `msgstr` and its message identifier in `msgid`. If the error was not defined with an identifier, `lasterr` returns an empty string for `msgid`. See "Message Identifiers" and "Using Message Identifiers with lasterr" in the MATLAB documentation for more information on the `msgid` argument and how to use it.

`lasterr('new_msgstr')` sets the last error message to a new string, `new_msgstr`, so that subsequent invocations of `lasterr` return the new error message string. You can also set the last error to an empty string with `lasterr('')`.

`lasterr('new_msgstr','new_msgid')` sets the last error message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lasterr` return the new error message and message identifier.

`[msgstr,msgid] = lasterr('new_msgstr','new_msgid')` returns the last error message and its identifier, also changing these values so that subsequent invocations of `lasterr` return the message and identifier strings specified by `new_msgstr` and `new_msgid` respectively.

Examples

Example 1

Here is a function that examines the `lasterr` string and displays its own message based on the error that last occurred. This example deals with two cases, each of which is an error that can result from a matrix multiply:

```
function matrix_multiply(A, B)
try
   A * B
```

```
catch
   errmsg = lasterr;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
   else
      if(strfind(errmsg, 'not defined for variables of class'))
         disp('** Both arguments must be double matrices')
      end
   end
end
```

If you call this function with matrices that are incompatible for matrix multiplication (e.g., the column dimension of `A` is not equal to the row dimension of `B`), MATLAB catches the error and uses `lasterr` to determine its source:

```
A = [1  2  3; 6  7  2; 0  -1  5];
B = [9  5  6; 0  4  9];

matrix_multiply(A, B)
** Wrong dimensions for matrix multiply
```

Example 2

Specify a message identifier and error message string with `error`:

```
error('MyToolbox:angleTooLarge', ...
      'The angle specified must be less than 90 degrees.');
```

In your error handling code, use `lasterr` to determine the message identifier and error message string for the failing operation:

```
[errmsg, msgid] = lasterr
errmsg =
   The angle specified must be less than 90 degrees.
msgid =
   MyToolbox:angleTooLarge
```

See Also

`error`, `lasterror`, `warning`, `lastwarn`

Purpose Return last error message and related information

Syntax

```
s = lasterror
s = lasterror(err)
```

Description

`s = lasterror` returns a structure `s` containing information about the last error issued by MATLAB. The return structure contains the following character array fields.

Fieldname	Description
`message`	Text of the error message
`identifier`	Message identifier of the error message

Note The `lasterror` return structure might contain additional fields in future versions of MATLAB.

If the last error issued by MATLAB had no message identifier, then the `message_id` field is an empty character array.

See "Message Identifiers" in the MATLAB documentation for more information on the syntax and usage of message identifiers.

`s = lasterror(err)` sets the last error information to the error message and identifier specified in the structure `err`. Subsequent invocations of `lasterror` or `lasterr` return this new error information. The optional return structure `s` contains information on the previous error.

The fields of the structure `err` are shown in the table above. If either of these fields is undefined, MATLAB uses an empty character array instead.

Example

`lasterror` is usually used in conjunction with the `rethrow` function in `try-catch` statements. For example,

```
try
   do_something
```

```
catch
   do_cleanup
   rethrow(lasterror)
end
```

See Also `error`, `rethrow`, `try`, `catch`, `lasterr`, `lastwarn`

lastwarn

Purpose Return last warning message

Syntax

```
msgstr = lastwarn
[msgstr,msgid] = lastwarn
lastwarn('new_msgstr')
lastwarn('new_msgstr','new_msgid')
[msgstr,msgid] = lastwarn('new_msgstr','new_msgid')
```

Description

`msgstr = lastwarn` returns the last warning message generated by MATLAB.

`[msgstr,msgid] = lastwarn` returns the last warning in `msgstr` and its message identifier in `msgid`. If the warning was not defined with an identifier, `lastwarn` returns an empty string for `msgid`. See “Message Identifiers” and “Warning Control” in the MATLAB documentation for more information on the `msgid` argument and how to use it.

`lastwarn('new_msgstr')` sets the last warning message to a new string, `new_msgstr`, so that subsequent invocations of `lastwarn` return the new warning message string. You can also set the last warning to an empty string with `lastwarn('')`.

`lastwarn('new_msgstr','new_msgid')` sets the last warning message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lastwarn` return the new warning message and message identifier.

`[msgstr,msgid] = lastwarn('new_msgstr','new_msgid')` returns the last warning message and its identifier, also changing these values so that subsequent invocations of `lastwarn` return the message and identifier strings specified by `new_msgstr` and `new_msgid`, respectively.

Remarks

`lastwarn` does not return warnings that are reported during the parsing of MATLAB commands. (Warning messages that include the failing file name and line number are parse-time warnings.)

Examples

Specify a message identifier and warning message string with `warning`:

```
warning('MATLAB:divideByZero', 'Divide by zero');
```

Use `lastwarn` to determine the message identifier and error message string for the operation:

```
[warnmsg, msgid] = lastwarn
warnmsg =
   Divide by zero
msgid =
   MATLAB:divideByZero
```

See Also

`warning`, `error`, `lasterr`, `lasterror`

Purpose Least common multiple

Syntax

```
L = lcm(A,B)
```

Description

`L = lcm(A,B)` returns the least common multiple of corresponding elements of arrays `A` and `B`. Inputs `A` and `B` must contain positive integer elements and must be the same size (or either can be scalar).

Examples

```
lcm(8,40)

ans =

    40

lcm(pascal(3),magic(3))

ans =
     8     1     6
     3    10    21
     4     9     6
```

See Also `gcd`

Purpose Left or right array division

Syntax

```
ldivide(A,B)     A.\B
rdivide(A,B)     A./B
```

Description

`ldivide(A,B)` and the equivalent `A.\B` divides each entry of `B` by the corresponding entry of `A`. `A` and `B` must be arrays of the same size. A scalar value for either `A` or `B` is expanded to an array of the same size as the other.

`rdivide(A,B)` and the equivalent `A./B` divides each entry of `A` by the corresponding entry of `B`. `A` and `B` must be arrays of the same size. A scalar value for either `A` or `B` is expanded to an array of the same size as the other.

Example

```
A = [1 2 3;4 5 6];
B = ones(2, 3);
A.\B

ans =

    1.0000    0.5000    0.3333
    0.2500    0.2000    0.1667
```

See Also Arithmetic operators, `mldivide`, `mrdivide`

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('toggle'), legend(axes_handle,'toggle')
legend('hide'), legend(axes_handle,'hide')
legend('show'), legend(axes_handle,'show')
legend('boxoff'), legend(axes_handle,'boxoff')
legend('boxon'), legend(axes_handle,'boxon')
legend_handle = legend(...)
legend
legend(legend_handle,...)
legend(...,'Location',location)
legend(...,'Orientation',orientation)
[legend_h,object_h,plot_h,text_strings] = legend(...)
legend(li_object,string1,string2,string3)
legend(li_object,M)
```

Description

`legend` places a legend on various types of graphs (line plots, bar graphs, pie charts, etc.). For each line plotted, the legend shows a sample of the line type, marker symbol, and color beside the text label you specify. When plotting filled areas (patch or surface objects), the legend contains a sample of the face color next to the text label.

The font size and font name for the legend strings match the Axes `FontSize` and `FontName` properties.

`legend('string1','string2',...)` displays a legend in the current axes using the specified strings to label each set of data.

`legend(h,'string1','string2',...)` displays a legend on the plot containing the objects identified by the handles in the vector `h` and using the specified strings to label the corresponding graphics object (line, barseries, 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 in the current axes or the axes specified by `axes_handle`.

`legend('toggle')`, `legend(axes_handle,'toggle')` toggles the legend on or off. If no legend exists for the current axes, one is created using default strings.

The *default string* for an object is the value of the object's `DisplayName` property, if you have defined a value for `DisplayName` (which you can do using the Property Editor or calling `set`). Otherwise, legend constructs a sting of the form `data1`, `data2`, etc.

`legend('hide')`, `legend(axes_handle,'hide')` makes the legend in the current axes or the axes specified by `axes_handle` invisible.

`legend('show')`, `legend(axes_handle,'show')` makes the legend in the current axes or the axes specified by `axes_handle` visible.

`legend('boxoff')`, `legend(axes_handle,'boxoff')` removes the box from the legend in the current axes or the axes specified by `axes_handle`.

`legend('boxon')`, `legend(axes_handle,'boxon')` adds a box to the legend in the current axes or the axes specified by `axes_handle`.

`legend_handle = legend` returns the handle to the legend on the current axes or empty if no legend exists.

`legend` with no arguments refreshes all the legends in the current figure.

`legend(legend_handle)` refreshes the specified legend.

`legend(...,'Location',location)` uses *`location`* to determine where to place the legend. *`location`* can be either a 1-by-4 position vector ([left bottom width height]) or one of the following strings.

Specifier	Location in Axes
`North`	inside plot box near top
`South`	inside bottom
`East`	inside right
`West`	inside left
`NorthEast`	inside top right (default)
`NorthWest`	inside top left
`SouthEast`	inside bottom right
`SouthWest`	inside bottom left
`NorthOutside`	outside plot box near top
`SouthOutside`	outside bottom
`EastOutside`	outside right
`WestOutside`	outside left
`NorthEastOutside`	outside top right
`NorthWestOutside`	outside top left
`SouthEastOutside`	outside bottom right
`SouthWestOutside`	outside bottom left
`Best`	least conflict with data in plot
`BestOutside`	least unused space outside plot

The *`location`* string can be all lower case and can be abbreviated by sentinel letter (e.g., N, NE, NEO, etc.).

Obsolete Location Values

Obsolete Specifier	Location in Axes
-1	outside axes on right side
0	inside axes
1	upper right corner of axes
2	upper left corner of axes
3	lower left corner of axes
4	lower right corner of axes

`legend(...,'Orientation','`*`orientation`*`')` creates a legend with the legend items arranged in the specified orientation. *`orientation`* can be `vertical` (the default) or `horizontal`.

`[legend_h,object_h,plot_h,text_strings] = legend(...)` returns

- `legend_h` — Handle of the legend axes
- `object_h` — Handles of the line, patch and text graphics objects used in the legend
- `plot_h` — Handles of the lines and other objects used in the plot
- `text_strings` — Cell array of the text strings used in the legend

These handles enable you to modify the properties of the respective objects.

`legend(li_object,string1,string2,string3)` creates a legend for legendinfo objects `li_objects` with strings `string1`, etc.

`legend(li_object,M)` creates a legend of legendinfo objects `li_objects` where `M` is a string matrix or cell array of strings corresponding to the legendinfo objects.

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.

`legend` installs a figure `ResizeFcn`, if there is not already a user-defined `ResizeFcn` assigned to the figure. This `ResizeFcn` attempts to keep the legend the same size.

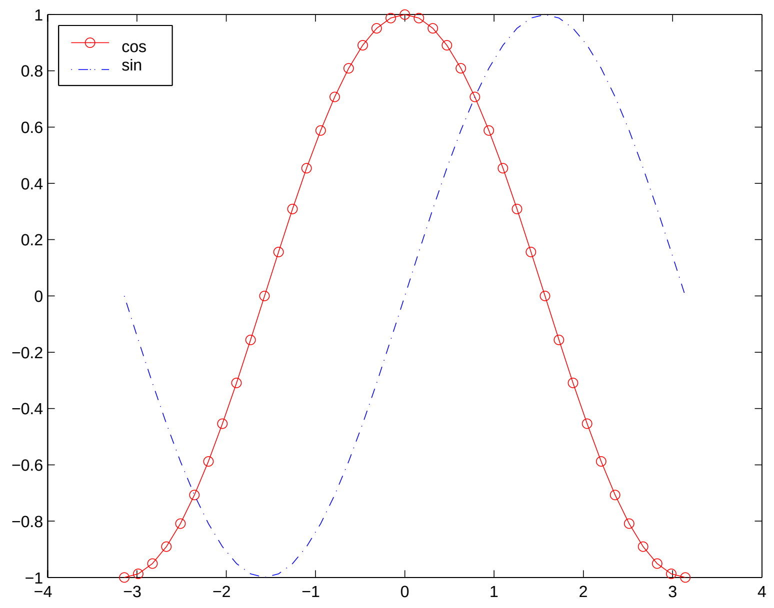
Moving the Legend

You can move the legend by pressing the left mouse button while the cursor is over the legend and dragging the legend to a new location. Double-clicking a label allows you to edit the label.

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`

Adding a Legend to a Graph for more information on using legends

"Annotating Plots" for related functions

legendre

Purpose

Associated Legendre functions

Syntax

```
P = legendre(n,X)
S = legendre(n,X,'sch')
N = legendre(n,X,'norm')
```

Definitions

Associated Legendre Functions. The Legendre functions are defined by

$$P_n^m(x) = (-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}P_n(x)$$

where

$$P_n(x)$$

is the Legendre polynomial of degree n.

$$P_n(x) = \frac{1}{2^n n!}\left[\frac{d^n}{dx^n}(x^2-1)^n\right]$$

Schmidt Seminormalized Associated Legendre Functions. The Schmidt seminormalized associated Legendre functions are related to the nonnormalized associated Legendre functions $P_n^m(x)$ by

$$P_n(x) \qquad \text{for } m = 0$$

$$S_n^m(x) = (-1)^m\sqrt{\frac{2(n-m)!}{(n+m)!}}P_n^m(x) \qquad \text{for } m > 0.$$

Fully Normalized Associated Legendre Functions. The fully normalized associated Legendre functions are normalized such that

$$\int_{-1}^{1}(N_n^m(x))^2 dx = 1$$

and are related to the unnormalized associated Legendre functions $P_n^m(x)$ by

$$N_n^m(x) = (-1)^m \sqrt{\frac{\left(n+\frac{1}{2}\right)(n-m)!}{(n+m)!}} P_n^m(x)$$

Description

`P = legendre(n,X)` computes the associated Legendre functions $P_n^m(x)$ of degree `n` and order `m = 0,1,...,n`, evaluated for each element of `X`. Argument `n` must be a scalar integer, and `X` must contain real values in the domain $-1 \le x \le 1$.

If `X` is a vector, then `P` is an `(n+1)`-by-`q` matrix, where `q = length(X)`. Each element `P(m+1,i)` corresponds to the associated Legendre function of degree `n` and order `m` evaluated at `X(i)`.

In general, the returned array `P` has one more dimension than `X`, and each element `P(m+1,i,j,k,...)` contains the associated Legendre function of degree `n` and order `m` evaluated at `X(i,j,k,...)`. Note that the first row of `P` is the Legendre polynomial evaluated at `X`, i.e., the case where `m` = 0.

`S = legendre(n,X,'sch')` computes the Schmidt seminormalized associated Legendre functions $S_n^m(x)$.

`N = legendre(n,X,'norm')` computes the fully normalized associated Legendre functions $N_n^m(x)$.

Examples

Example 1. The statement `legendre(2,0:0.1:0.2)` returns the matrix

	x = 0	x = 0.1	x = 0.2
m = 0	-0.5000	-0.4850	-0.4400
m = 1	0	-0.2985	-0.5879
m = 2	3.0000	2.9700	2.8800

Example 2. Given,

```
X = rand(2,4,5);
n = 2;
P = legendre(n,X)
```

then

```
size(P)
ans =
     3     2     4     5
```

and

```
P(:,1,2,3)
ans =
   -0.2475
   -1.1225
    2.4950
```

is the same as

```
legendre(n,X(1,2,3))
ans =
   -0.2475
   -1.1225
    2.4950
```

Algorithm

`legendre` uses a three-term backward recursion relationship in `m`. This recursion is on a version of the Schmidt seminormalized associated Legendre functions $Q_n^m(x)$, which are complex spherical harmonics. These functions are related to the standard Abramowitz and Stegun [1] functions $P_n^m(x)$ by

$$P_n^m(x) = \sqrt{\frac{(n+m)!}{(n-m)!}}\,Q_n^m(x)$$

They are related to the Schmidt form given previously by

$$S_n^m(x) = Q_n^0(x) \qquad \text{for } m = 0$$

$$S_n^m(x) = (-1)^m \sqrt{2}\,Q_n^m(x) \qquad \text{for } m > 0 .$$

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publlications, 1965, Ch.8.

[2] Jacobs, J. A., *Geomagnetism*, Academic Press, 1987, Ch.4.

Purpose Length of vector

Syntax

```
n = length(X)
```

Description The statement `length(X)` is equivalent to `max(size(X))` for nonempty arrays and `0` for empty arrays.

`n = length(X)` returns the size of the longest dimension of `X`. If `X` is a vector, this is the same as its length.

Examples

```
x = ones(1,8);
n = length(x)

n =
     8

x = rand(2,10,3);
n = length(x)

n =
    10
```

See Also `ndims`, `size`

license

Purpose

Display license number for MATLAB or list of licenses checked out

Syntax

```
license
license('inuse')
result = license('inuse')
result = license('test',feature)
license('test',feature,toggle)
license('checkout',feature)
```

Description

`license` displays the license number for this MATLAB as a string, or one of the following strings:

String	Description
`'demo'`	MATLAB is a demonstration version
`'student'`	MATLAB is the student version
`'unknown'`	License number cannot be determined

`license('inuse')` displays the list of licenses checked out in the current MATLAB session. In the list, products are identified by the license feature names, i.e., the text string used in the `INCREMENT` lines in a License File (license.dat). The `license` function uses only lower-case characters in the license feature names and sorts the list by alphabetical order.

`result = license('inuse')` returns an array of structures, where each structure represents a checked-out license. Each structure contains two fields: `feature` identifies the product and `user` is the username of the person who has the license checked out.

`result = license('test',feature)` tests if a license exists for the product identified by the text string `feature`, returning `1` if the license exists and `0` if the license does not exist.

In the `feature` argument, you must specify the product by license feature name, exactly as it appears in the `INCREMENT` lines in a License File (`license.dat`). For example, `'image_toolbox'` is the feature name for the

Image Processing Toolbox. The `feature` string is case insensitive and must not exceed 27 characters in length.

Note Testing for a license only confirms that the license exists. It does not confirm that the license can be checked out. If the license has expired or if a system administrator has excluded you from using the product in an options file, license will still return `1`, if the license exists.

`license('test',feature,toggle)` enables or disables license testing for the specified product, `feature`, depending on the value of `toggle`. The parameter `toggle` can have either of two values:

`'enable'`	Tests for the specified license return either `1` (license exists) or `0` (license does not exist).
`'disable'`	Tests for the specified license always return `0` (license does not exist)

Note Disabling a test for a particular product can impact all other tests for the existence of the license, not just tests performed using the `license` command.

`result = license('checkout',feature)` checks out a license for the product identified by the text string `feature`, returning `1` if the license was checked out and `0` if it could not be checked out.

Examples

Get a list of licenses currently being used.

```
license('inuse')

image_toolbox
map_toolbox
matlab
```

Get a list of licenses in use with information about who is using the license.

```
S = license('inuse')

S =

1x3 struct array with fields:
    feature
    user

S(1)

ans =

    feature: 'image_toolbox'
       user: 'juser'
```

Determine if a license exists for the Mapping Toolbox.

```
license('test','map_toolbox')

ans =

    1
```

Check out a license for the Control Toolbox.

```
license('checkout','control_toolbox')

ans =

    1
```

Determine if the license for the Control Toolbox is checked out.

```
license('inuse')

control_toolbox
image_toolbox
map_toolbox
matlab
```

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 objects.

`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.

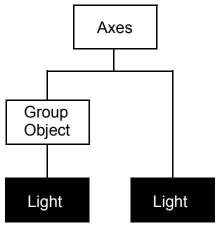
```
h = surf(peaks);
set(h,'FaceLighting','phong','FaceColor','interp',...
      'AmbientStrength',0.5)
light('Position',[1 0 0],'Style','infinite');
```

See Also `lighting`, `material`, `patch`, `surface`

Lighting as a Visualization Tool for more information about lighting

“Lighting” for related functions

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`
Controlling the Appearance		
`SelectionHighlight`	This property is not used by light objects.	Values: `on`, `off` Default: `on`

Property Name	Property Description	Property Value
`Visible`	Makes the effects of the light visible or invisible	Values: `on`, `off` Default: `on`
Controlling Access to Objects		
`HandleVisibility`	Determines if and when the light'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.	Value: `[]` (empty matrix)
`Parent`	The parent of a light object is an axes, hggroup, or hgtransform object.	Value: object 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	Value: any matrix Default: `[]` (empty matrix)
Properties Related to Callback Routine Execution		
`BeingDeleted`	Query to see if object is being deleted.	Values: on | off Read only
`BusyAction`	Specifies how to handle callback routine interruption	Values: `cancel`, `queue` Default: `queue`

Property Name	Property Description	Property Value
`ButtonDownFcn`	This property is not used by light objects.	Value: string or function handle Default: empty string
`CreateFcn`	Defines a callback routine that executes when a light is created	Value: string or function handle Default: empty string
`DeleteFcn`	Defines a callback routine that executes when the light is deleted (via `close` or `delete`)	Value: string or function handle Default: empty string
`Interruptible`	Determines if callback routine can be interrupted	Values: `on`, `off` Default: `on` (can be interrupted)
`UIContextMenu`	This property is not used by light objects.	Value: handle of a Uicontextmenu

Modifying Properties

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

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

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

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

Light Property Descriptions

This section lists property names along with the type of values each accepts.

`BeingDeleted` `on | {off}` Read Only

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

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

`BusyAction` `cancel | {queue}`

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

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

`ButtonDownFcn` string

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 a three-element RGB vector or one of the MATLAB predefined names. See the `ColorSpec` reference page for more information.

`CreateFcn` string or function handle

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 or in a call to the `light` function to create a new light object. 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`.

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

`DeleteFcn` string or function handle

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`.

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

HandleVisibility `{on} | callback | off`

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

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

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

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

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

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

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

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

HitTest `{on} | off`

This property is not used by light objects.

Interruptible {on} | off

Callback routine interruption mode. Light object callback routines defined for the DeleteFcn property are not affected by the Interruptible property.

Parent handle of parent axes, hggroup,or hgtransform

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

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

Position [x,y,z] in axes data units

Location of light object. This property specifies a vector defining the location of the light object. The vector is defined from the origin to the specified *x*-, *y*-, and *z*-coordinates. The placement of the light depends on the setting of the Style property:

- If the Style property is set to local, Position specifies the actual location of the light (which is then a point source that radiates from the location in all directions).
- If the Style property is set to infinite, Position specifies the direction from which the light shines in parallel rays.

Selected on | off

This property is not used by light objects.

SelectionHighlight {on} | off

This property is not used by light objects.

Style {infinite} | local

Parallel or divergent light source. This property determines whether MATLAB places the light object at infinity, in which case the light rays are parallel, or at the location specified by the Position property, in which case the light rays diverge in all directions. See the Position property.

Tag string

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

`Type` string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For light objects, `Type` is always `'light'`.

`UIContextMenu` handle of a uicontextmenu object

This property is not used by light objects.

`UserData` matrix

User-specified data. This property can be any data you want to associate with the light object. The light does not use this property, but you can access it using `set` and `get`.

`Visible` `{on} | off`

Light visibility. While light objects themselves are not visible, you can see the light on patch and surface objects. When you set `Visible` to `off`, the light emanating from the source is not visible. There must be at least one light object in the axes whose `Visible` property is `on` for any lighting features to be enabled (including the axes `AmbientLightColor` and patch and surface `AmbientStrength`).

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)
[az el] = lightangle(light_handle)
```

Description

`lightangle(az,el)` creates a light at the position specified by azimuth and elevation. `az` is the azimuthal (horizontal) rotation and `el` is the vertical elevation (both in degrees). The interpretation of azimuth and elevation is the same as that of the `view` command.

`light_handle = lightangle(az,el)` creates a light and returns the handle of the light in `light_handle`.

`lightangle(light_handle,az,el)` sets the position of the light specified by `light_handle`.

`[az,el] = lightangle(light_handle)` returns the azimuth and elevation of the light specified by `light_handle`.

Remarks

By default, when a light is created, its style is `infinite`. If the light handle passed in to `lightangle` refers to a local light, the distance between the light and the camera target is preserved as the position is changed.

Examples

```
surf(peaks)
axis vis3d
h = light;
for az = -50:10:50
   lightangle(h,az,30)
   drawnow
end
```

See Also

`light`, `camlight`, `view`

Lighting as a Visualization Tool for more information about lighting

"Lighting" for related functions

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 gouraud` 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`

Lighting as a Visualization Tool for more information about lighting

"Lighting" for related functions

lin2mu

Purpose Convert linear audio signal to mu-law

Syntax

```
mu = lin2mu(y)
```

Description `mu = lin2mu(y)` converts linear audio signal amplitudes in the range $-1 \le Y \le 1$ to mu-law encoded “flints” in the range $0 \le u \le 255$.

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 pairs. For example,

```
line('XData',x,'YData',y,'ZData',z,'Color','r','LineWidth',4)
```

Line properties control various aspects of the line object and are described in the "Line Properties" section. You can also set and query property values after creating the line using `set` and `get`.

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

Unlike high-level functions such as `plot`, `line` does not respect the settings of the figure and axes `NextPlot` properties. It simply adds line objects to the current axes. However, axes properties that are under automatic control, such as the axis limits, can change to accommodate the line within the current axes.

Examples

This example uses the `line` function to add a shadow to plotted data. First, plot some data and save the line's handle:

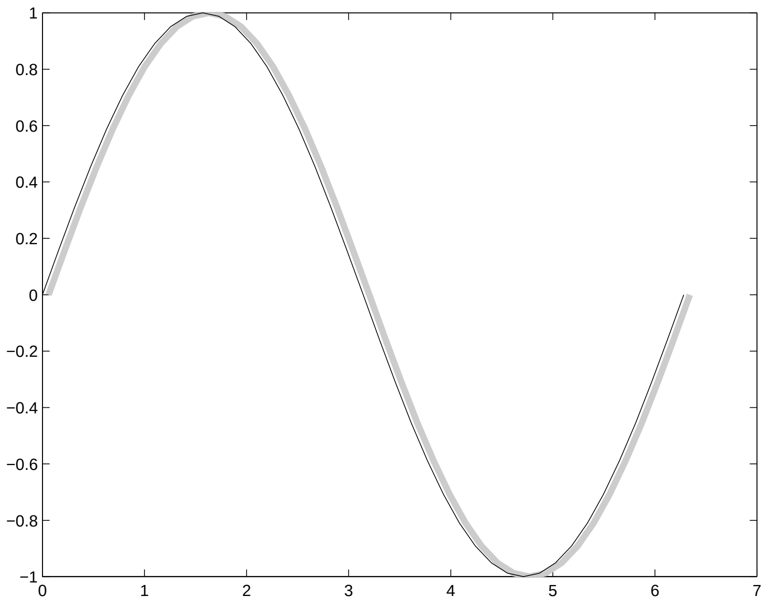
```
t = 0:pi/20:2*pi;
hline1 = plot(t,sin(t),'k');
```

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

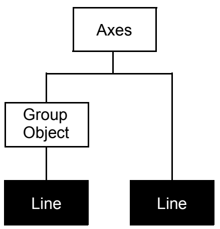
```
line(rand(4,2),rand(4,2),rand(4,1))
```

See Also

`axes`,`newplot`, `plot`, `plot3`

"Object Creation Functions" for related functions

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	Value: vector or matrix Default: [0 1]
YData	The *y*-coordinates defining the line	Value: vector or matrix Default: [0 1]
ZData	The *z*-coordinates defining the line	Value: 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	Value: 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 Dejfault: auto
MarkerFaceColor	Fill color for markers that are closed shapes	Values: ColorSpec, none, auto Default: none
MarkerSize	Size of marker in points	Value: size in points Default: 6
Controlling the Appearance		

Property Name	Property Description	Property Value
`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`	Highlights line when selected (`Selected` property set to `on`)	Values: `on`, `off` Default: `on`
`Visible`	Makes 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 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.	Value: `[]` (empty matrix)
`Parent`	The parent of a line object is an axes, hggroup, or hgtransform object.	Value: object handle
`Selected`	Indicates whether the line is in a selected state	Values: `on`, `off` Default: `on`
`Tag`	User-specified label	Value: any string Default: `''` (empty string)

Property Name	Property Description	Property Value
`Type`	The type of graphics object (read only)	Value: the string `'line'`
`UserData`	User-specified data	Value: any matrix Default: `[]` (empty matrix)
Properties Related to Callback Routine Execution		
`BeingDeleted`	Query to see if object is being deleted.	Values: on | off Read only
`BusyAction`	Specifies how to handle callback routine interruption	Values: `cancel`, `queue` Default: `queue`
`ButtonDownFcn`	Defines a callback routine that executes when a mouse button is pressed on over the line	Value: string or function handle Default: `''` (empty string)
`CreateFcn`	Defines a callback routine that executes when a line is created	Value: string or function handle Default: `''` (empty string)
`DeleteFcn`	Defines a callback routine that executes when the line is deleted (via `close` or `delete`)	Value: string or function handle Default: `''` (empty string)
`Interruptible`	Determines if callback routine can be interrupted	Values: `on`, `off` Default: `on` (can be interrupted)
`UIContextMenu`	Associates a context menu with the line	Value: handle of a Uicontextmenu

Modifying Properties

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

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

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

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

Line Property Descriptions

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

`BeingDeleted` `on | {off}` Read Only

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

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

`BusyAction` `cancel | {queue}`

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

- `cancel` — Discard the event that attempted to execute a second callback routine.

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

ButtonDownFcn string or function handle

Button press callback function. A callback function 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.

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

Children vector of handles

The empty matrix; line objects have no children.

Clipping {on} | off

Clipping mode. MATLAB clips lines to the axes plot box by default. If you set Clipping to off, lines are displayed outside the axes plot box. This can occur if you create a line, set hold to on, freeze axis scaling (set axis to manual), and then create a longer line.

Color ColorSpec

Line color. A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the ColorSpec reference page for more information on specifying color.

CreateFcn string or function handle

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 or in a call to the line function to create a new line object. 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.

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

DeleteFcn string or function handle

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`.

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

EraseMode `{normal} | none | xor | background`

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

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

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

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

`HitTest` `{on} | off`

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

`HandleVisibility` `{on} | callback | off`

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

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

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

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

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

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

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

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

`Interruptible` `{on} | off`

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

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

Line style. This property specifies the line style. Available line styles are shown in the table.

Symbol	Line Style
`' '`	Solid line (default)
`'--'`	Dashed line
`':'`	Dotted line
`' .'`	Dash-dot line
`'none'`	No line

You can use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth` scalar

The width of the line object. Specify this value in points (1 point = $^{1}/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker` character (see table)

Marker symbol. The `Marker` property specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the table.

Marker Specifier	Description
`'+'`	Plus sign
`'o'`	Circle
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No marker (default)

`MarkerEdgeColor` `ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the line's `Color` property.

`MarkerFaceColor` ColorSpec | {none} | auto

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or the figure color, if the axes `Color` property is set to `none` (which is the factory default for axes).

`MarkerSize` size in points

Marker size. A scalar specifying the size of the marker, in points. The default value for `MarkerSize` is six points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent` handle of axes, hggroup, or hgtransform

Parent of line object. This property contains the handle of the line object's parent. The parent of a line object is the axes that contains it. You can reparent line objects to other axes, hggroup, or hgtransform objects.

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

`Selected` on | off

Is object selected? When this property is `on`. MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight` {on} | off

Objects are highlighted when selected. When the `Selected` property is on, MATLAB indicates the selected state by drawing handles at each vertex. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag` string

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

`Type` string (read only)

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

`UIContextMenu` handle of a uicontextmenu object

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

`UserData` matrix

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

`Visible` `{on} | off`

Line visibility. By default, all lines are visible. When set to `off`, the line is not visible, but still exists, and you can `get` and `set` its properties.

`XData` vector of coordinates

X-coordinates. A vector of x-coordinates defining the line. `YData` and `ZData` must be the same length and have the same number of rows. (See Examples.)

`YData` vector or matrix of coordinates

Y-coordinates. A vector of y-coordinates defining the line. `XData` and `ZData` must be the same length and have the same number of rows.

`ZData` vector of coordinates

Z-coordinates. A vector of z-coordinates defining the line. `XData` and `YData` must have the same number of rows.

Lineseries Properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or with the property editor (`propertyeditor`).

See Plot Objects for more information on lineseries objects.

Note that you cannot define default properties for lineseries objects.

Lineseries Property Descriptions

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

`BeingDeleted` `on | {off}` Read Only

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

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

`BusyAction` `cancel | {queue}`

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

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

`ButtonDownFcn` string or function handle

Button press callback function. A callback function 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.

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

`Children` vector of handles

The empty matrix; line objects have no children.

`Clipping` `{on} | off`

Clipping mode. MATLAB clips lines to the axes plot box by default. If you set `Clipping` to `off`, lines are displayed outside the axes plot box. This can occur if you create a line, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a longer line.

`Color` `ColorSpec`

Line color. A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

`CreateFcn` string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates a lineseries object. You must specify the callback during the creation of the object. For example,

```
plot(1:10,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other lineseries properties. Setting this property on an existing lineseries object has no effect.

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

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

`DeleteFcn` string or function handle

Delete 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`.

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

`DisplayName` string

Label used by plot legends. The `legend` command and the figure's active legend use the text you specify for this property as labels for any bar objects appearing in these legends.

`EraseMode` `{normal} | none | xor | background`

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

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

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

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

`HitTest` `{on} | off`

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

`HandleVisibility` `{on} | callback | off`

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

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

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

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

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

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

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

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

`Interruptible` `{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether a lineseries 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`

Style of line drawn. This property specifies the style of the line used to draw the lineseries object. The following table shows available line styles.

Symbol	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth` scalar

The width of the lineseries 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 are displayed at data points. You can set values for the Marker property independently from the LineStyle property. Supported markers are shown in the following table.

Marker Specifier	Description
+	Plus sign
o	Circle
*	Asterisk
.	Point
x	Cross
'square' or s	Square
'diamond' or d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
'pentagram' or p	Five-pointed star (pentagram)
'hexagram' or h	Six-pointed star (hexagram)
none	No marker (default)

MarkerEdgeColor ColorSpec | none | {auto}

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none specifies no color, which makes nonfilled markers invisible. auto sets MarkerEdgeColor to the same color as the Color property.

MarkerFaceColor ColorSpec | {none} | auto

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the axes color, or the figure color, if the axes Color property is set to none (which is the factory default for axes).

MarkerSize size in points

Marker size. A scalar specifying the size of the marker, in points. The default value for MarkerSize is six points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the '.' symbol) at one-third the specified size.

Parent handle of axes, hggroup, or hgtransform

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

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

Selected on | off

Is object selected? When this property is on, MATLAB displays selection handles if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn callback to set this property, allowing users to select the object with the mouse.

SelectionHighlight {on} | off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing handles at each vertex. When SelectionHighlight is off, MATLAB does not draw the handles.

Tag string

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

Type string (read only)

Class of graphics object. For lineseries objects, `Type` is always the string `line`.

UIContextMenu handle of a uicontextmenu object

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

UserData matrix

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

Visible `{on} | off`

Lineseries object visibility. By default, all lineseries objects are visible. When set to `off`, the object 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 lineseries object. `YData` and `ZData` must be the same size.

XDataMode `{auto} | manual`

Use automatic or user-specified x-axis values. If you specify `XData`, MATLAB sets this property to `manual`.

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks and *x*-tick labels to the indices of the `YData`, overwriting any previous values.

XDataSource string (MATLAB variable)

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `XData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `XData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData` vector or matrix of coordinates

Y-coordinates. A vector of *y*-coordinates defining the lineseries object. `XData` and `ZData` must be the same length and have the same number of rows.

`YDataSource` string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData` vector of coordinates

Z-coordinates. A vector of *z*-coordinates defining the lineseries object. `XData` and `YData` must be the same length and have the same number of rows.

`ZDataSource` string (MATLAB variable)

Link `ZData` to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `ZData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `ZData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

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

Specifier	Marker Type
`'square'` or `s`	Square
`'diamond'` or `d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`'pentagram'` or `p`	Five-pointed star (pentagram)
`'hexagram'` or `h`	Six-pointed star (hexagram)

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.

Plotting Data Points with No Line

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:

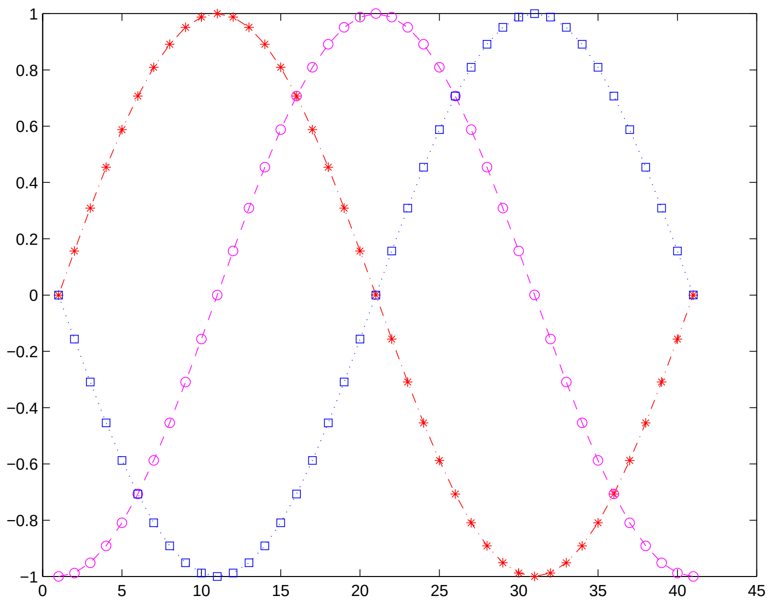
- `LineWidth` — Specifies the width (in points) of the line
- `MarkerEdgeColor` — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles)
- `MarkerFaceColor` — Specifies the color of the face of filled markers
- `MarkerSize`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

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

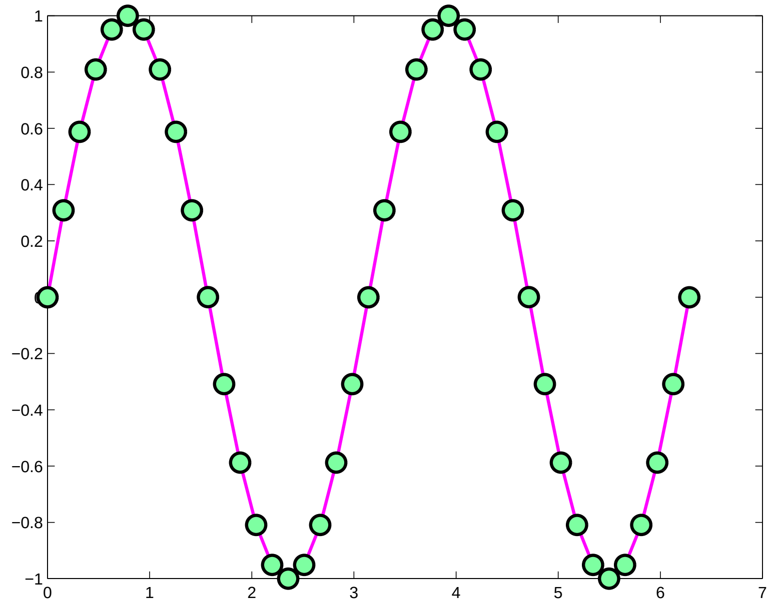
Plot the sine function over three different ranges using different line styles, colors, and markers.

```
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

"Basic Plots and Graphs" for related functions

Purpose

Synchronize limits of specified axes

Syntax

```
linkaxes
linkaxes(axes_handles)
linkaxes(axes_handles,'options)
```

Description

Use `linkaxes` to synchronize the individual axis limits on different subplots within a figure. This is useful when you want to zoom or pan in one subplot and display the same range of data in another subplot. `linkaxes` operates on 2-D plots.

`linkaxes` links the *x*- and *y*-axis limits of all axes (i.e., all subplots) in the current figure.

`linkaxes(axes_handles)` links the *x*- and *y*-axis limits of the axes specified in `axes_handles`.

`linkaxes(axes_handles,'`*`option`*`')` links the axes specified in `axes_handles` according to the specified option. The *`option`* argument can be one of the following strings:

- `x` — Link *x*-axes only
- `y` — Link *y*-axes only
- `xy` — Link *x*- and *y*-axes
- `off` — Remove linking

See the `linkprop` function for more advanced capabilities enabling you to link object properties on any graphics objects.

Examples

This example creates two subplots and links the *x*-axis limits of the two axes. You can use interactive zooming or panning (selected from the figure toolbar) to see the effect of axes linking. For example, pan in one graph and notice how the *x*-axis also changes in the other.

```
ax(1) = subplot(2,2,1);
plot(rand(1,10)*10,'Parent',ax(1));
ax(2) = subplot(2,2,2);
plot(rand(1,10)*100,'Parent',ax(2));
linkaxes(ax,'x');
```

See Also

linkprop

Purpose

Keep same value for corresponding properties

Syntax

```
hlink = linkprop(obj_handles,'PropertyName')
hlink = linkprop(obj_handles,{'PropertyName1','PropertyName2',...})
```

Description

Use `linkprop` to maintain the same values for the corresponding properties of different objects.

`hlink = linkprop(obj_handles,'PropertyName')` maintains the same value for the property *`PropertyName`* on all objects whose handles appear in `obj_handles`. `linkprop` returns the link object in `hlink`. See "Link Object" for more information.

`hlink = linkprop(obj_handles,{'PropertyName1','PropertyName2',...})` maintains the same respective values for all properties passed as a cell array on all objects whose handles appear in `obj_handles`.

Note that the linked properties of all linked objects are updated immediately when `linkprop` is called. The first object in the list (`obj_handles`) determines the property values for the rest of the objects.

Link Object

The mechanism to link the properties of different graphics objects is stored in the link object, which is returned by `linkprop`. Therefore, the link object must exist within the context where you want property linking to occur (such as in the base workspace if users are to interact with the objects from the command line or figure tools).

The following list describes ways to maintain a reference to the link object.

- Return the link object as an output argument from a function and keep it in the base workspace while interacting with the linked objects.
- Make the `hlink` variable global.
- Store the `hlink` variable in an object's `UserData` property or in application data. See the "Examples" section for an example that uses application data.

Modifying Link Object

If you want to change either the graphics objects or the properties that are linked, you need to use the link object methods designed for that purpose.

These methods are functions that operate only on link objects. To use them, you must first create a link object using `linkprop`.

Method	Purpose
`addtarget`	Add specified graphics object to the link object's targets.
`removetarget`	Remove specified graphics object from the link object's targets.
`addprop`	Add specified property to the linked properties.
`removeprop`	Remove specified property from the linked properties.

Method Syntax

```
addtarget(hlink,obj_handles)
removetarget(hlink,obj_handles)
addprop(hlink,'PropertyName')
removeprop(hlink,'PropertyName')
```

Arguments

- `hlink` — Link object returned by `linkprop`
- `obj_handles` — One or more graphic object handles
- *`PropertyName`* — Name of a property common to all target objects

Examples

This example creates four isosurface graphs of fluid flow data, each displaying a different isovalue. The `CameraPosition` and `CameraUpVector` properties of each subplot axes are linked so that the user can rotate all subplots in unison.

After running the example, select **Rotate 3D** from the figure **Tools** menu and observe how all subplots rotate together.

Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

The property linking code is in step 3.

1 Define the data using the `flow` M-file and specify property values for the isosurface (which is a patch object).

```
function linkprop_example
[x y z v] = flow;
isoval = [-3 -1 0 1];
props.FaceColor = [0 0 .5];
props.EdgeColor = 'none';
props.AmbientStrength = 1;
props.FaceLighting = 'gouraud';
```

2 Create four subplot axes and add an isosurface graph to each one. Add a title and set viewing and lighting parameters using a local function (`set_view`). (`subplot`, `patch`, `isosurface`, `title`, `num2str`)

```
for k = 1:4
    h(k) = subplot(2,2,k);
    patch(isosurface(x,y,z,v,isoval(k)),props)
    title(h(k),['Isovalue = ',num2str(k)])
    set_view(h(k))
end
```

3 Link the `CameraPosition` and `CameraTarget` properties of all subplot axes. Since this example function will have completed execution when the user is rotating the subplots, the link object is stored in the first subplot axes application data. See `setappdata` for more information on using application data.

```
hlink = linkprop(h,{'CameraPosition','CameraUpVector'});
key = 'graphics_linkprop';
% Store link object on first subplot axes
setappdata(h(1),key,hlink);
```

4 The following local function contains viewing and lighting commands issued on each axes. It is called with the creation of each subplot (`view`, `axis`, `camlight`).

```
function set_view(ax)
% Set the view and add lighting
view(ax,3); axis(ax,'tight','equal')
camlight left; camlight right
```

```
% Make axes invisible and title visible
axis(ax,'off')
set(get(ax,'title'),'Visible','on')
```

Linking an Additional Property

Suppose you want to add the axes `PlotBoxAspectRatio` to the linked properties in the previous example. You can do this by modifying the link object that is stored in the first subplot axes' application data.

1. First click the first subplot axes to make it the current axes (since its handle was saved only within the creating function). Then get the link object's handle from application data (`getappdata`).

   ```
   hlink = getappdata(gca,'graphics_linkprop');
   ```

2. Use the `addprop` method to add a new property to the link object.

   ```
   addprop(hlink,'PlotBoxAspectRatio')
   ```

Since `hlink` is a reference to the link object (i.e., not a copy), `addprop` can change the object that is stored in application data.

See Also

`getappdata`, `linkaxes`, `setappdata`

Purpose

Solve a linear system of equations

Syntax

```
X = linsolve(A,B)
X = linsolve(A,B,opts)
```

Description

`X = linsolve(A,B)` solves the linear system `A*X = B` using LU factorization with partial pivoting when A is square and QR factorization with column pivoting otherwise. The number of columns of `A` must equal the number of rows of `B` must have the same number of rows. If `A` is m-by-n and `B` is n-by-k, then `X` is m-by-k. `linsolve` returns a warning if `A` is square and ill conditioned or if it is not square and rank deficient.

`[X, R] = linsolve(A,B)` suppresses these warnings and returns `R`, which is the reciprocal of the condition number of `A` if `A` is square, or the rank of `A` if `A` is not square.

`X = linsolve(A,B,opts)` solves the linear system `A*X = B` or `A'*X = B`, using the solver that is most appropriate given the properties of the matrix `A`, which you specify in `opts`. For example, if `A` is upper triangular, you can set `opts.UT = true` to make `linsolve` use a solver designed for upper triangular matrices. If `A` has the properties in `opts`, `linsolve` is faster than `mldivide`, because `linsolve` does not perform any tests to verify that `A` has the specified properties.

Caution If `A` does not have the properties that you specify in `opts`, `linsolve` returns incorrect results and does not return an error message. If you are not sure whether `A` has the specified properties, use `mldivide` instead.

The `TRANSA` field of the `opts` structure specifies the form of the linear system you want to solve:

- If you set `opts.TRANSA = false`, `linsolve(A,B,opts)` solves `A*X = B`.
- If you set `opts.TRANSA` = `true`, `linsolve(A,B,opts)` solves `A'*X = B`.

The following table lists all the field of `opts` and their corresponding matrix properties. The values of the fields of `opts` must be `logical` and the default value for all fields is `false`.

Field Name	Matrix Property
`LT`	Lower triangular
`UT`	Upper triangular
`UHESS`	Upper Hessenberg
`SYM`	Real symmetric or complex Hermitian
`POSDEF`	Positive definite
`RECT`	General rectangular
`TRANSA`	Conjugate transpose — specifies whether the function solves `A*X = B` or `A'*X = B`

The following table lists all combinations of field values in `opts` that are valid for `linsolve`. A true/false entry indicates that `linsolve` accepts either true or false.

LT	UT	UHESS	SYM	POSDEF	RECT	TRANS
`true`	`false`	`false`	`false`	`false`	`true/false`	`true/false`
`false`	`true`	`false`	`false`	`false`	`true/false`	`true/false`
`false`	`false`	`true`	`false`	`false`	`false`	`true/false`
`false`	`false`	`false`	`true`	`true`	`false`	`true/false`
`false`	`false`	`false`	`false`	`false`	`true/false`	`true/false`

Example

The following code solves the system `A'x = b` for an upper triangular matrix `A` using both `mldivide` and `linsolve`.

```
A = triu(rand(5,3)); x = [1 1 1 0 0]'; b = A'*x;
y1 = (A')\b
```

```
opts.UT = true; opts.TRANSA = true;
y2 = linsolve(A,b,opts)

y1 =

    1.0000
    1.0000
    1.0000
         0
         0


y2 =

    1.0000
    1.0000
    1.0000
         0
         0
```

Note If you are working with matrices having different properties, it is useful to create an options structure for each type of matrix, such as `opts_sym`. This way you do not need to change the fields whenever you solve a system with a different type of matrix `A`.

See Also

`mldivide`, `slash`

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`

"Predefined Dialog Boxes" for related functions

Purpose Load workspace variables from disk

Syntax

```
load
load('filename')
load('filename', 'X', 'Y', 'Z')
load('filename', '-regexp', exprlist)
load('-mat', 'filename')
load('-ascii', 'filename')
S = load(...)
load filename -regexp expr1 expr2 ...
```

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 a file named `filename.mat` and treats it as a binary MAT-file. If `filename` has an extension other than `.mat`, `load` treats the file as ASCII data.

`load('filename', 'X', 'Y', 'Z')` loads just the specified variables from the MAT-file. The wildcard `'*'` loads variables that match a pattern (MAT-file only).

`load('filename', '-regexp', exprlist)` loads those variables that match any of the regular expressions in `exprlist`, where `exprlist` is a comma-delimited list of quoted regular expressions.

`load('-mat', 'filename')` forces `load` to treat the file as a MAT-file, regardless of file extension. If the file is not a MAT-file, `load` returns an error.

`load('-ascii', 'filename')` forces `load` to treat the file as an ASCII file, regardless of file extension. If the file is not numeric text, `load` returns an error.

`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 retrieved. When the file contains ASCII data, `S` is a double-precision array.

`load filename -regexp expr1 expr2 ...` is the command form of the syntax.

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 a command-line option with this functional form, specify any option as a string argument, including the hyphen. For example,

```
load('myfile.dat', '-mat')
```

Remarks

For information on any of the following topics related to saving to MAT-files, see "Importing Data from MAT-Files" in the "MATLAB Programming" documentation:

- Previewing MAT-file contents
- Loading binary data
- Loading ASCII data

Examples

Example 1 — Loading From a Binary MAT-file

To see what is in the MAT-file prior to loading it, use `whos -file`:

```
whos -file mydata.mat
  Name           Size            Bytes  Class

  javArray      10x1                     java.lang.Double[][]
  spArray        5x5                 84  double array (sparse)
  strArray       2x5                678  cell array
  x              3x2x2               96  double array
  y              4x5               1230  cell array
```

Clear the workspace and load it from MAT-file `mydata.mat`:

```
clear
load mydata

whos
  Name           Size             Bytes  Class

  javArray      10x1                     java.lang.Double[][]
  spArray        5x5                 84  double array (sparse)
  strArray       2x5                678  cell array
  x              3x2x2               96  double array
  y              4x5               1230  cell array
```

Example 2 — Loading From an ASCII File

Create several 4-columnn matrices and save them to an ASCII file:

```
a = magic(4);  b = ones(2, 4) * -5.7;  c = [8 6 4 2];
save -ascii mydata.dat
```

Clear the workspace and load it from the file mydata.dat. If the filename has an extension other than .mat, MATLAB assumes that it is ASCII:

```
clear
load mydata.dat
```

MATLAB loads all data from the ASCII file, merges it into a single matrix, and assigns the matrix to a variable named after the filename:

```
mydata
mydata =
   16.0000    2.0000    3.0000   13.0000
    5.0000   11.0000   10.0000    8.0000
    9.0000    7.0000    6.0000   12.0000
    4.0000   14.0000   15.0000    1.0000
   -5.7000   -5.7000   -5.7000   -5.7000
   -5.7000   -5.7000   -5.7000   -5.7000
    8.0000    6.0000    4.0000    2.0000
```

Example 3 — Using Regular Expressions

Using regular expressions, load from MAT-file mydata.mat those variables with names that begin with Mon, Tue, or Wed:

```
load('mydata', '-regexp', '^Mon|^Tue|^Wed');
```

Here is another way of doing the same thing. In this case, there are three separate expression arguments:

```
load('mydata', '-regexp', '^Mon', '^Tue', '^Wed');
```

See Also

clear, fprintf, fscanf, partialpath, save, spconvert, who

loadobj

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 that 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 data types (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.

A child object does not inherit the `loadobj` method of its parent class. To implement `loadobj` for any class, including a class that inherits from a parent, you must define a `loadobj` method within that class directory.

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 Compute `log(1+x)` accurately for small values of `x`

Syntax

```
y = log1p(x)
```

Description `y = log1p(x)` computes `log(1+x)`, compensating for the roundoff in `1+x`. log1p(x) is more accurate than log(1+x) for small values of x. For small `x`, `log1p(x)` is approximately `x`, whereas `log(1+x)` can be zero.

See Also `log`, `expm1`

Purpose

Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

Syntax

```
Y = log2(X)
[F,E] = log2(X)
```

Description

`Y = log2(X)` computes the base 2 logarithm of the elements of `X`.

`[F,E] = log2(X)` returns arrays `F` and `E`. Argument `F` is an array of real values, usually in the range `0.5 <= abs(F) < 1`. For real `X`, `F` satisfies the equation: `X = F.*2.^E`. Argument `E` is an array of integers that, for real `X`, satisfy the equation: `X = F.*2.^E`.

Remarks

This function corresponds to the ANSI C function `frexp()` and the IEEE floating-point standard function `logb()`. Any zeros in `X` produce `F = 0` and `E = 0`.

Examples

For IEEE arithmetic, the statement `[F,E] = log2(X)` yields the values:

X	F	E
1	1/2	1
pi	pi/4	2
-3	-3/4	2
eps	1/2	-51
realmax	1-eps/2	1024
realmin	1/2	-1021

See Also

`log`, `pow2`

Purpose 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`

logical

Purpose Convert numeric values to logical

Syntax

```
K = logical(A)
```

Description `K = logical(A)` returns an array that can be used for logical indexing or logical tests.

`A(B)`, where `B` is a logical array, 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 array, `A`, to a numeric double array.

Logical arrays are also created by the relational operators (==,<,>,~, etc.) and functions like `any`, `all`, `isnan`, `isinf`, and `isfinite`.

Examples Given `A = [1 2 3; 4 5 6; 7 8 9]`, the statement `B = logical(eye(3))` returns a logical array

```
B =
     1     0     0
     0     1     0
     0     0     1
```

which can be used in logical indexing that returns `A`'s diagonal elements:

```
A(B)

ans =
     1
     5
     9
```

However, attempting to index into `A` using the *numeric* array `eye(3)` results in:

```
A(eye(3))
??? Subscript indices must either be real positive integers or
logicals.
```

See Also `islogical`, logical operators (elementwise and short-circuit)

Purpose Log-log scale plot

Syntax

```
loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,LineSpec,...)
loglog(...,'PropertyName',PropertyValue,...)
h = loglog(...)
hline = loglog('v6',...)
```

Description

`loglog(Y)` plots the columns of `Y` versus their index if `Y` contains real numbers. If `Y` contains complex numbers, `loglog(Y)` and `loglog(real(Y),imag(Y))` are equivalent. `loglog` ignores the imaginary component in all other uses of this function.

`loglog(X1,Y1,...)` plots all `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `loglog` plots the vector argument versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`loglog(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples, where `LineSpec` determines line type, marker symbol, and color of the plotted lines. You can mix `Xn,Yn,LineSpec` triples with `Xn,Yn` pairs, for example,

```
loglog(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

`loglog(...,'`*`PropertyName`*`',PropertyValue,...)` sets property values for all lineseries graphics objects created by `loglog`. See the `line` reference page for more information.

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

Backward Compatible Version

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

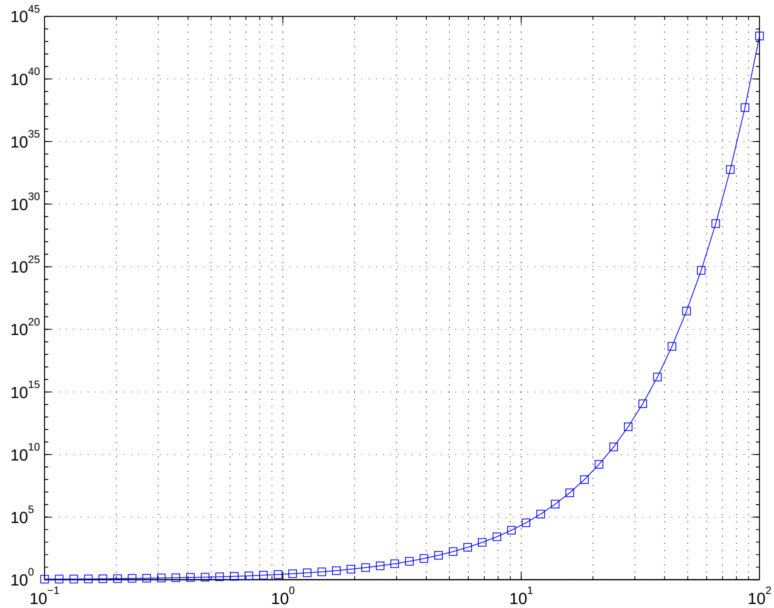
Remarks

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

`LineSpec`, `plot`, `semilogx`, `semilogy`

"Basic Plots and Graphs" for related functions

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.

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 `:`

lookfor

Purpose Search for 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 the MATLAB search path. For all files in which a match occurs, `lookfor` displays the H1 line.

`lookfor topic -all` searches the entire first comment block of an M-file looking for `topic`.

Examples For example

```
lookfor inverse
```

finds at least a dozen matches, including H1 lines containing "inverse hyperbolic cosine," "two-dimensional inverse FFT," and "pseudoinverse." Contrast this with

```
which inverse
```

or

```
what inverse
```

These functions run more quickly, but probably fail to find anything because MATLAB does not have a function `inverse`.

In summary, `what` lists the functions in a given 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. For instructions, see Finding Files and Content Within Files.

See Also `dir`, `doc`, `filebrowser`, `findstr`, `help`, `helpdesk`, `helpwin`, `regexp`, `what`, `which`, `who`

Purpose Convert string to lowercase

Syntax

```
t = lower('str')
B = lower(A)
```

Description

`t = lower('str')` returns the string formed by converting any uppercase characters in `str` to the corresponding lowercase characters and leaving all other characters unchanged.

`B = lower(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `lower` to each string within `A`.

Examples

`lower('MathWorks')` is `mathworks`.

Remarks

Character sets supported:

- PC: 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`

Purpose Least squares solution in the presence of known covariance

Syntax

```
x = lscov(A,b)
x = lscov(A,b,w)
x = lscov(A,b,V)
[x,stdx] = lscov(A,b,V)
[x,stdx,mse] = lscov(...)
[x,stdx,mse,S] = lscov(...)
```

Description

`x = lscov(A,b)` returns the ordinary least squares solution to the linear system of equations `A*x = b`, i.e., `x` is the n-by-1 vector that minimizes the sum of squared errors `(b - A*x)'*(b - A*x)`, where `A` is m-by-n, and `b` is m-by-1. `b` can also be an m-by-k matrix, and `lscov` returns one solution for each column of `b`. When `rank(A) < n`, `lscov` sets the maximum possible number of elements of `x` to zero to obtain a "basic solution".

`x = lscov(A,b,w)`, where `w` is a vector length m of real positive weights, returns the weighted least squares solution to the linear system `A*x = b`, that is, `x` minimizes `(b - A*x)'*diag(w)*(b - A*x)`. `w` typically contains either counts or inverse variances.

`x = lscov(A,b,V)`, where `V` is an m-by-m real symmetric positive definite matrix, returns the generalized least squares solution to the linear system `A*x = b` with covariance matrix proportional to `V`, that is, `x` minimizes `(b - A*x)'*inv(V)*(b - A*x)`.

More generally, `V` can be positive semidefinite, and `lscov` returns `x` that minimizes `e'*e`, subject to `A*x + T*e = b`, where the minimization is over `x` and `e`, and `T*T' = V`. When `V` is semidefinite, this problem has a solution only if `b` is consistent with `A` and `V` (that is, `b` is in the column space of `[A T]`), otherwise `lscov` returns an error.

By default, `lscov` computes the Cholesky decomposition of `V` and, in effect, inverts that factor to transform the problem into ordinary least squares. However, if `lscov` determines that `V` is semidefinite, it uses an orthogonal decomposition algorithm that avoids inverting `V`.

`x = lscov(A,b,V,alg)` specifies the algorithm used to compute `x` when `V` is a matrix. `alg` can have the following values:

- `'chol'` uses the Cholesky decomposition of `V`.

- 'orth' uses orthogonal decompositions, and is more appropriate when V is ill-conditioned or singular, but is computationally more expensive.

[x,stdx] = lscov(...) returns the estimated standard errors of x. When A is rank deficient, stdx contains zeros in the elements corresponding to the necessarily zero elements of x.

[x,stdx,mse] = lscov(...) returns the mean squared error.

[x,stdx,mse,S] = lscov(...) returns the estimated covariance matrix of x. When A is rank deficient, S contains zeros in the rows and columns corresponding to the necessarily zero elements of x. lscov cannot return S if it is called with multiple right-hand sides, that is, if size(B,2) > 1.

The standard formulas for these quantities, when A and V are full rank, are

- x = inv(A'*inv(V)*A)*A'*inv(V)*B
- mse = B'*(inv(V) - inv(V)*A*inv(A'*inv(V)*A)*A'*inv(V))*B./(m-n)
- S = inv(A'*inv(V)*A)*mse
- stdx = sqrt(diag(S))

However, lscov uses methods that are faster and more stable, and are applicable to rank deficient cases.

lscov assumes that the covariance matrix of B is known only up to a scale factor. mse is an estimate of that unknown scale factor, and lscov scales the outputs S and stdx appropriately. However, if V is known to be exactly the covariance matrix of B, then that scaling is unnecessary. To get the appropriate estimates in this case, you should rescale S and stdx by 1/mse and sqrt(1/mse), respectively.

Algorithm

The vector x minimizes the quantity (A*x-b)'*inv(V)*(A*x-b). The classical linear algebra solution to this problem is

```
x = inv(A'*inv(V)*A)*A'*inv(V)*b
```

but the lscov function instead computes the QR decomposition of A and then modifies Q by V.

See Also

lsqnonneg, qr

The arithmetic operator \

Reference

[1] Strang, G., *Introduction to Applied Mathematics*, Wellesley-Cambridge, 1986, p. 398.

lsqnonneg

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.

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,...)
[x,flag,relres,iter,resvec,lsvec] = 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 it should be large and sparse. 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. You can suppress these messages by calling `lsqr` with the syntax

```
[x,flag] = lsqr(A,b,...)
```

which returns an integer `flag` instead of the message, as described in the following table.

`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 you specify the `flag` output.

`[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns an estimate of the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of the residual norm estimates at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of estimates of the scaled normal equations residual at each iteration: `norm((A*inv(M))'*(B-A*X))/norm(A*inv(M),'fro')`. Note that the estimate of `norm(A*inv(M),'fro')` changes, and hopefully improves, at each iteration.

Examples

```
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 11 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`, `norm`, `pcg`, `qmr`, `symmlq`

`@` (function handle)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "LSQR: An Algorithm for Sparse Linear Equations And Sparse Least Squares," *ACM Trans. Math. Soft.*, Vol.8, 1982, pp. 43-71.

Purpose LU matrix factorization

Syntax

```
[L,U] = lu(X)
[L,U,P] = lu(X)
Y = lu(X)
[L,U,P,Q] = lu(X)
[L,U,P] = lu(X,thresh)
[L,U,P,Q] = lu(X,thresh)
```

Description

The `lu` function expresses a 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. `X` can be rectangular. For a full matrix `X`, `lu` uses the Linear Algebra Package (LAPACK) routines described in "Algorithm" on page 2-1392.

`[L,U] = lu(X)` returns an upper triangular matrix in `U` and a permuted lower triangular matrix `L` (that is, a product of lower triangular and permutation matrices), such that `X = L*U`.

`[L,U,P] = lu(X)` returns an upper triangular matrix in `U`, a lower triangular matrix `L` with a unit diagonal, and a permutation matrix `P`, so that `L*U = P*X`.

`Y = lu(X)` returns a matrix `Y`, which contains the strictly lower triangular `L`, i.e., without its unit diagonal, and the upper triangular `U` as submatrices. That is, if `[L,U,P] = lu(X)`, then `Y = U+L-eye(size(X))`. The permutation matrix `P` is not returned by `Y = lu(X)`.

`[L,U,P,Q] = lu(X)` for sparse nonempty `X`, returns a unit lower triangular matrix `L`, an upper triangular matrix `U`, a row permutation matrix `P`, and a column reordering matrix `Q`, so that `P*X*Q = L*U`. This syntax uses UMFPACK and is significantly more time and memory efficient than the other syntaxes, even when used with `colamd`. If `X` is empty or not sparse, `lu` displays an error message.

`[L,U,P] = lu(X,thresh)` controls pivoting in sparse matrices, where `thresh` is a pivot threshold in the interval `[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` (conventional partial pivoting) is the default.

`[L,U,P,Q] = lu(X,thresh)` controls pivoting in UMFPACK, where `thresh` is a pivot threshold in the interval `[0,1]`. Given a pivot column `j`, UMFPACK selects the sparsest candidate pivot row `i` such that the absolute value of the pivot entry is greater than or equal to `thresh` times the absolute value of the largest entry in the column `j`. For complex matrices, absolute values are computed as `abs(real(a)) + abs(imag(a))`. The magnitude of entries in `L` is limited to `1/thresh`.

Setting `thresh` to `1.0` results in conventional partial pivoting. The default value is `0.1`. Smaller values of `thresh` lead to sparser LU factors, but the solution might be inaccurate. Larger values usually (but not always) lead to a more accurate solution, but increase the number of steps the algorithm performs.

Note In rare instances, incorrect factorization results in `P*X*Q` ≠ `L*U`. Increase `thresh`, to a maximum of `1.0` (regular partial pivoting), and try again.

Remarks

Most of the algorithms for computing LU factorization are variants of Gaussian elimination. The factorization is a key step in obtaining the inverse with `inv` and the determinant with `det`. It is also the basis for the linear equation solution or matrix division obtained with `\` and `/`.

Arguments

`X` Rectangular matrix to be factored.

`thresh` Pivot threshold for sparse matrices. Valid values are in the interval `[0,1]`. If you specify the fourth output `Q`, the default is `0.1`. Otherwise the default is `1.0`.

`L` Factor of `X`. Depending on the form of the function, `L` is either a unit lower triangular matrix, or else the product of a unit lower triangular matrix with `P'`.

`U` Upper triangular matrix that is a factor of `X`.

`P` Row permutation matrix satisfying the equation `L*U = P*X`, or `L*U = P*X*Q`. Used for numerical stability.

`Q` Column permutation matrix satisfying the equation `P*X*Q = L*U`. Used to reduce fill-in in the sparse case.

Examples

Example 1. Start with

```
A = [ 1    2    3
      4    5    6
      7    8    0 ];
```

To see the LU factorization, call `lu` with two output arguments.

```
[L1,U] = lu(A)

L1 =
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
    1.0000         0         0

U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000
```

Notice that `L1` is a permutation of a lower triangular matrix: if you switch rows 2 and 3, and then switch rows 1 and 2, the resulting matrix is lower triangular and has `1`s on the diagonal. Notice also that `U` is upper triangular. To check that the factorization does its job, compute the product

```
L1*U
```

which returns the original `A`. The inverse of the example matrix, `X = inv(A)`, is actually computed from the inverses of the triangular factors

```
X = inv(U)*inv(L1)
```

Using three arguments on the left side to get the permutation matrix as well

```
[L2,U,P] = lu(A)
```

returns a truly lower triangular `L2`, the the same value of `U`, and the permutation matrix `P`.

```
L2 =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000

U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000

P =
    0    0    1
    1    0    0
    0    1    0
```

Note that `L2 = P*L1`.

```
P*L1

ans =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
```

To verify that `L2*U` is a permuted version of `A`, compute `L2*U` and subtract it from `P*A`:

```
P*A - L2*U

ans =
     0     0     0
     0     0     0
     0     0     0
```

In this case, `inv(U)*inv(L)` results in the permutation of `inv(A)` given by `inv(P)*inv(A)`.

The determinant of the example matrix is

```
d = det(A)

d =
    27
```

It is computed from the determinants of the triangular factors

```
d = det(L)*det(U)
```

The solution to $Ax = b$ is obtained with matrix division

```
x = A\b
```

The solution is actually computed by solving two triangular systems

```
y = L\b
x = U\y
```

Example 2. Generate a 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster-Fuller geodesic dome.

```
B = bucky;
```

Use the sparse matrix syntax with four outputs to get the row and column permutation matrices.

```
[L,U,P,Q] = lu(B);
```

Apply the permutation matrices to `B`, and subtract the product of the lower and upper triangular matrices.

```
Z = P*B*Q - L*U;
norm(Z,1)

ans =
  7.9936e-015
```

The 1-norm of their difference is within roundoff error, indicating that `L*U = P*B*Q`.

Algorithm

For full matrices `X`, `lu` uses the LAPACK routines listed in the following table.

	Real	Complex
X double	DGETRF	ZGETRF
X single	SGETRF	CGETRF

For sparse `X`, with four outputs, `lu` uses UMFPACK. With three or fewer outputs, `lu` uses code introduced in MATLAB 4.

See Also

`cond`, `det`, `inv`, `luinc`, `qr`, `rref`

The arithmetic operators `\` and `/`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

[2] Davis, T. A., *UMFPACK Version 4.0 User Guide* (`http://www.cise.ufl.edu/research/sparse/umfpack/v4.0/UserGuide.pdf`), Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, 2002.

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.

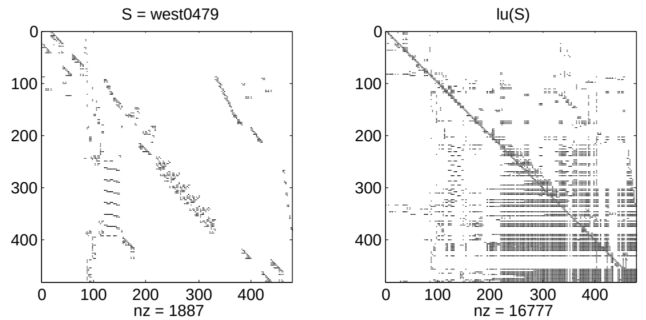
Limitations

`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);
```

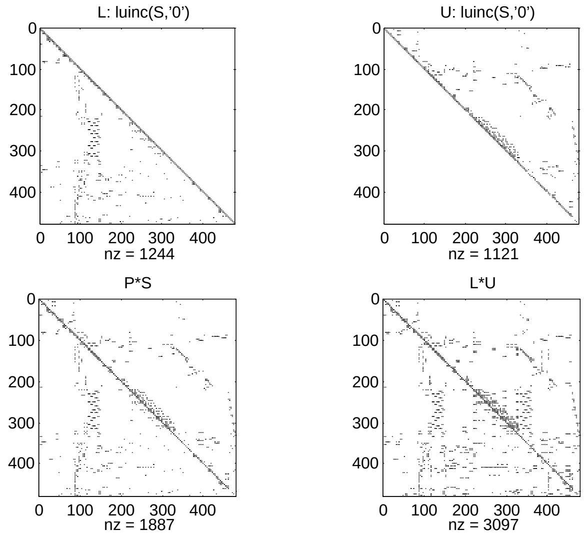


Compute the incomplete LU factorization of level 0.

```
[L,U,P] = luinc(S,'0');
D = (L*U).*spones(P*S)-P*S;
```

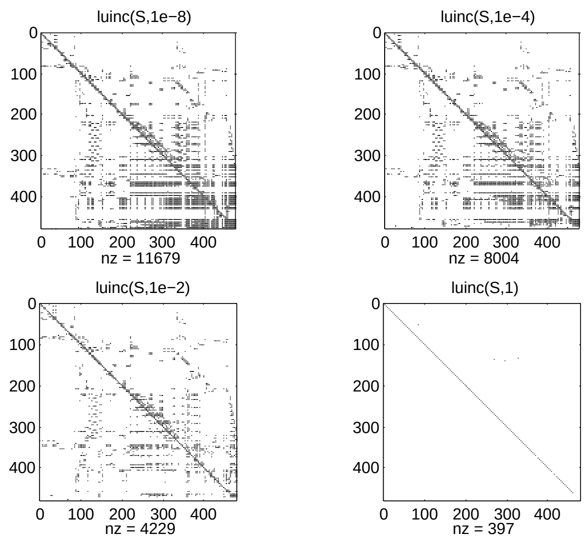
`spones(U)` and `spones(triu(P*S))` are identical.

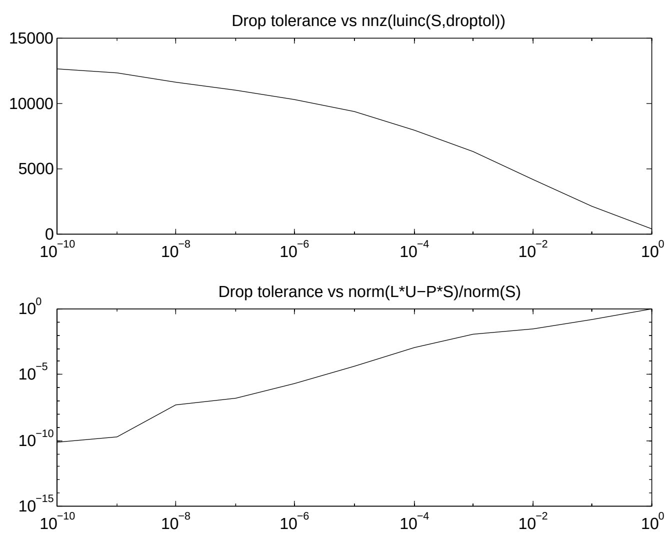
`spones(L)` and `spones(tril(P*S))` disagree at 73 places on the diagonal, where `L` is `1` and `P*S` is `0`, and also at position (206,113), where `L` is `0` due to cancellation, and `P*S` is `-1`. `D` has entries of the order of eps.




```
[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)

2magic

Purpose Magic square

Syntax

```
M = magic(n)
```

Description

`M = magic(n)` returns an `n`-by-`n` matrix constructed from the integers `1` through `n^2` with equal row and column sums. The order `n` must be a scalar greater than or equal to `3`.

Remarks

A magic square, scaled by its magic sum, is doubly stochastic.

Examples

The magic square of order 3 is

```
M = magic(3)

M =

     8     1     6
     3     5     7
     4     9     2
```

This is called a magic square because the sum of the elements in each column is the same.

```
sum(M) =

     15     15     15
```

And the sum of the elements in each row, obtained by transposing twice, is the same.

```
sum(M')' =

     15
     15
     15
```

This is also a special magic square because the diagonal elements have the same sum.

```
sum(diag(M)) =
```

```
        15
```

The value of the characteristic sum for a magic square of order n is

```
sum(1:n^2)/n
```

which, when n = 3, is 15.

Algorithm

There are three different algorithms:

- n odd
- n even but not divisible by four
- n divisible by four

To make this apparent, type

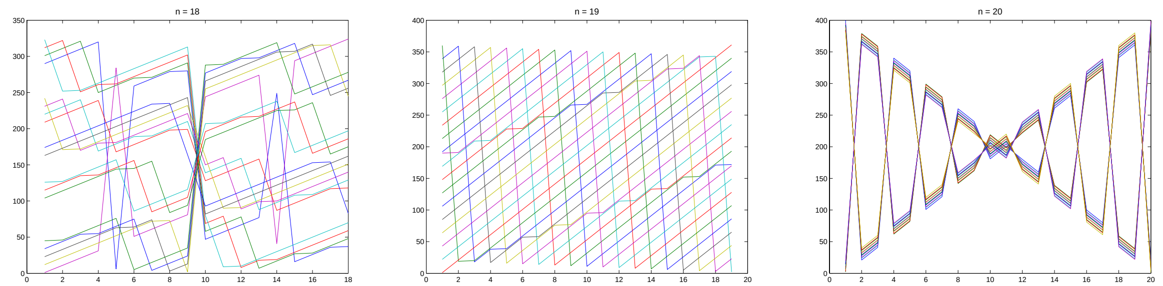
```
for n = 3:20
    A = magic(n);
    r(n) = rank(A);
end
```

For n odd, the rank of the magic square is n. For n divisible by 4, the rank is 3. For n even but not divisible by 4, the rank is n/2 + 2.

```
[(3:20)',r(3:20)']
ans =
     3     3
     4     3
     5     5
     6     5
     7     7
     8     3
     9     9
    10     7
    11    11
    12     3
    13    13
    14     9
    15    15
    16     3
    17    17
    18    11
    19    19
```

20 3

Plotting A for n = 18, 19, 20 shows the characteristic plot for each category.



Limitations

If you supply `n` less than `3`, `magic` returns either a nonmagic square, or else the degenerate magic squares `1` and `[]`.

See Also

`ones`, `rand`

makehgtform

Purpose

Create 4-by-4 transform matrix

Syntax

```
M = makehgtform
M = makehgtform('translate',[tx ty tz])
M = makehgtform('scale',s)
M = makehgtform('scale',[sx,sy,sz])
M = makehgtform('xrotate',t)
M = makehgtform('yrotate',t)
M = makehgtform('zrotate',t)
M = makehgtform('axisrotate',[ax,ay,az],t)
```

Description

Use `makehgtform` to create transform matrices for translation, scaling, and rotation of graphics objects. Apply the transform to graphics objects by assigning the transform to the `Matrix` property of a parent hgtransform object. See Examples for more information.

`M = makehgtform` returns an identity transform.

`M = makehgtform('translate',[tx ty tz])` or `M = makehgtform('translate',tx,ty,tz)` returns a transform that translates along the *x*-axis by `tx`, along the *y*-axis by `ty`, and along the *z*-axis by `tz`.

`M = makehgtform('scale',s)` returns a transform that scales uniformly along the *x*-, *y*-, and *z*-axes.

`M = makehgtform('scale',[sx,sy,sz])` returns a transform that scales along the *x*-axis by `sx`, along the *y*-axis by `sy`, and along the *z*-axis by `sz`.

`M = makehgtform('xrotate',t)` returns a transform that rotates around the *x*-axis by `t` radians.

`M = makehgtform('yrotate',t)` returns a transform that rotates around the *y*-axis by `t` radians.

`M = makehgtform('zrotate',t)` returns a transform that rotates around the *z*-axis by `t` radians.

Purpose

Divide matrix into cell array of matrices

Syntax

```
c = mat2cell(x,m,n)
c = mat2cell(x,d1,d2,d3,...,dn)
c = mat2cell(x,r)
```

Description

`c = mat2cell(x,m,n)` divides the two-dimensional matrix `x` into adjacent submatrices, each contained in a cell of the returned cell array `c`. Vectors `m` and `n` specify the number of rows and columns, respectively, to be assigned to the submatrices in `c`.

The example shown below divides a 60-by-50 matrix into six smaller matrices. MATLAB returns the new matrices in a 3-by-2 cell array:

```
mat2cell(x, [10 20 30], [25 25])
```

The sum of the element values in `m` must equal the total number of rows in `x`. And the sum of the element values in `n` must equal the number of columns in `x`.

The elements of `m` and `n` determine the size of each cell in `c` by satisfying the following formula for `i = 1:length(m)` and `j = 1:length(n)`:

```
size(c{i,j}) == [m(i) n(j)]
```

`c = mat2cell(x,d1,d2,d3,...,dn)` divides the multidimensional array `x` and returns a multidimensional cell array of adjacent submatrices of `x`. Each of the vector arguments `d1` through `dn` should sum to the respective dimension sizes of `x` such that, for `p = 1:n`,

```
size(x,p) == sum(dp)
```

The elements of `d1` through `dn` determine the size of each cell in `c` by satisfying the following formula for `ip = 1:length(dp)`:

```
size(c{i1,i2,i3,...,in}) == [d1(i1) d2(i2) d3(i3) ... dn(in)]
```

If `x` is an empty array, `mat2cell` returns an empty cell array. This requires that all `dn` inputs that correspond to the zero dimensions of `x` be equal to `[]`.

For example,

```
a = rand(3,0,4);
c = mat2cell(a, [1 2], [], [2 1 1]);
```

`c = mat2cell(x,r)` divides an array `x` by returning a single-column cell array containing full rows of `x`. The sum of the element values in vector `r` must equal the number of rows of `x`.

The elements of `r` determine the size of each cell in `c`, subject to the following formula for `i = 1:length(r)`:

```
size(c{i},1) == r(i)
```

Remarks

`mat2cell` supports all array types.

Examples

Divide matrix `X` into 2-by-3 and 2-by-2 matrices contained in a cell array:

```
X = [1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15; 16 17 18 19 20]
X =
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20

C = mat2cell(X, [2 2], [3 2])
C =
    [2x3 double]    [2x2 double]
    [2x3 double]    [2x2 double]

C{1,1}                          C{1,2}
ans =                           ans =
     1     2     3                   4     5
     6     7     8                   9    10

C{2,1}                          C{2,2}
ans =                           ans =
    11    12    13                  14    15
    16    17    18                  19    20
```

See Also

`cell2mat`, `num2cell`

Purpose

Convert a matrix into a string

Syntax

```
str = mat2str(A)
str = mat2str(A, n)
str = mat2str(A, 'class')
str = mat2str(A, n, 'class')
```

Description

`str = mat2str(A)` converts matrix `A` into a string, suitable for input to the `eval` function, using full precision.

`str = mat2str(A,n)` converts matrix `A` using `n` digits of precision.

`str = mat2str(A, 'class')` creates a string with the name of the class of `A` included. This option ensures that the result of evaluating `str` will also contain the class information.

`str = mat2str(A, n, 'class')` uses `n` digits of precision and includes the class information.

Limitations

The `mat2str` function is intended to operate on scalar, vector, or rectangular array inputs only. An error will result if `A` is a multidimensional array.

Examples

Example 1

Consider the matrix

```
x = [3.85 2.91; 7.74 8.99]
x =
    3.8500    2.9100
    7.7400    8.9900
```

The statement

```
A = mat2str(x)
```

produces

```
A =
   [3.85 2.91;7.74 8.99]
```

where `A` is a string of 21 characters, including the square brackets, spaces, and a semicolon.

`eval(mat2str(x))` reproduces `x`.

Example 2

Create a 1-by-6 matrix of signed 16-bit integers, and then use `mat2str` to convert the matrix to a 1-by-33 character array, `A`. Note that output string `A` includes the class name, `int16`:

```
x1 = int16([-300 407 213 418 32 -125]);

A = mat2str(x1, 'class')
A =
   int16([-300 407 213 418 32 -125])

class(A)
ans =
   char
```

Evaluating the string `A` gives you an output `x2` that is the same as the original `int16` matrix:

```
x2 = eval(A);

if isnumeric(x2) && isa(x2, 'int16') && all(x2 == x1)
   disp 'Conversion back to int16 worked'
end

Conversion back to int16 worked
```

See Also

`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 to the diffuse and ambient light, and the color of the specular light depends only on the color of the light source.

`material dull` sets the reflectance properties so that the object reflects more diffuse light and has no specular highlights, but the color of the reflected light depends only on the light source.

`material metal` sets the reflectance properties so that the object has a very high specular reflectance, very low ambient and diffuse reflectance, and the color of the reflected light depends on both the color of the light source and the color of the object.

`material([ka kd ks])` sets the ambient/diffuse/specular strength of the objects.

`material([ka kd ks n])` sets the ambient/diffuse/specular strength and specular exponent of the objects.

`material([ka kd ks n sc])` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects.

`material default` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects to their defaults.

Remarks The `material` command sets the `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, and `SpecularColorReflectance` properties of all surface and patch objects in the axes. There must be visible light objects in the axes for lighting to be enabled. Look at the `materal.m` M-file to see the actual values set (enter the command `type material`).

See Also

`light`, `lighting`, `patch`, `surface`

Lighting as a Visualization Tool for more information on lighting

"Lighting" for related functions

Purpose

Start MATLAB (UNIX systems only)

Syntax

```
matlab helpOption
matlab archOption
matlab dispOption
matlab modeOption
matlab mgrOption
matlab -c licensefile
matlab -r MATLAB_command
matlab -logfile filename
matlab -mwvisual visualid
matlab -nosplash
matlab -timing
matlab -debug
matlab -Ddebugger options
```

Note You can enter more than one of these options in the same MATLAB command. If you use **`-D`**`debugger` to start MATLAB in debug mode, the first option in the command must be **`-D`**`debugger`.

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). Before actually initiating the execution of MATLAB, this script configures the runtime environment by

- Determining the MATLAB root directory
- Determining the host machine architecture
- Processing any command line options
- Reading the MATLAB startup file, `.matlab7rc.sh`
- Setting MATLAB environment variables

There are two ways in which you can control the way the `matlab` script works:

- By specifying command line options
- By assigning values in the MATLAB startup file, `.matlab7rc.sh`

Specifying Options at the Command Line

Options that you can enter at the command line are as follows:

`matlab helpOption` displays information that matches the specified `helpOption` argument without starting MATLAB. `helpOption` can be any one of the keywords shown in the table below. Enter only one `helpOption` keyword in a `matlab` command.

Values for helpOption

Option	Description
`-help`	Display `matlab` command usage.
`-h`	The same as **`-help`**.
`-n`	Display all the final values of the environment variables and arguments passed to the MATLAB executable as well as other diagnostic information.
`-e`	Display *all* environment variables and their values just prior to exiting. This argument must have been parsed before exiting for anything to be displayed. The last possible exiting point is just before the MATLAB image would have been executed and a status of `0` is returned. If the exit status is not `0` on return, then the variables and values may not be correct.

`matlab archOption` starts MATLAB and assumes that you are running on the system architecture specified by `arch`, or using the MATLAB version specified by `variant`, or both. The values for the `archOption` argument are shown in the table below. Enter only one of these options in a `matlab` command.

Values for archOption

Option	Description
`-arch`	Run MATLAB assuming this architecture rather than the actual architecture of the machine you are using. Replace the term `arch` with a string representing a recognized system architecture.
`v=``variant`	Execute the version of MATLAB found in the directory `bin/$ARCH/variant` instead of `bin/$ARCH`. Replace the term `variant` with a string representing a MATLAB version.
`v=``arch/variant`	Execute the version of MATLAB found in the directory `bin/arch/variant` instead of `bin/$ARCH`. Replace the terms `arch` and `variant` with strings representing a specific architecture and MATLAB version.

`matlab dispOption` starts MATLAB using one of the display options shown in the table below. Enter only one of these options in a `matlab` command.

Values for dispOption

Option	Description
`-display` `xDisp`	Send X commands to X Window Server display `xDisp`. This supersedes the value of the `DISPLAY` environment variable.
`-nodisplay`	Start the Java virtual machine (unless the `-nojvm` option is also specified), but do not start the MATLAB desktop. Do not display any X commands, and ignore the `DISPLAY` environment variable,

`matlab modeOption` starts MATLAB without its desktop or Java virtual machine components. Enter only one of the options shown below.

Values for modeOption

Option	Descripton
-nodesktop	Do not start the MATLAB desktop. Use the current window for commands. The Java virtual machine will be started.
-nojvm	Shut off all Java support by not starting the Java virtual machine. In particular, the MATLAB desktop will not be started.

`matlab mgrOption` starts MATLAB in the memory management mode specified by `mgrOption`. Enter only one of the options shown below.

Values for mgrOption

Option	Description
-memmgr `manager`	Set environment variable `MATLAB_MEM_MGR` to `manager`. The `manager` argument can have one of the following values: • **cache** — The default. • **compact** — This is useful for large models or MATLAB code that uses many structure or object variables. It is not helpful for large arrays. (This option applies only to 32-bit architectures.) • **debug** — Does memory integrity checking and is useful for debugging memory problems caused by user-created MEX files.
-check_malloc	The same as using '**-memmgr debug**'.

`matlab -c licensefile` starts MATLAB using the specified license file. The `licensefile` argument can have the form `port@host` or it can be a colon separated list of license filenames. This option causes the `LM_LICENSE_FILE` and `MLM_LICENSE_FILE` environment variables to be ignored.

`matlab` **`-r`** `command` starts MATLAB and executes the specified MATLAB `command`.

`matlab` **`-logfile`** `filename` starts MATLAB and makes a copy of any output to the command window in file log. This includes all crash reports.

`matlab` **`-mwvisual`** `visualid` starts MATLAB and uses `visualid` as the default X visual for figure windows. `visualid` is a hexadecimal number that can be found using `xdpyinfo`.

`matlab` **`-nosplash`** starts MATLAB but does not display the splash screen during startup.

`matlab` **`-timing`** starts MATLAB and prints a summary of startup time to the command window. This information is also recorded in a timing log, the name of which is printed to the shell window in which MATLAB is started. This option should be used only when working with a Technical Support Representative from The MathWorks, Inc. (This option applies to `glnx86` systems only.)

`matlab` **`-debug`** starts MATLAB and displays debugging information that can be useful, especially for X based problems. This option should be used only when working with a Technical Support Representative from The MathWorks, Inc.

`matlab` **`-D`**`debugger options` starts MATLAB in debug mode, using the named `debugger` (e.g., `dbx`, `gdb`, `dde`, `xdb`, `cvd`). A full path can be specified for debugger.

The `options` argument can include *only* those options that follow the debugger name in the syntax of the actual debug command. For most debuggers, there is a very limited number of such options. Options that would normally be passed to the MATLAB executable should be used as parameters of a command inside the debugger (like `run`). They should not be used when running the MATLAB script.

If any other `matlab` command options are placed before the **-D**`debugger` argument, they will be handled as if they were part of the options after the **-D**`debugger` argument and will be treated as illegal options by most debuggers. The `MATLAB_DEBUG` environment variable is set to the filename part of the debugger argument.

To customize your debugging session, use a startup file. See your debugger documentation for details.

Note For certain debuggers like `gdb`, the `SHELL` environment variable is *always* set to `/bin/sh`.

Specifying Options in the MATLAB Startup File

The `.matlab7rc.sh` shell script contains definitions for a number of variables that the `matlab` script uses. These variables are defined within the `matlab` script, but can be redefined in `.matlab7rc.sh`. When invoked, `matlab` looks for the first occurrence of `.matlab7rc.sh` in the current directory, in the home directory (`$HOME`), and in the `$MATLAB/bin` directory, where the template version of `.matlab7rc.sh` is located.

You can edit the template file to redefine information used by the `matlab` script. If you do not want your changes applied systemwide, copy the edited version of the script to your current or home directory. Ensure that you edit the section that applies to your machine architecture.

The following table lists the variables defined in the `.matlab7rc.sh` file. See the comments in the `.matlab7rc.sh` file for more information about these variables.

Variable	Definition and Standard Assignment Behavior
`ARCH`	The machine architecture. The value `ARCH` passed with the `-arch` or `-arch/ext` argument to the script is tried first, then the value of the environment variable `MATLAB_ARCH` is tried next, and finally it is computed. The first one that gives a valid architecture is used.
`AUTOMOUNT_MAP`	Path prefix map for automounting. The value set in `.matlab7rc.sh` (initially by the installer) is used unless the value differs from that determined by the script, in which case the value in the environment is used.
`DISPLAY`	The hostname of the X Window display MATLAB uses for output. The value of `Xdisplay` passed with the `-display` argument to the script is used; otherwise, the value in the environment is used. `DISPLAY` is ignored by MATLAB if the `-nodisplay` argument is passed.

Variable	Definition and Standard Assignment Behavior (Continued)
LD_LIBRARY_PATH	Final Load library path. The name LD_LIBRARY_PATH is platform dependent. The final value is normally a colon-separated list of four sublists, each of which could be empty. The first sublist is defined in .matlab7rc.sh as LDPATH_PREFIX. The second sublist is computed in the script and includes directories inside the MATLAB root directory and relevant Java directories. The third sublist contains any nonempty value of LD_LIBRARY_PATH from the environment possibly augmented in .matlab7rc.sh. The final sublist is defined in .matlab7rc.sh as LDPATH_SUFFIX.
LM_LICENSE_FILE	The FLEX lm license variable. The license file value passed with the -c argument to the script is used; otherwise it is the value set in .matlab7rc.sh. In general, the final value is a colon-separated list of license files and/or port@host entries. The shipping .matlab7rc.sh file starts out the value by prepending LM_LICENSE_FILE in the environment to a default license.file. Later in the MATLAB script if the -c option is not used, the $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 the end of the current list.

Variable	Definition and Standard Assignment Behavior (Continued)
MATLAB	The MATLAB root directory. The default computed by the script is used unless MATLABdefault is reset in .matlab7rc.sh. Currently MATLABdefault is not reset in the shipping .matlab7rc.sh.
MATLAB_DEBUG	Normally set to the name of the debugger. The -Ddebugger argument passed to the script sets this variable. Otherwise, a nonempty value in the environment is used.
MATLAB_JAVA	The path to the root of the Java Runtime Environment. The default set in the script is used unless MATLAB_JAVA is already set. Any nonempty value from .matlab7rc.sh is used first, then any nonempty value from the environment. Currently there is no value set in the shipping .matlab67rc.sh, so that environment alone is used.
MATLAB_MEM_MGR	Turns on MATLAB memory integrity checking. The -check_malloc argument passed to the script sets this variable to 'debug'. Otherwise, a nonempty value set in .matlab7rc.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.

Variable	Definition and Standard Assignment Behavior (Continued)
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.
SHELL	The shell to use when the "!" or unix command is issued in MATLAB. This is taken from the environment unless SHELL is reset in .matlab7rc.sh. Currently SHELL is not reset in the shipping .matlab7rc.sh. If SHELL is empty or not defined, 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 .matlab7rc.sh. Currently TOOLBOX is not reset in the shipping .matlab7rc.sh.

Variable	Definition and Standard Assignment Behavior (Continued)
XAPPLRESDIR	The X application resource directory. A nonempty value in the environment is used first unless XAPPLRESDIR is reset in .matlab7rc.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 .matlab7rc.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 can involve deleting part of the pathname from the front of the MATLAB root path. The MATLAB variable is then used to locate all files within the MATLAB directory tree.

The matlab script determines the path of the MATLAB root directory by looking up the directory tree from the $MATLAB/bin directory (where the matlab script is located). The MATLAB variable is then used to locate all files within the MATLAB directory tree.

You can change the definition of MATLAB if, for example, you want to run a different version of MATLAB or if, for some reason, the path determined by the matlab script is not correct. (This can happen when certain types of automounting schemes are used by your system.)

AUTOMOUNT_MAP is used to modify the MATLAB root directory path. The pathname that is assigned to AUTOMOUNT_MAP is deleted from the front of the MATLAB root path. (It is unlikely that you will need to use this option.)

See Also

mex

Purpose

Start MATLAB (Windows systems only)

Syntax

```
matlab helpOption
matlab modeOption
matlab mgrOption
matlab -c licensefile
matlab -r MATLAB_command
matlab -logfile filename
matlab -nosplash
matlab -timing
matlab -noFigureWindows
matlab -automation
matlab -regserver
matlab -unregserver
```

Note You can enter more than one of these options in the same MATLAB command.

Description

`matlab` is a starter program (currently a DOS batch script) that starts the main MATLAB executable. (In this document, the term `matlab` refers to the starter program, and MATLAB refers to the main executable). Before actually initiating the execution of MATLAB, it configures the runtime environment by

- Determining the MATLAB root directory
- Determining the host machine architecture
- Selectively processing command line options with the rest passed to MATLAB.
- Setting certain MATLAB environment variables

There are two ways in which you can control the way the `matlab` starter program works:

- By specifying command line options
- By presetting environment variables before calling the program

Specifying Options at the Command Line

Options that you can enter at the command line are as follows:

`matlab helpOption` displays information that matches the specified `helpOption` argument without starting MATLAB. `helpOption` can be any one of the keywords shown in the table below. Enter only one `helpOption` keyword in a `matlab` command.

Values for helpOption

Option	Description
`-help`	Display `matlab` command usage.
`-h`	The same as **`-help`**.
`-?`	The same as **`-help`**.

`matlab modeOption` starts MATLAB without its desktop or Java virtual machine components. Enter only one of the options shown below.

Values for modeOption

Option	Descripton
`-nodesktop`	Do not start the MATLAB desktop. Use a V5 MATLAB command window for commands. The Java virtual machine will be started.
`-nojvm`	Shut off all Java support by not starting the Java virtual machine. In particular, the MATLAB desktop will not be started.

`matlab mgrOption` starts MATLAB in the memory management mode specified by `mgrOption`. Enter only one of the options shown below.

Values for mgrOption

Option	Description
-memmgr manager	Set environment variable `MATLAB_MEM_MGR` to `manager`. The `manager` argument can have one of the following values: • **cache** — The default. • **fast** — For large models or MATLAB code that uses many structure or object variables. It is not helpful for large arrays. • **debug** — Does memory integrity checking and is useful for debugging memory problems caused by user-created MEX files.
-check_malloc	The same as using '**-memmgr debug**'.

`matlab` **-c** `licensefile` starts MATLAB using the specified license file. The `licensefile` argument can have the form `port@host`. This option causes the `LM_LICENSE_FILE` and `MLM_LICENSE_FILE` environment variables to be ignored.

`matlab` **-r** `command` starts MATLAB and executes the specified MATLAB `command`. Any required M-file must be on the MATLAB path.

`matlab` **-logfile** `filename` starts MATLAB and makes a copy of any output to the command window in file log. This includes all crash reports.

`matlab` **-nosplash** starts MATLAB but does not display the splash screen during startup.

`matlab` **-timing** starts MATLAB and prints a summary of startup time to the command window. This information is also recorded in a timing log, the name of which is printed to the MATLAB command window. This option should be used only when working with a Technical Support Representative from The MathWorks, Inc.

`matlab -noFigureWindows` starts MATLAB but disables the display of any figure windows in MATLAB.

`matlab -automation` starts MATLAB as an automation server. The server window is minimized, and the MATLAB splash screen is not displayed on startup.

`matlab -regserver` registers MATLAB as a Component Object Model (COM) server.

`matlab -unregserver` removes all MATLAB COM server entries from the registry.

Presetting Environment Variables

You can set any of the following environment variables before starting MATLAB.

Variable Name	Description
`LM_LICENSE_FILE`	This is the FLEX lm license variable. The license file value passed with the **`-c`** argument to the script is used; otherwise it is the value set in the environment. The final value is a colon-separated list of license files and/or port@host entries.
`MATLAB`	This is the MATLAB root directory. It is used to determine the location of the MATLAB `bin` directory. If not defined in the environment, then the location of the script is used.
`MATLAB_MEM_MGR`	This determines the type of memory manager used by MATLAB. If not set in the environment, it is controlled by passing its value via the '**`-memmgr`**' option. If no value is predefined, then MATLAB uses '**`cache`**'.

See Also

`mex`

matlabcolon (matlab:)

Purpose Run specified function via hyperlink

Syntax

```
disp('<a href="matlab: stmnt_1; stmnt_n;">hyperlink_text</a>')
```

Description matlab: executes stmnt_1 through stmnt_n when you click (or press **Ctrl+Enter**) in hyperlink_text. This must be used with another function, such as disp, where disp creates and displays underlined and colored hyperlink_text in the Command Window. Use disp, error, fprintf, help or warning functions to display the hyperlink. The hyperlink_text is interpreted as HTML, so use HTML character entity references or ASCII values for special characters. Include the full hypertext string, from '<a href= to ' within a single line, that is, do not continue a long string on a new line.

Remarks The matlab: function behaves differently with diary, notebook, type, and similar functions than might be expected. For example, if you enter the following statement

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

the diary file, when viewed in a text editor, shows

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
<a href="matlab:magic(4)">Generate magic square</a>
```

If you view the output of diary in the Command Window, the Command Window interprets the <a href ...> statement and does display it as a hyperlink.

Examples

Single Function

The statement

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

displays

Generate magic square

in the Command Window. When you click the link Generate magic square, MATLAB runs magic(4).

Multiple Functions

You can include multiple functions in the statement, such as

```
disp('<a href="matlab: x=0:1:8;y=sin(x);plot(x,y)">Plot x,y</a>')
```

which displays

Plot x,y

in the Command Window. When you click the link, MATLAB runs

```
x = 0:1:8;
y = sin(x);
plot(x,y)
```

Clicking the Hyperlink Again

After running the statements in the hyperlink `Plot x,y` defined in the previous example, “Multiple Functions”, you can subsequently redefine `x` in the base workspace, for example, as

```
x = -2*pi:pi/16:2*pi;
```

If you then click the hyperlink, `Plot sin(x)`, it changes the current value of `x` back to

```
0:1:8
```

because the `matlab:` statement defines `x` in the base workspace. In the `matlab:` statement that displayed the hyperlink, `Plot x,y`, `x` was defined as `0:1:8`.

Presenting Options

Use multiple `matlab:` statements in an M-file to present options, such as

```
disp('<a href = "matlab:state = 0">Disable feature</a>')
disp('<a href = "matlab:state = 1">Enable feature</a>')
```

The Command Window displays

Disable feature
Enable feature

and depending on which link is clicked, will set `state` to `0` or `1`.

Special Characters

To create a string that includes a special character such as a greater than sign, `>`, you need to use the HTML character entity reference for the symbol, `>`. Otherwise, the symbol will be interpreted as ending of the `<a href = " ... "` element. For example, run

```
disp('<a href="matlab:str = ''Value &gt; 0''">Positive</a>')
```

and the Command Window displays

Positive

Instead of the HTML character entity reference, you can use the ASCII value for the symbol. For example, the greater than sign, `>`, is ASCII 62. The above example becomes

```
disp(...
'<a href="matlab:str=[''Value '' char(62) '' 0'']">Positive</a>')
```

Use these values for common special characters.

Character	HTML Character Entity Reference	ASCII Value
`>`	`>`	62
`<`	`<`	60
`&`	`&`	38
`"`	`"`	34

Links from M-File Help

For functions you create, you can include `matlab:` links within the M-file help, but you do not need to include a `disp` or similar statement because the help function already includes it for displaying hyperlinks. Use the links to display additional help in a browser when the user clicks them. The M-file, `soundspeed`, contains the following statements.

```
function c=soundspeed(s,t,p)

% Speed of sound in water, using
% <a href="matlab: web('http://www.zu.edu')">Wilson's formula</a>
% Where c is the speed of sound in water in m/s
```

etc.

Run `help soundspeed` and MATLAB displays the following in the Command Window.

```
>> help soundspeed
  Speed of sound in water, using
  Wilson's formula
  Where c is the speed of sound in water in m/s
```

When you click the link, `Wilson's formula`, MATLAB displays the HTML page `http://www.zu.edu` in the Web browser. Note that this URL is only an example and is invalid.

See Also

`disp`, `error`, `fprintf`, `input`, `run`, `warning`

More about HTML character entity references at `http://www.w3.org/`.

Purpose

MATLAB startup M-file for single-user systems or system administrators

Description

At startup time, MATLAB automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`. On multiuser or networked systems, `matlabrc.m` is reserved for use by the system manager. The file `matlabrc.m` invokes the file `startup.m` if it exists on the MATLAB 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

Turning Off the Figure Window Toolbar

If you do not want the toolbar to appear in the figure window, remove the comment marks from the following line in the `matlabrc.m` file, or create a similar line in your own `startup.m` file.

```
%  set(0,'defaultfiguretoolbar','none')
```

See Also

`matlabroot`, `quit`, `restoredefaultpath`, `startup`
"Startup Options"

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. In compiled M-code, it returns the path to the executable. Use `matlabroot` to create a path to MATLAB and toolbox directories that does not depend on a specific platform or MATLAB version.

`rd = matlabroot` returns the name of the directory in which the MATLAB software is installed and assigns it to `rd`.

Note The term `$matlabroot` represents the directory where MATLAB files are installed.

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.

See Also `fullfile`, `partialpath`, `path`

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 complex modulus (magnitude), computed with `max(abs(A))`, and ignores the phase angle, `angle(A)`. The `max` function ignores `NaN`s.

See Also

`isnan`, `mean`, `median`, `min`, `sort`

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`. For matrices, `mean(A,2)` is a column vector containing the mean value of each row. The default of `dim` is 1.

Examples

```
A = [1 2 3; 3 3 6; 4 6 8; 4 7 7];
mean(A)
ans =
    3.0000    4.5000    6.0000

mean(A,2)
ans =
    2.0000
    4.0000
    6.0000
    6.0000
```

See Also

`corrcoef`, `cov`, `max`, `median`, `min`, `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`

The Technical Support Guide to Memory Management at
`http://www.mathworks.com/support/tech-notes/1100/1106.shtml`.

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 thenumber of the selected menu item.

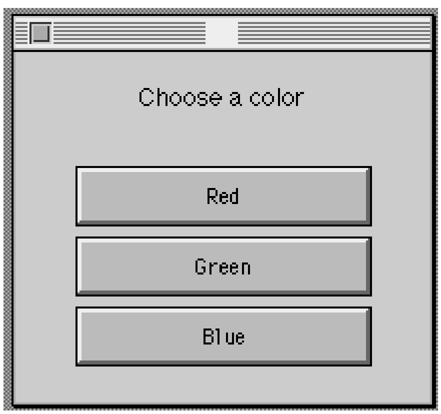
If the user's terminal provides a graphics capability, `menu` displays the menu items as push buttons in a figure window (Example 1), otherwise they will be given as a numbered list in the command window (Example 2).

Remarks To call `menu` from another ui object, set that object's `Interruptible` property to `'yes'`. For more information, see the MATLAB Graphics documentation.

Examples

Example 1

`k = menu('Choose a color','Red','Green','Blue')` displays



After input is accepted, use `k` to control the color of a graph.

```
color = ['r','g','b']
plot(t,s,color(k))
```

Example 2

```
K = menu('Choose a color','Red','Blue','Green')
```

displays on the Command Window

```
----- Choose a color -----
1) Red
2) Blue
3) Green
Select a menu number:
```

The number entered by the user in response to the prompt is returned as `K` (i.e. `K = 2` implies that the user selected `Blue`).

See Also

`guide`, `input`, `uicontrol`, `uimenu`

Purpose

Mesh plots

Syntax

```
mesh(X,Y,Z)
mesh(Z)
mesh(...,C)
mesh(...,'PropertyName',PropertyValue,...)
mesh(axes_handles,...)
meshc(...)
meshz(...)
h = mesh(...)
h = meshc(...)
h = meshz(...)
hsurface = mesh('v6'...), = meshc('v6'...), = meshz('v6'...)
```

Description

`mesh`, `meshc`, and `meshz` create wireframe parametric surfaces specified by `X`, `Y`, and `Z`, with color specified by `C`.

`mesh(X,Y,Z)` draws a wireframe mesh with color determined by `Z` so color is proportional to surface height. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, $(X(j), Y(i), Z(i,j))$ are the intersections of the wireframe grid lines; `X` and `Y` correspond to the columns and rows of `Z`, respectively. If `X` and `Y` are matrices, $(X(i,j), Y(i,j), Z(i,j))$ are the intersections of the wireframe grid lines.

`mesh(Z)` draws a wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`mesh(...,C)` draws a wireframe mesh with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`mesh(...,'PropertyName',PropertyValue,...)` sets the value of the specified surface property. Multiple property values can be set with a single statement.

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

meshc(...) draws a contour plot beneath the mesh.

meshz(...) draws a curtain plot (i.e., a reference plane) around the mesh.

h = mesh(...), h = meshc(...), and h = meshz(...) return a handle to a surfaceplot graphics object.

Backward Compatible Version

hsurface = mesh('v6',...) hsurface = meshc('v6',...), and hsurface = meshc('v6',...) returns the handles of surface objects instead of surfaceplot objects for compatibility with MATLAB 6.5 and earlier.

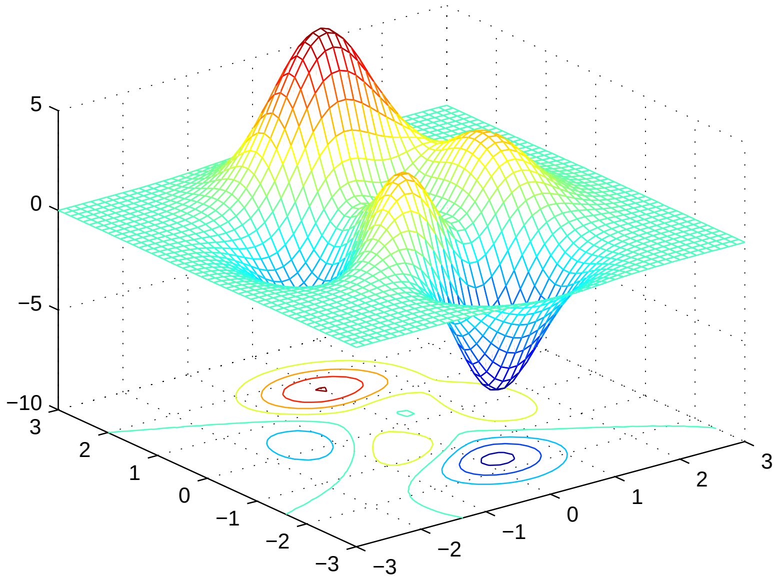
Remarks

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

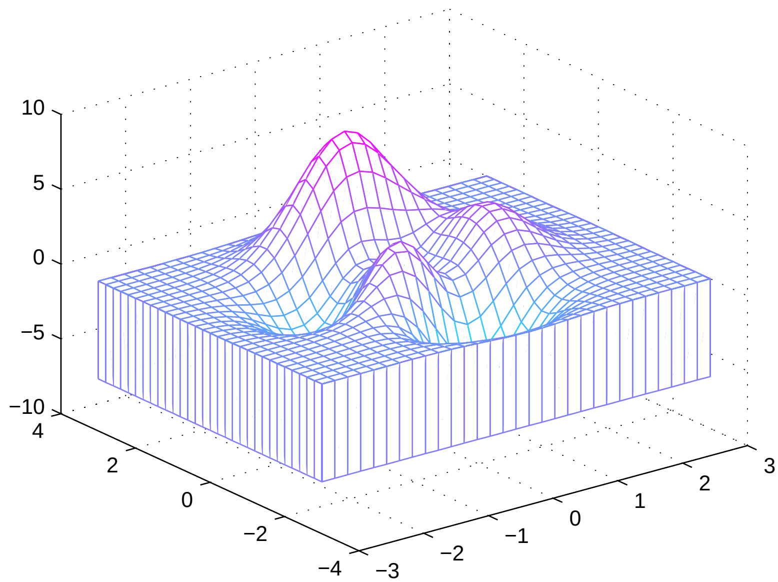
Produce a combination mesh and contour plot of the peaks surface:

```
[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 settings of the axes XLimMode, YLimMode, and ZLimMode properties determine the axis limits. axis sets these properties.

The range of C, or the current settings of the axes CLim and CLimMode properties (also set by the caxis function), determine the color scaling. The scaled color values are used as indices into the current colormap.

The mesh rendering functions produce color values by mapping the *z* data values (or an explicit color array) onto the current colormap. The MATLAB default behavior is to compute the color limits automatically using the minimum and maximum data values (also set using caxis auto). The minimum data value maps to the first color value in the colormap and the maximum data value maps to the last color value in the colormap. MATLAB performs a linear transformation on the intermediate values to map them to the current colormap.

meshc calls mesh, turns hold on, and then calls contour and positions the contour on the *x*-*y* plane. For additional control over the appearance of the contours, you can issue these commands directly. You can combine other types of graphs in this manner, for example surf and pcolor plots.

`meshc` assumes that `X` and `Y` are monotonically increasing. If `X` or `Y` is irregularly spaced, `contour3` calculates contours using a regularly spaced contour grid, then transforms the data to `X` or `Y`.

See Also

`contour`, `hidden`, `meshgrid`, `surface`, `surf`, `surfc`, `surfl`, `waterfall`

"Creating Surfaces and Meshes" for related functions

"Surfaceplot Properties" for a list of surfaceplot properties

The functions axis, `caxis`, `colormap`, `hold`, `shading`, and `view` all set graphics object properties that affect `mesh`, `meshc`, and `meshz`.

For a discussion of parametric surfaces plots, refer to `surf`.

Purpose

Generate `X` and `Y` 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

```
m = methods('classname')
m = methods('object')
m = methods(..., '-full')
```

Description

`m = methods('classname')` returns, in a cell array of strings, the names of all methods for the MATLAB, COM, or Java class `classname`.

`m = methods('object')` returns the names of all methods for the MATLAB, COM, or Java class of which `object` is an instance.

`m = methods(..., '-full')` returns the full description of the methods defined for the class, including inheritance information and, for COM and Java methods, attributes and signatures. For any overloaded method, the returned array includes a description of each of its signatures.

For MATLAB classes, inheritance information is returned only if that class has been instantiated.

Examples

List the methods of MATLAB class `stock`:

```
m = methods('stock')
m =
    'display'
    'get'
    'set'
    'stock'
    'subsasgn'
    'subsref'
```

Create a MathWorks sample COM control and list its methods:

```
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200]);
methods(h)

Methods for class com.mwsamp.mwsampctrl.1:

AboutBox          GetR8Array        SetR8             move
Beep              GetR8Vector       SetR8Array        propedit
FireClickEvent    GetVariantArray   SetR8Vector       release
```

```
GetBSTR          GetVariantVector  addproperty       save
GetBSTRArray     Redraw            delete            send
GetI4            SetBSTR           deleteproperty    set
GetI4Array       SetBSTRArray      events
GetI4Vector      SetI4             get
GetIDispatch     SetI4Array        invoke
GetR8            SetI4Vector       load
```

Display a full description of all methods on Java object `java.awt.Dimension`:

```
methods java.awt.Dimension -full

Dimension(java.awt.Dimension)
Dimension(int,int)
Dimension()
void wait() throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long,int) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
java.lang.Class getClass()  % Inherited from java.lang.Object
            .
            .
```

See Also

`methodsview`, `invoke`, `ismethod`, `help`, `what`, `which`

methodsview

Purpose Display information on all methods implemented by a class

Syntax

```
methodsview packagename.classname
methodsview classname
methodsview(object)
```

Description

`methodsview packagename.classname` displays information describing the Java class `classname` that is available from the package of Java classes `packagename`.

`methodsview classname` displays information describing the MATLAB, COM, or imported Java class `classname`.

`methodsview(object)` displays information describing the `object` instantiated from a COM or Java class.

MATLAB creates a new window in response to the `methodsview` command. This window displays all 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
```

MATLAB displays this information in a new window, as shown below

See Also `methods`, `import`, `class`, `javaArray`

Purpose Compile MEX-function from C or Fortran source code

Syntax `mex options filenames`

Description `mex options filenames` compiles a MEX-function from the C, C++, or Fortran source code files specified in `filenames`. All nonsource code `filenames` passed as arguments are passed to the linker without being compiled.

All valid `options` are shown in the MEX Script Switches table. These options are available on all platforms except where noted.

MEX's execution is affected both by command-line `options` and by an options file. The options file contains all compiler-specific information necessary to create a MEX-function. The default name for this options file, if none is specified with the `-f` option, is `mexopts.bat` (Windows) and `mexopts.sh` (UNIX).

Note The MathWorks provides an option, `setup`, for the `mex` script that lets you set up a default options file on your system.

On UNIX, the options file is written in the Bourne shell script language. The `mex` script searches for the first occurrence of the options file called `mexopts.sh` in the following list:

- The current directory
- The user profile directory (returned by the `prefdir` function)
- The directory specified by `[matlabroot '/bin']`

`mex` uses the first occurrence of the options file it finds. If no options file is found, `mex` displays an error message. You can directly specify the name of the options file using the `-f` switch.

Any variable specified in the options file can be overridden at the command line by use of the `<name>=<def>` command-line argument. If `<def>` has spaces in it, then it should be wrapped in single quotes (e.g., `OPTFLAGS='opt1 opt2'`). The definition can rely on other variables defined in the options file; in this case the variable referenced should have a prefixed `$` (e.g., `OPTFLAGS='$OPTFLAGS opt2'`).

On Windows, the options file is written in the Perl script language. The default options file is placed in your `user profile` directory after you configure your system by running `mex -setup`. The `mex` script searches for the first occurrence of the options file called `mexopts.bat` in the following list:

- The current directory
- The user profile directory (returned by the `prefdir` function)
- The directory specified by `[matlabroot '\bin\win32\mexopts']`

`mex` uses the first occurrence of the options file it finds. If no options file is found, `mex` searches your machine for a supported C compiler and uses the factory default options file for that compiler. If multiple compilers are found, you are prompted to select one.

No arguments can have an embedded equal sign (`=`); thus, `-DFOO` is valid, but `-DFOO=BAR` is not.

Remarks

`mex` compiles and links source files into a shared library called a MEX-file, executable from within MATLAB. The resulting file has a platform-dependent extension, as shown in the table below:

System Type	MEX File Extension
Sun Solaris	`.mexsol`
HP-UX	`.mexhpux`
Linux	`.mexglx`
MacIntosh	`.mexmac`
Windows	`.dll`

See Also

`dbmex`, `mexext`, `inmem`

Purpose Return the MEX-filename extension

Syntax

```
ext = mexext
```

Description `ext = mexext` returns the filename extension for the current platform.

Remarks The file built by the `mex` function has a platform-dependent extension, as shown in the table below:

System Type	MEX File Extension
Sun Solaris	.mexsol
HP-UX	.mexhpux
Linus	.mexglx
MacIntosh	.mexmac
Windows	.dll

Examples

```
ext = mexext

ext =
dll
```

See Also `mex`

mfilename

Purpose The name of the currently running M-file

Syntax

```
mfilename
p = mfilename('fullpath')
c = mfilename('class')
```

Description `mfilename` returns a string containing the name of the most recently invoked M-file. When called from within an M-file, it returns the name of that M-file, allowing an M-file to determine its name, even if the filename has been changed.

`p = mfilename('fullpath')` returns the full path and name of the M-file in which the call occurs, not including the filename extension.

`c = mfilename('class')` in a method, returns the class of the method, not including the leading `@` sign. If called from a nonmethod, it yields the empty string.

Remarks If `mfilename` is called with any argument other than the above two, it behaves as if it were called with no argument.

When called from the command line, `mfilename` returns an empty string.

To get the names of the callers of an M-file, use `dbstack` with an output argument.

See Also `dbstack`, `function`, `nargin`, `nargout`, `inputname`

Purpose Download file from FTP site

Syntax

```
mget(f,'filename')
mget(f,'dirname')
mget(f,'wildcard')
mget(...,'target')
```

Description

`mget(f,'filename')` retrieves `filename` from the FTP server `f` into the MATLAB current directory, where `f` was created using `ftp`.

`mget(f,'dirname')` retrieves the directory `dirname` and its contents from the FTP server `f` into the MATLAB current directory, where `f` was created using `ftp`. You can use a `wildcard` (`*`) in `dirname`.

`mget(...,'target')` retrieves the specified items from the FTP server `f`, where `f` was created using `ftp`, into the local directory specified by `target`, where `target` is an absolute pathname.

Examples

Connect to The MathWorks FTP server, change to the `pub/pentium` directory, and retrieve the file `Moler_1.txt` into the MATLAB current directory.

```
tmw=ftp('ftp.mathworks.com');
cd(tmw,'pub/pentium');
mget(tmw,'Moler_1.txt');
```

Then retrieve all files containing the term `Moler` into the directory `d:/myfiles`.

```
mget(tmw,'*Moler*','d:/myfiles');
```

See Also

`cd (ftp)`, `ftp`, `mput (ftp)`

min

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 largest complex modulus (magnitude), computed with `min(abs(A))`, and ignores the phase angle, `angle(A)`. The `min` function ignores `NaN`s.

See Also `max`, `mean`, `median`, `sort`

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. It should be large and sparse. 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. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

Purpose True if M-file or MEX-file cannot be cleared

Syntax

```
mislocked
mislocked(fun)
```

Description `mislocked` by itself returns logical `1` (`true`) if the currently running M-file or MEX-file is locked, and logical `0` (`false`) otherwise.

`mislocked(fun)` returns logical `1` (`true`) if the function named *`fun`* is locked in memory, and logical `0` (`false`) otherwise. Locked M-files and MEX-files cannot be removed with the `clear` function.

See Also `mlock`, `munlock`

mkdir

Purpose

Make new directory

Graphical Interface

As an alternative to the `mkdir` function, you can click the  icon in the Current Directory browser to add a directory.

Syntax

```
mkdir('dirname')
mkdir('parentdir','dirname')
[status,message,messageid] = mkdir(...,'dirname')
```

Description

`mkdir('dirname')` creates the directory `dirname` in the current directory, if `dirname` represents a relative path. Otherwise, `dirname` represents an absolute path and `dirname` attempts to create the absolute directory `dirname` in the root of the current volume. An absolute path starts in any one of a Windows drive letter, a UNC path `'\\'` string or a UNIX `'/'` character.

`mkdir('parentdir','dirname')` creates the directory `dirname` in the existing directory `parentdir`, where `parentdir` is an absolute or relative pathname.

`[status,message,messageid] = mkdir(...,'dirname')` creates the directory `dirname` in the existing directory `parentdir`, returning the status, a message, and the MATLAB error message ID (see `error` and `lasterr`). Here, `status` is `1` for success and is `0` for error. Only one output argument is required.

Examples

Create a Subdirectory in Current Directory

To create a subdirectory in the current directory called `newdir`, type

```
mkdir('newdir')
```

Create a Subdirectory in Specified Parent Directory

To create a subdirectory called `newdir` in the directory `testdata`, which is at the same level as the current directory, type

```
mkdir('../testdata','newdir')
```

Return Status When Creating Directory

In this example, the first attempt to create `newdir` succeeds, returning a status of `1`, and no error or warning message or message identifier:

```
[s, mess, messid] = mkdir('../testdata', 'newdir')
```

```
s =
   1
mess =
   ''
messid =
   ''
```

If you attempt to create the same directory again, `mkdir` again returns a success status, and also a warning and message identifier informing you that the directory already existed:

```
[s,mess,messid] = mkdir('../testdata','newdir')
s =
   1
mess =
   Directory "newdir" already exists.
messid =
   MATLAB:MKDIR:DirectoryExists
```

See Also

`copyfile`, `cd`, `dir`, `fileattrib`, `filebrowser`, `fileparts`, `ls`, `mfilename`, `movefile`, `rmdir`

mkdir (ftp)

Purpose Create new directory on FTP server

Syntax

```
mkdir(f,'dirname')
```

Description `mkdir(f,'dirname')` creates the directory `dirname` in the current directory of the FTP server `f`, where `f` was created using `ftp`, and where `dirname` is a pathname relative to the current directory on `f`.

Examples Connect to server `testsite`, view the contents, and create the directory `newdir` in the directory `testdir`.

```
test=ftp('ftp.testsite.com')
dir(test)
.               ..       otherfile.m          testdir
mkdir(test,'testdir/newdir');
dir(test,'testdir)
.               ..       newdir
```

See Also `dir (ftp)`, `ftp`, `rmdir (ftp)`

Purpose

Make a piecewise polynomial

Syntax

```
pp = mkpp(breaks,coefs)
pp = mkpp(breaks,coefs,d)
```

Description

`pp = mkpp(breaks,coefs)` builds a piecewise polynomial `pp` from its breaks and coefficients. `breaks` is a vector of length `L+1` with strictly increasing elements which represent the start and end of each of `L` intervals. `coefs` is an `L`-by-`k` matrix with each row `coefs(i,:)` containing the coefficients of the terms, from highest to lowest exponent, of the order `k` polynomial on the interval `[breaks(i),breaks(i+1)]`.

`pp = mkpp(breaks,coefs,d)` indicates that the piecewise polynomial `pp` is `d`-vector valued, i.e., the value of each of its coefficients is a vector of length `d`. `breaks` is an increasing vector of length `L+1`. `coefs` is a `d`-by-`L`-by-`k` array with `coefs(r,i,:)` containing the `k` coefficients of the `i`th polynomial piece of the `r`th component of the piecewise polynomial.

Use `ppval` to evaluate the piecwise polynomial at specific points. Use `unmkpp` to extract details of the piecewise polynomial.

Note. The *order* of a polynomial tells you the number of coefficients used in its description. A kth order polynomial has the form

$$c_1 x^{k-1} + c_2 x^{k-2} + \ldots + c_{k-1} x + c_k$$

It has k coefficients, some of which can be 0, and maximum exponent k-1. So the order of a polynomial is usually one greater than its degree. For example, a cubic polynomial is of order 4.

Examples

The first plot shows the quadratic polynomial

$$1 - \left(\frac{x}{2} - 1\right)^2 = \frac{-x^2}{4} + x$$

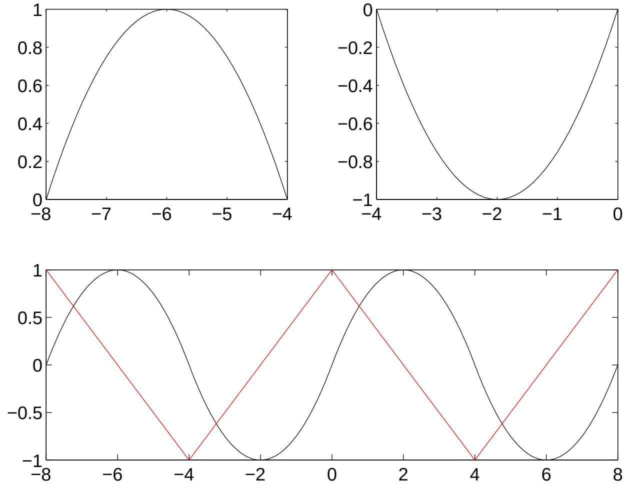
shifted to the interval [-8,-4]. The second plot shows its negative

$$\left(\frac{x}{2} - 1\right)^2 - 1 = \frac{x^2}{4} - x$$

but shifted to the interval [-4,0].

The last plot shows a piecewise polynomial constructed by alternating these two quadratic pieces over four intervals. It also shows its first derivative, which was constructed after breaking the piecewise polynomial apart using unmkpp.

```
subplot(2,2,1)
cc = [-1/4 1 0];
pp1 = mkpp([-8 -4],cc);
xx1 = -8:0.1:-4;
plot(xx1,ppval(pp1,xx1),'k-')

subplot(2,2,2)
pp2 = mkpp([-4 0],-cc);
xx2 = -4:0.1:0;
plot(xx2,ppval(pp2,xx2),'k-')

subplot(2,1,2)
pp = mkpp([-8 -4 0 4 8],[cc;-cc;cc;-cc]);
xx = -8:0.1:8;
plot(xx,ppval(pp,xx),'k-')
[breaks,coefs,l,k,d] = unmkpp(pp);
dpp = mkpp(breaks,repmat(k-1:-1:1,d*l,1).*coefs(:,1:k-1),d);
hold on, plot(xx,ppval(dpp,xx),'r-'), hold off
```



See Also `ppval`, `spline`, `unmkpp`

mldivide \, mrdivide /

Purpose Left or right matrix division

Syntax

```
mldivide(A,B)     A\B
mrdivide(B,A)     B/A
```

Description

mldivide(A,B) and the equivalent A\B perform matrix left division (back slash). A and B must be matrices that have the same number of rows, unless A is a scalar, in which case A\B performs element-wise division — that is, A\B = A.\B.

If A is a square matrix, A\B is roughly the same as inv(A)*B, except it is computed in a different way. If A is an n-by-n matrix and B is a column vector with n elements, or a matrix with several such columns, then X = A\B is the solution to the equation $AX = B$ computed by Gaussian elimination with partial pivoting (see “Algorithm” on page 2-1468 for details). A warning message is displayed if A is badly scaled or nearly singular.

If A is an m-by-n matrix with m ~= n and B is a column vector with m components, or a matrix with several such columns, then X = A\B is the solution in the least squares sense to the under- or overdetermined system of equations $AX = B$. In other words, X minimizes norm(A*X - B), the length of the vector $AX - B$. The rank k of A is determined from the QR decomposition with column pivoting (see “Algorithm” on page 2-1468 for details). The computed solution X has at most k nonzero elements per column. If k < n, this is usually not the same solution as x = pinv(A)*B, which returns a least squares solution.

mrdivide(B,A) and the equivalent B/A perform matrix right division (forward slash). B and A must have the same number of columns.

If A is a square matrix, B/A is roughly the same as B*inv(A). If A is an n-by-n matrix and B is a row vector with n elements, or a matrix with several such rows, then X = B/A is the solution to the equation $XA = B$ computed by Gaussian elimination with partial pivoting. A warning message is displayed if A is badly scaled or nearly singular.

If B is an m-by-n matrix with m ~= n and A is a column vector with m components, or a matrix with several such columns, then X = B/A is the solution in the least squares sense to the under- or overdetermined system of equations $XA = B$.

Note Matrix right division and matrix left division are related by the equation `B/A = (A'\B')'`.

Least Squares Solutions

If the equation $Ax = b$ does not have a solution (and A is not a square matrix), `x = A\b` returns a *least squares solution* — in other words, a solution that minimizes the length of the vector $Ax - b$, which is equal to `norm(A*x - b)`. See "Example 3" on page 2-1467 for an example of this.

Examples

Example 1

Suppose that `A` and `b` are the following.

```
A = magic(3)

A =

     8     1     6
     3     5     7
     4     9     2

b = [1;2;3]

b =

     1
     2
     3
```

To solve the matrix equation $Ax = b$, enter

```
x=A\b

x =

    0.0500
    0.3000
    0.0500
```

You can verify that x is the solution to the equation as follows.

```
A*x

ans =

    1.0000
    2.0000
    3.0000
```

Example 2 — A Singular

If A is singular, A\b returns the following warning.

```
Warning: Matrix is singular to working precision.
```

In this case, $Ax = b$ might not have a solution. For example,

```
A = magic(5);
A(:,1) = zeros(1,5); % Set column 1 of A to zeros
b = [1;2;5;7;7];
x = A\b
Warning: Matrix is singular to working precision.

ans =

   NaN
   NaN
   NaN
   NaN
   NaN
```

If you get this warning, you can still attempt to solve $Ax = b$ using the pseudoinverse function pinv.

```
x = pinv(A)*b

x =

         0
    0.0209
    0.2717
    0.0808
```

```
    -0.0321
```

The result x is least squares solution to $Ax = b$. To determine whether x is a exact solution — that is, a solution for which $Ax - b = 0$ — simply compute

```
A*x-b

ans =

    -0.0603
     0.6246
    -0.4320
     0.0141
     0.0415
```

The answer is not the zero vector, so x is not an exact solution.

"Pseudoinverses," in the online MATLAB documentation, provides more examples of solving linear systems using pinv.

Example 3

Suppose that

```
A = [1 0 0;1 0 0];
b = [1; 2];
```

Note that $Ax = b$ cannot have a solution, because A*x has equal entries for any x. Entering

```
x = A\b
```

returns the least squares solution

```
x =

     1.5000
          0
          0
```

along with a warning that A is rank deficient. Note that x is not an exact solution:

```
A*x-b
```

```
ans =

   0.5000
  -0.5000
```

Data Type Support

When computing `X = A\B` or `X = A/B`, the matrices `A` and `B` can have data type `double` or `single`. The following rules determine the data type of the result:

- If both `A` and `B` have type `double`, `X` has type `double`.
- If either `A` or `B` has type single, `X` has type `single`.

Algorithm

The specific algorithm used for solving the simultaneous linear equations denoted by `X = A\B` and `X = B/A` depends upon the structure of the coefficient matrix `A`. To determine the structure of `A` and select the appropriate algorithm, MATLAB follows this precedence:

1. **If A is sparse and diagonal**, `X` is computed by dividing by the diagonal elements of `A`.
2. **If A is sparse, square, and banded**, then banded solvers are used. Band density is (# nonzeros in the band)/(# nonzeros in a full band). Band density = `1.0` if there are no zeros on any of the three diagonals.
   - If `A` is real and tridiagonal, i.e., band density = `1.0`, and `B` is real with only one column, `X` is computed quickly using Gaussian elimination without pivoting.
   - If the tridiagonal solver detects a need for pivoting, or if `A` or `B` is not real, or if `B` has more than one column, but `A` is banded with band density greater than the `spparms` parameter `'bandden'` (default = `0.5`), then `X` is computed using the Linear Algebra Package (LAPACK) routines in the following table.

	Real	**Complex**
`A` and `B` `double`	`DGBTRF, DGBTRS`	`ZGBTRF, ZGBTRS`
`A` or `B` `single`	`SGBTRF, SGBTRS`	`CGBTRF, CGBTRS`

3. **If A is an upper or lower triangular matrix**, then `X` is computed quickly with a backsubstitution algorithm for upper triangular matrices, or a

forward substitution algorithm for lower triangular matrices. The check for triangularity is done for full matrices by testing for zero elements and for sparse matrices by accessing the sparse data structure.

If A is a full matrix, computations are performed using the Basic Linear Algebra Subprograms (BLAS) routines in the following table.

	Real	Complex
`A` and `B` `double`	`DTRSV`, `DTRSM`	`ZTRSV`, `ZTRSM`
`A` or `B` `single`	`STRSV`, `STRSM`	`CTRSV`, `CTRSM`

4. **If A is a permutation of a triangular matrix**, then `X` is computed with a permuted backsubstitution algorithm.
5. **If A is symmetric, or Hermitian, and has real positive diagonal elements**, then a Cholesky factorization is attempted (see `chol`). If `A` is found to be positive definite, the Cholesky factorization attempt is successful and requires less than half the time of a general factorization. Nonpositive definite matrices are usually detected almost immediately, so this check also requires little time.

   If successful, the Cholesky factorization for full `A` is

   ```
   A = R'*R
   ```

   where `R` is upper triangular. The solution `X` is computed by solving two triangular systems,

```
X = R\(R'\B)
```

Computations are performed using the LAPACK routines in the following table.

	Real	**Complex**
A and B double	DLANGE, DPOTRF, DPOTRS, DPOCON	ZLANGE, ZPOTRF, ZPOTRS, ZPOCON
A or B single	SLANGE, SPOTRF, SPOTRS, SDPOCON	CLANGE, CPOTRF, CPOTRS, CPOCON

If A is sparse, a symmetric minimum degree preordering is applied first (see symmmd and spparms) before X is computed. The algorithm is

```
perm = symmmd(A);          % Symmetric approximate minimum
                           % degree reordering
R = chol(A(perm,perm));    % Cholesky factorization
Y = R'\B(perm);            % Lower triangular solve
X(perm,:) = R\Y;           % Upper triangular solve
```

6. **If A is Hessenberg**, but not sparse, it is reduced to an upper triangular matrix and that system is solved via substitution.
7. **If A is square** and does not satisfy criteria 1 through 5, then a general triangular factorization is computed by Gaussian elimination with partial pivoting (see lu). This results in

   ```
   A = L*U
   ```

   where L is a permutation of a lower triangular matrix and U is an upper triangular matrix. Then X is computed by solving two permuted triangular systems.

```
X = U\(L\B)
```

If `A` is not sparse, computations are performed using the LAPACK routines in the following table.

	Real	Complex
`A` and `B double`	`DLANGE, DGESV, DGECON`	`ZLANGE, ZGESV, ZGECON`
`A` or `B single`	`SLANGE, SGESV, SGECON`	`CLANGE, CGESV, CGECON`

If `A` is sparse, then UMFPACK is used to compute `X`. The computations result in

```
P*A*Q = L*U
```

where `P` is a row permutation matrix and `Q` is a column reordering matrix. Then `X = Q*(U\L\(P*B))`.

8 **If A is not square**, then Householder reflections are used to compute an orthogonal-triangular factorization.

```
A*P = Q*R
```

where `P` is a permutation, `Q` is orthogonal and `R` is upper triangular (see `qr`). The least squares solution `X` is computed with

```
X = P*(R\(Q'*B))
```

If `A` is sparse, MATLAB computes a least squares solution using the sparse `qr` factorization of `A`.

If `A` is full, MATLAB uses the LAPACK routines listed in the following table to compute these matrix factorizations.

	Real	Complex
`A` and `B double`	`DGEQP3, DORMQR, DTRTRS`	`ZGEQP3, ZORMQR, ZTRTRS`
`A` or `B single`	`SGEQP3, SORMQR, STRTRS`	`CGEQP3, CORMQR, CTRTRS`

Note To see information about choice of algorithm and storage allocation for sparse matrices, , set the `spparms` parameter `'spumoni' = 1`.

Note `mldivide` and `mrdivide` are not implemented for sparse matrices `A` that are complex but not square.

See Also

Arithmetic operators, `linsolve`, `ldivide`, `rdivide`

Purpose

Check M-files for possible problems, and report results

Graphical Interface

In the Current Directory browser, select the M-Lint Code Check Report from the list of Directory Reports presented on the toolbar.

Syntax

```
mlint(filename)
info=mlint(filename,'-struct')
msg=mlint(filename,'-string')
[info,filepaths]=mlint(filename)
info=mlint(filename,'-id')
info=mlint(filename,'-fullpath')
```

Description

`mlint(filename)` displays M-Lint information about `filename`. If `filename` is a cell array, information is displayed for each file. `mlint(F1,F2,F3,...)`, where each input is a character array, displays information about each input filename. You cannot combine cell arrays and character arrays of filenames.

`info=mlint(filename,'-struct')` returns the M-Lint information in a structure array whose length is the number of suspicious constructs found. The structure has the following fields:

Field	Description
`line`	vector of line numbers to which the message refers
`column`	two-column array of column extents for each line
`message`	message describing the suspect that M-Lint caught

If multiple filenames are input, or if a cell array is input, `info` will contain a cell array of structures.

`msg=mlint(filename,'-string')` returns the M-Lint information as a string to the variable `msg`. If multiple filenames are input, or if a cell array is input, `msg` will contain a string where each file's information is separated by ten "=" characters, a space, the filename, a space, and ten "=" characters.

If the **`-struct`** or **`-string`** argument is omitted and an output argument is specified, the default behavior is **`-struct`**. If the argument is omitted and there are no output arguments, the default behavior is to display the information to the command line.

`[info,filepaths]=mlint(filename)` will additionally return `filepaths`, the absolute paths to the filenames in the same order as they were input.

`info=mlint(filename,'-id')` requests the message ID from M-Lint as well. When returned to a structure, the output will have the following additional field:

Field	Description
`id`	ID associated with the message

`info=mlint(filename,'-fullpath')` assumes that the input filenames are absolute paths, rather than requiring M-Lint to locate them.

To force M-Lint to ignore a line of code, add `%#ok` at the end of the line. This tag can be followed by comments. For example:

```
unsuppressed1 = 10   % This line will get caught
suppressed2 = 20    %#ok  These next two lines will not get caught
suppressed3 = 30     %#ok
```

Examples

`lengthofline.m` is an example M-file with suspicious M-Lint constructs. It is found in `$matlabroot/matlab/help/techdoc/matlab_env/examples`. To display to the command line, run

```
mlint lengthofline
```

To store to a struct with ID, run

```
info=mlint('lengthofline','-id')
```

See Also

`mlintrpt`

Purpose

Run mlint for file or directory, reporting results in Web browser

Graphical Interface

In the Current Directory browser, select the **M-Lint Code Check Report** button.

Syntax

```
mlintrpt
mlinkrpt(filename)
mlintrpt(dirname,'dir')
```

Description

`mlintrpt` scans all M-files in the current directory for M-Lint messages, and reports the results in a browser.

`mlintrpt(filename)` scans the M-file `filename` for messages as does the command `mlintrpt(filename,'file')` .

`mlintrpt(dirname,'dir')` scans the specified directory. Here, `dirname` can be in the current directory or can be a full pathname.

Examples

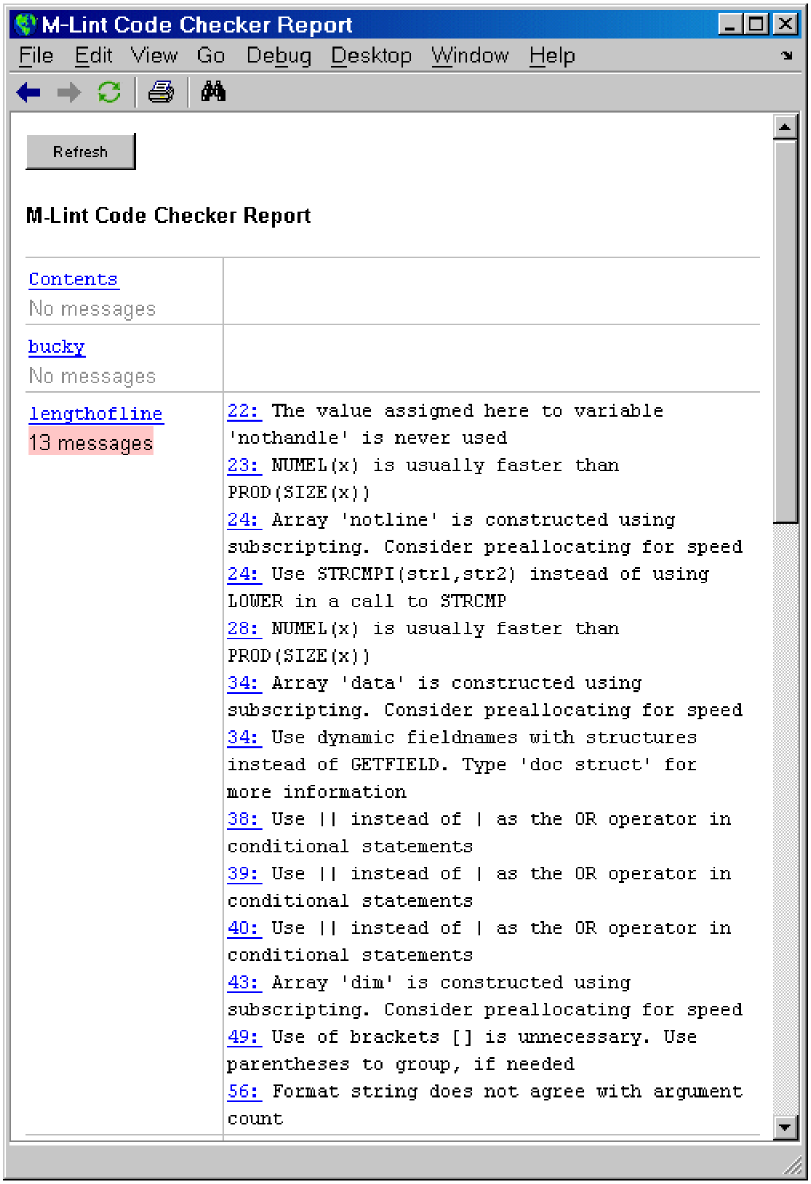
Run

```
mlintrpt('d:\MATLAB\work','dir')
```

See Also

`mlint`

and MATLAB displays a report of potential problems and improvements for all M-files in the `mydemos` directory.


M-Lint Code Checker Report
File Edit View Go Debug Desktop Window Help
Refresh
M-Lint Code Checker Report
Contents
No messages
bucky
No messages
lengthofline
13 messages
22: The value assigned here to variable 'nothandle' is never used
23: NUMEL(x) is usually faster than PROD(SIZE(x))
24: Array 'notline' is constructed using subscripting. Consider preallocating for speed
24: Use STRCMPI(str1,str2) instead of using LOWER in a call to STRCMP
28: NUMEL(x) is usually faster than PROD(SIZE(x))
34: Array 'data' is constructed using subscripting. Consider preallocating for speed
34: Use dynamic fieldnames with structures instead of GETFIELD. Type 'doc struct' for more information
38: Use || instead of | as the OR operator in conditional statements
39: Use || instead of | as the OR operator in conditional statements
40: Use || instead of | as the OR operator in conditional statements
43: Array 'dim' is constructed using subscripting. Consider preallocating for speed
49: Use of brackets [] is unnecessary. Use parentheses to group, if needed
56: Format string does not agree with argument count

For more information about using this report, see the M-Lint Graphical Interface documentation. (Although the `mlintrpt` results appear in the MATLAB Web browser and the M-Lint Graphical Interface uses the Current Directory browser, instructions for using the report are the same.)

See Also `mlint`

Purpose

Prevent M-file or MEX-file clearing

Syntax

`mlock`

Description

`mlock` locks the currently running M-file or MEX-file in memory so that subsequent `clear` functions do not remove it.

Use the `munlock` function to return the file to its normal, clearable state.

Locking an M-file or MEX-file in memory also prevents any `persistent` variables defined in the file from getting reinitialized.

Examples

The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
  .
  .
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked('testfun')
ans =
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock('testfun')

mislocked('testfun')
ans =
     0
```

See Also

`mislocked`, `munlock`, `persistent`

Purpose Information about a multimedia file

Syntax

```
info = mmfileinfo(filename)
```

Description

`info = mmfileinfo(filename)` returns a structure, `info`, whose fields contain information about the contents of the multimedia file identifed by the string `filename`.

Note `mmfileinfo` can be used only on Windows systems.

If `filename` is a URL, `mmfileinfo` might take a long time to return because it must first download the file. For large files, downloading can take several minutes. To avoid blocking the MATLAB command line while this processing takes place, download the file before calling `mmfileinfo`.

The `info` structure contains the following fields, listed in the order they appear in the structure.

Field	Description
`Filename`	String indicating the name of the file
`Duration`	Length of the file in seconds
`Audio`	Structure containing information about the audio data in the file. See “Audio Data” on page 2-1480 for more information about this data structure.
`Video`	Structure containing information about the video data in the file. See “Video Data” on page 2-1480 for more information about this data structure.

Audio Data

The `Audio` structure contains the following fields, listed in the order they appear in the structure. If the file does not contain audio data, the fields in the structure are empty.

Field	Description
`Format`	Text string, indicating the audio format
`NumberOfChannels`	Number of audio channels

Video Data

The `Video` structure contains the following fields, listed in the order they appear in the structure.

Field	Description
`Format`	Text string, indicating the video format
`Height`	Height of the video frame
`Width`	Width of the video frame

Examples

This example gets information about the contents of a file containing audio data.

```
info = mmfileinfo('my_audio_data.mp3')

info =

    Filename: 'my_audio_data.mp3'
    Duration: 1.6030e+002
       Audio: [1x1 struct]
       Video: [1x1 struct]
```

To look at the information returned about the audio data in the file, examine the fields in the `Audio` structure.

```
audio_data = info.Audio
```

```
audio_data =

              Format: 'MPEGLAYER3'
    NumberOfChannels: 2
```

Because the file contains only audio data, the fields in the `Video` structure are empty.

```
info.Video

ans =

    Format: ''
    Height: []
     Width: []
```

mod

Purpose Modulus after division

Syntax `M = mod(X,Y)`

Definition `mod(x,y)` is $x \bmod y$.

Description `M = mod(X,Y)` if `Y ~= 0`, returns `X - n.*Y` where `n = floor(X./Y)`. If `Y` is not an integer and the quotient `X./Y` is within roundoff error of an integer, then `n` is that integer. By convention, `mod(X,0)` is `X`. The inputs `X` and `Y` must be real arrays of the same size, or real scalars.

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
```

See Also `rem`

Purpose Display Command Window output one screenful at a time

Syntax

```
more on
more off
more(n)
```

Description `more on` enables paging of the output in the MATLAB Command Window. MATLAB displays output one screenful at a time.

`more off` disables paging of the output in the MATLAB Command Window.

`more(n)` displays `n` lines per page.

To see the status of `more`, type `get(0,'More')`. MATLAB returns either `on` or `off` indicating the `more` status. You can also set status for `more` by using `get(0,'More', 'status')`, where `'status'` is either `'on'` or `'off'`.

When you have enabled `more` and are examining output, you can do the following.

Press the...	To...
Return key	Advance to the next line of output.
Space bar	Advance to the next page of output.
Q (for quit) key	Terminate display of the text. Do not use **Ctrl+C** to terminate `more` or you might generate error messages in the Command Window.

By default, `more` is disabled. When enabled, `more` defaults to displaying 23 lines per page.

See Also `diary`

movefile

Purpose

Move file or directory

Graphical Interface

As an alternative to the `movefile` function, you can use the Current Directory browser to move files and directories.

Syntax

```
movefile('source')
movefile('source','destination')
movefile('source','destination','f')
[status,message,messageid] = movefile('source','destination','f')
```

Description

`movefile('source')` moves the file or directory named `source` to the current directory, where `source` is the absolute or relative pathname for the directory or file. Use the wildcard `*` at the end of `source` to move all matching files. Note that the archive attribute of `source` is not preserved.

`movefile('source','destination')` moves the file or directory named `source` to the location `destination`, where `source` and `destination` are the absolute or relative pathnames for the directory or files. To rename a file or directory when moving it, make `destination` a different name than `source`. Use the wildcard `*` at the end of `source` to move all matching files.

`movefile('source','destination','f')` moves the file or directory named `source` to the location `destination`, regardless of the read-only attribute of `destination`.

`[status,message,messageid]=movefile('source','destination','f')` moves the file or directory named `source` to the location `destination`, returning the status, a message, and the MATLAB error message ID (see `error` and `lasterr`). Here, `status` is `1` for success and is `0` for error. Only one output argument is required and the `f` input argument is optional.

The `*` wildcard in a path string is supported.

Examples

Move Source To Current Directory

To move the file `myfiles/myfunction.m` to the current directory, type

```
movefile('myfiles/myfunction.m')
```

If the current directory is projects/testcases and you want to move projects/myfiles and its contents to the current directory, use ../ in the source pathname to navigate up one level to get to the directory.

```
movefile('../myfiles')
```

Move All Matching Files By Using a Wildcard

To move all files in the directory myfiles whose names begin with my to the current directory, type

```
movefile('myfiles/my*')
```

Move Source to Destination

To move the file myfunction.m from the current directory to the directory projects, where projects and the current directory are at the same level, type

```
movefile('myfunction.m','../projects')
```

Move Directory Down One Level

This example moves the a directory down a level. For example to move the directory projects/testcases and all its contents down a level in projects to projects/myfiles, type

```
movefile('projects/testcases','projects/myfiles/')
```

The directory testcases and its contents now appear in the directory myfiles.

Rename When Moving File to Read-Only Directory

Move the file myfile.m from the current directory to d:/work/restricted, assigning it the name test1.m, where restricted is a read-only directory.

```
movefile('myfile.m','d:/work/restricted/test1.m','f')
```

The read-only file myfile.m is no longer in the current directory. The file test1.m is in d:/work/restricted and is read only.

Return Status When Moving Files

In this example, all files in the directory myfiles whose names start with new are to be moved to the current directory. However, if new* is accidentally written as nex*. As a result, the move is unsuccessful, as seen in the status and messages returned:

```
[s,mess,messid]=movefile('myfiles/nex*')

s =
     0

mess =

A duplicate filename exists, or the file cannot be found.

messid =

MATLAB:MOVEFILE:OSError
```

See Also cd, copyfile, delete, dir, fileattrib, filebrowser, ls, mkdir, rmdir

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 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('position')` move the callback figure (`gcbf`) or the current figure (`gcf`) to the specified position.

`movegui(h)` moves the figure identified by the handle `h` to the `onscreen` position.

`movegui` moves the callback figure (`gcbf`) or the current figure (`gcf`) to the `onscreen` position. This is useful as a string-based `CreateFcn` callback for a saved figure. It ensures the figure appears on screen when reloaded, regardless of its saved position.

Examples

This example demonstrates the usefulness of `movegui` to ensure that saved GUIs appear on screen when reloaded, regardless of the target computer's screen sizes and resolution. It creates a figure off the screen, assigns `movegui` as its `CreateFcn` callback, then saves and reloads the figure.

```
f = figure('Position',[10000,10000,400,300]);
set(f,'CreateFcn','movegui')
hgsave(f,'onscreenfig')
close(f)
f2 = hgload('onscreenfig');
```

See Also

`guide`

"Creating GUIs" in the MATLAB documentation

Purpose Play recorded movie frames

Syntax

```
movie(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 make up a list of frames to play in the movie.

For example, if `M` has four frames then `n = [10 4 4 2 1]` plays the movie ten times, and the movie consists of frame `4` followed by frame `4` again, followed by frame `2` and finally frame `1`.

`movie(M,n,fps)` plays the movie at `fps` frames per second. The default is 12 frames per second. Computers that cannot achieve the specified speed play as fast as possible.

`movie(h,...)` plays the movie centered in the figure or axes identified by the handle `h`.

`movie(h,M,n,fps,loc)` specifies a four-element location vector, [x y 0 0], where the lower left corner of the movie frame is anchored (only the first two elements in the vector are used). The location is relative to the lower left corner of the figure or axes specified by handle `h` and in units of pixels, regardless of the object's `Units` property.

Remarks The movie function 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

aviread, getframe, frame2im, im2frame

“Animation” for related functions

See Example – Visualizing an FFT as a Movie for another example

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 the compression codec to use.		

	On Windows:
	`'Indeo3'`
	`'Indeo5'`
	`'Cinepak'`
	`'MSVC'`
	`'RLE'`
	`'None'`
		On UNIX:
		`'None'`
			`'Indeo5'` 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.		
`'fps'`	A scalar value specifying the speed of the AVI movie in frames per second (fps).		15 fps

Parameter	Value	Default
`'keyframe'`	For compressors that support temporal compression, this is the number of key frames per second.	2 key frames per second.
`'quality'`	A number between 0 and 100 the specifies the desired quality of the output. Higher numbers result in higher video quality and larger file sizes. Lower numbers result in lower video quality and smaller file sizes. This parameter has no effect on uncompressed movies.	75
`'videoname'`	A descriptive name for the video stream. This parameter must be no greater than 64 characters long.	The default is the `filename`.

See Also

`avifile`, `aviread`, `aviinfo`, `movie`

Purpose

Upload file or directory to FTP server

Syntax

```
mput(f,'name')
mput(f,'wildcard')
```

Description

`mput(f,'filename')` uploads `name` from the MATLAB current directory to the current directory of the FTP server `f`, where `name` is a file or a directory and its contents, and where `f` was created using `ftp`. You can use a wildcard (`*`) in `filename`.

See Also

`ftp`, `methods`, `mkdir (ftp)`, `rename (ftp)`

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`
"Predefined Dialog Boxes" for related functions

Purpose Matrix multiplication

Syntax

```
C = A*B
```

Description

C = A*B is the linear algebraic product of the matrices A and B. The i,j entry of the product is defined by

$$C(i,j) = \sum_{k=1}^{p} A(i,k)B(k,j)$$

For nonscalar A and B, the number of columns of A must equal the number of rows of B. If A is m-by-p and B is p-by-n, the product C is m-by-n. You can multiply a scalar by a matrix of any size.

The preceding definition says that *C(i,j)* is the inner product of the *i*th row of *A* with the *j*th column of B. You can write this definition using the MATLAB colon operator as

```
C(i,j) = A(i,:)*B(:,j)
```

where A(i,:) is the ith row of A and B(:,j) is the jth row of B.

Note If A is an m-by-0 empty matrix and B is a 0-by-n empty matrix, where m and n are positive integers, A*B is an m-by-n matrix of all zeros.

Examples

Example 1

If A is a row vector and B is a column vector with the same number of elements as A, A*B is simply the inner product of A and B. For example,

```
A = [5 3 2 6]

A =

     5     3     2     6
B = [-4 9 0 1]'

B =
```

```
    -4
     9
     0
     1
A*B

ans =

    13
```

Example 2

```
A = [1 3 5; 2 4 7]

A =

     1     3     5
     2     4     7
B = [-5 8 11; 3 9 21;4 0 8]

B =

    -5     8    11
     3     9    21
     4     0     8
```

The product of A and B is

```
C = A*B

C =

    24    35   114
    30    52   162
```

Note that the second row of A is

```
A(2,:)

ans =

     2     4     7
```

while the third column of B is

```
B(:,3)

ans =

    11
    21
     8
```

The inner product of `A(2,:)` and `B(:,3)` is

```
A(2,:)*B(:,3)


ans =



   162
```

which is the same as `C(2,3)`.

See Also

Arithmetic operators

Purpose Convert mu-law audio signal to linear

Syntax

```
y = mu2lin(mu)
```

Description

`y = mu2lin(mu)` converts mu-law encoded 8-bit audio signals, stored as “flints” in the range `0` ≤ `mu` ≤ `255`, to linear signal amplitude in the range `-s < Y < s` where `s = 32124/32768 ~= .9803`. The input `mu` is often obtained using `fread(...,'uchar')` to read byte-encoded audio files. “Flints” are MATLAB integers — floating-point numbers whose values are integers.

See Also

`auread`, `lin2mu`

Purpose

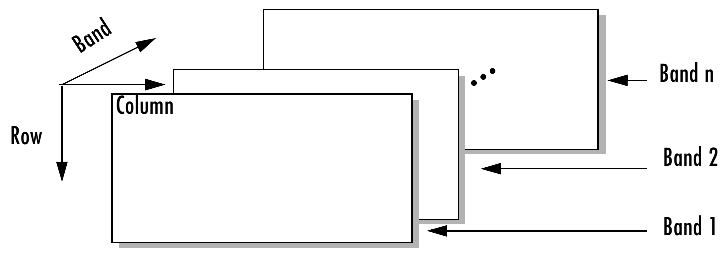
Read band interleaved data from a binary file

Syntax

```
X = multibandread(filename, size, precision, offset, interleave,
   byteorder)
X = multibandread(...,subset1,subset2,subset3)
```

Description

`X = multibandread(filename, size, precision, offset, interleave, byteorder)` reads multiband data from the binary file `filename`. This function defines *band* as the third dimension in a 3-D array, as shown in this figure.



You can use the parameters to `multibandread` to specify many aspects of the read operation, such as which bands to read. See "Parameters" on page 2-1500 for more information.

If you only read one band, the return value `X` is a 2-D array. If you read multiple bands, `X` is 3-D. By default, `X` is an array of type `double`; however, you can use the `precision` parameter to specify any other data type.

`X = multibandread(...,subset1,subset2,subset3)` reads a subset of the data in the file. You can use up to three subsetting parameters to specify the data subset along row, column, and band dimensions. See "Subsetting Parameters" on page 2-1501 for more information.

Parameters

This table describes the arguments accepted by `multibandread`.

`filename`	String containing the name of the file to be read.
`size`	Three-element vector of integers consisting of `[height, width, N]`, where • `height` is the total number of rows • `width` is the total number of elements in each row • `N` is the total number of bands. This will be the dimensions of the data if it is read in its entirety.
`precision`	String specifying the format of the data to be read, such as `'uint8'`, `'double'`, `'integer*4'`, or any of the other precisions supported by the `fread` function. Note: You can also use the `precision` parameter to specify the format of the output data. For example, to read `uint8` data and output a `uint8` array, specify a precision of `'uint8=>uint8'` (or `'*uint8'`). To read `uint8` data and output it in MATLAB in single precision, specify `'uint8=>single'`. See `fread` for more information.
`offset`	Scalar specifying the zero-based location of the first data element in the file. This value represents the number of bytes from the beginning of the file to where the data begins.

`interleave`	String specifying the format in which the data is stored • `'bsq'` — Band-Sequential • `'bil'` — Band-Interleaved-by-Line • `'bip'` — Band-Interleaved-by-Pixel For more information about these interleave methods, see the `multibandwrite` reference page.
`byteorder`	String specifying the byte ordering (machine format) in which the data is stored, such as • `'ieee-le'` — Little-endian • `'ieee-be'` — Big-endian See `fopen` for a complete list of supported formats.

Subsetting Parameters

You can specify up to three subsetting parameters. Each subsetting parameter is a three-element cell array, `{`*`dim`*`,` *`method`*`, index}`, where

`dim`	Text string specifying the dimension to subset along. It can have any of these values: • `'Column'` • `'Row'` • `'Band'`

`method`	Text string specifying the subsetting method. It can have either of these values: • `'Direct'` • `'Range'` If you leave out this element of the subset cell array, `multibandread` uses `'Direct'` as the default.
`index`	If `method` is `'Direct'`, `index` is a vector specifying the indices to read along the Band dimension. If `method` is `'Range'`, `index` is a three-element vector of `[start, increment, stop]` specifying the range and step size to read along the dimension specified in `dim`. If `index` is a two-element vector, `multibandread` assumes that the value of `increment` is `1`.

Examples

Read data from a multiband file into an 864-by-702-by-3 `uint8` matrix, `im`.

```
im = multibandread('bipdata.img',...
[864,702,3],'uint8=>uint8',0,'bip','ieee-le');
```

Read all rows and columns, but only bands 3, 4, and 6.

```
im = multibandread('bsqdata.img',...
[512,512,6],'uint8',0,'bsq','ieee-le',...
{'Band','Direct',[3 4 6]});
```

Read all bands and subset along the rows and columns.

```
im = multibandread('bildata.int',...
[350,400,50],'uint16',0,'bil','ieee-le',...
{'Row','Range',[2 2 350]},...
{'Column','Range',[1 4 350]});
```

See Also

`fread`, `fopen`, `multibandwrite`

Purpose Write multiband data to a file

Syntax

```
multibandwrite(data,filename,interleave)
multibandwrite(data,filename,interleave,start,totalsize)
multibandwrite(...,param,value,...)
```

Description

`multibandwrite(data,filename,interleave)` writes `data`, a two- or three-dimensional numeric or logical array, to the binary file specified by `filename`. The length of the third dimension of `data` determines the number of bands written to the file. The bands are written to the file in the form specified by `interleave`. See "Interleave Methods" on page 2-1504 for more information about this argument.

If `filename` already exists, `multibandwrite` overwrites it unless you specify the optional `offset` parameter. See the last alternate syntax for `multibandwrite` for information about other optional parameters.

`multibandwrite(data,filename,interleave,start,totalsize)` writes `data` to the binary file `filename` in chunks. In this syntax, `data` is a subset of the complete data set.

`start` is a 1-by-3 array `[firstrow firstcolumn firstband]` that specifies the location to start writing data. `firstrow` and `firstcolumn` specify the location of the upper left image pixel. `firstband` gives the index of the first band to write. For example, `data(I,J,K)` contains the data for the pixel at `[firstrow+I-1, firstcolumn+J-1]` in the `(firstband+K-1)`-th band.

`totalsize` is a 1-by-3 array, `[totalrows,totalcolumns,totalbands]`, which specifies the full, three-dimensional size of the data to be written to the file.

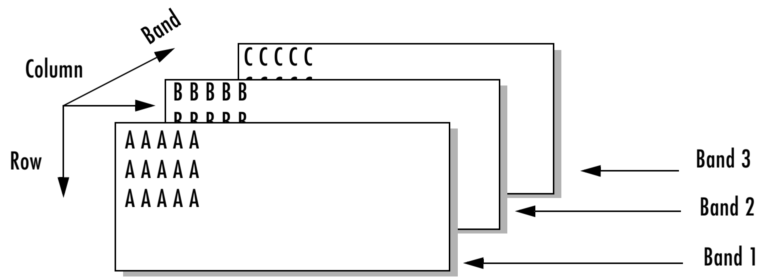
Note In this syntax, you must call `multibandwrite` multiple times to write all the data to the file. The first time it is called, `multibandwrite` writes the complete file, using the fill value for all values outside the data subset. In each subsequent call, `multibandwrite` overwrites these fill values with the data subset in `data`. The parameters `filename`, `interleave`, `offset`, and `totalsize` must remain constant throughout the writing of the file.

`multibandwrite(...,param,value...)` writes the multiband data to a file, specifying any of these optional parameter/value pairs.

Parameter	Description
`'precision'`	String specifying the form and size of each element written to the file. See the help for `fwrite` for a list of valid values. The default precision is the class of the data.
`'offset'`	The number of bytes to skip before the first data element. If the file does not already exist, `multibandwrite` writes ASCII null values to fill the space. To specify a different fill value, use the parameter `'fillvalue'`. This option is useful when you are writing a header to the file before or after writing the data. When writing the header to the file after the data is written, open the file with `fopen` using `'r+'` permission.
`machfmt`	String to control the format in which the data is written to the file. Typical values are `'ieee-le'` for little endian and `'ieee-be'` for big endian. See the help for `fopen` for a complete list of available formats. The default machine format is the local machine format.
`fillvalue`	A number specifying the value to use in place of missing data. `'fillvalue'` can be a single number, specifying the fill value for all missing data, or a 1-by-Number-of-bands vector of numbers specifying the fill value for each band. This value is used to fill space when data is written in chunks.

Interleave Methods

`interleave` is a string that specifies how `multibandwrite` interleaves the bands as it writes data to the file. If `data` is two-dimensional, `multibandwrite` ignores the `interleave` argument. The following table lists the supported methods and uses this example multiband file to illustrate each method.



Supported methods of interleaving bands include those listed below.

Method	String	Description	Example
Band-Interleaved-by-Line	`'bil'`	Write an entire row from each band	AAAAABBBBBCCCCC AAAAABBBBBCCCCC AAAAABBBBBCCCCC
Band-Interleaved-by-Pixel	`'bip'`	Write a pixel from each band	ABCABCABCABCABC...
Band-Sequential	`'bsq'`	Write each band in its entirety	AAAAA AAAAA AAAAA BBBBB BBBBB BBBBB CCCCC CCCCC CCCCC

Examples

In this example, all the data is written to the file with one function call. The bands are interleaved by line.

```
multibandwrite(data,'data.img','bil');
```

This example uses `multibandwrite` in a loop to write each band to a file separately.

```
for i=1:totalBands
```

```
    multibandwrite(bandData,'data.img','bip',[1 1 i],...
    [totalColumns, totalRows, totalBands]);
end
```

In this example, only a subset of each band is available for each call to `multibandwrite`. For example, an entire data set can have three bands with 1024-by-1024 pixels each (a 1024-by-1024-by-3 matrix). Only 128-by-128 chunks are available to be written to the file with each call to `multibandwrite`.

```
numBands = 3;
totalDataSize =  [1024 1024 numBands];
for i=1:numBands
   for k=1:8
       for j=1:8
          upperLeft = [(k-1)*128 (j-1)*128 i];
          multibandwrite(data,'banddata.img','bsq',...
                          upperLeft,totalDataSize);
       end
   end
end
```

See Also `multibandread`, `fwrite`, `fread`

Purpose Allow M-file or MEX-file clearing

Syntax

```
munlock
munlock fun
munlock('fun')
```

Description `munlock` unlocks the currently running M-file or MEX-file in memory so that subsequent `clear` functions can remove it.

`munlock fun` unlocks the M-file or MEX-file named `fun` from memory. By default, these files are unlocked so that changes to the file are picked up. Calls to `munlock` are needed only to unlock M-files or MEX-files that have been locked with `mlock`.

`munlock('fun')` is the function form of `munlock`.

Examples The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
  .
  .
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked testfun
ans =
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock testfun

mislocked testfun
ans =
     0
```

See Also

`mlock`, `mislocked`, `persistent`

2namelengthmax

Purpose Return maximum identifier length

Syntax

```
len = namelengthmax
```

Description

`len = namelengthmax` returns the maximum length allowed for MATLAB identifiers. MATLAB identifiers are

- Variable names
- Function and subfunction names
- Structure fieldnames
- Object names
- M-file names
- MEX-file names
- MDL-file names

Rather than hard-coding a specific maximum name length into your programs, use the `namelengthmax` function. This saves you the trouble of having to update these limits should the identifier length change in some future MATLAB release.

Examples

Call `namelengthmax` to get the maximum identifier length:

```
maxid = namelengthmax
maxid =
    63
```

See Also

`isvarname`, `genvarname`

NaN

Purpose

Not-a-Number

Syntax

`NaN`

Description

`NaN` returns the IEEE arithmetic representation for Not-a-Number (`NaN`). These result from operations which have undefined numerical results.

`NaN('double')` is the same as `NaN` with no inputs.

`NaN('single')` is the single precision representation of `NaN`.

`NaN(n)` is an `n`-by-`n` matrix of `NaN`s.

`NaN(m,n)` or `inf([m,n])` is an `m`-by-`n` matrix of `NaN`s.

`NaN(m,n,p,...)` or `NaN([m,n,p,...])` is an `m`-by-`n`-by-`p`-by-... array of `NaN`s.

`NaN(...,classname)` is an array of `NaN`s of class specified by `classname`. `classname` must be either `'single'` or `'double'`.

Examples

These operations produce `NaN`:

- Any arithmetic operation on a `NaN`, such as `sqrt(NaN)`
- Addition or subtraction, such as magnitude subtraction of infinities as `(+Inf)+(-Inf)`
- Multiplication, such as `0*Inf`
- Division, such as `0/0` and `Inf/Inf`
- Remainder, such as `rem(x,y)` where `y` is zero or `x` is infinity

Remarks

Because two `NaN`s are not equal to each other, logical operations involving `NaN`s always return false, except ~= (not equal). Consequently,

```
NaN ~= NaN
ans =
     1

NaN == NaN
ans =
     0
```

and the `NaN`s in a vector are treated as different unique elements.

```
unique([1 1 NaN NaN])
ans =
     1 NaN NaN
```

Use the `isnan` function to detect `NaN`s in an array.

```
isnan([1 1 NaN NaN])
ans =
     0     0     1     1
```

See Also

`Inf`, `isnan`

nargchk

Purpose Check number of input arguments

Syntax

```
msgstring = nargchk(minargs, maxargs, numargs)
msgstring = nargchk(minargs, maxargs, numargs, 'string')
msgstruct = nargchk(minargs, maxargs, numargs, 'struct')
```

Description Use `nargchk` inside an M-file function to check that the desired number of input arguments is specified in the call to that function.

`msgstring = nargchk(minargs, maxargs, numargs)` returns an error message string `msgstring` if the number of inputs specified in the call `numargs` is less than `minargs` or greater than `maxargs`. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty matrix.

It is common to use the `nargin` function to determine the number of input arguments specified in the call.

`msgstring = nargchk(minargs, maxargs, numargs, 'string')` is essentially the same as the command shown above, as `nargchk` returns a string by default.

`msgstruct = nargchk(minargs, maxargs, numargs, 'struct')` returns an error message structure `msgstruct` instead of a string. The fields of the return structure contain the error message string and a message identifier. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty structure.

When too few inputs are supplied, the message string and identifier are

```
       message: 'Not enough input arguments.'
    identifier: 'MATLAB:nargchk:notEnoughInputs'
```

When too many inputs are supplied, the message string and identifier are

```
       message: 'Too many input arguments.'
    identifier: 'MATLAB:nargchk:tooManyInputs'
```

Remarks `nargchk` is often used together with the `error` function. The `error` function accepts either type of return value from `nargchk`: a message string or message structure. For example, this command provides the `error` function with a message string and identifier regarding which error was caught:

```
error(nargchk(2, 4, nargin, 'struct'))
```

If nargchk detects no error, it returns an empty string or structure. When nargchk is used with the error function, as shown here, this empty string or structure is passed as an input to error. When error receives an empty string or structure, it simply returns and no error is generated.

Examples

Given the function foo,

```
function f = foo(x, y, z)
error(nargchk(2, 3, nargin))
```

Then typing foo(1) produces

```
Not enough input arguments.
```

See Also

nargoutchk, nargin, nargout, varargin, varargout, error

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 function *`fun`* or `-1` if the function has a variable number of input arguments.

`nargout` returns the number of output arguments specified for a function.

`nargout('`*`fun`*`')` returns the number of declared outputs for the 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(x, y, npts, angle, subdiv)
% MYPLOT  Plot a function.
% MYPLOT(x, y, npts, angle, subdiv)
%     The first two input arguments are
%     required; the other three have default values.
 ...
if nargin < 5, subdiv = 20; end
if nargin < 4, angle = 10; end
if nargin < 3, npts = 25; end
 ...
if nargout == 0
     plot(x, y)
else
     x0 = x;
     y0 = y;
```

```
end
```

See Also `inputname`, `varargin`, `varargout`, `nargchk`, `nargoutchk`

nargoutchk

Purpose Validate number of output arguments

Syntax

```
msgstring = nargoutchk(minargs, maxargs, numargs)
msgstring = nargoutchk(minargs, maxargs, numargs, 'string')
msgstruct = nargoutchk(minargs, maxargs, numargs, 'struct')
```

Description Use `nargoutchk` inside an M-file function to check that the desired number of output arguments is specified in the call to that function.

`msgstring = nargoutchk(minargs, maxargs, numargs)` returns an error message string `msgstring` if the number of outputs specified in the call, `numargs`, is less than `minargs` or greater than `maxargs`. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargoutchk` returns an empty matrix.

It is common to use the `nargout` function to determine the number of output arguments specified in the call.

`msgstring = nargoutchk(minargs, maxargs, numargs, 'string')` is essentially the same as the command shown above, as `nargoutchk` returns a string by default.

`msgstruct = nargoutchk(minargs, maxargs, numargs, 'struct')` returns an error message structure `msgstruct` instead of a string. The fields of the return structure contain the error message string and a message identifier. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargoutchk` returns an empty structure.

When too few outputs are supplied, the message string and identifier are

```
   message: 'Not enough output arguments.'
identifier: 'MATLAB:nargoutchk:notEnoughOutputs'
```

When too many outputs are supplied, the message string and identifier are

```
   message: 'Too many output arguments.'
identifier: 'MATLAB:nargoutchk:tooManyOutputs'
```

Remarks `nargoutchk` is often used together with the `error` function. The `error` function accepts either type of return value from `nargoutchk`: a message string or message structure. For example, this command provides the `error` function with a message string and identifier regarding which error was caught:

```
error(nargoutchk(2, 4, nargout, 'struct'))
```

If `nargoutchk` detects no error, it returns an empty string or structure. When `nargoutchk` is used with the `error` function, as shown here, this empty string or structure is passed as an input to `error`. When `error` receives an empty string or structure, it simply returns and no error is generated.

Examples

You can use `nargoutchk` to determine if an M-file has been called with the correct number of output arguments. This example uses `nargout` to return the number of output arguments specified when the function was called. The function is designed to be called with one, two, or three output arguments. If called with no arguments or more than three arguments, `nargoutchk` returns an error message:

```
function [s, varargout] = mysize(x)
msg = nargoutchk(1, 3, nargout);
if isempty(msg)
    nout = max(nargout, 1) - 1;
    s = size(x);
    for k = 1:nout, varargout(k) = {s(k)}; end
else
    disp(msg)
end
```

See Also

`nargchk`, `nargout`, `nargin`, `varargout`, `varargin`, `error`

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

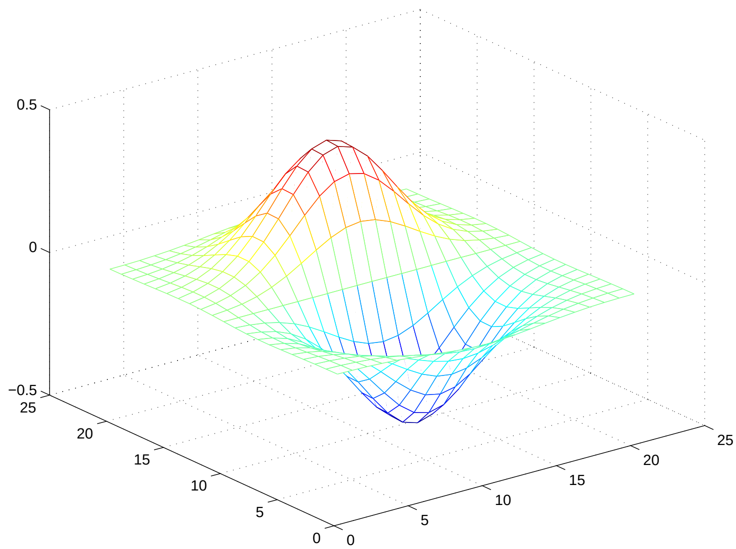
```
[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)
[X1,X2,...] = ndgrid(x)
```

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 `Xi` are copies of elements of the vector `xi`.

`[X1,X2,...] = ndgrid(x)` is the same as `[X1,X2,...] = ndgrid(x,x,...)`.

Examples Evaluate the function $x_1 e^{-x_1^2 - x_2^2}$ over the range $-2 < x_1 < 2, -2 < x_2 < 2$.

```
[X1,X2] = ndgrid(-2:.2:2, -2:.2:2);
Z = X1 .* exp(-X1.^2 - X2.^2);
mesh(Z)
```



Remarks

The `ndgrid` function is like `meshgrid` except that the order of the first two input arguments are switched. That is, the statement

```
[X1,X2,X3] = ndgrid(x1,x2,x3)
```

produces the same result as

```
[X2,X1,X3] = meshgrid(x2,x1,x3)
```

Because of this, `ndgrid` is better suited to multidimensional problems that aren't spatially based, while `meshgrid` is better suited to problems in two- or three-dimensional Cartesian space.

See Also

`meshgrid`, `interpn`

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`

newplot

Purpose Determine where to draw graphics objects

Syntax

```
newplot
h = newplot
h = newplot(hsave)
```

Description `newplot` prepares a figure and axes for subsequent graphics commands.

`h = newplot` prepares a figure and axes for subsequent graphics commands and returns a handle to the current axes.

`h = newplot(hsave)` prepares and returns an axes, but does not delete any objects whose handles appear in `hsave`. If `hsave` is specified, the figure and axes containing `hsave` are prepared for plotting instead of the current axes of the current figure. If `hsave` is empty, `newplot` behaves as if it were called without any inputs.

Remarks Use `newplot` at the beginning of high-level graphics M-files to determine which figure and axes to target for graphics output. Calling `newplot` can change the current figure and current axes. Basically, there are three options when you are drawing graphics in existing figures and axes:

- Add the new graphics without changing any properties or deleting any objects.
- Delete all existing objects whose handles are not hidden before drawing the new objects.
- Delete all existing objects regardless of whether or not their handles are hidden, and reset most properties to their defaults before drawing the new objects (refer to the following table for specific information).

The figure and axes `NextPlot` properties determine how `newplot` behaves. The following two tables describe this behavior with various property values.

First, `newplot` reads the current figure's `NextPlot` property and acts accordingly.

NextPlot	What Happens
`add`	Draw to the current figure without clearing any graphics objects already present.
`replacechildren`	Remove all child objects whose `HandleVisibility` property is set to `on` and reset figure `NextPlot` property to `add`. This clears the current figure and is equivalent to issuing the `clf` command.
`replace`	Remove all child objects (regardless of the setting of the `HandleVisibility` property) and reset figure properties to their defaults, except • `NextPlot` is reset to `add` regardless of user-defined defaults. • `Position`, `Units`, `PaperPosition`, and `PaperUnits` are not reset. This clears and resets the current figure and is equivalent to issuing the `clf reset` command.

After `newplot` establishes which figure to draw in, it reads the current axes' `NextPlot` property and acts accordingly.

NextPlot	Description
`add`	Draw into the current axes, retaining all graphics objects already present.
`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.

NextPlot	Description
replace	Remove all child objects (regardless of the setting of the HandleVisibility property) and reset axes properties to their defaults, except Position and Units. This clears and resets the current axes like the cla reset command.

See Also

axes, cla, clf, figure, hold, ishold, reset

The NextPlot property for figure and axes graphics objects

"Figure Windows" for related functions

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`

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 objects in the specified figure to `normal`. `state` can be the following strings:

- `'save'` — Set the values of the `EraseMode` properties to `normal` for all the appropriate objects in the designated figure.
- `'restore'` — Restore the `EraseMode` properties to the previous values (i.e., the values before calling `noanimate` with the `'save'` argument).

`noanimate(state)` operates on the current figure.

`noanimate` is useful if you want to print the figure to a TIFF or JPEG format.

See Also

`print`

"Animation" for related functions

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`

Purpose

Vector and matrix norms

Syntax

```
n = norm(A)
n = norm(A,p)
```

Description

The *norm* of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. The `norm` function calculates several different types of matrix norms:

`n = norm(A)` returns the largest singular value of `A`, `max(svd(A))`.

`n = norm(A,p)` returns a different kind of norm, depending on the value of `p`.

If `p` is...	**Then `norm` returns...**
`1`	The 1-norm, or largest column sum of `A`, `max(sum(abs(A)))`.
`2`	The largest singular value (same as `norm(A)`).
`inf`	The infinity norm, or largest row sum of `A`, `max(sum(abs(A')))`.
`'fro'`	The Frobenius-norm of matrix `A`, `sqrt(sum(diag(A'*A)))`.

When `A` is a vector:

`norm(A,p)`	Returns `sum(abs(A).^p)^(1/p)`, for any 1 <= `p` <= ∞.
`norm(A)`	Returns `norm(A,2)`.
`norm(A,inf)`	Returns `max(abs(A))`.
`norm(A,-inf)`	Returns `min(abs(A))`.

Remarks

Note that `norm(x)` is the Euclidean length of a vector `x`. On the other hand, MATLAB uses "length" to denote the number of elements `n` in a vector. This example uses `norm(x)/sqrt(n)` to obtain the root-mean-square (RMS) value of an `n`-element vector `x`.

```
x = [0 1 2 3]
x =
     0     1     2     3

sqrt(0+1+4+9)   % Euclidean length
ans =
    3.7417

norm(x)
ans =
    3.7417

n = length(x)   % Number of elements
n =
     4

rms = 3.7417/2  % rms = norm(x)/sqrt(n)
rms =
    1.8708
```

See Also

cond, condest, normest, rcond, svd

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 Open M-book in Microsoft Word (Windows only)

Syntax

```
notebook
notebook('filename')
notebook('-setup')
notebook('-setup', wordver, wordloc, templateloc)
```

Description

`notebook` by itself, launches Microsoft Word and creates a new M-book called `Document 1`.

`notebook('filename')` launches Microsoft Word and opens the M-book `filename`.

`notebook('-setup')` runs an interactive setup function for the Notebook. You are prompted for the version of Microsoft Word, and if necessary, for the locations of several files.

`notebook('-setup', wordver, wordloc, templateloc)` sets up the Notebook using the specified information.

`wordver`	Version of Microsoft Word, either 97, 2000, or 2002 (for XP)
`wordloc`	Directory containing `winword.exe`
`templateloc`	Directory containing Microsoft Word template directory

See Also Notebook for Publishing to Word

Purpose Current date and time

Syntax

```
t = now
```

Description `t = now` returns the current date and time as a serial date number. To return the time only, use `rem(now,1)`. To return the date only, use `floor(now)`.

Examples

```
t1 = now, t2 = rem(now,1)

t1 =

   7.2908e+05


t2 =

    0.4013
```

See Also `clock`, `date`, `datenum`

Purpose Real nth root of real numbers

Syntax

```
y = nthroot(X, n)
```

Description

`y = nthroot(X, n)` returns the real nth root of the elements of `X`. Both `X` and `n` must be real and `n` must be a scalar. If `X` has negative entries, `n` must be an odd integer.

Example

```
nthroot(-2, 3)
```

returns the real cube root of `-2`.

```
ans =

   -1.2599
```

By comparison,

```
(-2)^(1/3)
```

returns a complex cube root of `-2`.

```
ans =

   0.6300 + 1.0911i
```

See Also `power`

Purpose Null space of a matrix

Syntax

```
Z = null(A)
Z = null(A,'r')
```

Description

`Z = null(A)` is an orthonormal basis for the null space of `A` obtained from the singular value decomposition. That is, `A*Z` has negligible elements, `size(Z,2)` is the nullity of `A`, and `Z'*Z = I`.

`Z = null(A,'r')` is a "rational" basis for the null space obtained from the reduced row echelon form. `A*Z` is zero, `size(Z,2)` is an estimate for the nullity of `A`, and, if `A` is a small matrix with integer elements, the elements of the reduced row echelon form (as computed using `rref`) are ratios of small integers.

The orthonormal basis is preferable numerically, while the rational basis may be preferable pedagogically.

Example

Example 1. Compute the orthonormal basis for the null space of a matrix A.

```
A = [1     2     3
     1     2     3
     1     2     3];

Z = null(A)

Z =
    0.9636         0
   -0.1482   -0.8321
   -0.2224    0.5547

A*Z

ans =
  1.0e-015 *
     0.2220    0.2220
     0.2220    0.2220
     0.2220    0.2220

Z'*Z
```

```
ans =
    1.0000   -0.0000
   -0.0000    1.0000
```

Example 2. Compute the rational basis for the null space of the same matrix `A`.

```
ZR = null(A,'r')

ZR =
    -2    -3
     1     0
     0     1

A*ZR

ans =

     0     0
     0     0
     0     0
```

See Also

`orth`, `rank`, `rref`, `svd`

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`, `mat2cell`, `cell2mat`

Purpose Convert singles and doubles to IEEE hexadecimal strings.

Syntax `num2hex(X)`

Description If `X` is a single or double precision array with n elements, `num2hex(X)` is an n-by-8 or n-by-16 char array of the hexadecimal floating-point representation. The same representation is printed with format `hex`.

Examples

```
num2hex([1 0 0.1 -pi Inf NaN])
```

returns

```
ans =

3ff0000000000000
0000000000000000
3fb999999999999a
c00921fb54442d18
7ff0000000000000
fff8000000000000
```

```
num2hex(single([1 0 0.1 -pi Inf NaN]))
```

returns

```
ans =

3f800000
00000000
3dcccccd
c0490fdb
7f800000
ffc00000
```

See Also `hex2num`, `dec2hex`, `format`

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` with a format of `%10.5e\n` returns a matrix of strings in exponential format, having 5 decimal places, with each element separated by a `newline` character:

```
x = rand(3) * 9999;              % Create a 2-by-3 matrix.
x(3,:) = [];

A = num2str(x, '%10.5e\n')       % Convert to string array.
A =
   6.87255e+003
   1.55597e+003
   8.55890e+003

   3.46077e+003
```

```
1.91097e+003
4.90201e+003
```

See Also fprintf, int2str, sprintf

Purpose Number of elements in array or subscripted array expression

Syntax

```
n = numel(A)
n = numel(A,varargin)
```

Description `n = numel(A)` returns the the number of elements, `n`, in array `A`.

`n = numel(A,varargin)` returns the number of subscripted elements, `n`, in `A(index1,index2,...,indexn)`, where `varargin` is a cell array whose elements are `index1`, `index2`, `...`, `indexn`.

MATLAB implicitly calls the `numel` built-in function whenever an expression such as `A{index1,index2,...,indexN}` or `A.fieldname` generates a comma-separated list.

`numel` works with the overloaded `subsref` and `subsasgn` functions. It computes the number of expected outputs (`nargout`) returned from `subsref`. It also computes the number of expected inputs (`nargin`) to be assigned using `subsasgn`. The `nargin` value for the overloaded `subsasgn` function consists of the variable being assigned to, the structure array of subscripts, and the value returned by `numel`.

As a class designer, you must ensure that the value of `n` returned by the built-in `numel` function is consistent with the class design for that object. If `n` is different from either the `nargout` for the overloaded `subsref` function or the `nargin` for the overloaded `subsasgn` function, then you need to overload `numel` to return a value of `n` that is consistent with the class' `subsref` and `subsasgn` functions. Otherwise, MATLAB produces errors when calling these functions.

Examples Create a 4-by-4-by-2 matrix. `numel` counts 32 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
```

See Also `nargin`, `nargout`, `prod`, `size`, `subsasgn`, `subsref`

2ode15i

Purpose

Solve fully implicit differential equations, variable order method

Syntax

```
[t,Y] = ode15i(odefun,tspan,y0,yp0)
[t,Y] = ode15i(odefun,tspan,y0,yp0,options)
[t,Y,TE,YE,IE] = ode15i(...)
sol = ode15i(...)
```

Arguments

The following table lists the input arguments for `ode15i`.

`odefun`	A function that evaluates the left side of the differential equations, which are of the form $f(t, y, y') = 0$.
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.
`y0`, `yp0`	Vectors of initial conditions for y and y' respectively.
`options`	Optional integration argument created using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for `ode15i`.

`t`	Column vector of time points
`Y`	Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

Description

`[t,Y] = ode15i(odefun,tspan,y0,yp0)` with `tspan = [t0 tf]` integrates the system of differential equations $f(t, y, y') = 0$ from time `t0` to `tf` with initial conditions `y0` and `yp0`. Function `ode15i` solves ODEs and DAEs of index 1. The initial conditions must be consistent, meaning that $f(\mathrm{t0}, \mathrm{y0}, \mathrm{yp0}) = 0$. You can use the function `decic` to compute consistent initial conditions close to guessed values. Function `odefun(t,y,yp)`, for a scalar `t` and column vectors `y` and `yp`, must return a column vector corresponding to $f(t, y, y')$. Each row in the solution array `Y` corresponds to a time returned in the column vector `t`. To obtain solutions at specific times `t0,t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

Parameterizing Functions Called by Function Functions, in the online MATLAB documentation, explains how to provide addition parameters to the function `odefun`, if necessary.

`[t,Y] = ode15i(odefun,tspan,y0,yp0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used options include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components `1e-6` by default). See `odeset` for details.

`[t,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)` with the `'Events'` property in `options` set to a function `events`, solves as above while also finding where functions of (t, y, y'), called event functions, are zero. The function `events` is of the form `[value,isterminal,direction] = events(t,y,yp) and` includes the necessary event functions. Code the function `events` so that the `i`th element of each output vector corresponds to the `i`th event. For the `i`th event function in `events`:

- `value(i)` is the value of the function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Output `TE` is a column vector of times at which events occur. Rows of `YE` are the corresponding solutions, and indices in vector `IE` specify which event occurred. See "Changing ODE Integration Properties" in the MATLAB documentation for more information.

`sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)` returns a structure that can be used with `deval` to evaluate the solution at any point between `t0` and `tf`. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver. If you specify the `Events` option and a terminal event is detected, `sol.x(end)` contains the end of the step at which the event occurred.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Options

`ode15i` accepts the following parameters in `options`. For more information, see `odeset` and "Changing ODE Integration Properties" in the MATLAB documentation.

Error control	`RelTol`, `AbsTol`, `NormControl`
Solver output	`OutputFcn`, `OutputSel`, `Refine`, `Stats`
Event location	`Events`
Step size	`MaxStep`, `InitialStep`
Jacobian matrix	`Jacobian`, `JPattern`, `Vectorized`

Solver Output

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

Jacobian Matrices

The Jacobian matrices $\partial f/\partial y$ and $\partial f/\partial y'$ are critical to reliability and efficiency. You can provide these matrices as one of the following:

- Function of the form `[dfdy,dfdyp] = FJAC(t,y,yp)` that computes the Jacobian matrices. If `FJAC` returns an empty matix `[]` for either `dfdy` or `dfdyp`, then `ode15i` approximates that matrix by finite differences.
- Cell array of two constant matrices `{dfdy,dfdyp}`, either of which could be empty.

Use `odeset` to set the `Jacobian` option to the function or cell array. If you do not set the `Jacobian` option, `ode15i` approximates both Jacobian matrices by finite differences.

For `ode15i`, `Vectorized` is a two-element cell array. Set the first element to `'on'` if `odefun(t,[y1,y2,...],yp)` returns `[odefun(t,y1,yp),odefun(t,y2,yp),...]`. Set the second element to `'on'` if `odefun(t,y,[yp1,yp2,...])` returns `[odefun(t,y,yp1),odefun(t,y,yp2),...]`. The default value of `Vectorized` is `{'off','off'}`.

For `ode15i`, `JPattern` is also a two-element sparse matrix cell array. If $\partial f/\partial y$ or $\partial f/\partial y'$ is a sparse matrix, set `JPattern` to the sparsity patterns, `{SPDY,SPDYP}`. A sparsity pattern of $\partial f/\partial y$ is a sparse matrix `SPDY` with `SPDY(i,j) = 1` if component `i` of `f(t,y,yp)` depends on component `j` of `y`, and `0` otherwise. Use `SPDY = []` to indicate that $\partial f/\partial y$ is a full matrix. Similarly for $\partial f/\partial y'$ and `SPDYP`. The default value of `JPattern` is `{[],[]}`.

Examples

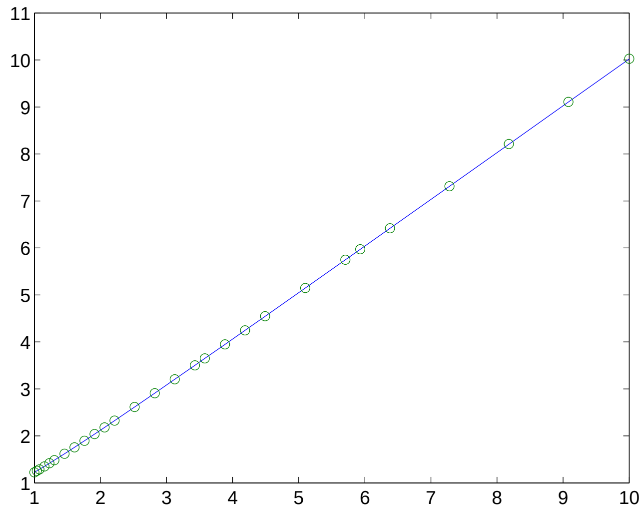
Example 1. This example uses uses a helper function `decic` to hold fixed the initial value for $y(t_0)$ and compute a consistent intial value for $y'(t_0)$ for the Weissinger implicit ODE. The Weissinger function evaluates the residual of the implicit ODE.

```
t0 = 1;
y0 = sqrt(3/2);
yp0 = 0;
```

```
[y0,yp0] = decic(@weissinger,t0,y0,1,yp0,0);
```

The example uses `ode15i` to solve the ODE, and then plots the numerical solution against the analytical solution.

```
[t,y] = ode15i(@weissinger,[1 10],y0,yp0);
ytrue = sqrt(t.^2 + 0.5);
plot(t,y,t,ytrue,'o');
```



Other Examples. These demos provide examples of implicit ODEs: `ihb1dae`, `iburgersode`.

See Also

`decic`, `deval`, `odeget`, `odeset`, `@` (function handle)

Other ODE initial value problem solvers: `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`

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,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

where `solver` is one of `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.

Arguments The following table describes the input arguments to the solvers.

`odefun`	A function that evaluates the right 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]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points.

Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.

`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`t`	Column vector of time points
`Y`	Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

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 the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

Parameterizing Functions Called by Function Functions, in the online MATLAB documentation, explains how to provide addition parameters to the function `odefun`, if necessary.

`[t, Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). See `odeset` for details.

`[t,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`:

- `value(i)` is the value of the function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimnesional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property 'on' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t, y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t, y)y' = f(t, y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to `'none'`.
- If the mass matrix depends weakly on y, set `MStateDependence` to `'weak'` (the default) 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	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 2-1556 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Changing ODE Integration Properties” in the MATLAB documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`RelTol`, `AbsTol`, `NormControl`	√	√	√	√	√	√	√
`OutputFcn`, `OutputSel`, `Refine`, `Stats`	√	√	√	√	√	√	√

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
`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

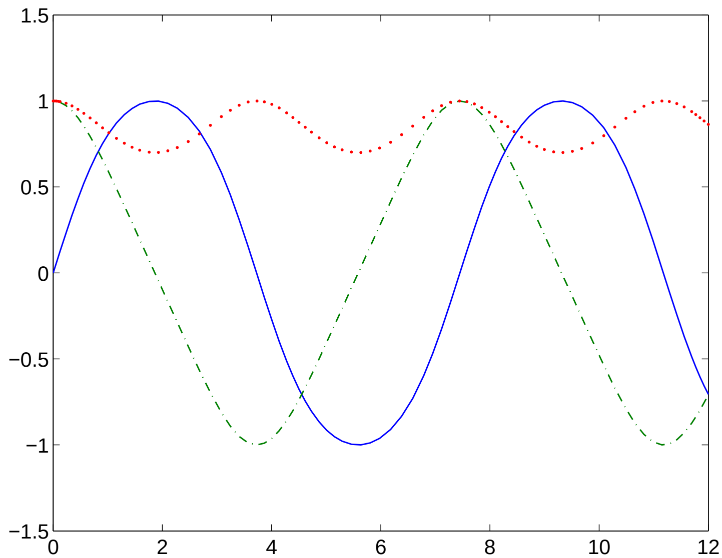
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[t,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array Y versus T shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2. An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$
\begin{array}{ll}
y'_1 = y_2 & y_1(0) = 0 \\
y'_2 = 1000(1-y_1^2)y_2 - y_1 & y_2(0) = 1
\end{array}
$$

To simulate this system, create a function vdp1000 containing the equations

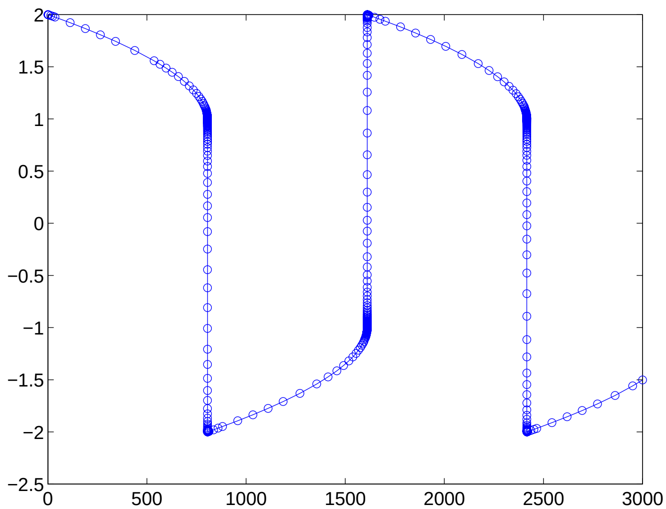
```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[t,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing `y(t`$_n$`)`, it needs only the solution at the immediately preceding time point, `y(t`$_{n-1}$`)`. In general, `ode45` is the best function to apply as a "first try" for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver – it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

See Also

`deval`, `ode15i`, `odeget`, `odeset`, `@` (function handle)

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp 436-451.

[2] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp 1-9.

[3] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp 19-26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp 1-22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp 538-552.

Purpose

Define 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 `odeset` or any of the ODE solvers.

Description

`odefile` is not a command or function. It is a help entry that describes how to create an M-file defining the system of equations to be solved. This definition is the first step in using any of the MATLAB ODE solvers. In MATLAB documentation, this M-file is referred to as an `odefile`, although you can give your M-file any name you like.

You can use the `odefile` M-file to define a system of differential equations in one of these forms

$$y' = f(t, y)$$

or

$$M(t, y)y' = f(t, y)v$$

where:

- t is a scalar independent variable, typically representing time.
- y is a vector of dependent variables.
- f is a function of t and y returning a column vector the same length as y.
- $M(t, y)$ is a time-and-state-dependent mass matrix.

The ODE file must accept the arguments `t` and `y`, although it does not have to use them. By default, the ODE file must return a column vector the same length as `y`.

All of the solvers of the ODE suite can solve $M(t, y)y' = f(t, y)$, except `ode23s`, which can only solve problems with constant mass matrices. The `ode15s` and `ode23t` solvers can solve some differential-algebraic equations (DAEs) of the form $M(t)y' = f(t, y)$.

Beyond defining a system of differential equations, you can specify an entire initial value problem (IVP) within the ODE M-file, eliminating the need to supply time and initial value vectors at the command line (see Examples on page 2-1562).

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.

$$y'_1 = y_2$$

$$y'_2 = \mu(1 - y_1^2)y_2 - y_1$$

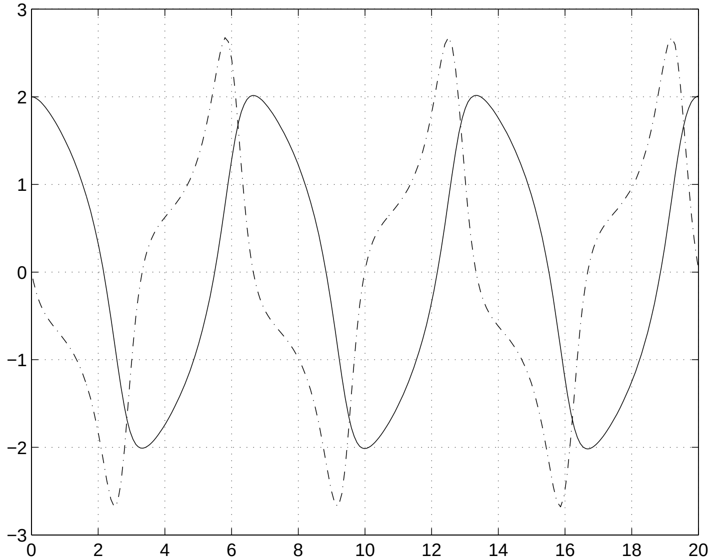
The M-file

```
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`

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.

ODE 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`.

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.

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 integers	Output selection indices. Specifies the components of the solution vector that the solver passes to the output function. `OutputSel` defaults to all components.	
`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.	

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`.

Event Location Property

Property	Value	Description
`Events`	Function	Locate events. Set this property to `@Events`, where `Events` is the name of the events function. See the ODE solvers for details.

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.

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. By default the solver determines an initial step size automatically.

In addition there are two options that apply only to the `ode15s` solver.

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

`deval`, `odeget`, `ode45`, `ode23`, `ode23t`, `ode23tb`, `ode113`, `ode15s`, `ode23s`, `@` (function handle)

odextend

Purpose

Extend the solution of an initial value problem for an ordinary differential equation (ODE)

Syntax

```
solext = odextend(sol, odefun, tfinal)
solext = odextend(sol, [], tfinal)
solext = odextend(sol, odefun, tfinal, yinit)
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
solext = odextend(sol, odefun, tfinal, yinit, options, P1, P2...)
```

Description

`solext = odextend(sol, odefun, tfinal)` extends the solution stored in `sol` to an interval with upper bound `tfinal` for the independent variable. `sol` is an ODE solution structure created using an ODE solver. The lower bound for the independent variable in `solext` is the same as in `sol`. If you created `sol` with an ODE solver other than `ode15i`, the function `odefun` computes the right-hand side of the ODE equation, which is of the form $y' = f(t, y)$. If you created `sol` using `ode15i`, the function `odefun` computes the left-hand side of the ODE equation, which is of the form $f(t, y, y') = 0$.

`odextend` extends the solution by integrating `odefun` from the upper bound for the independent variable in `sol` to `tfinal`, using the same ODE solver that created `sol`. By default, `odextend` uses

- The initial conditions `y = sol.y(:, end)` for the subsequent integration
- The same integration properties and additional input arguments the ODE solver originally used to compute `sol`. This information is stored as part of the solution structure `sol` and is subsequently passed to `solext`. Unless you want to change these values, you do not need to pass them to `odextend`.

`solext = odextend(sol, [], tfinal)` uses the same ODE function that the ODE solver uses to compute `sol` to extend the solution. It is not necessary to pass in `odefun` explicitly unless it differs from the original ODE function.

`solext = odextend(sol, odefun, tfinal, yinit)` uses the column vector `yinit` as new initial conditions for the subsequent integration, instead of the vector `sol.y(end)`.

Note To extend solutions obtained with `ode15i`, use the following syntax, in which the column vector `ypinit` is the initial derivative of the solution:

```
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
```

`solext = odextend(sol, odefun, tfinal, yinit, options)` uses the integration properties specified in `options` instead of the options the ODE solver originally used to compute `sol`. The new options are then stored within the structure `solext`. See `odeset` for details on setting `options` properties. Set `yinit = []` as a placeholder to specify the default initial conditions.

`solext = odextend(sol, odefun, tfinal, yinit, options, P1, P2...)` passes the additional parameters `P1`, `P2`,... to the ODE function as `odefun(t, y, P1, P2...)` and similarly to all functions you specify in `options`. You do not need to specify these parameters if their values are the same as those used to compute `sol`. Set `options = []` as a place holder to use the same `options` used to compute `sol`.

Example

The following command

```
sol=ode45(@vdp1,[0 10],[2 0]);
```

uses `ode45` to solve the system `y' = vdp1(t,y)`, where `vdp1` is an example of an ODE function provided with MATLAB, on the interval `[0 10]`. Then, the commands

```
sol=odextend(sol,@vdp1,20);
plot(sol.x,sol.y(1,:));
```

extend the solution to the interval `[0 20]` and plot the first component of the solution on `[0 20]`.

See Also

`deval`, `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `ode15i`, `odeset`, `odeget`, `deval`

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))
ones(m, n,...,classname)
ones([m,n,...],classname)
```

Description

`Y = ones(n)` returns an `n`-by-`n` matrix of `1`s. An error message appears if `n` is not a scalar.

`Y = ones(m,n)` or `Y = ones([m n])` returns an `m`-by-`n` matrix of ones.

`Y = ones(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`.

`ones(m, n,...,classname)` or `ones([m,n,...],classname)` is an `m`-by-`n`-by-... array of ones of data type `classname`. `classname` is a string specifying the data type of the output. `classname` can have the following values: `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, or `'uint32'`.

Example

```
x = ones(2,3,'int8');
```

See Also

`eye`, `zeros`

Purpose Open files based on extension

Syntax

```
open('name')
```

Description

`open('name')` opens the object specified by the string `name`. The specific action taken upon opening depends on the type of object specified by `name`.

name	Action
Variable	Open array `name` in the Array Editor (the array must be numeric).
M-file (`name.m`)	Open M-file `name` in M-file Editor.
Model (`name.mdl`)	Open model `name` in Simulink.
MAT-file (`name.mat`)	Open MAT-file and store variables in a structure in the workspace.
Figure file (`*.fig`)	Open figure in a figure window.
P-file (`name.p`)	Open the corresponding M-file, `name.m`, if it exists, in the M-file Editor.
HTML file (`*.html`)	Open HTML document in Help browser.
PDF file (`*.pdf`)	Open PDF document in Adobe Acrobat.
Other extensions (`name.xxx`)	Open `name.xxx` by calling the helper function `openxxx`, where `openxxx` is a user-defined function.
No extension (`name`)	Open `name` in the default editor. If `name` does not exist, then `open` checks to see if `name.mdl` or `name.m` is on the path or in the current directory and, if so, opens the file returned by `which('name')`.

If more than one file with the specified filename `name` exists on the MATLAB path, then `open` opens the file returned by `which('name')`.

If no such file `name` exists, then `open` displays an error message.

You can create your own `openxxx` functions to set up handlers for new file types. `open('filename.xxx')` calls the `openxxx` function it finds on the path. For example, create a function `openlog` if you want a handler for opening files with file extension `.log`.

Examples

Example 1 — Opening a File on the Path

To open the M-file `copyfile.m`, type

```
open copyfile.m
```

MATLAB opens the `copyfile.m` file that resides in `toolbox\matlab\general`. If you have a `copyfile.m` file in a directory that is before `toolbox\matlab\general` on the MATLAB path, then `open` opens that file instead.

Example 2 — Opening a File Not on the Path

To open a file that is not on the MATLAB path, enter the complete file specification. If no such file is found, then MATLAB displays an error message.

```
open('D:\temp\data.mat')
```

Example 3 — Specifying a File Without a File Extension

When you specify a file without including its file extension, MATLAB determines which file to open for you. It does this by calling

```
 which('filename')
```

In this example, `open matrixdemos` could open either an M-file or a Simulink model of the same name, since both exist on the path.

```
dir matrixdemos.*

  matrixdemos.m    matrixdemos.mdl
```

Because the call `which('matrixdemos')` returns the name of the Simulink model, `open` opens the `matrixdemos` model rather than the M-file of that name.

```
open matrixdemos          % Opens model matrixdemos.mdl
```

Example 4 — Opening a MAT-File

This example opens a MAT-file containing MATLAB data and then keeps just one of the variables from that file. The others are overwritten when ans is reused by MATLAB.

```
% Open a MAT-file containing miscellaneous data.
open D:\temp\data.mat

ans =

          x: [3x2x2 double]
          y: {4x5 cell}
          k: 8
    spArray: [5x5 double]
   dblArray: [4x1 java.lang.Double[][]]
   strArray: {2x5 cell}


% Keep the dblArray value by assigning it to a variable.
dbl = ans.dblArray

dbl =

java.lang.Double[][]:
    [ 5.7200]    [ 6.7200]    [ 7.7200]
    [10.4400]    [11.4400]    [12.4400]
    [15.1600]    [16.1600]    [17.1600]
    [19.8800]    [20.8800]    [21.8800]
```

Example 5 — Using a User-Defined Handler Function

If you create an M-file function called opencht to handle files with extension .cht, and then issue the command

```
open myfigure.cht
```

open calls your handler function with the following syntax:

```
opencht('myfigure.cht')
```

See Also

load, save, saveas, uiopen, which, file_formats, path

openfig

Purpose Open new copy or raise existing copy of saved figure

Syntax

```
openfig('filename.fig','new')
openfig('filename.fig','reuse')
openfig('filename.fig')
openfig('filename.fig','new','invisible')
openfig('filename.fig','new','visible')
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','new','invisible')` or `openfig('filename.fig','reuse','invisible')` opens the figure as in the preceding example, while forcing the figure to be invisible.

`openfig('filename.fig','new','visible')` or `openfig('filename.fig','new','visible')` opens the figure, while forcing the figure to be visible.

`openfig('filename.fig','reuse')` opens the figure contained in the FIG-file only if a copy is not currently open; otherwise `openfig` brings the existing copy forward, making sure it is still visible and completely on screen.

`openfig('filename.fig')` is the same as `openfig('filename.fig','new')`.

`openfig(...,'PropertyName',PropertyValue,...)` opens the FIG-file setting the specified figure properties before displaying the figure.

`figure_handle = openfig(...)` returns the handle to the figure.

Remarks

If the FIG-file contains an invisible figure, `openfig` returns its handle and leaves it invisible. The caller should make the figure visible when appropriate.

See Also

`guide`, `guihandles`, `movegui`, `open`, `hgload`, `save`

See "Deploying User Interfaces" in the MATLAB documentation for related functions

See "Understanding the Application M-File" in the MATLAB documentation for information on how to use `openfig`.

opengl

Purpose Change automatic selection mode of OpenGL rendering

Syntax

```
opengl selection_mode
opengl info
s = opengl data
```

Description The OpenGL autoselection mode applies when the `RendererMode` of the figure is `auto`. Possible values for *`selection_mode`* are

- `autoselect` – allows OpenGL to be automatically selected if OpenGL is available and if there is graphics hardware on the host machine.
- `neverselect` – disables autoselection of OpenGL.
- `advise` – prints a message to the command window if OpenGL rendering is advised, but `RenderMode` is set to `manual`.

`opengl`, by itself, returns the current autoselection state.

`opengl info` prints information with the version and vendor of the OpenGL on your system.

`s = opengl data` returns a structure containing the same data that is displayed when you call `opengl info`.

Note that the autoselection 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 the figure to `OpenGL`. For example,

```
set(gcf,'Renderer','OpenGL')
```

See Also Figure `Renderer` property

Purpose

Open workspace variable in the Array Editor or other tool for graphical editing

Graphical Interface

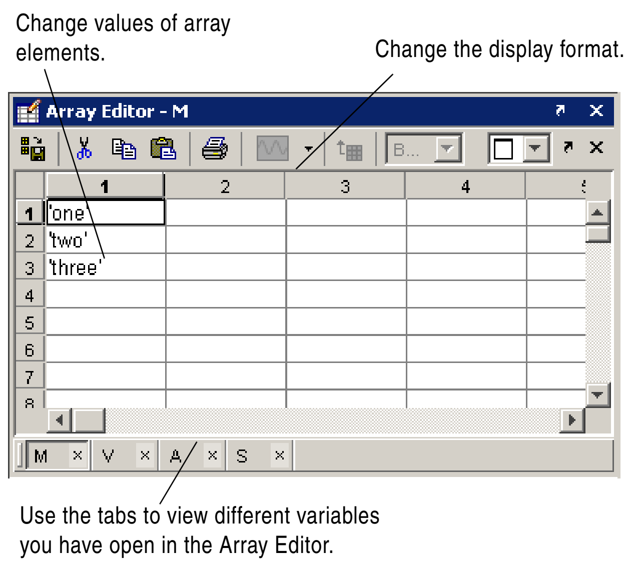
As an alternative to the `openvar` function, double-click on a variable in the Workspace browser.

Syntax

```
openvar('name')
```

Description

`openvar('name')` opens the workspace variable `name` in the Array Editor for graphical editing, where `name` is a numeric array, string, or cell array of strings. For some toolboxes, `openvar` instead opens a tool appropriate for viewing or editing that type of object.



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 an optimization `options` structure

Syntax

```
options = optimset('param1',value1,'param2',value2,...)
optimset
options = optimset
options = optimset(optimfun)
options = optimset(oldopts,'param1',value1,...)
options = optimset(oldopts,newopts)
```

Description

The function optimset creates an `options` structure that you can pass as an input argument to the following four MATLAB optimization functions:

- `fminbnd`
- `fminsearch`
- `fzero`
- `lsqnonneg`

You can use the `options` structure to change the default parameters for these functions.

Note If you have purchased the Optimization Toolbox, you can also use `optimset` to create an expanded `options` structure containing additional options specifically designed for the functions provided in that toolbox. See the reference page for the enhanced `optimset` function in the Optimization Toolbox for more information about these additional options.

`options = optimset('param1',value1,'param2',value2,...)` creates an optimization options structure called `options`, in which the specified parameters (`param`) have specified values. Any unspecified parameters are set to `[]` (parameters with value `[]` indicate to use the default value for that parameter when `options` is passed to the optimization function). It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`optimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = optimset` (with no input arguments) creates an options structure `options` where all fields are set to `[]`.

`options = optimset(optimfun)` creates an options structure `options` with all parameter names and default values relevant to the optimization function `optimfun`.

`options = optimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = optimset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

Options

The following table lists the available options for the MATLAB optimization functions.

Option	Value	Description
`Display`	`'off'` | `'iter'` | `{'final'}` | `'notify'`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` displays output only if the function does not converge.
`FunValCheck`	`{'off'}` | `'on'`	Check whether objective function values are valid. `'on'` displays a warning when the objective function returns a value that is `complex` or `NaN`. `'off'` displays no warning.
`MaxFunEvals`	positive integer	Maximum number of function evaluations allowed.
`MaxIter`	positive integer	Maximum number of iterations allowed.

Option	Value	Description
`OutputFcn`	function | `{[]}`	User-defined function that an opimization function calls at each iteration.
`TolFun`	positive scalar	Termination tolerance on the function value.
`TolX`	positive scalar	Termination tolerance on x.

Examples

This statement creates an optimization options structure called `options` in which the `Display` parameter is set to `'iter'` and the `TolFun` parameter is set to `1e-8`.

```
options = optimset('Display','iter','TolFun',1e-8)
```

This statement makes a copy of the options structure called `options`, changing the value of the `TolX` parameter and storing new values in `optnew`.

```
optnew = optimset(options,'TolX',1e-4);
```

This statement returns an optimization options structure that contains all the parameter names and default values relevant to the function `fminbnd`.

```
optimset('fminbnd')
```

See Also

`optimset` (Optimization Toolbox version), `optimget`, `fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`

orderfields

Purpose Order fields of a structure array

Syntax

```
s = orderfields(s1)
s = orderfields(s1, s2)
s = orderfields(s1, c)
s = orderfields(s1, perm)
[s, perm] = orderfields(...)
```

Description

`s = orderfields(s1)` orders the fields in `s1` so that the new structure array `s` has field names in ASCII dictionary order.

`s = orderfields(s1, s2)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in `s2`. Structures `s1` and `s2` must have the same fields.

`s = orderfields(s1, c)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in the cell array of field name strings `c`. Structure `s1` and cell array `c` must contain the same field names.

`s = orderfields(s1, perm)` orders the fields in `s1` so that the new structure array `s` has fieldnames in the order specified by the indices in permutation vector `perm`.

If `s1` has `N` fieldnames, the elements of `perm` must be an arrangement of the numbers from `1` to `N`. This is particularly useful if you have more than one structure array that you would like to reorder in the same way.

`[s, perm] = orderfields(...)` returns a permutation vector representing the change in order performed on the fields of the structure array that results in `s`.

Remarks `orderfields` only orders top-level fields. It is not recursive.

Examples Create a structure `s`. Then create a new structure from `s`, but with the fields ordered alphabetically:

```
s = struct('b', 2, 'c', 3, 'a', 1)
s =
    b: 2
```

```
    c: 3
    a: 1

snew = orderfields(s)
snew =
    a: 1
    b: 2
    c: 3
```

Arrange the fields of s in the order specified by the second (cell array) argument of `orderfields`. Return the new structure in `snew` and the permutation vector used to create it in `perm`:

```
[snew, perm] = orderfields(s, {'b', 'a', 'c'})
snew =
    b: 2
    a: 1
    c: 3
perm =
     1
     3
     2
```

Now create a new structure, `s2`, having the same fieldnames as `s`. Reorder the fields using the permutation vector returned in the previous operation:

```
s2 = struct('b', 3, 'c', 7, 'a', 4)
s2 =
    b: 3
    c: 7
    a: 4

snew = orderfields(s2, perm)
snew =
    b: 3
    a: 4
    c: 7
```

See Also

`struct`, `fieldnames`, `setfield`, `getfield`, `isfield`, `rmfield`, dynamic field names

ordqz

Purpose

Reorder eigenvalues in QZ factorization

Syntax

```
[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)
[...] = ordqz(AA,BB,Q,Z,keyword)
[...] = ordqz(AA,BB,Q,Z,clusters)
```

Description

`[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)` reorders the QZ factorizations `Q*A*Z = AA` and `Q*B*Z = BB` produced by the `qz` function for a matrix pair `(A,B)`. It returns the reordered pair `(AAS,BBS)` and the cumulative orthogonal transformations `QS` and `ZS` such that `QS*A*ZS = AAS` and `QS*B*ZS = BBS`. In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular pair `(AAS,BBS)`, and the corresponding invariant subspace is spanned by the leading columns of `ZS`. The logical vector `select` specifies the selected cluster as `E(select)` where `E = eig(AA,BB)`. Set `Q = []` or `Z = []` to get the incremental `QS` and `ZS` that transforms `(AA,BB)` into `(AAS,BBS)`.

`[...] = ordqz(AA,BB,Q,Z,keyword)` sets the selected cluster to include all eigenvalues in the region specified by `keyword`:

keyword	Selected Region
`'lhp'`	Left-half plane (`real(E) < 0`)
`'rhp'`	Right-half plane (`real(E) > 0`)
`'udi'`	Interior of unit disk (`abs(E) < 1`)
`'udo'`	Exterior of unit disk (`abs(E) > 1`)

`[...] = ordqz(AA,BB,Q,Z,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices commensurate with `E = eig(AA,BB)`, such that all eigenvalues with the same `clusters` value form one cluster, `ordqz` sorts the specified clusters in descending order along the diagonal of `(AAS,BBS)`. The cluster with highest index appears in the upper left corner.

See Also

`eig`, `ordschur`, `qz`

Purpose

Reorder eigenvalues in Schur factorization

Syntax

```
[US,TS] = ordschur(U,T,select)
[US,TS] = ordschur(U,T,keyword)
[US,TS] = ordschur(U,T,clusters)
```

Description

`[US,TS] = ordschur(U,T,select)` reorders the Schur factorization `X = U*T*U'` produced by the `schur` function and returns the reordered Schur matrix `TS` and the cumulative orthogonal transformation `US` such that `X = US*TS*US'`. In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular Schur matrix `TS`, and the corresponding invariant subspace is spanned by the leading columns of `US`. The logical vector `select` specifies the selected cluster as `E(select)` where `E = eig(T)`. Set `U = []` to get the incremental transformation `T = US*TS*US'`.

`[US,TS] = ordschur(U,T,keyword)` sets the selected cluster to include all eigenvalues in one of the following regions:

keyword	Selected Region
`'lhp'`	Left-half plane (`real(E) < 0`)
`'rhp'`	Right-half plane (`real(E) > 0`)
`'udi'`	Interior of unit disk (`abs(E) < 1`)
`'udo'`	Exterior of unit disk (`abs(E) > 1`)

`[US,TS] = ordschur(U,T,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices, commensurate with `E = eig(T)`, and such that all eigenvalues with the same `clusters` value form one cluster, `ordschur` sorts the specified clusters in descending order along the diagonal of `TS`, the cluster with highest index appearing in the upper left corner.

See Also

`eig`, `ordqz`, `schur`

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: `portrait`, `landscape`, or `tall`.

`orient landscape` sets the paper orientation of the current figure to full-page landscape, orienting the longest page dimension horizontally. The figure is centered on the page and scaled to fit the page with a 0.25 inch border.

`orient portrait` sets the paper orientation of the current figure to portrait, orienting the longest page dimension vertically. The `portrait` option returns the page orientation to the MATLAB default. (Note that the result of using the `portrait` option is affected by changes you make to figure properties. See the "Algorithm" section for more specific information.)

`orient tall` maps the current figure to the entire page in portrait orientation, leaving a 0.25 inch border.

`orient(fig_handle)`, `orient(simulink_model)` returns the current orientation of the specified figure or Simulink model.

`orient(fig_handle,`*`orientation`*`)`, `orient(simulink_model,`*`orientation`*`)` sets the orientation for the specified figure or Simulink model to the specified orientation (`landscape`, `portrait`, or `tall`).

Algorithm

`orient` sets the `PaperOrientation`, `PaperPosition`, and `PaperUnits` properties of the current figure. Subsequent print operations use these properties. The result of using the `portrait` option can be affected by default property values as follows:

- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the current values of `PaperOrientation` and `PaperPosition` to place the figure on the page.

- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the default figure `PaperPosition` with the x, y and width, height values reversed (i.e., [y,x,height,width]) to position the figure on the page.
- If the current figure `PaperType` is different from the default figure `PaperType`, then the `orient portrait` command uses the current figure `PaperPosition` with the x, y and width, height values reversed (i.e., [y,x,height,width]) to position the figure on the page.

See Also

`print`, `set`

`PaperOrientation`, `PaperPosition`, `PaperSize`, `PaperType`, and `PaperUnits` properties of figure graphics objects

"Printing" for related functions

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`

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) matches the switch expression (sw_expr).

Examples The general form of the switch statement is

```
switch sw_expr
    case case_expr
        statement
        statement
    case {case_expr1,case_expr2,case_expr3}
        statement
        statement
    otherwise
        statement
        statement
end
```

See switch for more details.

See Also switch

Index

D

E

I

M

P

R

S

T

Z