MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

Language Reference Manual

Version 5

How to Contact The MathWorks:

	(508) 647-7000	Phone
	(508) 647-7001	Fax
	(508) 647-7022	Technical Support Faxback Server
	The MathWorks, Inc. 24 Prime Park Way Natick, MA 01760-1500	Mail
	`http://www.mathworks.com`	Web
	`ftp.mathworks.com`	Anonymous FTP server
	`support@mathworks.com`	Technical support
	`suggest@mathworks.com`	Product enhancement suggestions
	`bugs@mathworks.com`	Bug reports
	`doc@mathworks.com`	Documentation error reports
	`subscribe@mathworks.com`	Subscribing user registration
	`service@mathworks.com`	Order status, license renewals, passcodes
	`info@mathworks.com`	Sales, pricing, and general information

Language Reference Manual (November 1996)

Preface

What Is MATLAB?

MATLAB® is a technical computing environment for high-performance numeric computation and visualization. MATLAB integrates numerical analysis, matrix computation, signal processing, and graphics in an easy-to-use environment where problems and solutions are expressed just as they are written mathematically – without traditional programming.

The name MATLAB stands for *matrix laboratory*. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represent the state of the art in software for matrix computation.

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many numerical problems in a fraction of the time it would take to write a program in a language such as Fortran, Basic, or C.

MATLAB has evolved over a period of years with input from many users. In university environments, it has become the standard instructional tool for introductory courses in applied linear algebra, as well as advanced courses in other areas. In industrial settings, MATLAB is used for research and to solve practical engineering and mathematical problems. Typical uses include general purpose numeric computation, algorithm prototyping, and special purpose problem solving with matrix formulations that arise in disciplines such as automatic control theory, statistics, and digital signal processing (time-series analysis).

MATLAB also features a family of application-specific solutions that we call *toolboxes*. Very important to most users of MATLAB, toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment in order to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems design, dynamic systems simulation, systems identification, neural networks, and others.

Probably the most important feature of MATLAB, and one that we took care to perfect, is its easy extensibility. This allows you to become a contributing author too, creating your own applications. In the years that MATLAB has been available, we have enjoyed watching many scientists, mathematicians, and engineers develop new and interesting applications, all without writing a single line of Fortran or other low-level code.

Who Wrote MATLAB?

The original MATLAB was written in Fortran by Cleve Moler, in an evolutionary process over several years. The underlying matrix algorithms are from the many people who worked on the LINPACK and EISPACK projects.

The current MATLAB program was written in C by The MathWorks. The first release was written by Steve Bangert, who wrote the parser/interpreter, Steve Kleiman, who implemented the graphics, and John Little and Cleve Moler, who wrote the analytical routines, the user's guide, and most of the M-files. Since the first release, many other people have joined the MATLAB development team and have made substantial contributions.

MATLAB Documentation

MATLAB comes with an extensive set of both online and printed documentation. The online *MATLAB Function Reference* is a compendium of all MATLAB language and graphics commands plus mathematical functions. You can access this documentation from the MATLAB Help Desk. Users on all platforms can access this facility via the doc command.Windows and Macintosh users can additionally access this facility via the **Help** menu or the **?** icon on the Command Window toolbar. From the Help Desk main menu, choose "MATLAB Functions " to display the *Function Reference.*

The online documentation is augmented with a full set of printed documents, consisting of the following titles:

- *Getting Started with MATLAB* , which explains how to get started with the fundamentals of MATLAB.
- *Using MATLAB*, which explains how to use MATLAB as both a programming language and a command-line application.
- *MATLAB Graphics* , which describes how to use MATLAB's graphics and visualization tools.
- *MATLAB Application Program Interface Guide* , which explains how to write C or Fortran programs that interact with MATLAB.
- *MATLAB 5 New Features Guide* , which provides information useful in making the transition from MATLAB 4 to MATLAB 5.
- *MATLAB Installation Guide* , which decribes how to install MATLAB on your platform.

If one or more of the printed documents is unavailable to you, you can locate an online version of the same document via the Help Desk.

Contents

Preface

1 Command Summary

1

Command Summary

This chapter lists MATLAB commands by functional area.

General Purpose Commands

Managing Commands and Functions

`addpath`	Add directories to MATLAB's search path	page 2-23
`doc`	Load hypertext documentation	page 2-194
`help`	Online help for MATLAB functions and M-files	page 2-337
`lasterr`	Last error message	page 2-394
`lookfor`	Keyword search through all help entries	page 2-411
`path`	Control MATLAB's directory search path	page 2-490
`profile`	Measure and display M-file execution profiles	page 2-520
`rmpath`	Remove directories from MATLAB's search path	page 2-556
`type`	List file	page 2-664
`version`	MATLAB version number	page 2-672
`what`	Directory listing of M-files, MAT-files, and MEX-files	page 2-680
`whatsnew`	Display README files for MATLAB and toolboxes	page 2-681
`which`	Locate functions and files	page 2-682

Managing Variables and the Workspace

`clear`	Remove items from memory	page 2-108
`disp`	Display text or array	page 2-190
`length`	Length of vector	page 2-399
`load`	Retrieve variables from disk	page 2-402
`pack`	Consolidate workspace memory	page 2-486
`save`	Save workspace variables on disk	page 2-565
`size`	Array dimensions	page 2-580
`who, whos`	List directory of variables in memory	page 2-685

Controlling the Command Window

`echo`	Echo M-files during execution	page 2-197
`format`	Control the output display format	page 2-267
`more`	Control paged output for the command window	page 2-435

Working with Files and the Operating Environment

`applescript`	Load a compiled AppleScript from a file and execute it	page 2-32
`cd`	Change working directory	page 2-85
`delete`	Delete files and graphics objects	page 2-183
`diary`	Save session in a disk file	page 2-186
`dir`	Directory listing	page 2-189
`edit`	Edit an M-file	page 2-198

Starting and Quitting MATLAB

Operators and Special Characters

Logical Functions

Language Constructs and Debugging

MATLAB as a Programming Language

Control Flow

Interactive Input

Object-Oriented Programming

Debugging

Elementary Matrices and Matrix Manipulation

Elementary Matrices and Arrays

Special Variables and Constants

Time and Dates

Matrix Manipulation

Specialized Matrices

Elementary Math Functions

Specialized Math Functions

Coordinate System Conversion

Matrix Functions - Numerical Linear Algebra

Matrix Analysis

Linear Equations

Eigenvalues and Singular Values

Matrix Functions

Low Level Functions

Data Analysis and Fourier Transform Functions

Basic Operations

Finite Differences

Correlation

Filtering and Convolution

Fourier Transforms

Vector Functions

Polynomial and Interpolation Functions

Polynomials

Data Interpolation

Function Functions – Nonlinear Numerical Methods

Sparse Matrix Functions

Elementary Sparse Matrices

Full to Sparse Conversion

Working with Nonzero Entries of Sparse Matrices

Visualizing Sparse Matrices

Reordering Algorithms

Norm, Condition Number, and Rank

Sparse Systems of Linear Equations

Sparse Eigenvalues and Singular Values

Miscellaneous

Sound Processing Functions

General Sound Functions

SPARCstation-specific Sound Functions

.WAV Sound Functions

Character String Functions

General

String Manipulation

String to Number Conversion

Radix Conversion

Low-Level File I/O Functions

File Opening and Closing

Unformatted I/O

Structure Functions

Object Functions

Cell Array Functions

Multidimensional Array Functions

`squeeze`	Remove singleton dimensions	page 2-615
`sub2ind`	Single index from subscripts	page 2-635

Graphics Functions

Color Operations and Lighting

`brighten`	Brighten or darken color map
`caxis`	Pseudocolor axis scaling
`colorbar`	Display color bar (color scale)
`colorcube`	Enhanced color-cube color map
`colordef`	Set up color defaults
`colormap`	Set the color look-up table
`diffuse`	Diffuse reflectance
`graymon`	Graphics figure defaults set for gray-scale monitor
`hsv2rgb`	Hue-saturation-value to red-green-blue conversion
`lighting`	Lighting mode
`material`	Material reflectance mode
`rgb2hsv`	RGB to HSVconversion
`rgbplot`	Plot color map
`shading`	Color shading mode
`specular`	Specular reflectance
`spinmap`	Spin the colormap
`surfnorm`	3-D surface normals
`whitebg`	Change axes background color for plots

Colormaps

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

`winter`	Shades of blue and green color map

Basic Plots and Graphs

`bar`	Vertical bar chart
`barh`	Horizontal bar chart
`hist`	Plot histograms
`hold`	Hold current graph
`loglog`	Plot using log-log scales
`pie`	Pie plot
`plot`	Plot vectors or matrices.
`polar`	Polar coordinate plot
`semilogx`	Semi-log scale plot
`semilogy`	Semi-log scale plot
`subplot`	Create axes in tiled positions

Hardcopy/File Output

`hardcopy`	Save figure window to file
`orient`	Hardcopy paper orientation
`print`	Print graph or save graph to file
`printopt`	Configure local printer defaults
`savtoner`	Modify graphic objects to print on a white background

Surface, Mesh, and Contour Plots

`contour`	Contour (level curves) plot.
`contourc`	Contour computation
`contourf`	Filled contour plot
`hidden`	Mesh hidden line removal mode
`meshc`	Combination mesh/contourplot
`mesh`	3-D mesh with reference plane
`surf`	3-D shaded surface graph
`surface`	Create surface low-level objects
`surfc`	Combination surf/contourplot
`surfl`	3-D shaded surface with lighting
`trimesh`	Triangular mesh plot
`trisurf`	Triangular surface plot

Domain Generation for Function Visualization

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

Specialized Plotting

`area`	Area plot
`box`	Axis box for 2-D and 3-D plots
`comet`	Comet plot
`compass`	Compass plot
`errorbar`	Plot graph with error bars
`ezplot`	Easy to use function plotter
`feather`	Feather plot
`fill`	Draw filled 2-D polygons
`fplot`	Plot a function
`pareto`	Pareto chart
`pie3`	3-D Pie plot
`plotmatrix`	Scatter plot matrix
`pcolor`	Pseudocolor (checkerboard) plot
`rose`	Plot rose or angle histogram
`quiver`	Quiver (or velocity) plot
`ribbon`	Ribbon plot
`stairs`	Stairstep graph
`stem`	Plot discrete sequence data

Three-Dimensional Plotting

`bar3`	Vertical 3-D bar chart
`bar3h`	Horizontal 3-D bar chart
`comet3`	3-D Comet plot
`cylinder`	Generate cylinder
`fill3`	Draw filled 3-D polygons in 3-space
`plot3`	Plot lines and points in 3-D space
`quiver3`	3-D Quiver (or velocity) plot
`slice`	Volumetric slice plot
`sphere`	Generate sphere
`stem3`	Plot discrete surface data
`view`	3-D graph viewpoint specification.
`viewmtx`	Generate view transformation matrices
`waterfall`	Waterfall plot

Plot Annotation and Grids

`clabel`	Add contour labels to a contour plot
`datetick`	Date formatted tick labels
`grid`	Grid lines for 2-D and 3-D plots
`gtext`	Place text on a 2-D graph using a mouse
`legend`	Graph legend for lines and patches

`plotyy`	Plot graphs with Y tick labels on the left and right
`title`	Titles for 2-D and 3-D plots
`xlabel`	X-axis labels for 2-D and 3-D plots
`ylabel`	Y-axis labels for 2-D and 3-D plots
`zlabel`	Z-axis labels for 3-D plots

Handle Graphics, General

`bwcontr`	Contrasting black and/or color
`copyobj`	Make a copy of a graphics object and its children
`findobj`	Find objects with specified property values
`gcbo`	Return object whose callback is currently executing
`gco`	Return handle of current object
`get`	Get object properties
`rotate`	Rotate objects about specified origin and direction
`ishandle`	True for graphics objects
`set`	Set object properties
`treediag`	Tree diagram of objects

Handle Graphics, Object Creation

`axes`	Create axis at arbitrary positions
`figure`	Create Figures (graph windows)
`image`	Display image (create image object)
`light`	Create light object
`line`	Create line low-level objects
`patch`	Create patch low-level objects
`text`	Add text to the current plot

Handle Graphics, Figure Windows

`capture`	Screen capture of the current figure
`clc`	Clear figure window
`clf`	Clear Figure
`clg`	Clear Figure (graph window)
`close`	Close specified window
`gcf`	Get current figure handle
`newplot`	Graphics M-file preamble for NextPlot property
`refresh`	Refresh figure

Handle Graphics, Axes

`axis`	Plot axis scaling and appearance

`cla`	Clear axis
`gca`	Get current axis handle

Object Manipulation

`propedit`	Edit all properties of any selected object
`reset`	Reset axis or figure
`rotate3d`	Interactively rotate the view of a 3-D plot
`selectmoveresize`	Interactively select, move, or resize objects
`shg`	Show graph window

Graphical User Interface Creation

`dialog`	Create a dialog box
`errordlg`	Create error dialog box
`helpdlg`	Display help dialog box
`inputdlg`	Create input dialog
`menu`	Generate a menu of choices for user input
`menuedit`	Menu Editor
`msgbox`	Create message dialog box
`questdlg`	Create question dialog box
`textwrap`	Return wrapped string matrix for given UI Control
`uicontrol`	Create user interface control
`uigetfile`	Display dialog box to retrieve name of file for reading
`uimenu`	Create user interface menu
`uiputfile`	Display dialog box to retrieve name of file for writing
`uiresume`	Used with `uiwait`, controls program execution
`uisetcolor`	Interactively set a ColorSpec via a dialog box
`uisetfont`	Interactively set a font by displaying a dialog box
`uiwait`	Used with `uiresume`, controls program execution
`waitbar`	Display wait bar
`waitforbuttonpress`	Wait for key/buttonpress over figure
`warndlg`	Create warning dialog box

Interactive User Input

`ginput`	Graphical input from a mouse or cursor
`zoom`	Zoom in and out on a 2-D plot

Interface Design

`algntool`	Align uicontrols and axes
`cbedit`	Callback Editor

`guide`	functions
`toolpal`	Initialization for Tool Palette

Region of Interest

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

Micellaneous Graphics Commands

`convhull`	Convex hull
`delaunay`	Delaunay triangulation
`dsearch`	Search Delaunay triangulation for nearest point
`inpolygon`	True for points inside a polygonal region
`polyarea`	Area of polygon
`tsearch`	Search for enclosing Delaunay triangle
`voronoi`	Voronoi diagram

2

Reference

This chapter describes all MATLAB operators, commands, and functions in alphabetical order.

Arithmetic Operators + - * / \ ^ '

Purpose Matrix and array arithmetic

Syntax

```
A+B
A–B
A*B     A.*B
A/B     A./B
A\B     A.\B
A^B     A.^B
A'      A.'
```

Description

+
-
*
/
\
^
'

MATLAB has two different types of arithmetic operations. Matrix arithmetic operations are defined by the rules of linear algebra. Array arithmetic operations are carried out element-by-element. The period character (.) distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition and subtraction, the character pairs `.+` and `.–` are not used.

`+` Addition or unary plus. `A+B` adds `A` and `B`. `A` and `B` must have the same size, unless one is a scalar. A scalar can be added to a matrix of any size.

`–` Subtraction or unary minus. `A–B` subtracts `B` from `A`. `A` and `B` must have the same size, unless one is a scalar. A scalar can be subtracted from a matrix of any size.

`*` Matrix multiplication. `C = A*B` is the linear algebraic product of the matrices `A` and `B`. More precisely,

$$C(i, j) = \sum_{k=1}^{n} A(i, k)B(k, j)$$

For nonscalar `A` and `B`, the number of columns of `A` must equal the number of rows of `B`. A scalar can multiply a matrix of any size.

`.*` Array multiplication. `A.*B` is the element-by-element product of the arrays `A` and `B`. `A` and `B` must have the same size, unless one of them is a scalar.

`/` Slash or matrix right division. `B/A` is roughly the same as `B*inv(A)`. More precisely, `B/A = (A'\B')'`. See `\`.

`./` Array right division. `A./B` is the matrix with elements `A(i,j)/B(i,j)`. `A` and `B` must have the same size, unless one of them is a scalar.

`\` Backslash or matrix left division. 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` components, or a matrix with several such columns, then `X = A\B` is the solution to the equation $AX = B$ computed by Gaussian elimination (see "Algorithm" for details). A warning message prints 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 $AX = B$. The effective rank, `k`, of `A`, is determined from the QR decomposition with pivoting (see "Algorithm" for details). A solution `X` is computed which has at most `k` nonzero components per column. If `k < n`, this is usually not the same solution as `pinv(A)*B`, which is the least squares solution with the smallest norm, $||X||$.

`.\` Array left division. `A.\B` is the matrix with elements `B(i,j)/A(i,j)`. `A` and `B` must have the same size, unless one of them is a scalar.

`^` Matrix power. `X^p` is `X` to the power `p`, if `p` is a scalar. If `p` is an integer, the power is computed by repeated multiplication. If the integer is negative, `X` is inverted first. For other values of `p`, the calculation involves eigenvalues and eigenvectors, such that if `[V,D] = eig(X)`, then `X^p = V*D.^p/V`.

If `x` is a scalar and `P` is a matrix, `x^P` is `x` raised to the matrix power `P` using eigenvalues and eigenvectors. `X^P`, where `X` and `P` are both matrices, is an error.

`.^` Array power. `A.^B` is the matrix with elements `A(i,j)` to the `B(i,j)` power. `A` and `B` must have the same size, unless one of them is a scalar.

`'` Matrix transpose. `A'` is the linear algebraic transpose of `A`. For complex matrices, this is the complex conjugate transpose.

`.'` Array transpose. `A.'` is the array transpose of `A`. For complex matrices, this does not involve conjugation.

Arithmetic Operators + - * / \ ^ '

Remarks

The arithmetic operators have M-file function equivalents, as shown:

Binary addition	`A+B`	`plus(A,B)`
Unary plus	`+A`	`uplus(A)`
Binary subtraction	`A–B`	`minus(A,B)`
Unary minus	`–A`	`uminus(A)`
Matrix multiplication	`A*B`	`mtimes(A,B)`
Array-wise multiplication	`A.*B`	`times(A,B)`
Matrix right division	`A/B`	`mrdivide(A,B)`
Array-wise right division	`A./B`	`rdivide(A,B)`
Matrix left division	`A\B`	`mldivide(A,B)`
Array-wise left division	`A.\B`	`ldivide(A,B)`
Matrix power	`A^B`	`mpower(A,B)`
Array-wise power	`A.^B`	`power(A,B)`
Complex transpose	`A'`	`ctranspose(A)`
Matrix transpose	`A.'`	`transpose(A)`

Examples

Here are two vectors, and the results of various matrix and array operations on them, printed with `format rat`.

Matrix Operations		Array Operations	
`x`	1 2 3	`y`	4 5 6
`x'`	1 2 3	`y'`	4 5 6
`x+y`	5 7 9	`x-y`	-3 -3 -3
`x + 2`	3 4 5	`x-2`	-1 0 1
`x * y`	Error	`x.*y`	4 10 18
`x'*y`	32	`x'.*y`	Error
`x*y'`	4 5 6 8 10 12 12 15 18	`x.*y'`	Error
`x*2`	2 4 6	`x.*2`	2 4 6
`x\y`	16/7	`x.\y`	4 5/2 2
`2\x`	1/2 1 3/2	`2./x`	2 1 2/3

Arithmetic Operators + - * / \ ^ '

Matrix Operations		Array Operations	
x/y	0 0 1/6 0 0 1/3 0 0 1/2	x./y	1/4 2/5 1/2
x/2	1/2 1 3/2	x./2	1/2 1 3/2
x^y	Error	x.^y	1 32 729
x^2	Error	x.^2	1 4 9
2^x	Error	2.^x	2 4 8
(x+i*y)'	1 – 4i 2 – 5i 3 – 6i		
(x+i*y).'	1 + 4i 2 + 5i 3 + 6i		

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`.

- If `A` is a triangular matrix, or a permutation of a triangular matrix, then `X` can be computed quickly by a permuted backsubstitution algorithm. The check for triangularity is done for full matrices by testing for zero elements and for sparse matrices by accessing the sparse data structure. Most nontriangular matrices are detected almost immediately, so this check requires a negligible amount of time.
- If `A` is symmetric, or Hermitian, and has positive diagonal elements, then a Cholesky factorization is attempted (see `chol`). If `A` is sparse, a symmetric minimum degree preordering is applied (see `symmmd` and `spparms`). 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 is

```
A = R'*R
```

where R is upper triangular. The solution X is computed by solving two triangular systems,

```
X = R\(R'\B)
```

- If A is square, but not a permutation of a triangular matrix, or is not Hermitian with positive elements, or the Cholesky factorization fails, then a general triangular factorization is computed by Gaussian elimination with partial pivoting (see lu). If A is sparse, a nonsymmetric minimum degree preordering is applied (see colmmd and spparms). 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 square and is full, 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 not square and is sparse, then the augmented matrix is formed by:

  ```
  S = [c*I A; A' 0]
  ```

  The default for the residual scaling factor is `c = max(max(abs(A)))/1000` (see spparms). The least squares solution X and the residual `R = B–A*X` are computed by

  ```
  S * [R/c; X] = [B; 0]
  ```

  with minimum degree preordering and sparse Gaussian elimination with numerical pivoting.

The various matrix factorizations are computed by MATLAB implementations of the algorithms employed by LINPACK routines ZGECO, ZGEFA and ZGESL for square matrices and ZQRDC and ZQRSL for rectangular matrices. See the *LINPACK Users' Guide* for details.

Diagnostics

From matrix division, if a square A is singular:

```
Matrix is singular to working precision.
```

From element-wise division, if the divisor has zero elements:

```
Divide by zero.
```

On machines without IEEE arithmetic, like the VAX, the above two operations generate the error messages shown. On machines with IEEE arithmetic, only warning messages are generated. The matrix division returns a matrix with each element set to Inf; the element-wise division produces NaNs or Infs where appropriate.

If the inverse was found, but is not reliable:

```
Warning: Matrix is close to singular or badly scaled.
    Results may be inaccurate.  RCOND = xxx
```

From matrix division, if a nonsquare A is rank deficient:

```
Warning: Rank deficient, rank = xxx tol = xxx
```

See Also

det	Matrix determinant
inv	Matrix inverse
lu	LU matrix factorization
orth	Range space of a matrix
qr	Orthogonal-triangular decomposition
rref	Reduced row echelon form

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose

Relational operations

Syntax

```
A < B
A > B
A <= B
A >= B
A == B
A ~= B
```

Description

The relational operators are <, ≤, >, ≥, ==, and ~=. Relational operators perform element-by-element comparisons between two arrays. They return an array of the same size, with elements set to logical true (1) where the relation is true, and elements set to logical false (0) where it is not.

The operators <, ≤, >, and ≥ use only the real part of their operands for the comparison. The operators == and ~= test real and imaginary parts.

The relational operators have precedence midway between the logical operators and the arithmetic operators.

To test if two strings are equivalent, use `strcmp`, which allows vectors of dissimilar length to be compared.

Examples

If one of the operands is a scalar and the other a matrix, the scalar expands to the size of the matrix. For example, the two pairs of statements:

```
X = 5; X >= [1 2 3; 4 5 6; 7 8 10]
X = 5*ones(3,3); X >= [1 2 3; 4 5 6; 7 8 10]
```

produce the same result:

```
ans =

    1    1    1
    1    1    0
    0    0    0
```

See Also

Logical Operators & | ~

`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzeros
`find`	Find indices and values of nonzero elements
`strcmp`	Compare strings

Purpose Logical operations

Syntax

```
A & B
A | B
~A
```

Description The symbols `&`, `|`, and `~` are the logical operators AND, OR, and NOT. They work element-wise on arrays, with 0 representing logical false (`F`), and anything nonzero representing logical true (`T`). The `&` operator does a logical AND, the `|` operator does a logical OR, and `~A` complements the elements of `A`. The function `xor(A,B)` implements the exclusive OR operation. Truth tables for these operators and functions follow.

Inputs A	B	and A&B	or A|B	xor xor(A,B)	NOT ~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

The logical operators have the lowest precedence, with arithmetic operators and relational operators being evaluated first.

The precedence for the logical operators with respect to each other is:

1. `not` has the highest precedence.
2. `and` and `or` have equal precedence, and are evaluated from left to right.

Remarks The logical operators have M-file function equivalents, as shown:

and	`A&B`	`and(A,B)`
or	`A|B`	`or(A,B)`
not	`~A`	`not(A)`

Examples

Here are two scalar expressions that illustrate precedence relationships for arithmetic, relational, and logical operators:

```
1 & 0 + 3
3 > 4 & 1
```

They evaluate to 1 and 0 respectively, and are equivalent to:

```
1 & (0 + 3)
(3 > 4) & 1
```

Here are two examples that illustrate the precedence of the logical operators to each other:

```
1 | 0 & 0 = 0
0 & 0 | 1 = 1
```

See Also

The relational operators: <, <=, >, >=, ==, ~=, as well as:

`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzeros
`find`	Find indices and values of nonzero elements
`logical`	Convert numeric values to logical
`xor`	Exclusive or

Purpose Special characters

Syntax `[ ] ( ) {} = ' . ... , ; % !`

Description

`[ ]` Brackets are used to form vectors and matrices. `[6.9 9.64 sqrt(-1)]` is a vector with three elements separated by blanks. `[6.9, 9.64, i]` is the same thing. `[1+j 2-j 3]` and `[1 +j 2 -j 3]` are not the same. The first has three elements, the second has five.

`[11 12 13; 21 22 23]` is a 2-by-3 matrix. The semicolon ends the first row.

Vectors and matrices can be used inside `[ ]` brackets. `[A B;C]` is allowed if the number of rows of `A` equals the number of rows of `B` and the number of columns of `A` plus the number of columns of `B` equals the number of columns of `C`. This rule generalizes in a hopefully obvious way to allow fairly complicated constructions.

`A = [ ]` stores an empty matrix in `A`. `A(m,:) = [ ]` deletes row `m` of `A`. `A(:,n) = [ ]` deletes column `n` of `A`. `A(n) = [ ]` reshapes `A` into a column vector and deletes the third element.

`[A1,A2,A3...] = function` assigns function output to multiple variables.

For the use of `[` and `]` on the left of an "=" in multiple assignment statements, see `lu`, `eig`, `svd`, and so on.

`{ }` Curly braces are used in cell array assignment statements. For example.,

`A(2,1) = {[1 2 3; 4 5 6]}`, or `A{2,2} = ('str')`. See `help paren` for more information about `{ }`.

`( )` Parentheses are used to indicate precedence in arithmetic expressions in the usual way. They are used to enclose arguments of functions in the usual way. They are also used to enclose subscripts of vectors and matrices in a manner somewhat more general than usual. If `X` and `V` are vectors, then `X(V)` is `[X(V(1)), X(V(2)), ..., X(V(n))]`. The components of `V` must be integers to be used as subscripts. An error occurs if any such subscript is less than 1 or greater than the size of `X`. Some examples are

- `X(3)` is the third element of `X`.
- `X([1 2 3])` is the first three elements of `X`.

See `help paren` for more information about `( )`.

If `X` has `n` components, `X(n:–1:1)` reverses them. The same indirect subscripting works in matrices. If `V` has `m` components and `W` has `n` components, then `A(V,W)` is the m-by-n matrix formed from the elements of `A` whose subscripts are the elements of `V` and `W`. For example, `A([1,5],:) = A([5,1],:)` interchanges rows 1 and 5 of `A`.

`=` Used in assignment statements. `B = A` stores the elements of `A` in `B`. == is the relational equals operator. See the Relational Operators page.

`'` Matrix transpose. `X'` is the complex conjugate transpose of `X`. `X.'` is the nonconjugate transpose.

Quotation mark. `'any text'` is a vector whose components are the ASCII codes for the characters. A quotation mark within the text is indicated by two quotation marks.

`.` Decimal point. `314/100`, `3.14` and `.314e1` are all the same.
Element-by-element operations. These are obtained using `.*` , `.^` , `./`, or `.\`. See the Arithmetic Operators page.

`.` Field access. `A.(field)` and `A(i).field`, when `A` is a structure, access the contents of `field`.

`..` Parent directory. See `cd`.

`...` Continuation. Three or more points at the end of a line indicate continuation.

`,` Comma. Used to separate matrix subscripts and function arguments. Used to separate statements in multistatement lines. For multi-statement lines, the comma can be replaced by a semicolon to suppress printing.

`;` Semicolon. Used inside brackets to end rows. Used after an expression or statement to suppress printing or to separate statements.

`%` Percent. The percent symbol denotes a comment; it indicates a logical end of line. Any following text is ignored. MATLAB displays the first contiguous comment lines in a M-file in response to a `help` command.

`!` Exclamation point. Indicates that the rest of the input line is issued as a command to the operating system.

Remarks

Some uses of special characters have M-file function equivalents, as shown:

Horizontal concatenation	`[A,B,C...]`	`horzcat(A,B,C...)`
Vertical concatenation	`[A;B;C...]`	`vertcat(A,B,C...)`
Subscript reference	`A(i,j,k...)`	`subsref(A,S)`. See `help subsref`.
Subscript assignment	`A(i,j,k...)= B`	`subsasgn(A,S,B)`. See `help subsasgn`.

See Also

Arithmetic, relational, and logical operators.

Colon :

Purpose

Create vectors, array subscripting, and *for* iterations

Description

The colon is one of the most useful operators in MATLAB. It can create vectors, subscript arrays, and specify `for` iterations.

The colon operator uses the following rules to create regularly spaced vectors:

`j:k`	is the same as `[j,j+1,...,k]`
`j:k`	is empty if `j > k`
`j:i:k`	is the same as `[j,j+i,j+2i, ...,k]`
`j:i:k`	is empty if `i > 0` and `j > k` or if `i < 0` and `j < k`

where `i`,`j`, and `k` are all scalars.

Below are the definitions that govern the use of the colon to pick out selected rows, columns, and elements of vectors, matrices, and higher-dimensional arrays:

`A(:,j)`	is the `j`-th column of `A`
`A(i,:)`	is the `i`-th row of `A`
`A(:,:)`	is the equivalent two-dimensional array. For matrices this is the same as `A`.
`A(j:k)`	is `A(j), A(j+1),...,A(k)`
`A(:,j:k)`	is `A(:,j), A(:,j+1),...,A(:,k)`
`A(:,:,k)`	is the `k`th page of three-dimensional array `A`.
`A(i,j,k,:)`	is a vector in four-dimensional array `A`. The vector includes `A(i,j,k,1)`, `A(i,j,k,2)`, `A(i,j,k,3)`, and so on.
`A(:)`	is all the elements of `A`, regarded as a single column. On the left side of an assignment statement, `A(:)` fills `A`, preserving its shape from before. In this case, the right side must contain the same number of elements as `A`.

Examples

Using the colon with integers,

```
D = 1:4
```

results in

```
D =
     1     2     3     4
```

Using two colons to create a vector with arbitrary real increments between the elements,

```
E = 0:.1:.5
```

results in

```
E =
     0    0.1000    0.2000    0.3000    0.4000    0.5000
```

The command

```
A(:,:,2) = pascal(3)
```

generates a three-dimensional array whose first page is all zeros.

```
A(:,:,1) =
      0     0     0
      0     0     0
      0     0     0

A(:,:,2) =
      1     1     1
      1     2     3
      1     3     6
```

See Also

`for`	Repeat statements a specific number of times
`linspace`	Generate linearly spaced vectors
`logspace`	Generate logarithmically spaced vectors
`reshape`	Reshape array

abs

Purpose Absolute value and complex magnitude

Syntax

```
Y = abs(X)
```

Description `abs(X)` returns the absolute value, $|X|$, for each element of `X`.

If `X` is complex, `abs(X)` returns the complex modulus (magnitude):

```
abs(X) = sqrt(real(X).^2 + imag(X).^2)
```

Examples

```
abs(-5) = 5
abs(3+4i) = 5
```

See Also

`angle`	Phase angle
`sign`	Signum function
`unwrap`	Correct phase angles

Purpose

Inverse cosine and inverse hyperbolic cosine

Syntax

```
Y = acos(X)
Y = acosh(X)
```

Description

The acos and acosh functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

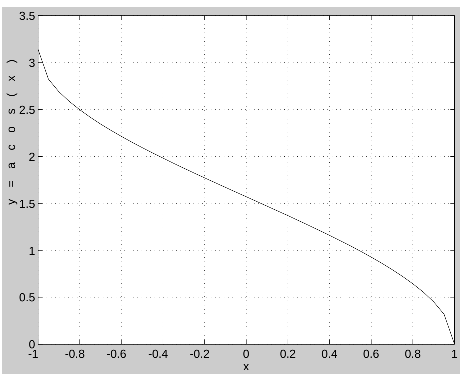
`Y = acos(X)` returns the inverse cosine (arccosine) for each element of `X`. For real elements of `X` in the domain $[-1, 1]$, `acos(X)` is real and in the range $[0, \pi]$. For real elements of `X` outside the domain $[-1, 1]$, `acos(X)` is complex.

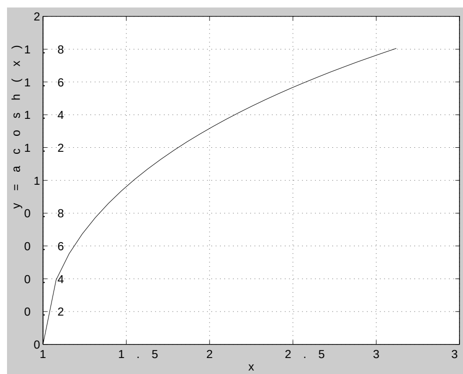
`Y = acosh(X)` returns the inverse hyperbolic cosine for each element of `X`.

Examples

Graph the inverse cosine function over the domain $-1 \le x \le 1$, and the inverse hyperbolic cosine function over the domain $1 \le x \le \pi$.

```
x = -1:.05:1; plot(x,acos(x))
x = 1:pi/40:pi; plot(x,acosh(x))
```





Algorithm

$$\cos^{-1}(z) = -i \log\left[z + i(1 - z^2)^{\frac{1}{2}}\right]$$

$$\cosh^{-1}(z) = \log\left[z + (z^2 - 1)^{\frac{1}{2}}\right]$$

See Also

`cos`, `cosh`	Cosine and hyperbolic cosine

acot, acoth

Purpose

Inverse cotangent and inverse hyperbolic cotangent

Syntax

```
Y = acot(X)
Y = acoth(X)
```

Description

The `acot` and `acoth` functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

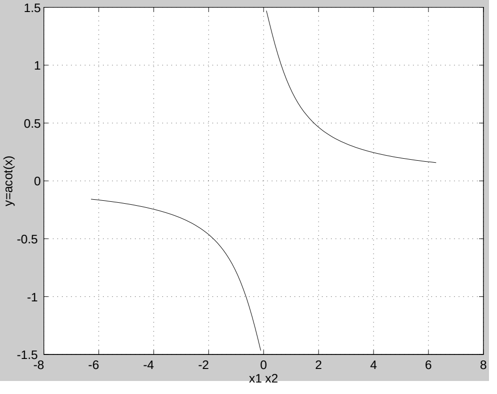
`Y = acot(X)` returns the inverse cotangent (arccotangent) for each element of `X`.

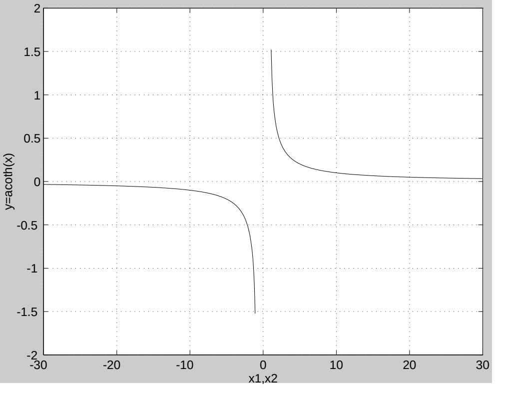
`Y = acoth(X)` returns the inverse hyperbolic cotangent for each element of `X`.

Examples

Graph the inverse cotangent over the domains $-2\pi \le x < 0$ and $0 < x \le 2\pi$, and the inverse hyperbolic cotangent over the domains $-30 \le x < -1$ and $1 < x \le 30$.

```
x1 = -2*pi:pi/30:-0.1; x2 = 0.1:pi/30:2*pi;
plot(x1,acot(x1),x2,acot(x2))
x1 = -30:0.1:-1.1; x2 = 1.1:0.1:30;
plot(x1,acoth(x1),x2,acoth(x2))
```





Algorithm

$$\cot^{-1}(z) = \tan^{-1}\left(\frac{1}{z}\right)$$

$$\coth^{-1}(z) = \tanh^{-1}\left(\frac{1}{z}\right)$$

See Also

`cot`, `coth` Cotangent and hyperbolic cotangent

Purpose

Inverse cosecant and inverse hyperbolic cosecant

Syntax

```
Y = acsc(X)
Y = acsch(X)
```

Description

The acsc and acsch functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

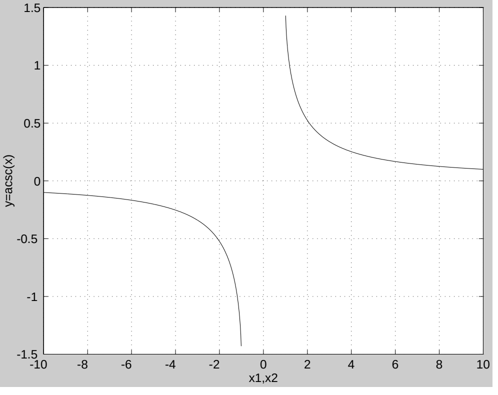
`Y = acsc(X)` returns the inverse cosecant (arccosecant) for each element of `X`.

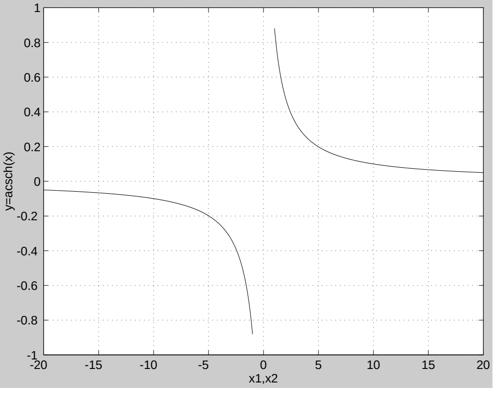
`Y = acsch(X)` returns the inverse hyperbolic cosecant for each element of `X`.

Examples

Graph the inverse cosecant over the domains $-10 \le x < -1$ and $1 < x \le 10$, and the inverse hyperbolic cosecant over the domains $-20 \le x \le -1$ and $1 \le x \le 20$.

```
x1 = -10:0.01:-1.01; x2 = 1.01:0.01:10;
plot(x1,acsc(x1),x2,acsc(x2))
x1 = -20:0.01:-1; x2 = 1:0.01:20;
plot(x1,acsch(x1),x2,acsch(x2))
```





Algorithm

$$\csc^{-1}(z) = \sin^{-1}\left(\frac{1}{z}\right)$$

$$\operatorname{csch}^{-1}(z) = \sinh^{-1}\left(\frac{1}{z}\right)$$

See Also

csc, csch	Cosecant and hyperbolic cosecant

Purpose Add directories to MATLAB's search path

Syntax

```
addpath('directory')
addpath('dir1','dir2','dir3',...)
addpath(...,'-flag')
```

Description

`addpath ('`*directory*`')` prepends the specified directory to MATLAB's current search path.

`addpath ('`*dir1*`','`*dir2*`','`*dir3*`',...)` prepends all the specified directories to the path.

`addpath (...,'-`*flag*`')` either prepends or appends the specified directories to the path depending the value of *flag*:

`0` or `begin`	Prepend specified directories
`1` or `end`	Append specified directories

Examples

```
path
              MATLABPATH
       c:\matlab\toolbox\general
       c:\matlab\toolbox\ops
       c:\matlab\toolbox\strfun

addpath('c:\matlab\myfiles')

path
             MATLABPATH
       c:\matlab\myfiles
       c:\matlab\toolbox\general
       c:\matlab\toolbox\ops
       c:\matlab\toolbox\strfun
```

See Also

`path`	Control MATLAB's directory search path
`rmpath`	Remove directories from MATLAB's search path

airy

Purpose

Airy functions

Syntax

```
W = airy(Z)
W = airy(k,Z)
[W,ierr] = airy(k,Z)
```

Definition

The Airy functions form a pair of linearly independent solutions to:

$$\frac{d^2W}{dZ^2} - ZW = 0$$

The relationship between the Airy and modified Bessel functions is:

$$Ai(Z) = \left[\frac{1}{\pi}\sqrt{Z/3}\right] K_{1/3}(\zeta)$$

where,

$$\zeta = \frac{2}{3}Z^{3/2}$$

Description

`W = airy(Z)` returns the Airy function, $Ai(Z)$, for each element of the complex array `Z`.

`W = airy(k,Z)` returns different results depending on the value of `k`:

k	Returns
`0`	The same result as `airy(Z)`.
`1`	The derivative, $Ai'(Z)$.
`2`	The Airy function of the second kind, $Bi(Z)$.
`3`	The derivative, $Bi'(Z)$.

`[W,ierr] = airy(k,Z)` also returns an array of error flags.

`ierr = 1`	Illegal arguments.
`ierr = 2`	Overflow. Return `Inf`.
`ierr = 3`	Some loss of accuracy in argument reduction.
`ierr = 4`	Unacceptable loss of accuracy, `Z` too large.
`ierr = 5`	No convergence. Return `NaN`.

See Also

`besseli`	Modified Bessel functions of the first kind
`besselj`	Bessel functions of the first kind
`besselk`	Modified Bessel functions of the third kind
`bessely`	Bessel functions of the second kind

References

[1] Amos, D. E., “A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order,” *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[2] Amos, D. E., “A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order,” *Trans. Math. Software*, 1986.

all

Purpose

Test to determine if all elements are nonzero

Syntax

```
B = all(A)
B = all(A,dim)
```

Description

`B = all(A)` tests whether *all* the elements along various dimensions of an array are nonzero or logical true (`1`).

If `A` is a vector, `all(A)` returns logical true (`1`) if all of the elements are nonzero, and returns logical false (`0`) if one or more elements are zero.

If `A` is a matrix, `all(A)` treats the columns of `A` as vectors, returning a row vector of `1`s and `0`s.

If `A` is a multidimensional array, `all(A)` treats the values along the first non-singleton dimension as vectors, returning a logical condition for each vector.

`B = all(A,`*`dim`*`)` tests along the dimension of `A` specified by scalar *dim*.

1	1	1
1	1	0

A

1	1	0

all(A,1)

1
0

all(A,2)

Examples

Given,

```
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69]
```

then `B = (A < 0.5)` returns logical true (`1`) only where `A` is less than one half:

```
0   0   1   1   1   1   0
```

The `all` function reduces such a vector of logical conditions to a single condition. In this case, `all(B)` yields `0`.

This makes `all` particularly useful in `if` statements,

```
if all(A < 0.5)
    do something
end
```

where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

Applying the `all` function twice to a matrix, as in `all(all(A))`, always reduces it to a scalar condition.

```
all(all(eye(3)))
ans =
   0
```

See Also

The logical operators: `&`, `|`, `~`, and:

any	Test for any nonzeros

Other functions that collapse an array's dimensions include:

`max`, `mean`, `median`, `min`, `prod`, `std`, `sum`, `trapz`

angle

Purpose Phase angle

Syntax

```
P = angle(Z)
```

Description

`P = angle(Z)` returns the phase angles, in radians, for each element of complex array `Z`. The angles lie between $\pm\pi$.

For complex `Z`, the magnitude and phase angle are given by

```
R = abs(Z)          % magnitude
theta = angle(Z)    % phase angle
```

and the statement

```
Z = R.*exp(i*theta)
```

converts back to the original complex `Z`.

Examples

```
Z =
1.0000 - 1.0000i   2.0000 + 1.0000i   3.0000 - 1.0000i   4.0000 + 1.0000i
1.0000 + 2.0000i   2.0000 - 2.0000i   3.0000 + 2.0000i   4.0000 - 2.0000i
1.0000 - 3.0000i   2.0000 + 3.0000i   3.0000 - 3.0000i   4.0000 + 3.0000i
1.0000 + 4.0000i   2.0000 - 4.0000i   3.0000 + 4.0000i   4.0000 - 4.0000i

P = angle(Z)
P =

   -0.7854    0.4636   -0.3218    0.2450
    1.1071   -0.7854    0.5880   -0.4636
   -1.2490    0.9828   -0.7854    0.6435
    1.3258   -1.1071    0.9273   -0.7854
```

Algorithm

`angle` can be expressed as:

```
angle(z) = imag(log(z)) = atan2(imag(z),real(z))
```

See Also

`abs`	Absolute value and complex magnitude
`unwrap`	Correct phase angles

Purpose The most recent answer

Syntax

```
ans
```

Description The ans variable is created automatically when no output argument is specified.

Examples The statement

```
2+2
```

is the same as

```
ans = 2+2
```

any

Purpose

Test for any nonzeros

Syntax

```
B = any(A)
B = any(A,dim)
```

Description

`B = any(A)` tests whether *any* of the elements along various dimensions of an array are nonzero or logical true (`1`).

If `A` is a vector, `any(A)` returns logical true (`1`) if any of the elements of `A` are nonzero, and returns logical false (`0`) if all the elements are zero.

If `A` is a matrix, `any(A)` treats the columns of `A` as vectors, returning a row vector of `1`s and `0`s.

If `A` is a multidimensional array, `any(A)` treats the values along the first non-singleton dimension as vectors, returning a logical condition for each vector.

`B = any(A,`*`dim`*`)` tests along the dimension of `A` specified by scalar *`dim`*.

1	0	1
0	0	0

A

1	0	1

any(A,1)

1
0

any(A,2)

Examples

Given,

```
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69]
```

then `B = (A < 0.5)` returns logical true (`1`) only where `A` is less than one half:

```
0    0    1    1    1    1    0
```

The `any` function reduces such a vector of logical conditions to a single condition. In this case, `any(B)` yields `1`.

This makes `any` particularly useful in `if` statements,

```
if any(A < 0.5)
    do something
end
```

where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

Applying the `any` function twice to a matrix, as in `any(any(A))`, always reduces it to a scalar condition.

```
any(any(eye(3)))
ans =
   1
```

See Also

The logical operators `&`, `|`, `~`, and:

`all`	Test to determine if all elements are nonzero

Other functions that collapse an array's dimensions include:

`max`, `mean`, `median`, `min`, `prod`, `std`, `sum`, `trapz`

applescript

Purpose

Load a compiled AppleScript from a file and execute it

Syntax

```
applescript(filename)
result = applescript(filename)
applescript(filename,'VarName1','VarValue1', ...)
```

Description

`applescript(`*`filename`*`)` loads a compiled AppleScript from the file *filename* and executes it. If *filename* is not a full path name, then `applescript` searches for *filename* along the MATLAB path.

`result = applescript(`*`filename`*`)` returns in `result` the value that the AppleScript returns, converted to a string.

`applescript(`*`filename`*`, 'VarName1', 'VarValue1',...)` sets the value of the AppleScript's property or variable whose name is specified in `VarName` to the value specified in `VarValue`.

Remarks

`applescript` is available on the Macintosh only.

Examples

Compile an AppleScript and save it to the file `rename`:

```
tell application "Finder"
   set name of item itemName to newName
end tell
```

The `applescript` command renames file `hello` on volume `MyDisk` to the new name `world`.

```
applescript('rename', 'itemName', '"MyDisk:hello"', ...
                      'newName', '"world"');
```

Purpose

Inverse secant and inverse hyperbolic secant

Syntax

```
Y = asec(X)
Y = asech(X)
```

Description

The asec and asech functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

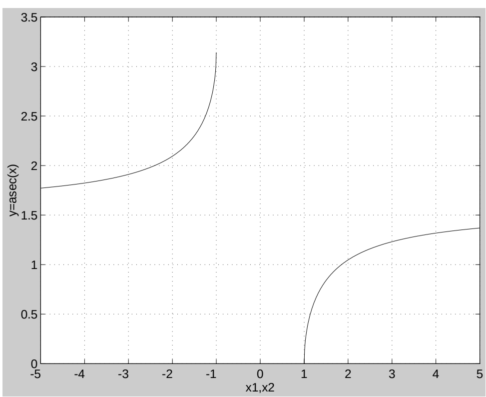
`Y = asec(X)` returns the inverse secant (arcsecant) for each element of `X`.

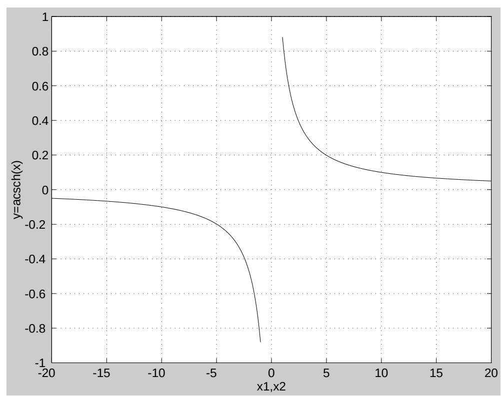
`Y = asech(X)` returns the inverse hyperbolic secant for each element of `X`.

Examples

Graph the inverse secant over the domains $1 \le x \le 5$ and $-5 \le x \le -1$, and the inverse hyperbolic secant over the domain $0 < x \le 1$.

```
x1 = -5:0.01:-1; x2 = 1:0.01:5;
plot(x1,asec(x1),x2,asec(x2))
x = 0.01:0.001:1; plot(x,asech(x))
```





Algorithm

$$\sec^{-1}(z) = \cos^{-1}\left(\frac{1}{z}\right)$$

$$\operatorname{sech}^{-1}(z) = \cosh^{-1}\left(\frac{1}{z}\right)$$

See Also

sec, sech	Secant and hyperbolic secant

asin, asinh

Purpose

Inverse sine and inverse hyperbolic sine

Syntax

```
Y = asin(X)
Y = asinh(X)
```

Description

The `asin` and `asinh` functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

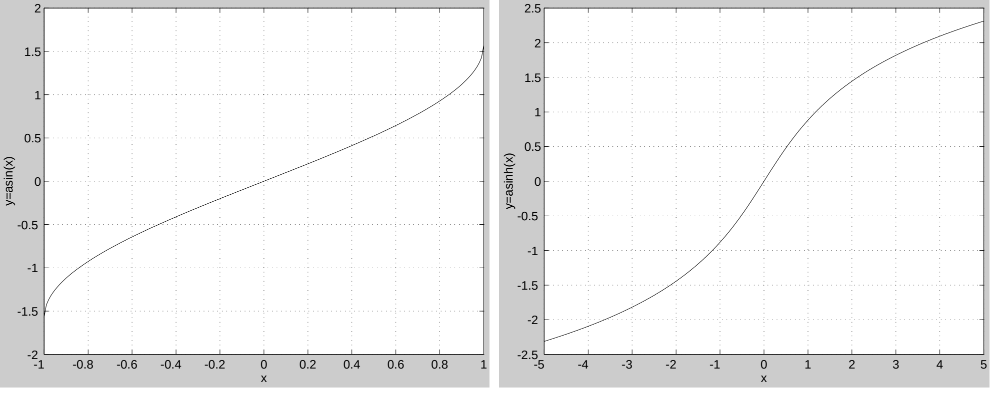
`Y = asin(X)` returns the inverse sine (arcsine) for each element of `X`. For real elements of `X` in the domain $[-1, 1]$, `asin(X)` is in the range $[-\pi/2, \pi/2]$. For real elements of `x` outside the range $[-1, 1]$, `asin(X)` is complex.

`Y = asinh(X)` returns the inverse hyperbolic sine for each element of `X`.

Examples

Graph the inverse sine function over the domain $-1 \le x \le 1$, and the inverse hyperbolic sine function over the domain $-5 \le x \le 5$.

```
x = -1:.01:1; plot(x,asin(x))
x = -5:.01:5; plot(x,asinh(x))
```



Algorithm

$$\sin^{-1}(z) = -i \log\left[iz + (1-z^2)^{\frac{1}{2}}\right]$$

$$\sinh^{-1}(z) = \log\left[z + (z^2+1)^{\frac{1}{2}}\right]$$

See Also

`sin`, `sinh` — Sine and hyperbolic sine

Purpose Assign value to variable in workspace

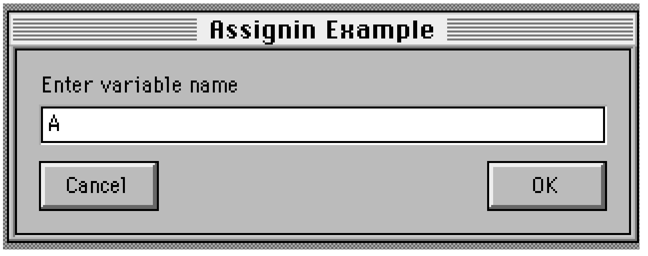
Syntax

```
assignin(ws,'name',v)
```

Description `assignin(ws,'name',v)` assigns the variable `'name'` in the workspace `ws` the value `v`. `'name'` is created if it doesn't exist. `ws` can be either `'caller'` or `'base'`.

Examples Here's a function that creates a variable with a user-chosen name in the base workspace. The variable is assigned the value $\sqrt{\pi}$.

```
function sqpi
var = inputdlg('Enter variable name','Assignin Example',1,{'A'})
assignin('base','var',sqrt(pi))
```



See Also

`evalin`	Evaluate expression in workspace.

atan, atanh

Purpose Inverse tangent and inverse hyperbolic tangent

Syntax

```
Y = atan(X)
Y = atanh(X)
```

Description The `atan` and `atanh` functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

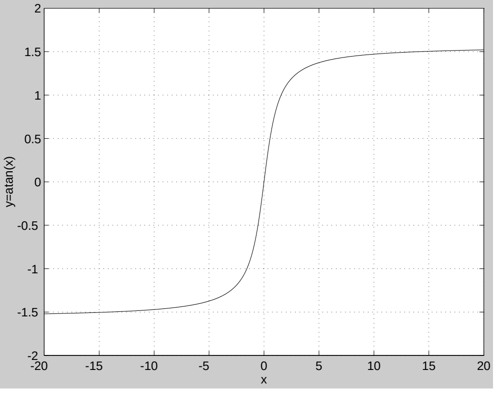
`Y = atan(X)` returns the inverse tangent (arctangent) for each element of `X`.

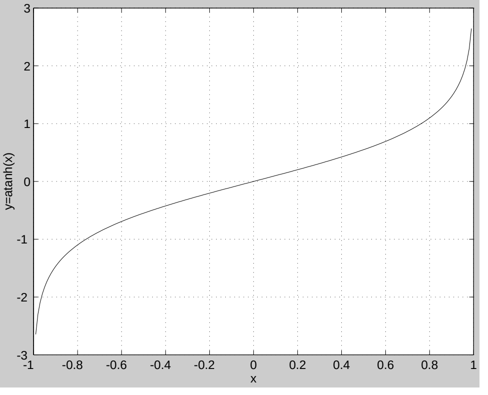
For real elements of `X`, `atan(X)` is in the range $[-\pi/2, \pi/2]$.

`Y = atanh(X)` returns the inverse hyperbolic tangent for each element of `X`.

Examples Graph the inverse tangent function over the domain $-20 \le x \le 20$, and the inverse hyperbolic tangent function over the domain $-1 < x < 1$.

```
x = -20:0.01:20; plot(x,atan(x))
x = -0.99:0.01:0.99; plot(x,atanh(x))
```





Algorithm

$$\tan^{-1}(z) = \frac{i}{2}\log\left(\frac{i+z}{i-z}\right)$$

$$\tanh^{-1}(z) = \frac{1}{2}\log\left(\frac{1+z}{1-z}\right)$$

See Also

`atan2` Four-quadrant inverse tangent

`tan`, `tanh`	Tangent and hyperbolic tangent

Purpose Four-quadrant inverse tangent

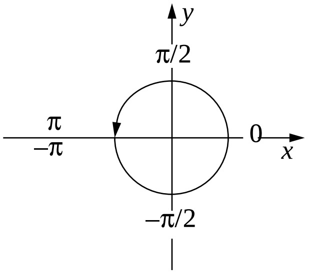
Syntax

```
P = atan2(Y,X)
```

Description

`P = atan2(Y,X)` returns an array `P` the same size as `X` and `Y` containing the element-by-element, four-quadrant inverse tangent (arctangent) of the real parts of `Y` and `X`. Any imaginary parts are ignored.

Elements of `P` lie in the half-open interval $[-\pi, \pi]$. The specific quadrant is determined by `sign(Y)` and `sign(X)`:



This contrasts with the result of `atan(Y/X)`, which is limited to the interval $[-\pi/2, \pi/2]$, or the right side of this diagram.

Examples

Any complex number $z = x+iy$ is converted to polar coordinates with

```
r = abs(z)
theta = atan2(imag(z),real(z))
```

To convert back to the original complex number:

```
z = r *exp(i *theta)
```

This is a common operation, so MATLAB provides a function, `angle(z)`, that simply computes `atan2(imag(z),real(z))`.

See Also

`atan`, `atanh`	Inverse tangent and inverse hyperbolic tangent
`tan`, `tanh`	Tangent and hyperbolic tangent

Purpose Read NeXT/SUN (.au) sound file

Syntax

```
y = auread(aufile)
[y,Fs,bits] = auread(aufile)
[...] = auread(aufile,N)
[...] = auread(aufile,[N1,N2])
siz = auread(aufile,'size')
```

Description Supports multi-channel data in the following formats:

- 8-bit mu-law
- 8-, 16-, and 32-bit linear
- floating-point

`y = auread(`*`aufile`*`)` loads a sound file specified by the string *aufile*, returning the sampled data in `y`. The `.au` extension is appended if no extension is given. Amplitude values are in the range `[-1,+1]`.

`[y,Fs,bits] = auread(`*`aufile`*`)` returns the sample rate (`Fs`) in Hertz and the number of bits per sample (`bits`) used to encode the data in the file.

`[...] = auread(`*`aufile`*`,N)` returns only the first `N` samples from each channel in the file.

`[...] = auread(`*`aufile`*`,[N1 N2])` returns only samples `N1` through `N2` from each channel in the file.

`siz = auread(`*`aufile`*`,'size')` returns the size of the audio data contained in the file in place of the actual audio data, returning the vector `siz = [samples channels]`.

See Also

`auwrite`	Write NeXT/SUN (`.au`) sound file
`wavread`	Read Microsoft WAVE (`.wav`) sound file

auwrite

Purpose

Write NeXT/SUN (.au) sound file

Syntax

```
auwrite(y,aufile)
auwrite(y,Fs,aufile)
auwrite(y,Fs,N,aufile)
auwrite(y,Fs,N,method,aufile)
```

Description

`auwrite` supports multi-channel data for 8-bit mu-law, and 8- and 16-bit linear formats.

`auwrite(y,`*aufile*`)` writes a sound file specified by the string *aufile*. The data should be arranged with one channel per column. Amplitude values outside the range `[-1,+1]` are clipped prior to writing.

`auwrite(y,Fs,`*aufile*`)` specifies the sample rate of the data in Hertz.

`auwrite(y,Fs,N,`*aufile*`)` selects the number of bits in the encoder. Allowable settings are `N = 8` and `N = 16`.

`auwrite(y,Fs,N,`*method*`,`*aufile*`)` allows selection of the encoding method, which can be either `'mu'` or `'linear'`. Note that mu-law files must be 8-bit. By default, `method='mu'`.

See Also

`auread`	Read NeXT/SUN (.au) sound file
`wavwrite`	Write Microsoft WAVE (.wav) sound file

balance

Purpose Improve accuracy of computed eigenvalues

Syntax

```
[D,B] = balance(A)
B = balance(A)
```

Description `[D,B] = balance(A)` returns a diagonal matrix `D` whose elements are integer powers of two, and a balanced matrix `B` so that `B = D\A*D`. If `A` is symmetric, then `B == A` and `D` is the identity matrix.

`B = balance(A)` returns just the balanced matrix `B`.

Remarks Nonsymmetric matrices can have poorly conditioned eigenvalues. Small perturbations in the matrix, such as roundoff errors, can lead to large perturbations in the eigenvalues. The quantity which relates the size of the matrix perturbation to the size of the eigenvalue perturbation is the condition number of the eigenvector matrix,

```
cond(V) = norm(V)*norm(inv(V))
```

where

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

(The condition number of `A` itself is irrelevant to the eigenvalue problem.)

Balancing is an attempt to concentrate any ill conditioning of the eigenvector matrix into a diagonal scaling. Balancing usually cannot turn a nonsymmetric matrix into a symmetric matrix; it only attempts to make the norm of each row equal to the norm of the corresponding column. Furthermore, the diagonal scale factors are limited to powers of two so they do not introduce any roundoff error.

MATLAB's eigenvalue function, `eig(A)`, automatically balances `A` before computing its eigenvalues. Turn off the balancing with `eig(A,'nobalance')`.

Examples

This example shows the basic idea. The matrix A has large elements in the upper right and small elements in the lower left. It is far from being symmetric.

```
A = [1  100  10000; .01  1  100; .0001  .01  1]
A =
   1.0e+04 *
    0.0001    0.0100    1.0000
    0.0000    0.0001    0.0100
    0.0000    0.0000    0.0001
```

Balancing produces a diagonal D matrix with elements that are powers of two and a balanced matrix B that is closer to symmetric than A.

```
[D,B] = balance(A)
D =
   1.0e+03 *
    2.0480         0         0
         0    0.0320         0
         0         0    0.0003
B =
    1.0000    1.5625    1.2207
    0.6400    1.0000    0.7812
    0.8192    1.2800    1.0000
```

To see the effect on eigenvectors, first compute the eigenvectors of A.

```
[V,E] = eig(A); V
V =
   –1.0000    0.9999   –1.0000
    0.0050    0.0100    0.0034
    0.0000    0.0001    0.0001
```

Note that all three vectors have the first component the largest. This indicates V is badly conditioned; in fact cond(V) is 1.7484e+05. Next, look at the eigenvectors of B.

```
[V,E] = eig(B); V
V =
   -0.8873    0.6933    0.8919
    0.2839    0.4437   -0.3264
    0.3634    0.5679   -0.3129
```

Now the eigenvectors are well behaved and `cond(V)` is `31.9814`. The ill conditioning is concentrated in the scaling matrix; `cond(D)` is `8192`.

This example is small and not really badly scaled, so the computed eigenvalues of `A` and `B` agree within roundoff error; balancing has little effect on the computed results.

Algorithm

`balance` is built into the MATLAB interpreter. It uses the algorithm in [1] originally published in Algol, but popularized by the Fortran routines `BALANC` and `BALBAK` from EISPACK.

Successive similarity transformations via diagonal matrices are applied to `A` to produce `B`. The transformations are accumulated in the transformation matrix `D`.

The `eig` function automatically uses balancing to prepare its input matrix.

Limitations

Balancing can destroy the properties of certain matrices; use it with some care. If a matrix contains small elements that are due to roundoff error, balancing may scale them up to make them as significant as the other elements of the original matrix.

Diagnostics

If `A` is not a square matrix:

```
Matrix must be square.
```

See Also

`condeig`	Condition number with respect to eigenvalues
`eig`	Eigenvalues and eigenvectors
`hess`	Hessenberg form of a matrix
`schur`	Schur decomposition

References

[1] Parlett, B. N. and C. Reinsch, “Balancing a Matrix for Calculation of Eigenvalues and Eigenvectors,” *Handbook for Auto. Comp.*, Vol. II, Linear Algebra, 1971,pp. 315-326.

base2dec

Purpose

Base to decimal number conversion

Syntax

```
d = base2dec('strn',base)
```

Description

`d = base2dec('strn',base)` converts the string number *strn* of the specified `base` into its decimal (base 10) equivalent. `base` must be an integer between 2 and 36. If *'strn'* is a character array, each row is interpreted as a string in the specified `base`.

Examples

The expression `base2dec('212',3)` converts 212_3 to decimal, returning 23.

See Also

`dec2base`

Purpose

Bessel functions of the third kind (Hankel functions)

Syntax

```
H = besselh(nu,K,Z)
H = besselh(nu,Z)
H = besselh(nu,1,Z,1)
H = besselh(nu,2,Z,1)
[H,ierr] = besselh(...)
```

Definitions

The differential equation

$$z^2\frac{d^2y}{dz^2} + z\frac{dy}{dz} - (z^2 + \nu^2)y = 0$$

where ν is a nonnegative constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*. $J_\nu(z)$ and $J_{-\nu}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger ν. $Y_\nu(z)$ is a second solution of Bessel's equation—linearly independent of $J_\nu(z)$ — defined by:

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

The relationship between the Hankel and Bessel functions is:

$$H1_\nu(z) = J_\nu(z) + i\ Y_\nu(z)$$

Description

`H = besselh(nu,K,Z)` for `K` = 1 or 2 computes the Hankel functions $H1_\nu(z)$ or $H2_\nu(z)$ for each element of the complex array `Z`. If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`H = besselh(nu,Z)` uses `K` = 1.

`H = besselh(nu,1,Z,1)` scales $H1_\nu(z)$ by `exp(-i*z)`.

`H = besselh(nu,2,Z,1)` scales $H2_\nu(z)$ by `exp(+i*z)`.

`[H,ierr] = besselh(...)` also returns an array of error flags:

`ierr = 1`	Illegal arguments.
`ierr = 2`	Overflow. Return `Inf`.
`ierr = 3`	Some loss of accuracy in argument reduction.
`ierr = 4`	Unacceptable loss of accuracy, `Z` or `nu` too large.
`ierr = 5`	No convergence. Return `NaN`.

Purpose

Modified Bessel functions

Syntax

```
I = besseli(nu,Z)        Modified Bessel function of the 1st kind
K = besselk(nu,Z)        Modified Bessel function of the 3rd kind
E = besseli(nu,Z,1)
K = besselk(nu,Z,1)
[I,ierr] = besseli(...)
[K,ierr] = besselk(...)
```

Definitions

The differential equation

$$z^2 \frac{d^2 y}{dz^2} + z\frac{dy}{dz} - (z^2 + \nu^2)y = 0$$

where ν is a nonnegative constant, is called the *modified Bessel's equation*, and its solutions are known as *modified Bessel functions*.

$I_\nu(z)$ and $I_{-\nu}(z)$ form a fundamental set of solutions of the modified Bessel's equation for noninteger ν. $K_\nu(z)$ is a second solution, independent of $I_\nu(z)$.

$I_\nu(z)$ and $K_\nu(z)$ are defined by:

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k!\ \Gamma(\nu + k + 1)}, \quad \text{where } \Gamma(a) = \int_0^\infty e^{-t}\, t^{a-1}\, dt$$

$$K_\nu(z) = \left(\frac{\pi}{2}\right) \frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)}$$

Description

`I = besseli(nu,Z)` computes modified Bessel functions of the first kind, $I_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size.If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`K = besselk(nu,Z)` computes modified Bessel functions of the second kind, $K_\nu(z)$, for each element of the complex array `Z`.

`E = besseli(nu,Z,1)` computes `besseli(nu,Z).*exp(-Z)`.

`K = besselk(nu,Z,1)` computes `besselk(nu,Z).*exp(-Z)`.

`[I,ierr] = besseli(...)` and `[K,ierr] = besselk(...)` also return an array of error flags.

`ierr = 1`	Illegal arguments.
`ierr = 2`	Overflow. Return `Inf`.
`ierr = 3`	Some loss of accuracy in argument reduction.
`ierr = 4`	Unacceptable loss of accuracy, `Z` or `nu` too large.
`ierr = 5`	No convergence. Return `NaN`.

Algorithm

The `besseli` and `besselk` functions use a Fortran MEX-file to call a library developed by D. E. Amos [3] [4].

See Also

`airy`	Airy functions
`besselj`, `bessely`	Bessel functions

References

[1] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sections 9.1.1, 9.1.89 and 9.12, formulas 9.1.10 and 9.2.5.

[2] Carrier, Krook, and Pearson, *Functions of a Complex Variable: Theory and Technique*, Hod Books, 1983, section 5.5.

[3] Amos, D. E., “A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order,” *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[4] Amos, D. E., “A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order,” *Trans. Math. Software*, 1986.

Purpose

Bessel functions

Syntax

```
J = besselj(nu,Z)      Bessel function of the 1st kind
Y = bessely(nu,Z)      Bessel function of the 2nd kind
[J,ierr] = besselj(nu,Z)
[Y,ierr] = bessely(nu,Z)
```

Definition

The differential equation

$$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} - (z^2 + \nu^2) y = 0$$

where ν is a nonnegative constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*.

$J_\nu(z)$ and $J_{-\nu}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger ν.

$Y_\nu(z)$ is a second solution of Bessel's equation—linearly independent of $J_\nu(z)$ — defined by:

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

Description

`J = besselj(nu,Z)` computes Bessel functions of the first kind, $J_\nu(z)$, for each element of the complex array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`Y = bessely(nu,Z)` computes Bessel functions of the second kind, $Y_\nu(z)$, for real, nonnegative order `nu` and argument `Z`.

`[J,ierr] = besselj(nu,Z)` and `[Y,ierr] = bessely(nu,Z)` also return an array of error flags.

`ierr = 1`	Illegal arguments.
`ierr = 2`	Overflow. Return `Inf`.
`ierr = 3`	Some loss of accuracy in argument reduction.
`ierr = 4`	Unacceptable loss of accuracy, `Z` or `nu` too large.
`ierr = 5`	No convergence. Return `NaN`.

Remarks

The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind:

$$H_\nu^{(1)}(z) = J_\nu(z) + i\ Y_\nu(z)$$

$$H_\nu^{(2)}(z) = J_\nu(z) - i\ Y_\nu(z)$$

where $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see `besselh`).

Examples

`besselj(3:9,(0:.2:10)')` generates the entire table on page 398 of Abramowitz and Stegun, *Handbook of Mathematical Functions*..

Algorithm

The `besselj` and `bessely` functions use a Fortran MEX-file to call a library developed by D. E. Amos [3] [4].

See Also

`airy`	Airy functions
`besseli`, `besselk`	Modified Bessel functions

References

[1] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sections 9.1.1, 9.1.89 and 9.12, formulas 9.1.10 and 9.2.5.

[2] Carrier, Krook, and Pearson, *Functions of a Complex Variable: Theory and Technique*, Hod Books, 1983, section 5.5.

[3] Amos, D. E., "A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[4] Amos, D. E., "A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Trans. Math. Software*, 1986.

beta, betainc, betaln

Purpose Beta functions

Syntax

```
B = beta(Z,W)
I = betainc(X,Z,W)
L = betaln(Z,W)
```

Definition The beta function is:

$$B(z, w) = \int_0^1 t^{z-1}(1-t)^{w-1}\, dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$$

where $\Gamma(z)$ is the gamma function. The incomplete beta function is:

$$I_x(z, w) = \frac{1}{B(z, w)}\int_0^x t^{z-1}(1-t)^{w-1}\, dt$$

Description `B = beta(Z,W)` computes the beta function for corresponding elements of the complex arrays `Z` and `W`. The arrays must be the same size (or either can be scalar).

`I = betainc(X,Z,W)` computes the incomplete beta function. The elements of `X` must be in the closed interval [0,1].

`L = betaln(Z,W)` computes the natural logarithm of the beta function, `log(beta(Z,W))`, without computing `beta(Z,W)`. Since the `beta` function can range over very large or very small values, its logarithm is sometimes more useful.

Examples

```
format rat
beta((0:10)',3)

ans =

     1/0
     1/3
     1/12
     1/30
     1/60
     1/105
     1/168
     1/252
     1/360
     1/495
     1/660
```

In this case, with integer arguments,

```
beta(n,3)
= (n–1)!*2!/(n+2)!
= 2/(n*(n+1)*(n+2))
```

is the ratio of fairly small integers and the rational format is able to recover the exact result.

For `x = 510`, `betaln(x,x) = –708.8616`, which, on a computer with IEEE arithmetic, is slightly less than `log(realmin)`. Here `beta(x,x)` would underflow (or be denormal).

Algorithm

```
beta(z,w) = exp(gammaln(z)+gammaln(w)–gammaln(z+w))
betaln(z,w) = gammaln(z)+gammaln(w)–gammaln(z+w)
```

bicg

Purpose BiConjugate Gradients method

Syntax

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

Description

x = bicg(A,b) attempts to solve the system of linear equations A*x = b for x. The coefficient matrix A must be square and the right hand side (column) vector b must have length n, where A is n-by-n. bicg will start iterating from an initial estimate that by default is an all zero vector of length n. Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate x has relative residual norm(b-A*x)/norm(b) less than or equal to the tolerance of the method. The default tolerance is 1e-6. The default maximum number of iterations is the minimum of n and 20. No preconditioning is used.

bicg(A,b,tol) specifies the tolerance of the method, tol.

bicg(A,b,tol,maxit) additionally specifies the maximum number of iterations, maxit.

bicg(A,b,tol,maxit,M) and bicg(A,b,tol,maxit,M1,M2) use left preconditioner M or M = M1*M2 and effectively solve the system inv(M)*A*x = inv(M)*b for x. If M1 or M2 is given as the empty matrix ([]), it is considered to be the identity matrix, equivalent to no preconditioning at all. Since systems of equations of the form M*y = r are solved using backslash within bicg, it is wise to factor preconditioners into their LU factors first. For example, replace bicg(A,b,tol,maxit,M) with:

```
[M1,M2] = lu(M);
bicg(A,b,tol,maxit,M1,M2).
```

`bicg(A,b,tol,maxit,M1,M2,x0)` specifies the initial estimate `x0`. If `x0` is given as the empty matrix (`[]`), the default all zero vector is used.

`x = bicg(A,b,tol,maxit,M1,M2,x0)` returns a solution `x`. If `bicg` converged, a message to that effect is displayed. If `bicg` failed to converge after the maximum number of iterations or halted 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.

`[x,flag] = bicg(A,b,tol,maxit,M1,M2,x0)` returns a solution `x` and a flag that describes the convergence of `bicg`:

Flag	Convergence
`0`	`bicg` converged to the desired tolerance `tol` within `maxit` iterations without failing for any reason.
`1`	`bicg` iterated `maxit` times but did not converge.
`2`	One of the systems of equations of the form `M*y = r` involving the preconditioner was ill-conditioned and did not return a useable result when solved by `\` (backslash).
`3`	The method stagnated. (Two consecutive iterates were the same.)
`4`	One of the scalar quantities calculated during `bicg` 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] = bicg(A,b,tol,maxit,M1,M2,x0)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, then `relres` ≤ `tol`.

`[x,flag,relres,iter] = bicg(A,b,tol,maxit,M1,M2,x0)` also returns the iteration number at which `x` was computed. This always satisfies `0` ≤ `iter` ≤ `maxit`.

`[x,flag,relres,iter,resvec] = bicg(A,b,tol,maxit,M1,M2,x0)` also returns a vector of the residual norms at each iteration, starting from `resvec(1) = norm(b-A*x0)`. If `flag` is `0`, `resvec` is of length `iter+1` and `resvec(end)` ≤ `tol*norm(b)`.

Examples

Start with `A = west0479` and make the true solution the vector of all ones.

```
load west0479
A = west0479
b = sum(A,2)
```

We could accurately solve `A*x = b` using backslash since `A` is not so large.

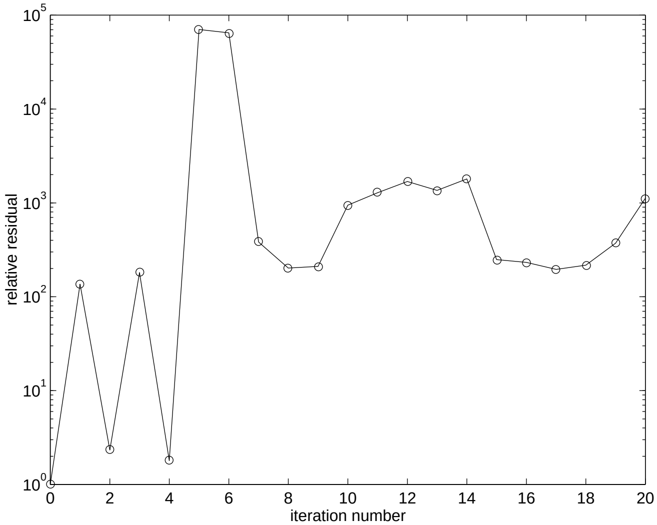
```
x = A \ b
norm(b-A*x) / norm(b) =
6.8476e-18
```

Now try to solve `A*x = b` with `bicg`.

```
[x,flag,relres,iter,resvec] = bicg(A,b)
flag =
1
relres =
1
iter =
0
```

The value of `flag` indicates that `bicg` iterated the default 20 times without converging. The value of `iter` shows that the method behaved so badly that the initial all zero guess was better than all the subsequent iterates. The value of `relres` supports this: `relres = norm(b-A*x)/norm(b)` = `norm(b)/norm(b)` = `1`.

The plot semilogy(0:20,resvec/norm(b),'-o') below confirms that the unpreconditioned method oscillated rather wildly.



Try an incomplete LU factorization with a drop tolerance of 1e-5 for the preconditioner.

```
[L1,U1] = luinc(A,1e-5)
nnz(A) =
1887
nnz(L1) =
5562
nnz(U1) =
4320
```

A warning message indicates a zero on the main diagonal of the upper triangular `U1`. Thus it is singular. When we try to use it as a preconditioner:

```
[x,flag,relres,iter,resvec] = bicg(A,b,1e-6,20,L1,U1)
flag =
2
relres =
1
iter =
0
resvec =
7.0557e+005
```

the method fails in the very first iteration when it tries to solve a system of equations involving the singular `U1` with backslash. It is forced to return the initial estimate since no other iterates were produced.

Try again with a slightly less sparse preconditioner:

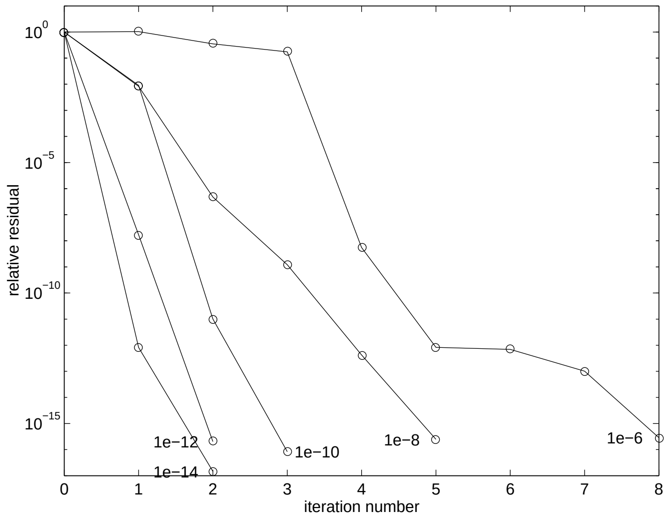
```
[L2,U2] = luinc(A,1e-6)
nnz(L2) =
6231
nnz(U2) =
4559
```

This time there is no warning message. All entries on the main diagonal of `U2` are nonzero

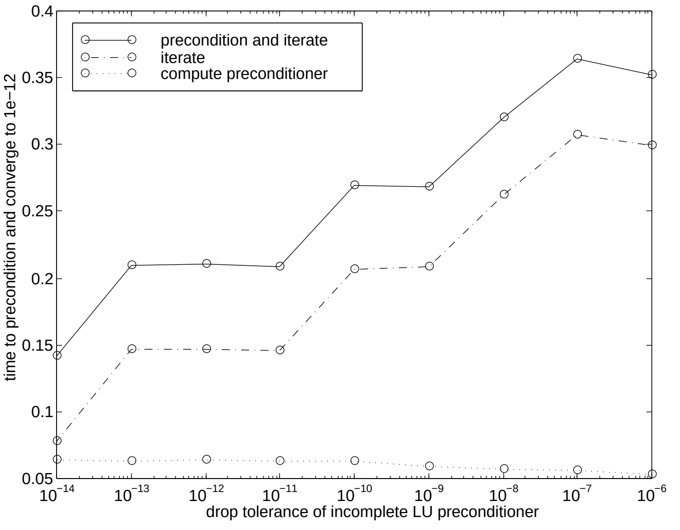
```
[x,flag,relres,iter,resvec] = bicg(A,b,1e-15,10,L2,U2)
flag =
0
relres =
2.8664e-16
iter =
8
```

and `bicg` converges to within the desired tolerance at iteration number 8. Decreasing the value of the drop tolerance increases the fill-in of the incomplete factors but also increases the accuracy of the approximation to the original matrix. Thus, the preconditioned system becomes closer to `inv(U)*inv(L)*L*U*x = inv(U)*inv(L)*b`, where `L` and `U` are the true LU factors, and closer to being solved within a single iteration.

The next graph shows the progress of `bicg` using six different incomplete LU factors as preconditioners. Each line in the graph is labelled with the drop tolerance of the preconditioner used in `bicg`.



This does not give us any idea of the time involved in creating the incomplete factors and then computing the solution. The following graph plots drop tolerance of the incomplete LU factors against the time to compute the preconditioner, the time to iterate once the preconditioner has been computed, and their sum, the total time to solve the problem. The time to produce the factors does not increase very quickly with the fill-in, but it does slow down the average time for an iteration. Since fewer iterations are performed, the total time to solve the problem decreases. `west0479` is quite a small matrix, only 139-by-139, and preconditioned `bicg` still takes longer than backslash.



See Also

`bicgstab`	BiConjugate Gradients Stabilized method
`cgs`	Conjugate Gradients Squared method
`gmres`	Generalized Minimum Residual method (with restarts)
`luinc`	Incomplete LU matrix factorizations
`pcg`	Preconditioned Conjugate Gradients method
`qmr`	Quasi-Minimal Residual method
`\`	Matrix left division

References

Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM, Philadelphia, 1994.

Purpose BiConjugate Gradients Stabilized method

Syntax

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

Description

`x = bicgstab(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The coefficient matrix `A` must be square and the right hand side (column) vector `b` must have length `n`, where `A` is `n`-by-`n`. `bicgstab` will start iterating from an initial estimate that by default is an all zero vector of length `n`. Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate `x` has relative residual `norm(b–A*x)/norm(b)` less than or equal to the tolerance of the method. The default tolerance is `1e–6`. The default maximum number of iterations is the minimum of `n` and 20. No preconditioning is used.

`bicgstab(A,b,tol)` specifies the tolerance of the method, `tol`.

`bicgstab(A,b,tol,maxit)` additionally specifies the maximum number of iterations, `maxit`.

`bicgstab(A,b,tol,maxit,M)` and `bicgstab(A,b,tol,maxit,M1,M2)` use left preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M1` or `M2` is given as the empty matrix (`[]`), it is considered to be the identity matrix, equivalent to no preconditioning at all. Since systems of equations of the form `M*y = r` are solved using backslash within `bicgstab`, it is wise to factor preconditioners into their LU factors first. For example, replace `bicgstab(A,b,tol,maxit,M)` with:

```
[M1,M2] = lu(M);
bicgstab(A,b,tol,maxit,M1,M2).
```

`bicgstab(A,b,tol,maxit,M1,M2,x0)` specifies the initial estimate `x0`. If `x0` is given as the empty matrix (`[]`), the default all zero vector is used.

`x = bicgstab(A,b,tol,maxit,M1,M2,x0)` returns a solution `x`. If `bicgstab` converged, a message to that effect is displayed. If `bicgstab` failed to converge after the maximum number of iterations or halted 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.

`[x,flag] = bicgstab(A,b,tol,maxit,M1,M2,x0)` returns a solution `x` and a flag that describes the convergence of `bicgstab`:

Flag	Convergence
0	`bicgstab` converged to the desired tolerance `tol` within `maxit` iterations without failing for any reason.
1	`bicgstab` iterated `maxit` times but did not converge.
2	One of the systems of equations of the form `M*y = r` involving the preconditioner was ill-conditioned and did not return a useable result when solved by \ (backslash).
3	The method stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `bicgstab` 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] = bicgstab(A,b,tol,maxit,M1,M2,x0)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, then `relres` ≤ `tol`.

`[x,flag,relres,iter] = bicgstab(A,b,tol,maxit,M1,M2,x0)` also returns the iteration number at which `x` was computed. This always satisfies `0` ≤ `iter` ≤ `maxit`. `iter` may be an integer or an integer + 0.5, since `bicgstab` may converge half way through an iteration.

`[x,flag,relres,iter,resvec] = bicgstab(A,b,tol,maxit,M1,M2,x0)` also returns a vector of the residual norms at each iteration, starting from `resvec(1) = norm(b-A*x0)`. If `flag` is `0`, `resvec` is of length `2*iter+1`, whether `iter` is an integer or not. In this case, `resvec(end)` ≤ `tol*norm(b)`.

Example

```
load west0479
A = west0479
b = sum(A,2)
[x,flag] = bicgstab(A,b)
```

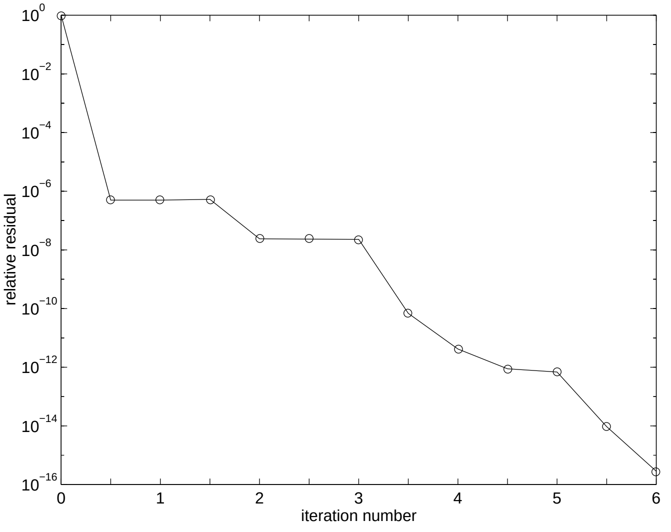
`flag` is `1` since `bicgstab` will not converge to the default tolerance `1e-6` within the default 20 iterations.

```
[L1,U1] = luinc(A,1e-5)
[x1,flag1] = bicgstab(A,b,1e-6,20,L1,U1)
```

`flag1` is 2 since the upper triangular `U1` has a zero on its diagonal so `bicgstab` fails in the first iteration when it tries to solve a system such as `U1*y = r` with backslash.

```
[L2,U2] = luinc(A,1e-6)
[x2,flag2,relres2,iter2,resvec2] = bicgstab(A,b,1e-15,10,L2,U2)
```

`flag2` is `0` since `bicgstab` will converge to the tolerance of `2.9e-16` (the value of `relres2`) at the sixth iteration (the value of `iter2`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. `resvec2(1) = norm(b)` and `resvec2(7) = norm(b-A*x2)`. You may follow the progress of `bicgstab` by plotting the relative residuals at the half way point and end of each iteration starting from the intial estimate (iterate number 0) with `semilogy(0:0.5:iter2,resvec2/norm(b),'-o')`



See Also

`bicg`	BiConjugate Gradients method
`cgs`	Conjugate Gradients Squared method
`gmres`	Generalized Minimum Residual method (with restarts)
`luinc`	Incomplete LU matrix factorizations
`pcg`	Preconditioned Conjugate Gradients method
`qmr`	Quasi-Minimal Residual method
`\`	Matrix left division

References

van der Vorst, H. A., *BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., March 1992,Vol. 13, No. 2, pp. 631-644.

Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM, Philadelphia, 1994.

Purpose Binary to decimal number conversion

Syntax

`bin2dec(`*binarystr*`)`

Description

`bin2dec(`*binarystr*`)` interprets the binary string *binarystr* and returns the equivalent decimal number.

Examples

`bin2dec('010111')` returns 23.

See Also

`dec2bin`

bitand

Purpose Bit-wise AND

Syntax

```
C = bitand(A,B)
```

Description

`C = bitand(A,B)` returns the bit-wise AND of two nonnegative integer arguments `A` and `B`. To ensure the operands are integers, use the `ceil`, `fix`, `floor`, and `round` functions.

Examples

The five-bit binary representations of the integers 13 and 27 are 01101 and 11011, respectively. Performing a bit-wise AND on these numbers yields 01001, or 9.

```
C = bitand(13,27)

C =

     9
```

See Also

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

Purpose

Complement bits

Syntax

```
C = bitcmp(A,n)
```

Description

`C = bitcmp(A,n)` returns the bit-wise complement of `A` as an `n`-bit floating-point integer (flint).

Example

With eight-bit arithmetic, the ones' complement of 01100011 (99, decimal) is 10011100 (156, decimal).

```
C = bitcmp(99,8)

C =

   156
```

See Also

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

bitget

Purpose Get bit

Syntax

```
C = bitget(A,bit)
```

Description

`C = bitget(A,bit)` returns the value of the bit at position *bit* in `A`. Operand `A` must be a nonnegative integer, and *bit* must be a number between 1 and the number of bits in the floating-point integer (flint) representation of `A` (52 for IEEE flints). To ensure the operand is an integer, use the `ceil`, `fix`, `floor`, and `round` functions.

Example

The `dec2bin` function converts decimal numbers to binary. However, you can also use the `bitget` function to show the binary representation of a decimal number. Just test successive bits from most to least significant:

```
disp(dec2bin(13))
1101
C = bitget(13,4:–1:1)

C =
     1     1     0     1
```

See Also

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

Purpose Maximum floating-point integer

Syntax

```
bitmax
```

Description

`bitmax` returns the maximum unsigned floating-point integer for your computer. It is the value when all bits are set. On IEEE machines, this is the value $2^{53} - 1$.

See Also

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

bitor

Purpose Bit-wise OR

Syntax

```
C = bitor(A,B)
```

Description `C = bitor(A,B)` returns the bit-wise OR of two nonnegative integer arguments `A` and `B`. To ensure the operands are integers, use the `ceil`, `fix`, `floor`, and `round` functions.

Examples The five-bit binary representations of the integers 13 and 27 are 01101 and 11011, respectively. Performing a bit-wise OR on these numbers yields 11111, or 31.

```
C = bitor(13,27)

C =

    31
```

See Also

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

Purpose Set bit

Syntax

```
C = bitset(A,bit)
C = bitset(A,bit,v)
```

Description

`C = bitset(A,bit)` sets bit position *bit* in `A` to 1 (on). `A` must be a nonnegative integer and *bit* must be a number between 1 and the number of bits in the floating-point integer (flint) representation of `A` (52 for IEEE flints). To ensure the operand is an integer, use the `ceil`, `fix`, `floor`, and `round` functions.

`C = bitset(A,bit,v)` sets the bit at position *bit* to the value `v`, which must be either 0 or 1.

Examples

Setting the fifth bit in the five-bit binary representation of the integer 9 (01001) yields 11001, or 25.

```
C = bitset(9,5)

C =

    25
```

See Also

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

bitshift

Purpose

Bit-wise shift

Syntax

```
C = bitshift(A,n)
```

Description

`C = bitshift(A,n)` returns the value of `A` shifted by `n` bits. If `n>0`, this is same as a multiplication by 2^n (left shift). If `n<0`, this is the same as a division by 2^n (right shift). `A` must be a nonnegative integer, which you can ensure by using the `ceil`, `fix`, `floor`, and `round` functions.

Examples

Shifting 1100 (12, decimal) to the left two bits yields 110000 (48, decimal).

```
C = bitshift(12,2)

C =

    48
```

See Also

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

Purpose

Bit-wise XOR

Syntax

```
C = bitxor(A,B)
```

Description

`C = bitxor(A,B)` returns the bit-wise XOR of the two arguments `A` and `B`. Both `A` and `B` must be integers. You can ensure this by using the `ceil`, `fix`, `floor`, and `round` functions.

Examples

The five-bit binary representations of the integers 13 and 27 are 01101 and 11011, respectively. Performing a bit-wise XOR on these numbers yields 10110, or 22.

```
C = bitxor(13,27)

C =
    22
```

See Also

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

blanks

Purpose A string of blanks

Syntax

```
blanks(n)
```

Description `blanks(n)` is a string of `n` blanks.

Examples `blanks` is useful with the `display` function. For example,

```
disp(['xxx' blanks(20) 'yyy'])
```

displays twenty blanks between the strings `'xxx'` and `'yyy'`.

`disp(blanks(n)')` moves the cursor down `n` lines.

See Also

`clc`	Clear command window
`home`	Send the cursor home
`format`	See `compact` option for suppression of blank lines

Purpose Break out of flow control structures

Syntax

```
break
```

Description `break` terminates the execution of `for` and `while` loops. In nested loops, `break` exits from the innermost loop only.

Examples The indented statements are repeatedly executed until nonpositive `n` is entered.

```
while 1
    n = input('Enter n. n <= 0 quits. n = ')
    if n <= 0,break,end
    r = rank(magic(n))
end
disp('That''s all.')
```

See Also

`end`	Terminate `for`, `while`, and `if` statements and indicate the last index
`error`	Display error messages
`for`	Repeat statements a specific number of times
`if`	Conditionally execute statements
`return`	Return to the invoking function
`switch`	Switch among several cases based on expression
`while`	Repeat statements an indefinite number of times

builtin

Purpose Execute builtin function from overloaded method

Syntax

```
builtin(function,x1,...,xn)
[y1,..,yn] = builtin(function,x1,...,xn)
```

Description `builtin` is used in methods that overload builtin functions to execute the original builtin function. If *function* is a string containing the name of a builtin function,then:

`builtin(`*function*`,x1,...,xn)` evaluates that function at the given arguments.

`[y1,..,yn] = builtin(`*function*`,x1,...,xn)` returns multiple output arguments.

Remarks `builtin`(...) is the same as `feval`(...) except that it calls the original builtin version of the function even if an overloaded one exists. (For this to work you must never overload `builtin`.)

See Also

`feval`	Function evaluation

calendar

Purpose Calendar

Syntax

```
c = calendar
c = calendar(d)
c = calendar(y,m)

calendar(...)
```

Description

`c = calendar` returns a 6-by-7 matrix containing a calendar for the current month. The calendar runs Sunday (first column) to Saturday.

`c = calendar(d)`, where `d` is a serial date number or a date string, returns a calendar for the specified month.

`c = calendar(y,m)`, where `y` and `m` are integers, returns a calendar for the specified month of the specified year.

`calendar(...)` displays the calendar on the screen.

Examples

The command:

```
calendar(1957,10)
```

reveals that the Space Age began on a Friday (on October 4, 1957, when Sputnik 1 was launched).

```
                      Oct 1957
     S      M     Tu      W     Th      F      S
     0      0      1      2      3      4      5
     6      7      8      9     10     11     12
    13     14     15     16     17     18     19
    20     21     22     23     24     25     26
    27     28     29     30     31      0      0
     0      0      0      0      0      0      0
```

See Also

`datenum` Serial date number

cart2pol

Purpose

Transform Cartesian coordinates to polar or cylindrical

Syntax

```
[THETA,RHO,Z] = cart2pol(X,Y,Z)
[THETA,RHO] = cart2pol(X,Y)
```

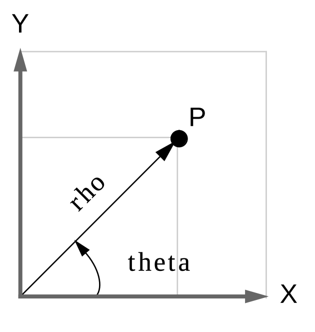
Description

`[THETA,RHO,Z] = cart2pol(X,Y,Z)` transforms three-dimensional Cartesian coordinates stored in corresponding elements of arrays `X`, `Y`, and `Z`, into cylindrical coordinates. `THETA` is a counterclockwise angular displacement in radians from the positive *x*-axis, `RHO` is the distance from the origin to a point in the *x*-*y* plane, and `Z` is the height above the *x*-*y* plane. Arrays `X`, `Y`, and `Z` must be the same size (or any can be scalar).

`[THETA,RHO] = cart2pol(X,Y)` transforms two-dimensional Cartesian coordinates stored in corresponding elements of arrays `X` and `Y` into polar coordinates.

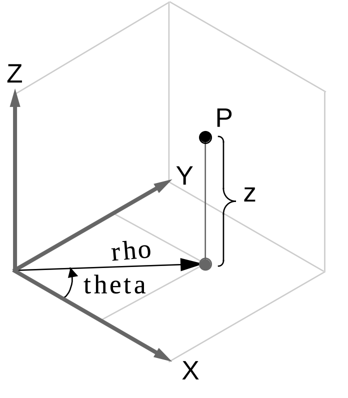
Algorithm

The mapping from two-dimensional Cartesian coordinates to polar coordinates, and from three-dimensional Cartesian coordinates to cylindrical coordinates is:



Two-Dimensional Mapping

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



Three-Dimensional Mapping

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

See Also

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

cart2sph

Purpose Transform Cartesian coordinates to spherical

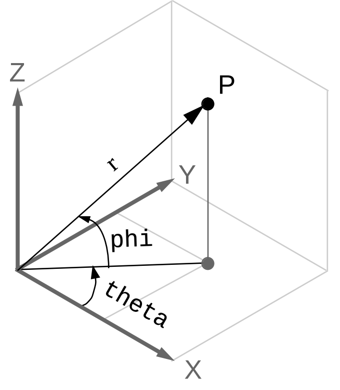
Syntax

```
[THETA,PHI,R] = cart2sph(X,Y,Z)
```

Description `[THETA,PHI,R] = cart2sph(X,Y,Z)` transforms Cartesian coordinates stored in corresponding elements of arrays `X`, `Y`, and `Z` into spherical coordinates. Azimuth `THETA` and elevation `PHI` are angular displacements in radians measured from the positive *x*-axis, and the *x*-*y* plane, respectively; and `R` is the distance from the origin to a point.

Arrays `X`, `Y`, and `Z` must be the same size.

Algorithm The mapping from three-dimensional Cartesian coordinates to spherical coordinates is:




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

See Also

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

Purpose

Case switch

Description

case is part of the switch statement syntax, which allows for conditional execution.

A particular case consists of the case statement itself, followed by a case expression, and one or more statements.

A case is executed only if its associated case expression (case_expr) is the first to match the switch expression (switch_expr).

Examples

The general form of the switch statement is:

```
switch switch_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	Switch among several cases based on expression

cat

Purpose

Concatenate arrays

Syntax

```
C = cat(dim,A,B)
C = cat(dim,A1,A2,A3,A4...)
```

Description

`C = cat(dim,A,B)` concatenates the arrays `A` and `B` along *dim*.

`C = cat(dim,A1,A2,A3,A4,...)` concatenates all the input arrays (`A1`, `A2`, `A3`, `A4`, and so on) along *dim*.

`cat(2,A,B)` is the same as `[A,B]` and `cat(1,A,B)` is the same as `[A;B]`.

Remarks

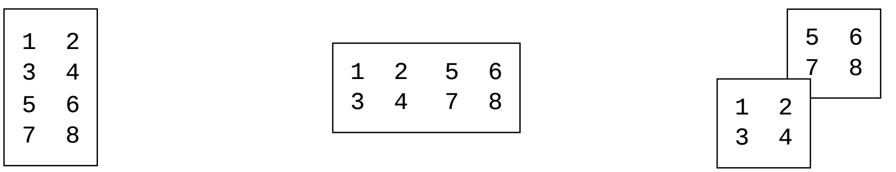
When used with comma separated list syntax, `cat(dim,C{:})` or `cat(dim,C.field)` is a convenient way to concatenate a cell or structure array containing numeric matrices into a single matrix.

Examples

Given,

```
A =                    B =
     1     2                         5     6
     3     4                         7     8
```

concatenating along different dimensions produces:



`C = cat(1,A,B)`   **`C = cat(2,A,B)`**   **`C = cat(3,A,B)`**

The commands

```
A = magic(3); B = pascal(3);
C = cat(4,A,B);
```

produce a 3-by-3-by-1-by-2 array.

See Also

`[]`	(Special characters) Build arrays
`num2cell`	Convert a numeric array into a cell array

Purpose Change working directory

Syntax

```
cd
cd directory
cd ..
```

Description

`cd`, by itself, prints out the current directory.

`cd` *directory* sets the current directory to the one specified. On UNIX platforms, the character ~ is interpreted as the user's root directory.

`cd ..` changes to the directory above the current one.

Examples

UNIX: `cd /usr/local/matlab/toolbox/demos`

DOS: `cd C:MATLAB\DEMOS`

VMS: `cd DISK1:[MATLAB.DEMOS]`

Macintosh: `cd Toolbox:Demos`

To specify a Macintosh directory name that includes spaces, enclose the name in single quotation marks, as in `'Toolbox:New M-Files'`.

See Also

`dir`	Directory listing
`path`	Control MATLAB's directory search path
`what`	Directory listing of M-files, MAT-files, and MEX-files

cdf2rdf

Purpose

Convert complex diagonal form to real block diagonal form

Syntax

```
[V,D] = cdf2rdf(V,D)
```

Description

If the eigensystem `[V,D] = eig(X)` has complex eigenvalues appearing in complex-conjugate pairs, `cdf2rdf` transforms the system so `D` is in real diagonal form, with 2-by-2 real blocks along the diagonal replacing the complex pairs originally there. The eigenvectors are transformed so that

```
X = V*D/V
```

continues to hold. The individual columns of `V` are no longer eigenvectors, but each pair of vectors associated with a 2-by-2 block in `D` spans the corresponding invariant vectors.

Examples

The matrix

```
X =
    1     2     3
    0     4     5
    0    -5     4
```

has a pair of complex eigenvalues.

```
[V,D] = eig(X)

V =
    1.0000     0.4002 - 0.0191i       0.4002 + 0.0191i
         0     0.6479                 0.6479
         0          0 + 0.6479i            0 - 0.6479i

D =
    1.0000            0                  0
         0       4.0000 + 5.0000i        0
         0            0             4.0000 - 5.0000i
```

Converting this to real block diagonal form produces

```
[V,D] = cdf2rdf(V,D)

V =
    1.0000     0.4002     -0.0191
         0     0.6479           0
         0          0      0.6479
D =
    1     0     0
    0     4     5
    0    -5     4
```

Algorithm

The real diagonal form for the eigenvalues is obtained from the complex form using a specially constructed similarity transformation.

See Also

`eig`	Eigenvalues and eigenvectors
`rsf2csf`	Convert real Schur form to complex Schur form

ceil

Purpose Round toward infinity

Syntax

```
B = ceil(A)
```

Description

`B = ceil(A)` rounds the elements of `A` to the nearest integers greater than or equal to `A`. For complex `A`, the imaginary and real parts are rounded independently.

Examples

```
a =

  Columns 1 through 4

 -1.9000         -0.2000          3.4000          5.6000

  Columns 5 through 6

   7.0000            2.4000 + 3.6000i

ceil(a)

ans =

  Columns 1 through 4

  -1.0000          0             4.0000            6.0000

  Columns 5 through 6

   7.0000            3.0000 + 4.0000i
```

See Also

`fix`	Round toward zero
`floor`	Round toward minus infinity
`round`	Round to nearest integer

Purpose Create cell array

Syntax

```
c = cell(n)
c = cell(m,n)
c = cell([m n])
c = cell(m,n,p,...)
c = cell([m n p ...])
c = cell(size(A))
```

Description

`c = cell(n)` creates an n-by-n cell array of empty matrices. An error message appears if `n` is not a scalar.

`c = cell(m,n)` or `c = cell([m,n])` creates an m-by-n cell array of empty matrices. Arguments `m` and `n` must be scalars.

`c = cell(m,n,p,...)` or `c = cell([m n p ...])` creates an m-by-n-by-p-... cell array of empty matrices. Arguments `m`, `n`, `p`,... must be scalars.

`c = cell(size(A))` creates a cell array the same size as `A` containing all empty matrices.

Examples

```
A = ones(2,2)

A =
     1     1
     1     1

c = cell(size(A))

c =

     []     []
     []     []
```

See Also

`ones`	Create an array of all ones
`rand`	Uniformly distributed random numbers and arrays
`randn`	Normally distributed random numbers and arrays
`zeros`	Create an array of all zeros

Purpose Cell array to structure array conversion

Syntax

```
s = cell2struct(c,fields,dim)
```

Description

`s = cell2struct(c,fields,dim)` converts the cell array `c` into the structure `s` by folding the dimension `dim` of `c` into fields of `s`. The length of `c` along the specified dimension (`size(c,dim)`) must match the number of fields names in `fields`. Argument `fields` can be a character array or a cell array of strings.

Examples

```
c = {'tree',37.4,'birch'};
f = {'category','height','name'};
s = cell2struct(c,f,2)

s =

    category: 'tree'
      height: 37.4000
        name: 'birch'
```

See Also

`fieldnames`	Field names of a structure
`struct2cell`	Structure to cell array conversion

Purpose Display cell array contents.

Syntax `celldisp(C)`

Description `celldisp(c)` recursively displays the contents of a cell array.

Example Use `celldisp` to display the contents of a 2-by-3 cell array:

```
C = {[1 2] 'Tony' 3+4i; [1 2;3 4] –5 'abc'};
celldisp(C)

C{1,1} =

     1     2

C{2,1} =

     1     2
     3     4

C{1,2} =

Tony

C{2,2} =

    –5

C{1,3} =

   3.0000+ 4.0000i


C{2,3} =

abc
```

See Also

`cellplot`	Graphically display the structure of cell arrays

Purpose

Graphically display the structure of cell arrays

Syntax

```
cellplot(c)
cellplot(c,'legend')
handles = cellplot(...)
```

Description

`cellplot(c)` displays a figure window that graphically represents the contents of `c`. Filled rectangles represent elements of vectors and arrays, while scalars and short text strings are displayed as text.

`cellplot(c,'legend')` also puts a legend next to the plot.

`handles = cellplot(c)` displays a figure window and returns a vector of surface handles.

Limitations

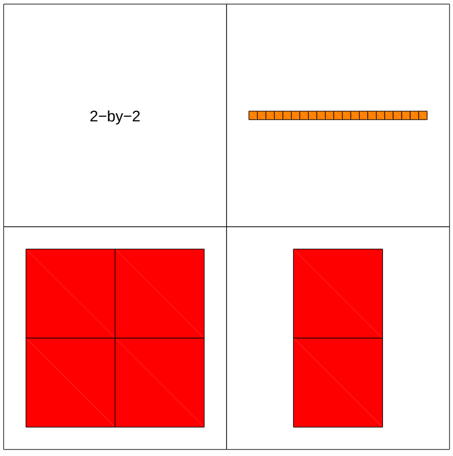
The `cellplot` function can display only two-dimensional cell arrays.

Examples

Consider a 2-by-2 cell array containing a matrix, a vector, and two text strings:

```
c{1,1} = '2-by-2';
c{1,2} = 'eigenvalues of eye(2)';
c{2,1} = eye(2);
c{2,2} = eig(eye(2));
```

The command `cellplot(c)` produces:



Purpose Create cell array of strings from character array

Syntax

```
c = cellstr(S)
```

Description `c = cellstr(S)` places each row of the character array `S` into separate cells of `c`. Use the `string` function to convert back to a string matrix.

Examples Given the string matrix

```
S =
abc
defg
hi
```

The command `c = cellstr(S)` returns the 3-by-1 cell array:

```
c =
    'abc'
    'defg'
    'hi'
```

See Also

`iscellstr`	True for cell array of strings
`string`	Convert numeric values to string

Purpose Conjugate Gradients Squared method

Syntax

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

Description

`x = cgs(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The coefficient matrix `A` must be square and the right hand side (column) vector `b` must have length `n`, where `A` is `n`-by-`n`. `cgs` will start iterating from an initial estimate that by default is an all zero vector of length `n`. Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate `x` has relative residual `norm(b–A*x)/norm(b)` less than or equal to the tolerance of the method. The default tolerance is `1e–6`. The default maximum number of iterations is the minimum of `n` and 20. No preconditioning is used.

`cgs(A,b,tol)` specifies the tolerance of the method, `tol`.

`cgs(A,b,tol,maxit)` additionally specifies the maximum number of iterations, `maxit`.

`cgs(A,b,tol,maxit,M)` and `cgs(A,b,tol,maxit,M1,M2)` use left preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M1` or `M2` is given as the empty matrix (`[]`), it is considered to be the identity matrix, equivalent to no preconditioning at all. Since systems of equations of the form `M*y = r` are solved using backslash within `cgs`, it is wise to factor preconditioners into their LU factors first. For example, replace `cgs(A,b,tol,maxit,M)` with:

```
[M1,M2] = lu(M);
cgs(A,b,tol,maxit,M1,M2).
```

`cgs(A,b,tol,maxit,M1,M2,x0)` specifies the initial estimate `x0`. If `x0` is given as the empty matrix (`[]`), the default all zero vector is used.

`x = cgs(A,b,tol,maxit,M1,M2,x0)` returns a solution `x`. If `cgs` converged, a message to that effect is displayed. If `cgs` failed to converge after the maximum number of iterations or halted 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.

`[x,flag] = cgs(A,b,tol,maxit,M1,M2,x0)` returns a solution `x` and a flag that describes the convergence of `cgs`:

Flag	Convergence
`0`	`cgs` converged to the desired tolerance `tol` within `maxit` iterations without failing for any reason.
`1`	`cgs` iterated `maxit` times but did not converge.
`2`	One of the systems of equations of the form `M*y = r` involving the preconditioner was ill-conditioned and did not return a useable result when solved by \ (backslash).
`3`	The method stagnated. (Two consecutive iterates were the same.)
`4`	One of the scalar quantities calculated during `cgs` 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] = cgs(A,b,tol,maxit,M1,M2,x0)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, then `relres` $\leq$ `tol`.

`[x,flag,relres,iter] = cgs(A,b,tol,maxit,M1,M2,x0)` also returns the iteration number at which `x` was computed. This always satisfies `0` $\leq$ `iter` $\leq$ `maxit`.

`[x,flag,relres,iter,resvec] = cgs(A,b,tol,maxit,M1,M2,x0)` also returns a vector of the residual norms at each iteration, starting from `resvec(1) = norm(b-A*x0)`. If `flag` is `0`, `resvec` is of length `iter+1` and `resvec(end)` ≤ `tol*norm(b)`.

Examples

```
load west0479
A = west0479
b = sum(A,2)
[x,flag] = cgs(A,b)
```

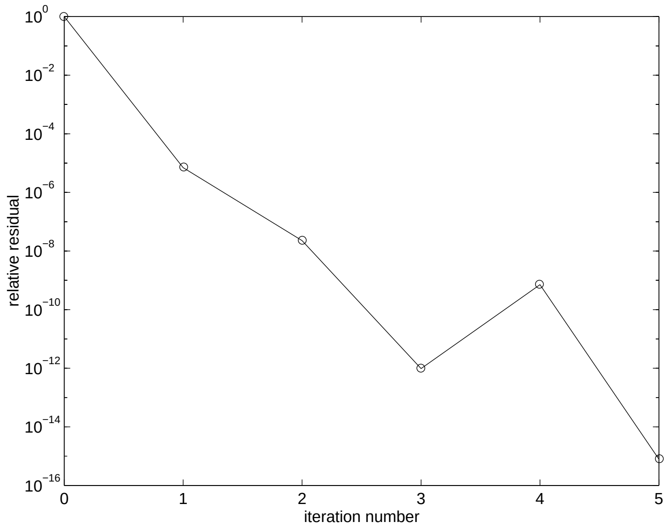
`flag` is `1` since `cgs` will not converge to the default tolerance `1e-6` within the default 20 iterations.

```
[L1,U1] = luinc(A,1e-5)
[x1,flag1] = cgs(A,b,1e-6,20,L1,U1)
```

`flag1` is `2` since the upper triangular `U1` has a zero on its diagonal so `cgs` fails in the first iteration when it tries to solve a system such as `U1*y = r` for `y` with backslash.

```
[L2,U2] = luinc(A,1e-6)
[x2,flag2,relres2,iter2,resvec2] = cgs(A,b,1e-15,10,L2,U2)
```

`flag2` is `0` since `cgs` will converge to the tolerance of `7.9e-16` (the value of `relres2`) at the fifth iteration (the value of `iter2`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. `resvec2(1) = norm(b)` and `resvec2(6) = norm(b-A*x2)`. You may follow the progress of `cgs` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0) with `semilogy(0:iter2,res2/norm(b),'-o')`.



See Also

`bicg`	BiConjugate Gradients method
`bicgstab`	BiConjugate Gradients Stabilized method
`gmres`	Generalized Minimum Residual method (with restarts)
`luinc`	Incomplete LU matrix factorizations
`pcg`	Preconditioned Conjugate Gradients method
`qmr`	Quasi-Minimal Residual method
`\`	Matrix left division

References

Sonneveld, Peter, *CGS: A fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., January 1989, Vol. 10, No. 1, pp. 36-52

Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM, Philadelphia, 1994.

char

Purpose Create character array (string)

Syntax

```
S = char(X)
S = char(C)
S = char(t1,t2.t3...)
```

Description `S = char(X)` converts the array `X` that contains positive integers representing character codes into a MATLAB character array (the first 127 codes are ASCII). The actual characters displayed depend on the character set encoding for a given font. The result for any elements of `X` outside the range from 0 to 65535 is not defined (and may vary from platform to platform). Use `double` to convert a character array into its numeric codes.

`S = char(C)` when `C` is a cell array of strings, places each element of `C` into the rows of the character array `s`. Use `cellstr` to convert back.

`S = char(t1,t2,t3,..)` forms the character array `S` containing the text strings `T1`,`T2`,`T3`,... as rows, automatically padding each string with blanks to form a valid matrix. Each text parameter,`T`*i*, can itself be a character array. This allows the creation of arbitarily large character arrays. Empty strings are significant.

Remarks Ordinarily, the elements of `A` are integers in the range 32:127, which are the printable ASCII characters, or in the range 0:255, which are all 8-bit values. For noninteger values, or values outside the range 0:255, the characters printed are determined by `fix(rem(A,256))`.

Examples To print a 3-by-32 display of the printable ASCII characters:

```
ascii = char(reshape(32:127,32,3)')
ascii =
 ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
' a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

See Also

`get`, `set`, and `text` in the online *MATLAB Function Reference* , and:

`cellstr`	Create cell array of strings from character array
`double`	Convert to double precision
`strings`	MATLAB string handling
`strvcat`	Vertical concatenation of strings

Purpose

Cholesky factorization

Syntax

```
R = chol(X)
[R,p] = chol(X)
```

Description

The `chol` function uses only the diagonal and upper triangle of `X`. The lower triangular is assumed to be the (complex conjugate) transpose of the upper. That is, `X` is Hermitian.

`R = chol(X)`, where `X` is positive definite produces an upper triangular `R` so that `R'*R = X`. If `X` is not positive definite, an error message is printed.

`[R,p] = chol(X)`, with two output arguments, never produces an error message. If `X` is positive definite, then `p` is 0 and `R` is the same as above. If `X` is not positive definite, then `p` is a positive integer and `R` is an upper triangular matrix of order `q = p–1` so that `R'*R = X(1:q,1:q)`.

Examples

The binomial coefficients arranged in a symmetric array create an interesting positive definite matrix.

```
n = 5;
X = pascal(n)
X =
    1    1    1    1    1
    1    2    3    4    5
    1    3    6   10   15
    1    4   10   20   35
    1    5   15   35   70
```

It is interesting because its Cholesky factor consists of the same coefficients, arranged in an upper triangular matrix.

```
R = chol(X)
R =
    1    1    1    1    1
    0    1    2    3    4
    0    0    1    3    6
    0    0    0    1    4
    0    0    0    0    1
```

Destroy the positive definiteness (and actually make the matrix singular) by subtracting 1 from the last element.

```
X(n,n) = X(n,n)–1
X =
     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    69
```

Now an attempt to find the Cholesky factorization fails.

Algorithm

`chol` uses the algorithm from the LINPACK subroutine `ZPOFA`. For a detailed description of the use of the Cholesky decomposition, see Chapter 8 of the *LINPACK Users' Guide*.

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

cholinc

Purpose

Incomplete Cholesky factorizations

Syntax

```
cholinc(X,'0')
R = cholinc(X,'0')
[R,p] = cholinc(X,'0')
R = cholinc(X,droptol)
R = cholinc(X,options)
```

Description

`cholinc(X,'0')` produces the incomplete Cholesky factorization of a real symmetric positive definite sparse matrix with 0 level of fill-in. `cholinc(X,'0')` produces an upper triangular matrix. The lower triangle of `X` is assumed to be the transpose of the upper (`X` is symmetric).

`R = cholinc(X,'0')` returns an upper triangular matrix which has the same sparsity pattern as the upper triangle of `X`. The product `R'*R` agrees with `X` over its sparsity pattern. The positive definiteness of `X` is not sufficient to guarantee the existence of the incomplete factor, and, in this case, an error message is printed.

`[R,p] = cholinc(X,'0')` never produces an error message. If the incomplete factor exists, then `p = 0` and `R` is the upper triangular factor. If the calculation of `R` breaks down due to a zero or negative pivot, then `p` is a positive integer and `R` is an upper triangular matrix of size `q`-by-`n` where `q = p–1`. The sparsity pattern of `R` is the same as the `q`-by-`n` upper triangle of `X` and the `n`-by-`n` product `R'*R` agrees with `X` over the sparsity pattern of its first `q` rows and columns `X(1:q,:)` and `X(:,1:q)`.

`R = cholinc(X,droptol)` computes the incomplete Cholesky factorization of any sparse matrix using a drop tolerance. `droptol` must be a non-negative scalar. `cholinc(X,droptol)` produces an approximation to the complete Cholesky factor returned by `chol(X)`. For increasingly smaller values of the drop tolerance, this approximation improves, until the drop tolerance is 0, at which time the complete Cholesky factorization is produced, as in `chol(X)`.

The off-diagonal entries `R(i,j)` which are smaller in magnitude than the local drop tolerance, which is given by `droptol*norm(X(:,j))/R(i,i)`, are dropped from the factor. The diagonal entries are preserved even if they are too small in an attempt to avoid a singular factor.

`R = cholinc(X,options)` specifies a structure with up to three fields which may be used in any combination: `droptol`, `michol`, `rdiag`. Additional fields are ignored.

`droptol` is the drop tolerance of the incomplete factorization.

If `michol` is `1`, `cholinc` produces the modified incomplete Cholesky factorization which subtracts the dropped elements in any column from the diagonal element of the upper triangular factor. The default value is `0`.

If `rdiag` is `1`, any zeros on the diagonal of the upper triangular factor are replaced by the square root of the product of the drop tolerance and the norm of that column of `X`, `sqrt(droptol*norm(X(:,j)))`. The default is `0`. Note that the `thresh` option available in the incomplete LU factorization (see `luinc`) is not here as it is always set to `0`. There are never any row interchanges during the formation of the incomplete Cholesky factor.

`R = cholinc(X,droptol)` and `R = cholinc(X,options)` return an upper triangular matrix in `R`. The product `R'*R` is an approximation to the matrix `X`.

Remarks

These incomplete factorizations may be useful as preconditioners for solving large sparse systems of linear equations. 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 `rdiag` option to replace a zero diagonal only gets rid of the symptoms of the problem, but it does not solve it. The preconditioner may not be singular, but it probably is not useful, and a warning message is printed.

Examples

Start with a symmetric positive definite matrix, `S`.

```
S = delsq(numgrid('C',15));
```

`S` is the two-dimensional, five-point discrete negative Lapacian on the grid generated by `numgrid('C',15)`.

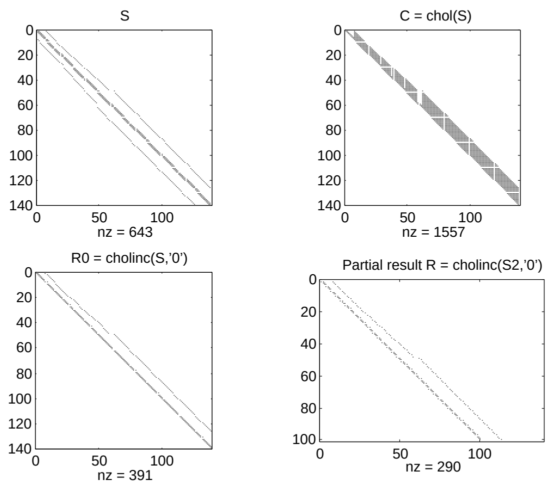
Compute the Cholesky factorization and the incomplete Cholesky factorization of level 0 to compare the fill-in. Make S singular by zeroing out a diagonal entry and compute the (partial) incomplete Cholesky factorization of level 0.

```
C = chol(S);
R0 = chol(S,'0');
S2 = S; S2(101,101) = 0;
[R,p] = cholinc(S2,'0');
```

There is fill-in within the bands of S in the complete Cholesky factor, but none in the incomplete Cholesky factor. The incomplete factorization of the singular S2 stopped at row p = 101 resulting in a 100-by-139 partial factor.

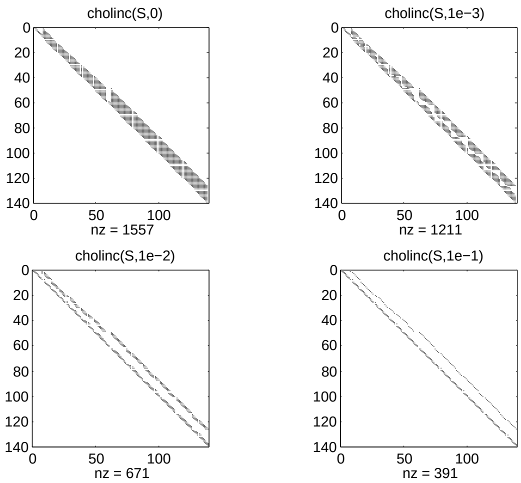
```
D1 = (R0'*R0).*spones(S)-S;
D2 = (R'*R).*spones(S2)-S2;
```

D1 has elements of the order of eps, showing that R0'*R0 agrees with S over its sparsity pattern. D2 has elements of the order of eps over its first 100 rows and first 100 columns, D2(1:100,:) and D2(:,1:100).

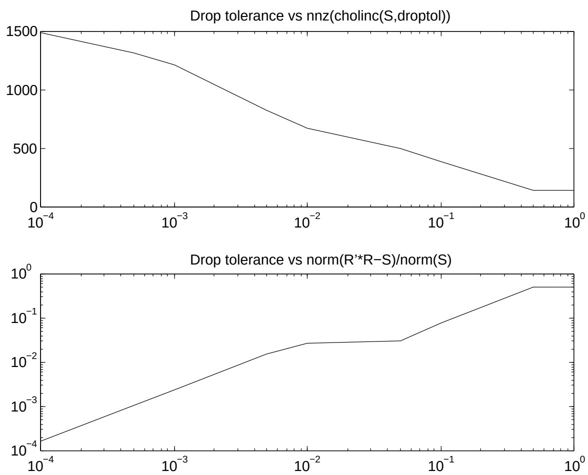


The first subplot below shows that cholinc(S,0), the incomplete Cholesky factor with a drop tolerance of 0, is the same as the Cholesky factor of S.

Increasing the drop tolerance increases the sparsity of the incomplete factors, as seen below.



Unfortunately, the sparser factors are poor approximations, as is seen by the plot of drop tolerance versus `norm(R'*R-S,1)/norm(S,1)` in the next figure.



Limitations

`cholinc` works on square matrices only. For `cholinc(X,'0')`, `X` must be real.

Algorithm

`R = cholinc(X,droptol)` is obtained from `[L,U] = luinc(X,options)`, where `options.droptol = droptol` and `options.thresh = 0`. The rows of the upper-triangular `U` are scaled by the square root of the diagonal in that row, and this scaled factor becomes `R`.

`R = cholinc(X,options)` is produced in a similar manner, except the `rdiag` option translates into the `udiag` option and the `milu` option takes the value of the `michol` option.

`cholinc(X,'0')` is based on the "KJI" variant of the Cholesky factorization. Updates are made only to positions which are nonzero in the upper triangle of `X`.

See Also

`chol`	Cholesky factorization
luinc	Incomplete LU matrix factorizations
pcg	Preconditioned Conjugate Gradients method

References

Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

Purpose Create object or return class of object

Syntax

```
str = class(object)
obj = class(s,'class_name')
obj = class(s,'class_name',parent1,parent2...)
```

Description `str = class(`*`object`*`)` returns a string specifying the class of *object*.

The possible object classes are:

`cell`	Multidimensional cell array
`double`	Multidimensional double precision array
`sparse`	Two-dimensional real (or complex) sparse array
`char`	Array of alphanumeric characters
`struct`	Structure
`'`*`class_name`*`'`	User-defined object class

`obj = class(s,'`*`class_name`*`')` creates an object of class `'`*`class_name`*`'` using structure `s` as a template. This syntax is only valid in a function named *class_name*`.m` in a directory named `@`*`class_name`* (where `'`*`class_name`*`'` is the same as the string passed into `class`).

NOTE On VMS, the method directory is named `#class_name`.

`obj = class(s,'`*`class_name`*`',`*`parent1`*`,`*`parent2`*`,...)` creates an object of class `'`*`class_name`*`'` using structure `s` as a template, and also ensures that the newly created object inherits the methods and fields of the parent objects *parent1*, *parent2*, and so on.

See Also

`inferiorto`	Inferior class relationship
`isa`	Detect an object of a given class
`superiorto`	Superior class relationship

clear

Purpose

Remove items from memory

Syntax

```
clear
clear name
clear name1 name2 name3...
clear global name
```

`clear` *`keyword`* where *keyword* is one of: `functions`, `variables`, `mex`, `global`, `all`

Description

`clear`, by itself, clears all variables from the workspace.

`clear` *`name`* removes just the M-file or MEX-file function or variable *name* from the workspace. If *name* is global, it is removed from the current workspace, but left accessible to any functions declaring it global.

`clear` *`name1 name2 name3`* removes *name1*, *name2*, and *name3* from the workspace.

`clear global` *`name`* removes the global variable *name*.

`clear` *`keyword`* clears the indicated items:

`clear functions`	Clears all the currently compiled M-functions from memory.
`clear variables`	Clears all variables from the workspace.
`clear mex`	Clears all MEX-files from memory.
`clear global`	Clears all global variables.
`clear all`	Removes all variables, functions, and MEX-files from memory, leaving the workspace empty.

Remarks

You can use wildcards (*) to remove items selectively. For instance, `clear my*` removes any variables whose names begin with the string "my." The function form of the syntax, `clear('name')`, is also permitted.

Limitations

`clear` doesn't affect the amount of memory allocated to the MATLAB process under UNIX.

See Also

`pack`	Consolidate workspace memory

clock

Purpose Current time as a date vector

Syntax

```
c = clock
```

Description

`c = clock` returns a 6-element date vector containing the current date and time in decimal form:

```
c = [year month day hour minute seconds]
```

The first five elements are integers. The seconds element is accurate to several digits beyond the decimal point. The statement `fix(clock)` rounds to integer display format.

See Also

`cputime`	CPU time in seconds
`datenum`	Serial date number
`datevec`	Date components
`etime`	Elapsed time
`tic`	Start a stopwatch timer
`toc`	Read the stopwatch timer

Purpose

Sparse column minimum degree permutation

Syntax

```
p = colmmd(S)
```

Description

`p = colmmd(S)` returns the column minimum degree permutation vector for the sparse matrix `S`. For a nonsymmetric matrix `S`, this is a column permutation `p` such that `S(:,p)` tends to have sparser LU factors than `S`.

The `colmmd` permutation is automatically used by `\` and `/` for the solution of nonsymmetric and symmetric indefinite sparse linear systems.

Use `spparms` to change some options and parameters associated with heuristics in the algorithm.

Algorithm

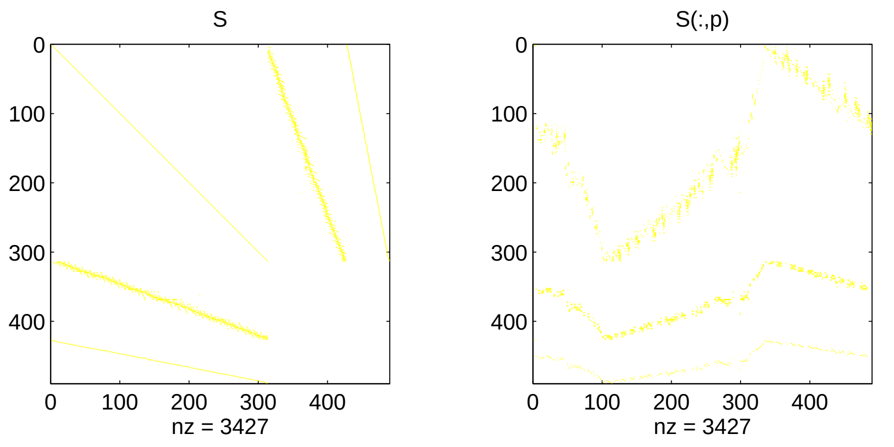
The minimum degree algorithm for symmetric matrices is described in the review paper by George and Liu [1]. For nonsymmetric matrices, MATLAB's minimum degree algorithm is new and is described in the paper by Gilbert, Moler, and Schreiber [2]. It is roughly like symmetric minimum degree for `A'*A`, but does not actually form `A'*A`.

Each stage of the algorithm chooses a vertex in the graph of `A'*A` of lowest degree (that is, a column of `A` having nonzero elements in common with the fewest other columns), eliminates that vertex, and updates the remainder of the graph by adding fill (that is, merging rows). If the input matrix `S` is of size `m`-by-`n`, the columns are all eliminated and the permutation is complete after `n` stages. To speed up the process, several heuristics are used to carry out multiple stages simultaneously.

Examples

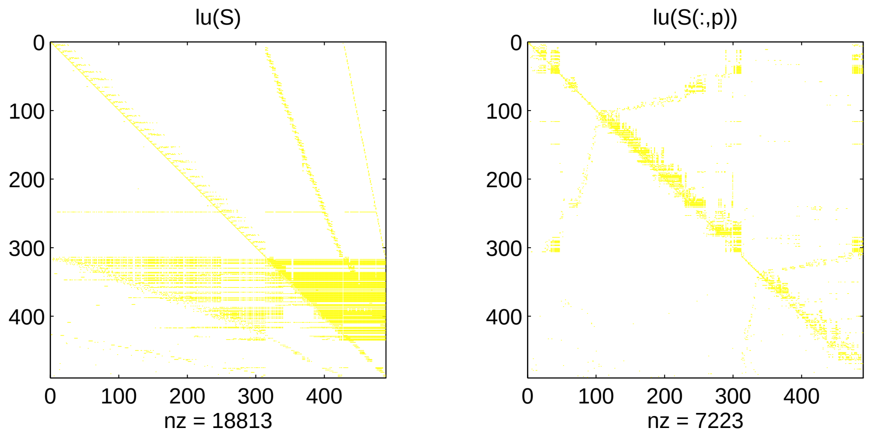
The Harwell-Boeing collection of sparse matrices includes a test matrix ABB313. It is a rectangular matrix, of order 313-by-176, associated with least squares adjustments of geodesic data in the Sudan. Since this is a least squares problem, form the augmented matrix (see `spaugment`), which is square and of order 489. The spy plot shows that the nonzeros in the original matrix are concentrated in two stripes, which are reflected and supplemented with a scaled identity in the augmented matrix. The `colmmd` ordering scrambles this structure. (Note that this example requires the Harwell-Boeing collection of software.)

```
load('abb313.mat')
S = spaugment(A);
p = colmmd(S);
spy(S)
spy(S(:,p))
```



Comparing the spy plot of the LU factorization of the original matrix with that of the reordered matrix shows that minimum degree reduces the time and storage requirements by better than a factor of 2.6. The nonzero counts are 18813 and 7223, respectively.

```
spy(lu(S))
spy(lu(S(:,p)))
```



See Also

`\`	Backslash or matrix left division
`colperm`	Sparse column permutation based on nonzero count
`lu`	LU matrix factorization
`spparms`	Set parameters for sparse matrix routines
`symmmd`	Sparse symmetric minimum degree ordering
`symrcm`	Sparse reverse Cuthill-McKee ordering

References

[1] George, Alan and Liu, Joseph, “The Evolution of the Minimum Degree Ordering Algorithm,” *SIAM Review*, 1989, 31:1-19,.

[2] Gilbert, John R., Cleve Moler, and Robert Schreiber, “Sparse Matrices in MATLAB: Design and Implementation,” *SIAM Journal on Matrix Analysis and Applications 13*, 1992, pp. 333-356.

colperm

Purpose Sparse column permutation based on nonzero count

Syntax

```
j = colperm(S)
```

Description `j = colperm(S)` generates a permutation vector `j` such that the columns of `S(:,j)` are ordered according to increasing count of nonzero entries. This is sometimes useful as a preordering for LU factorization; in this case use `lu(S(:,j))`.

If `S` is symmetric, then `j = colperm(S)` generates a permutation `j` so that both the rows and columns of `S(j,j)` are ordered according to increasing count of nonzero entries. If `S` is positive definite, this is sometimes useful as a preordering for Cholesky factorization; in this case use `chol(S(j,j))`.

Algorithm The algorithm involves a sort on the counts of nonzeros in each column.

Examples The `n`-by-`n` *arrowhead* matrix

```
A = [ones(1,n); ones(n–1,1) speye(n–1,n–1)]
```

has a full first row and column. Its LU factorization, `lu(A)`, is almost completely full. The statement

```
j = colperm(A)
```

returns `j = [2:n 1]`. So `A(j,j)` sends the full row and column to the bottom and the rear, and `lu(A(j,j))` has the same nonzero structure as `A` itself.

On the other hand, the Bucky ball example, `B = bucky`,

has exactly three nonzero elements in each row and column, so `j = colperm(B)` is the identity permutation and is no help at all for reducing fill-in with subsequent factorizations.

See Also

`chol`	Cholesky factorization
`colmmd`	Sparse minimum degree ordering
`lu`	LU matrix factorization
`symrcm`	Sparse reverse Cuthill-McKee ordering

Purpose Companion matrix

Syntax

```
A = compan(u)
```

Description

`A = compan(u)` returns the corresponding companion matrix whose first row is `-u(2:n)/u(1)`, where `u` is a vector of polynomial coefficients. The eigenvalues of `compan(u)` are the roots of the polynomial.

Examples

The polynomial $(x-1)(x-2)(x+3) = x^3 - 7x + 6$ has a companion matrix given by

```
u = [1  0  -7  6]
A = compan(u)
A =
    0    7   -6
    1    0    0
    0    1    0
```

The eigenvalues are the polynomial roots:

```
eig(compan(u))
ans =
    -3.0000
     2.0000
     1.0000
```

This is also `roots(u)`.

See Also

`eig`	Eigenvalues and eigenvectors
`poly`	Polynomial with specified roots
`polyval`	Polynomial evaluation
`roots`	Polynomial roots

computer

Purpose Identify the computer on which MATLAB is running

Syntax

```
str = computer
[str,maxsize] = computer
```

Description `str = computer` returns a string with the computer type on which MATLAB is running.

`[str,maxsize] = computer` returns the integer `maxsize`, which contains the maximum number of elements allowed in an array with this version of MATLAB.

The list of supported computers changes as new computers are added and others become obsolete.

String	Computer
`SUN4`	Sun4 SPARC workstation
`SOL2`	Solaris 2 SPARC workstation
`PCWIN`	MS-Windows
`MAC2`	All Macintosh
`HP700`	HP 9000/700
`ALPHA`	DEC Alpha
`AXP_VMSG`	Alpha VMS G_float
`AXP_VMSIEEE`	Alpha VMS IEEE
`VAX_VMSD`	VAX/VMS D_float

String	Computer
VAX_VMSG	VAX/VMS G_float
LNX86	Linux Intel
SGI	Silicon Graphics (R4000)
SGI64	Silicon Graphics (R8000)
IBM_RS	IBM RS6000 workstation

See Also isieee, isunix, isvms

cond

Purpose Condition number with respect to inversion

Syntax

```
c = cond(X)
c = cond(X,p)
```

Description The *condition number* of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from matrix inversion and the linear equation solution. Values of `cond(X)` and `cond(X,p)` near 1 indicate a well-conditioned matrix.

`c = cond(X)` returns the 2-norm condition number, the ratio of the largest singular value of `X` to the smallest.

`c = cond(X,p)` returns the matrix condition number in p-norm:

```
norm(X,p) * norm(inv(X),p
```

If *p* is...	Then `cond(X,p)` returns the...
`1`	1-norm condition number
`2`	2-norm condition number
`'fro'`	Frobenius norm condition number
`inf`	Infinity norm condition number

Algorithm The algorithm for `cond` (when `p = 2`) uses the singular value decomposition, `svd`.

See Also

`condeig`	Condition number with respect to eigenvalues
`condest`	1-norm matrix condition number estimate
`norm`	Vector and matrix norms
`rank`	Rank of a matrix
`svd`	Singular value decomposition

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose Condition number with respect to eigenvalues

Syntax

```
c = condeig(A)
[V,D,s] = condeig(A)
```

Description `c = condeig(A)` returns a vector of condition numbers for the eigenvalues of `A`. These condition numbers are the reciprocals of the cosines of the angles between the left and right eigenvectors.

`[V,D,s] = condeig(A)` is equivalent to: `[V,D] = eig(A); s = condeig(A);`.

Large condition numbers imply that `A` is near a matrix with multiple eigenvalues.

See Also

`balance`	Improve accuracy of computed eigenvalues
`cond`	Condition number with respect to inversion
`eig`	Eigenvalues and eigenvectors

condest

Purpose

1-norm matrix condition number estimate

Syntax

```
c = condest(A)
[c,v] = condest(A)
```

Description

`c = condest(A)` uses Higham's modification of Hager's method to estimate the condition number of a matrix. The computed `c` is a lower bound for the condition of `A` in the 1-norm.

`[c,v] = condest(A)` estimates the condition number and also computes a vector `v` such that $\|Av\| = \|A\|\|v\|/c$.

Thus, `v` is an approximate null vector of `A` if `c` is large.

This function handles both real and complex matrices. It is particularly useful for sparse matrices.

See Also

`cond`	Condition number with respect to inversion
`normest`	2-norm estimate

Reference

[1] Higham, N.J. "Fortran Codes for Estimating the One-Norm of a Real or Complex Matrix, with Applications to Condition Estimation." *ACM Trans. Math. Soft.*, 14, 1988, pp. 381-396.

Purpose Complex conjugate

Syntax

```
ZC = conj(Z)
```

Description `ZC = conj(Z)` returns the complex conjugate of the elements of `Z`.

Algorithm If `Z` is a complex array:

```
conj(Z) = real(Z) - i*imag(Z)
```

See Also

`i`, `j`	Imaginary unit ($\sqrt{-1}$)
`imag`	Imaginary part of a complex number
`real`	Real part of a complex number

conv

Purpose Convolution and polynomial multiplication

Syntax

```
w = conv(u,v)
```

Description

`w = conv(u,v)` convolves vectors `u` and `v`. Algebraically, convolution is the same operation as multiplying the polynomials whose coefficients are the elements of `u` and `v`.

Definition

Let `m = length(u)` and `n = length(v)`. Then `w` is the vector of length `m+n-1` whose kth element is

$$w(k) = \sum_j u(j)v(k+1-j)$$

The sum is over all the values of `j` which lead to legal subscripts for `u(j)` and `v(k+1-j)`, specifically `j = max(1,k+1-n):min(k,m)`. When `m = n`, this gives

```
w(1) = u(1)*v(1)
w(2) = u(1)*v(2)+u(2)*v(1)
w(3) = u(1)*v(3)+u(2)*v(2)+u(3)*v(1)
...
w(n) = u(1)*v(n)+u(2)*v(n-1)+ ... +u(n)*v(1)
...
w(2*n-1) = u(n)*v(n)
```

Algorithm

The convolution theorem says, roughly, that convolving two sequences is the same as multiplying their Fourier transforms. In order to make this precise, it is necessary to pad the two vectors with zeros and ignore roundoff error. Thus, if

`X = fft([x zeros(1,length(y)-1)])` and `Y = fft([y zeros(1,length(x)-1)])`

then `conv(x,y) = ifft(X.*Y)`

See Also

`convmtx`, `xconv2`, `xcorr`, in the Signal Processing Toolbox, and:

`deconv`	Deconvolution and polynomial division
`filter`	Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter

Purpose Two-dimensional convolution

Syntax

```
C = conv2(A,B)
C = conv2(hcol,hrow,A)
C = conv2(...,'shape')
```

Description

`C = conv2(A,B)` computes the two-dimensional convolution of matrices `A` and `B`. If one of these matrices describes a two-dimensional FIR filter, the other matrix is filtered in two dimensions.

The size of `C` in each dimension is equal to the sum of the corresponding dimensions of the input matrices, minus one. That is, if the size of `A` is `[ma,na]` and the size of `B` is `[mb,nb]`, then the size of `C` is `[ma+mb-1,na+nb-1]`.

`C = conv2(hcol,hrow,A)` convolves `A` separably with `hcol` in the column direction and `hrow` in the row direction. `hcol` and `hrow` should both be vectors.

`C = conv2(...,'`*shape*`')` returns a subsection of the two-dimensional convolution, as specified by the *shape* parameter:

`full`	Returns the full two-dimensional convolution (default).
`same`	Returns the central part of the convolution of the same size as `A`.
`valid`	Returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, `C` has size `[ma-mb+1,na-nb+1]` when `size(A)` > `size(B)`.

Examples

In image processing, the Sobel edge finding operation is a two-dimensional convolution of an input array with the special matrix

```
s = [1 2 1; 0 0 0; -1 -2 -1];
```

These commands extract the horizontal edges from a raised pedestal:

```
A = zeros(10);
A(3:7,3:7) = ones(5);
H = conv2(A,s);
mesh(H)
```

These commands display first the vertical edges of A, then both horizontal and vertical edges.

```
V = conv2(A,s');
mesh(V)
mesh(sqrt(H.^2+V.^2))
```

See Also

conv	Convolution and polynomial multiplication
deconv	Deconvolution and polynomial division
filter2	Two-dimensional digital filtering
xcorr2	Two-dimensional cross-correlation (see Signal Processing Toolbox)

Purpose Convex hull

Syntax

```
K = convhull(x,y)
K = convhull(x,y,TRI)
```

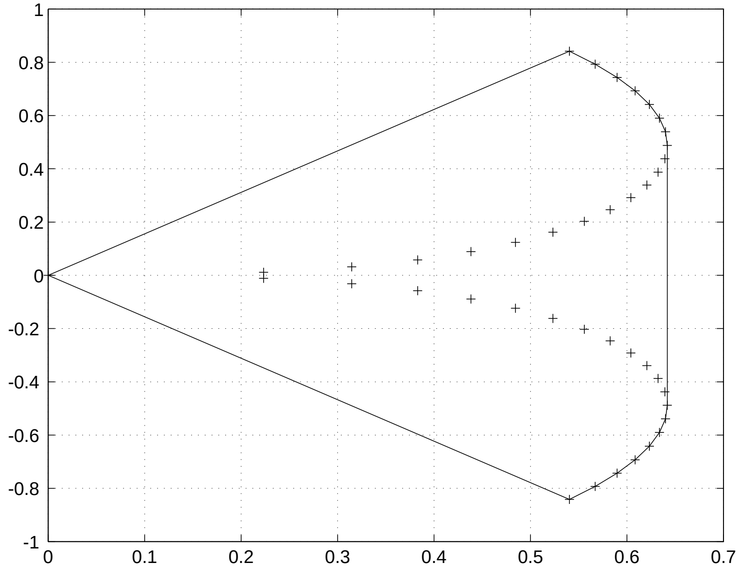
Description

`K = convhull(x,y)` returns indices into the `x` and `y` vectors of the points on the convex hull.

`K = convhull(x,y,TRI)` uses the triangulation (as obtained from `delaunay`) instead of computing it each time.

Examples

```
xx = -1:.05:1; yy = abs(sqrt(xx));
[x,y] = pol2cart(xx,yy);
k = convhull(x,y);
plot(x(k),y(k),'r-',x,y,'b+')
```



See Also

`delaunay`	Delauney triangulation
`polyarea`	Area of polygon
`voronoi`	Voronoi diagram

convn

Purpose

N-dimensional convolution

Syntax

```
C = convn(A,B)
C = convn(A,B,'shape')
```

Description

`C = convn(A,B)` computes the N-dimensional convolution of the arrays `A` and `B`. The size of the result is `size(A)+size(B)–1`.

`C = convn(A,B,'`*shape*`')` returns a subsection of the N-dimensional convolution, as specified by the *shape* parameter:

- `'full'` returns the full N-dimensional convolution (default).
- `'same'` returns the central part of the result that is the same size as `A`.
- `'valid'` returns only those parts of the convolution that can be computed without assuming that the array `A` is zero-padded. The size of the result is

  `max(size(A)–size(B) + 1, 0).`

See Also

`conv`	Convolution and polynomial multiplication
`conv2`	Two-dimensional convolution

Purpose

Correlation coefficients

Syntax

```
S = corrcoef(X)
S = corrcoef(x,y)
```

Description

`S = corrcoef(X)` returns a matrix of correlation coefficients calculated from an input matrix whose rows are observations and whose columns are variables. The matrix `S = corrcoef(X)` is related to the covariance matrix `C = cov(X)` by

$$S(i,\ j) = \frac{C(i,\ j)}{\sqrt{C(i,\ i)C(j,\ j)}}$$

`corrcoef(X)` is the zeroth lag of the covariance function, that is, the zeroth lag of `xcov(x,'coeff')` packed into a square array.

`S = corrcoef(x,y)` where `x` and `y` are column vectors is the same as `corrcoef([x y])`.

See Also

`xcorr`, `xcov` in the Signal Processing Toolbox, and:

`cov`	Covariance matrix
`mean`	Average or mean value of arrays
`std`	Standard deviation

cos, cosh

Purpose Cosine and hyperbolic cosine

Syntax

```
Y = cos(X)
Y = cosh(X)
```

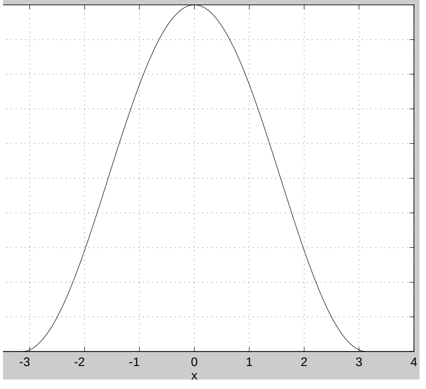
Description The cos and cosh functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

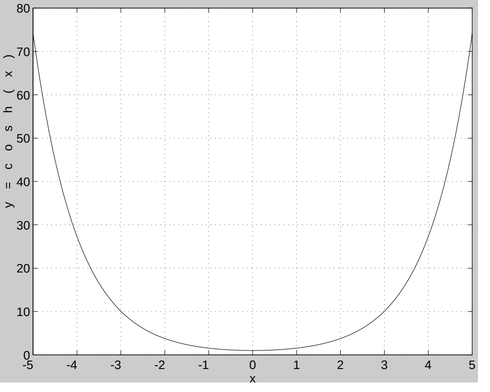
`Y = cos(X)` returns the circular cosine for each element of X.

`Y = cosh(X)` returns the hyperbolic cosine for each element of X.

Examples Graph the cosine function over the domain $-\pi \le x \le \pi$, and the hyperbolic cosine function over the domain $-5 \le x \le 5$.

```
x = -pi:0.01:pi; plot(x,cos(x))
x = -5:0.01:5; plot(x,cosh(x))
```





The expression cos(pi/2) is not exactly zero but a value the size of the floating-point accuracy, eps, because pi is only a floating-point approximation to the exact value of π.

Algorithm

$$\cos(x+iy) = \cos(x)\cosh(y) - i\sin(x)\sin(y)$$

$$\cos(z) = \frac{e^{iz}+e^{-iz}}{2}$$

$$\cosh(z) = \frac{e^{z}+e^{-z}}{2}$$

See Also acos, acosh Inverse cosine and inverse hyperbolic cosine

Purpose Cotangent and hyperbolic cotangent

Syntax

```
Y = cot(X)
Y = coth(X)
```

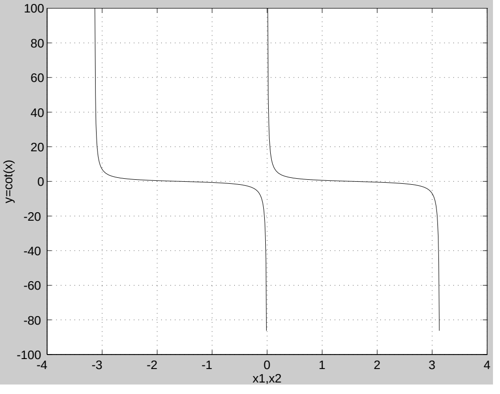
Description The cot and coth functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

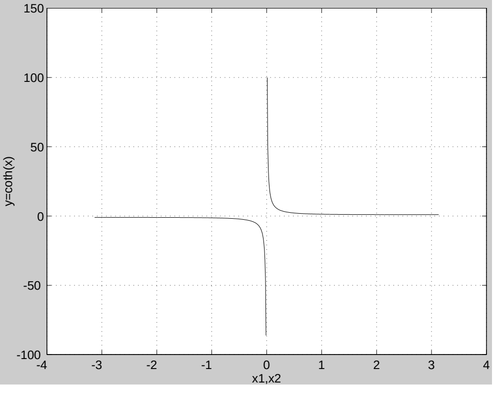
`Y = cot(X)` returns the cotangent for each element of `X`.

`Y = coth(X)` returns the hyperbolic cotangent for each element of `X`.

Examples Graph the cotangent and hyperbolic cotangent over the domains $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01; x2 = 0.01:0.01:pi-0.01;
plot(x1,cot(x1),x2,cot(x2))
plot(x1,coth(x1),x2,coth(x2))
```





Algorithm

$$\cot(z) = \frac{1}{\tan(z)}$$

$$\coth(z) = \frac{1}{\tanh(z)}$$

See Also acot, acoth Inverse cotangent and inverse hyperbolic cotangent

cov

Purpose

Covariance matrix

Syntax

```
C = cov(X)
C = cov(x,y)
```

Description

`C = cov(x)` where `x` is a vector returns the variance of the vector elements. For matrices where each row is an observation and each column a variable, `cov(x)` is the covariance matrix. `diag(cov(x))` is a vector of variances for each column, and `sqrt(diag(cov(x)))` is a vector of standard deviations.

`C = cov(x,y)`, where `x` and `y` are column vectors of equal length, is equivalent to `cov([x y])`.

Remarks

`cov` removes the mean from each column before calculating the result.

The *covariance* function is defined as

where E is the mathematical expectation and $\mu_i = Ex_i$.

Examples

Consider `A = [-1 1 2 ; -2 3 1 ; 4 0 3]`. To obtain a vector of variances for each column of `A`:

```
v = diag(cov(A))'
v =
   10.3333    2.3333    1.0000
```

Compare vector `v` with covariance matrix `C`:

```
C =
   10.3333   -4.1667    3.0000
   -4.1667    2.3333   -1.5000
    3.0000   -1.5000    1.0000
```

The diagonal elements `C(i,i)` represent the variances for the columns of `A`. The off-diagonal elements `C(i,j)` represent the covariances of columns `i` and `j`.

See Also

`xcorr`, `xcov` in the Signal Processing Toolbox, and:

`corrcoef`	Correlation coefficients
`mean`	Average or mean value of arrays
`std`	Standard deviation

Purpose

Sort complex numbers into complex conjugate pairs

Syntax

```
B = cplxpair(A)
B = cplxpair(A,tol)
B = cplxpair(A,[],dim)
B = cplxpair(A,tol,dim)
```

Description

`B = cplxpair(A)` sorts the elements along different dimensions of a complex array, grouping together complex conjugate pairs.

The conjugate pairs are ordered by increasing real part. Within a pair, the element with negative imaginary part comes first. The purely real values are returned following all the complex pairs. The complex conjugate pairs are forced to be exact complex conjugates. A default tolerance of `100*eps` relative to `abs(A(i))` determines which numbers are real and which elements are paired complex conjugates.

If `A` is a vector, `cplxpair(A)` returns `A` with complex conjugate pairs grouped together.

If `A` is a matrix, `cplxpair(A)` returns `A` with its columns sorted and complex conjugates paired.

If `A` is a multidimensional array, `cplxpair(A)` treats the values along the first non-singleton dimension as vectors, returning an array of sorted elements.

`B = cplxpair(A,`*`tol`*`)` overrides the default tolerance.

`B = cplxpair(A,[],`*`dim`*`)` sorts `A` along the dimension specified by scalar *`dim`*.

`B = cplxpair(A,`*`tol`*`,`*`dim`*`)` sorts `A` along the specified dimension and overrides the default tolerance.

Diagnostics

If there are an odd number of complex numbers, or if the complex numbers cannot be grouped into complex conjugate pairs within the tolerance, `cplxpair` generates the error message:

```
Complex numbers can't be paired.
```

cputime

Purpose Elapsed CPU time

Syntax

```
cputime
```

Description

`cputime` returns the total CPU time (in seconds) used by MATLAB from the time it was started. This number can overflow the internal representation and wrap around.

Examples

For example

```
t = cputime; surf(peaks(40)); e = cputime-t

e =

     0.4667
```

returns the CPU time used to run `surf(peaks(40))`.

See Also

`clock`	Current time as a date vector
`etime`	Elapsed time
`tic`, `toc`	Stopwatch timer

Purpose

Vector cross product

Syntax

```
W = cross(U,V)
W = cross(U,V,dim)
```

Description

`W = cross(U,V)` returns the cross product of the vectors `U` and `V`. That is, `W = U x V`. `U` and `V` are usually 3-element vectors. If `U` and `V` are multidimensional arrays, `cross` returns the cross product of `U` and `V` along the first dimension of length 3.

If `U` and `V` are arrays, `cross(U,V)` treats the first size 3 dimension of `U` and `V` as vectors, returning pages whose columns are cross products.

`W = cross(U,V,dim)` where `U` and `V` are multidimensional arrays, returns the cross product of `U` and `V` in dimension `dim`. `U` and `V` must have the same size, and both `size(U,dim)` and `size(V,dim)` must be 3.

Remarks

To perform a dot (scalar) product of two vectors of the same size, use:

`c = sum(a.*b)` or, if `a` and `b` are row vectors, `c = a.'*b`.

Examples

The cross and dot products of two vectors are calculated as shown:

```
a = [1 2 3]; b = [4 5 6];
c = cross(a,b)

c =

    -3     6    -3

d = sum(a.*b)

d =

    32
```

csc, csch

Purpose Cosecant and hyperbolic cosecant

Syntax

```
Y = csc(x)
Y = csch(x)
```

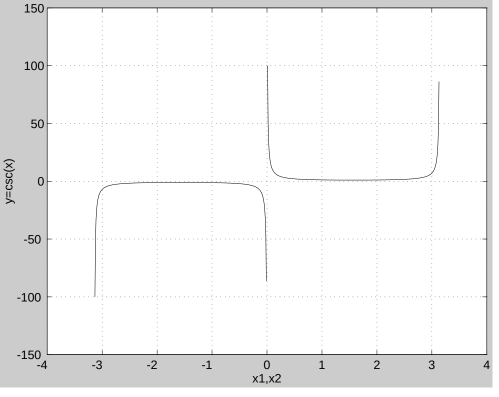
Description The csc and csch functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

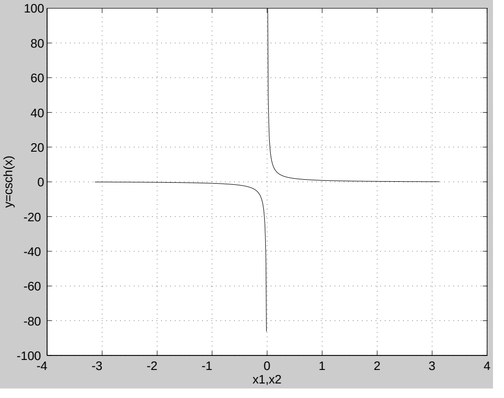
`Y = csc(x)` returns the cosecant for each element of `x`.

`Y = csch(x)` returns the hyperbolic cosecant for each element of `x`.

Examples Graph the cosecant and hyperbolic cosecant over the domains $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01; x2 = 0.01:0.01:pi-0.01;
plot(x1,csc(x1),x2,csc(x2))
plot(x1,csch(x1),x2,csch(x2))
```





Algorithm

$$\csc(z) = \frac{1}{\sin(z)}$$

$$\operatorname{csch}(z) = \frac{1}{\sinh(z)}$$

See Also acsc, acsch Inverse cosecant and inverse hyperbolic cosecant

Purpose

Cumulative product

Syntax

```
B = cumprod(A)
B = cumprod(A,dim)
```

Description

`B = cumprod(A)` returns the cumulative product along different dimensions of an array.

If `A` is a vector, `cumprod(A)` returns a vector containing the cumulative product of the elements of `A`.

If `A` is a matrix, `cumprod(A)` returns a matrix the same size as `A` containing the cumulative products for each column of `A`.

If `A` is a multidimensional array, `cumprod(A)` works on the first nonsingleton dimension.

`B = cumprod(A,dim)` returns the cumulative product of the elements along the dimension of `A` specified by scalar `dim`. For example, `cumprod(A,1)` increments the first (row) index, thus working along the rows of `A`.

Examples

```
cumprod(1:5)  =  [1  2  6  24  120]

A = [1 2 3; 4 5 6];

disp(cumprod(A))
     1     2     3
     4    10    18

disp(cumprod(A,2))
     1     2     6
     4    20   120
```

See Also

`cumsum`	Cumulative sum
`prod`	Product of array elements
`sum`	Sum of array elements

cumsum

Purpose Cumulative sum

Syntax

```
B = cumsum(A)
B = cumsum(A,dim)
```

Description

`B = cumsum(A)` returns the cumulative sum along different dimensions of an array.

If `A` is a vector, `cumsum(A)` returns a vector containing the cumulative sum of the elements of `A`.

If `A` is a matrix, `cumsum(A)` returns a matrix the same size as `A` containing the cumulative sums for each column of `A`.

If `A` is a multidimensional array, `cumsum(A)` works on the first nonsingleton dimension.

`B = cumsum(A,dim)` returns the cumulative sum of the elements along the dimension of `A` specified by scalar `dim`. For example, `cumsum(A,1)` works across the first dimension (the rows).

Examples

```
cumsum(1:5)  =  [1  3  6  10  15]

A = [1 2 3; 4 5 6];

disp(cumsum(A))
     1     2     3
     5     7     9

disp(cumsum(A,2))
     1     3     6
     4     9    15
```

See Also

`sum`	Sum of array elements
`prod`	Product of array elements
`cumprod`	Cumulative product of elements

Purpose Cumulative trapezoidal numerical integration

Syntax

```
Z = cumtrapz(Y)
Z = cumtrapz(X,Y)
Z = cumtrapz(... dim)
```

Description `Z = cumtrapz(Y)` computes an approximation of the cumulative integral of `Y` via the trapezoidal method with unit spacing. (This is similar to `cumsum(Y)`, except that trapezoidal approximation is used.) To compute the integral with other than unit spacing, multiply `Z` by the spacing increment.

For vectors, `cumtrapz(Y)` is the cumulative integral of `Y`.

For matrices, `cumtrapz(Y)` is a row vector with the cumulative integral over each column.

For multidimensional arrays, `cumtrapz(Y)` works across the first nonsingleton dimension.

`Z = cumtrapz(X,Y)` computes the cumulative integral of `Y` with respect to `X` using trapezoidal integration. `X` and `Y` must be vectors of the same length, or `X` must be a column vector and `Y` an array.

If `X` is a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`, `cumtrapz(X,Y)` operates across this dimension.

`Z = cumtrapz(... dim)` integrates across the dimension of `Y` specified by scalar `dim`. The length of `X` must be the same as `size(Y,dim)`.

See Also

`cumsum`	Cumulative sum
`trapz`	Trapezoidal numerical integration

date

Purpose Current date string

Syntax

```
str = date
```

Description `str = date` returns a string containing the date in `dd-mmm-yyyy` format.

See Also

`clock`	Current time as a date vector
`datenum`	Serial date number
`now`	Current date and time

Purpose

Serial date number

Syntax

```
N = datenum(str)
N = datenum(Y,M,D)
N = datenum(Y,M,D,H,MI,S)
```

Description

The `datenum` function converts date strings and date vectors into serial date numbers. Date numbers are serial days elapsed from some reference date. By default, the serial day 1 corresponds to 1-Jan-0000.

`N = datenum(`*`str`*`)` converts the date string *`str`* into a serial date number.

NOTE The string *`str`* must be in one of the date formats `0`, `1`, `2`, `6`, `13`, `14`, `15`, or `16` as defined by `datestr`.

`N = datenum(Y,M,D)` returns the serial date number for corresponding elements of the `Y`, `M`, and `D` (year, month, day) arrays. `Y`, `M`, and `D` must be arrays of the same size (or any can be a scalar). Values outside the normal range of each array are automatically "carried" to the next unit.

`N = datenum(Y,M,D,H,MI,S)` returns the serial date number for corresponding elements of the `Y`, `M`, `D`, `H`, `MI`, and `S` (year, month, hour, minute, and second) array values. `Y`, `M`, `D`, `H`, `MI`, and `S` must be arrays of the same size (or any can be a scalar).

Examples

`n = datenum('19-May-1995')` returns `n = 728798`.

`n = datenum(1994,12,19)` returns `n = 728647`.

`n = datenum(1994,12,19,18,0,0)` returns `n = 7.2865e+05` .

See Also

`datestr`	Date string format
`datevec`	Date components
`now`	Current date and time

datestr

Purpose Date string format

Syntax

```
str = datestr(D,dateform)
```

Description

`str = datestr(D,dateform)` converts each element of the array of serial date numbers (`D`) to a string. Optional argument *dateform* specifies the date format of the result, where *dateform* can be either a number or a string:

dateform **(number)**	*dateform* **(string)**	**Example**
`0`	`'dd-mmm-yyyy HH:MM:SS'`	`01-Mar-1995 03:45`
`1`	`'dd-mmm-yyyy'`	`01-Mar-1995`
`2`	`'mm/dd/yy'`	`03/01/95`
`3`	`'mmm'`	`Mar`
`4`	`'m'`	`M`
`5`	`'mm'`	`3`
`6`	`'mm/dd'`	`03/01`
`7`	`'dd'`	`1`
`8`	`'ddd'`	`Wed`
`9`	`'d'`	`W`
`10`	`'yyyy'`	`1995`
`11`	`'yy'`	`95`
`12`	`'mmmyy'`	`Mar95`
`13`	`'HH:MM:SS'`	`15:45:17`

dateform (number)	*dateform* (string)	Example
14	'HH:MM:SS PM'	03:45:17 PM
15	'HH:MM'	15:45
16	'HH:MM PM'	03:45 PM
17	'QQ-YY'	Q1–96
18	'QQ'	Q1

NOTE *dateform* numbers `0`, `1`, `2`, `6`, `13`, `14`, `15`, and `16` produce a string suitable for input to `datenum` or `datevec`. Other date string formats will not work with these functions.

Time formats like `'h:m:s'`, `'h:m:s.s'`, `'h:m pm'`, ... may also be part of the input array `D`. If you do not specify *dateform*, the date string format defaults to

- 1, if `D` contains date information only (01-Mar-1995)
- 16, if `D` contains time information only (03:45 PM)
- 0, if `D` contains both date and time information (01-Mar-1995 03:45)

See Also

`date`	Current date string
`datenum`	Serial date number
`datevec`	Date components

datevec

Purpose

Date components

```
C = datevec(A)
[Y,M,D,H,MI,S] = datevec(A)
```

Description

`C = datevec(A)` splits its input into an n-by-6 array with each row containing the vector `[Y,M,D,H,MI,S]`. The first five date vector elements are integers. Input `A` can either consist of strings of the sort produced by the `datestr` function, or scalars of the sort produced by the `datenum` and `now` functions.

`[Y,M,D,H,MI,S] = datevec(A)` returns the components of the date vector as individual variables.

When creating your own date vector, you need not make the components integers. Any components that lie outside their conventional ranges affect the next higher component (so that, for instance, the anomalous June 31 becomes July 1). A zeroth month, with zero days, is allowed.

Examples

Let

```
d = '12/24/1984'
t = '725000.00',
```

Then `datevec(d)` and `datevec(t)` generate `[1984 12 24 0 0 0]`.

See Also

`clock`	Current time as date vector
`datenum`	Serial date number
`datestr`	Date string format

Purpose

Clear breakpoints

Syntax

```
dbclear
dbclear at lineno in function
dbclear all in function
dbclear all
dbclear in mfilename
dbclear if keyword
```

where *keyword* is one of: error, naninf, infnan, warning

Description

The `at`, `in`, and `if` keywords, familiar to users of the UNIX debugger dbx, are optional.

`dbclear`, by itself, clears the breakpoint(s) set by a corresponding `dbstop` command.

`dbclear at` *lineno* `in` *function* clears the breakpoint set at the specified line in the specified M-file. *function* must be the name of an M-file function or a `MATLABPATH` relative partial pathname.

`dbclear all in` *function* clears all breakpoints in the specified M-file.

`dbclear all` clears all breakpoints in all M-file functions, except for errors and warning breakpoints.

`dbclear in` *function* clears the breakpoint set at the first executable line in the specified M-file.

`dbclear if` *keyword* clears the indicated statement or breakpoint:

`dbclear if error`	Clears the `dbstop error` statement, if set. If a runtime error occurs after this command, MATLAB terminates the current operation and returns to the base workspace.
`dbclear if naninf`	Clears the `dbstop naninf` statement, if set.

`dbclear if infnan`	Clears the `dbstop infnan` statement, if set.
`dbclear if warning`	Clears warning breakpoints.

See Also

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

See also partialpath.

Purpose

Resume execution

Syntax

```
dbcont
```

Description

`dbcont` resumes execution of an M-file from a breakpoint. Execution continues until either another breakpoint is encountered, an error occurs, or MATLAB returns to the base workspace prompt.

See Also

`dbclear`	Clear breakpoints
`dbdown`	Change local workspace context (down)
`dbquit`	Quit debug mode
`dbstack`	Display function call stack
`dbstatus`	List all breakpoints
`dbstep`	Execute one or more lines from a breakpoint
`dbstop`	Set breakpoints in an M-file function
`dbtype`	List M-file with line numbers
`dbup`	Change local workspace context (up)

dbdown

Purpose

Change local workspace context

Syntax

```
dbdown
```

Description

`dbdown` changes the current workspace context to the workspace of the called M-file when a breakpoint is encountered. You must have issued the `dbup` command at least once before you issue this command. `dbdown` is the opposite of `dbup`.

Multiple `dbdown` commands change the workspace context to each successively executed M-file on the stack until the current workspace context is the current breakpoint. It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.

See Also

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

Purpose Numerical double integration

Syntax

```
result = dblquad('fun',inmin,inmax,outmin,outmax)
result = dblquad('fun',inmin,inmax,outmin,outmax,tol,trace)
result = dblquad('fun',inmin,inmax,outmin,outmax,tol,trace,order)
```

Description

`result = dblquad('`*`fun`*`',inmin,inmax,outmin,outmax)` evaluates the double integral *fun*(`inner`,`outer`) using the `quad` quadrature function. `inner` is the inner variable, ranging from `inmin` to `inmax`, and `outer` is the outer variable, ranging from `outmin` to `outmax`. The first argument '*fun*' is a string representing the integrand function. This function must be a function of two variables of the form `fout = `*`fun`*`(inner,outer)`. The function must take a vector `inner` and a scalar `outer` and return a vector `fout` that is the function evaluated at `outer` and each value of `inner`.

`result = dblquad('`*`fun`*`',inmin,inmax,outmin,outmax,tol,trace)` passes `tol` and `trace` to the `quad` function. See the help entry for `quad` for a description of the `tol` and `trace` parameters.

`result = dblquad('`*`fun`*`',inmin,inmax,outmin,outmax,tol,trace,order)` passes `tol` and `trace` to the `quad` or `quad8` function depending on the value of the string `order`. Valid values for `order` are `'quad'` and `'quad8'` or the name of any user-defined quadrature method with the same calling and return arguments as `quad` and `quad8`.

Example

`result = dblquad('integrnd',pi,2*pi,0,pi)` integrates the function `y*sin(x)+x*cos(y)`, where `x` ranges from π to 2π, and `y` ranges from 0 to π, assuming:

- `x` is the inner variable in the integration.
- `y` is the outer variable.
- the M-file `integrnd.m` is defined as:

```
function out = integrnd(x, y)
out = y*sin(x)+x*cos(y);
```

Note that `integrnd.m` is valid when `x` is a vector and `y` is a scalar. Also, `x` must be the first argument to `integrnd.m` since it is the inner variable.

See Also	`quad`, `quad8`	Numerical evaluation of integrals

Purpose Enable MEX-file debugging

Syntax

```
dbmex on
dbmex off
dbmex stop
dbmex print
```

Description

`dbmex on` enables MEX-file debugging. To use this option, first start MATLAB from within a debugger by typing: `matlab –Ddebugger`, where `debugger` is the name of the debugger.

`dbmex off` disables MEX-file debugging.

`dbmex stop` returns to the debugger prompt.

`dbmex print` displays MEX-file debugging information.

`dbmex` is not available on the Macintosh or the PC.

See Also

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

dbquit

Purpose Quit debug mode

Syntax

```
dbquit
```

Description `dbquit` immediately terminates the debugger and returns control to the base workspace prompt. The M–file being processed is *not* completed and no results are returned.

All breakpoints remain in effect.

See Also

`dbclear`	Clear breakpoints
`dbcont`	Resume execution
`dbdown`	Change local workspace context (down)
`dbstack`	Display function call stack
`dbstatus`	List all breakpoints
`dbstep`	Execute one or more lines from a breakpoint
`dbstop`	Set breakpoints in an M-file function
`dbtype`	List M-file with line numbers
`dbup`	Change local workspace context (up)

Purpose Display function call stack

Syntax

```
dbstack
[ST,I] = dbstack
```

Description `dbstack` displays the line numbers and M-file names of the function calls that led to the current breakpoint, listed in the order in which they were executed. In other words, the line number of the most recently executed function call (at which the current breakpoint occurred) is listed first, followed by its calling function, which is followed by its calling function, and so on, until the topmost M-file function is reached.

`[ST,I] = dbstack` returns the stack trace information in an `m`-by-`1` structure `ST` with the fields:

`name`	function name
`line`	function line number

The current workspace index is returned in `I`.

Examples

```
>> dbstack
> In /usr/local/matlab/toolbox/matlab/cond.m at line 13
  In test1.m at line 2
  In test.m at line 3
```

See Also

`dbclear`	Clear breakpoints
`dbcont`	Resume execution
`dbdown`	Change local workspace context (down)
`dbquit`	Quit debug mode
`dbstatus`	List all breakpoints
`dbstep`	Execute one or more lines from a breakpoint
`dbstop`	Set breakpoints in an M-file function
`dbtype`	List M-file with line numbers
`dbup`	Change local workspace context (up)

dbstatus

Purpose List all breakpoints

Syntax

```
dbstatus
dbstatus function
s = dbstatus(...)
```

Description

`dbstatus`, by itself, lists all breakpoints in effect including `error`, `warning`, and `naninf`.

`dbstatus` *function* displays a list of the line numbers for which breakpoints are set in the specified M-file.

`s = dbstatus(...)` returns the breakpoint information in an `m`-by-`1` structure with the fields:

`name`	function name
`line`	vector of breakpoint line numbers
`cond`	condition string (`error`, `warning`, or `naninf`)

Use `dbstatus` *class/function* or `dbstatus private`/*function* or `dbstatus` *class*/`private`/*function* to determine the status for methods, private functions, or private methods (for a class named *class*). In all of these forms you can further qualify the function name with a subfunction name as in `dbstatus` *function/subfunction*.

See Also

`dbclear`	Clear breakpoints
`dbcont`	Resume execution
`dbdown`	Change local workspace context (down)
`dbquit`	Quit debug mode
`dbstack`	Display function call stack
`dbstep`	Execute one or more lines from a breakpoint
`dbstop`	Set breakpoints in an M-file function
`dbtype`	List M-file with line numbers
`dbup`	Change local workspace context (up)

Purpose

Execute one or more lines from a breakpoint

Syntax

```
dbstep
dbstep nlines
dbstep in
```

Description

This command allows you to debug an M-file by following its execution from the current breakpoint. At a breakpoint, the `dbstep` command steps through execution of the current M-file one line at a time or at the rate specified by `nlines`.

`dbstep`, by itself, executes the next executable line of the current M-file. `dbstep` steps over the current line, skipping any breakpoints set in functions called by that line.

`dbstep nlines` executes the specified number of executable lines.

`dbstep in` steps to the next executable line. If that line contains a call to another M-file, execution resumes with the first executable line of the called file. If there is no call to an M-file on that line, `dbstep in` is the same as `dbstep`.

See Also

`dbclear`	Clear breakpoints
`dbcont`	Resume execution
`dbdown`	Change local workspace context (down)
`dbquit`	Quit debug mode
`dbstack`	Display function call stack
`dbstatus`	List all breakpoints
`dbstop`	Set breakpoints in an M-file function
`dbtype`	List M-file with line numbers
`dbup`	Change local workspace context (up)

dbstop

Purpose Set breakpoints in an M-file function

Syntax

```
dbstop at lineno in function
dbstop in function
```

`dbstop if` *`keyword`* where *keyword* is one of: `error`, `naninf`, `infnan`, `warning`

Description The `dbstop` command sets up MATLAB's debugging mode. `dbstop` sets a breakpoint at a specified location in an M-file function or causes a break in case an error or warning occurs during execution. When the specified `dbstop` condition is met, the MATLAB prompt is displayed and you can issue any valid MATLAB command.

`dbstop at` *`lineno`* `in` *`function`* stops execution just prior to execution of that line of the specified M-file function. *`function`* must be the name of an M-file function or a `MATLABPATH` relative partial pathname.

`dbstop in` *`function`* stops execution before the first executable line in the M-file function when it is called.

`dbstop if` *`keyword`* stops execution under the specified conditions:

`dbstop if error`	Stops execution if a runtime error occurs in any M-file function. You can examine the local workspace and sequence of function calls leading to the error, but you cannot resume M-file execution after a runtime error.
`dbstop if naninf`	Stops execution when it detects Not-a-Number (`NaN`) or Infinity (`Inf`).
`dbstop if infnan`	Stops execution when it detects Not-a-Number (`NaN`) or Infinity (`Inf`).
`dbstop if warning`	Stops execution if a runtime warning occurs in any M-file function.

Regardless of the form of the `dbstop` command, when a stop occurs, the line or error condition that caused the stop is displayed. To resume M-file function execution, issue a dbcont command or step to another line in the file with the dbstep command.

Any breakpoints set by the first two forms of the dbstop command are cleared if the M-file function is edited or cleared.

The at, in, and if keywords, familiar to users of the UNIX debugger dbx, are optional.

Examples

Here is a short example, printed with the dbtype command to produce line numbers.

```
dbtype buggy
1   function z = buggy(x)
2   n = length(x);
3   z = (1:n)./x;
```

The statement

```
dbstop in buggy
```

causes execution to stop at line 2, the first executable line. The command

```
dbstep
```

then advances to line 3 and allows the value of n to be examined.

The example function only works on vectors; it produces an error if the input x is a full matrix. So the statements

```
dbstop if error
buggy(magic(3))
```

produce

```
Error using ==>./
Matrix dimensions must agree.
Error in ==> buggy.m
On line 3 ==> z = (1:n)./x;
```

Finally, if any of the elements of the input x are zero, a division by zero occurs. For example, consider

```
dbstop if naninf
buggy(0:2)
```

which produces

```
Warning: Divide by zero
NaN/Inf debugging breakpoint hit on line 2.
Stopping at next line.
2   n = length(x);
3   z = (1:n)./x;
```

See Also

`dbclear`	Clear breakpoints
`dbcont`	Resume execution
`dbdown`	Change local workspace context (down)
`dbquit`	Quit debug mode
`dbstack`	Display function call stack
`dbstatus`	List all breakpoints
`dbstep`	Execute one or more lines from a breakpoint
`dbtype`	List M-file with line numbers
`dbup`	Change local workspace context (up)

See also partialpath.

Purpose

List M-file with line numbers

Syntax

```
dbtype function
dbtype function start:end
```

Description

`dbtype` *function* displays the contents of the specified M-file function with line numbers preceding each line. *function* must be the name of an M-file function or a `MATLABPATH` relative partial pathname.

`dbtype` *function start:end* displays the portion of the file specified by a range of line numbers.

See Also

`dbclear`	Clear breakpoints
`dbcont`	Resume execution
`dbdown`	Change local workspace context (down)
`dbquit`	Quit debug mode
`dbstack`	Display function call stack
`dbstatus`	List all breakpoints
`dbstep`	Execute one or more lines from a breakpoint
`dbstop`	Set breakpoints in an M-file function
`dbup`	Change local workspace context (up)

See also partialpath.

dbup

Purpose Change local workspace context

Syntax

```
dbup
```

Description This command allows you to examine the calling M-file by using any other MATLAB command. In this way, you determine what led to the arguments being passed to the called function.

`dbup` changes the current workspace context (at a breakpoint) to the workspace of the calling M-file.

Multiple `dbup` commands change the workspace context to each previous calling M-file on the stack until the base workspace context is reached. (It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.)

See Also

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

Purpose

Set up advisory link

Syntax

```
rc = ddeadv(channel,'item','callback')
rc = ddeadv(channel,'item','callback','upmtx')
rc = ddeadv(channel,'item','callback','upmtx',format)
rc = ddeadv(channel,'item','callback','upmtx',format,timeout)
```

Description

`ddeadv` sets up an advisory link between MATLAB and a server application. When the data identified by the `item` argument changes, the string specified by the `callback` argument is passed to the `eval` function and evaluated. If the advisory link is a hot link, DDE modifies `upmtx`, the update matrix, to reflect the data in `item`.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

Arguments

`rc`	Return code: `0` indicates failure, `1` indicates success.
`channel`	Conversation channel from `ddeinit`.
item	String specifying the DDE item name for the advisory link. Changing the data identified by `item` at the server triggers the advisory link.
callback	String specifying the callback that is evaluated on update notification. Changing the data identified by `item` at the server causes `callback` to get passed to the `eval` function to be evaluated.
upmtx (*optional*)	String specifying the name of a matrix that holds data sent with an update notification. If `upmtx` is included, changing `item` at the server causes `upmtx` to be updated with the revised data. Specifying `upmtx` creates a hot link. Omitting `upmtx` or specifying it as an empty string creates a warm link. If `upmtx` exists in the workspace, its contents are overwritten. If `upmtx` does not exist, it is created.

`format` (*optional*)	Two-element array specifying the format of the data to be sent on update. The first element specifies the Windows clipboard format to use for the data. The only currently supported format is `cf_text`, which corresponds to a value of 1. The second element specifies the type of the resultant matrix. Valid types are `numeric` (the default, which corresponds to a value of `0`) and `string` (which corresponds to a value of `1`). The default format array is `[1 0]`.
`timeout` (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). If advisory link is not established within `timeout` milliseconds, the function fails. The default value of `timeout` is three seconds.

Examples

Set up a hot link between a range of cells in Excel (Row 1, Column 1 through Row 5, Column 5) and the matrix `x`. If successful, display the matrix:

```
rc = ddeadv(channel, 'r1c1:r5c5', 'disp(x)', 'x');
```

Communication with Excel must have been established previously with a `ddeinit` command.

See Also

`ddeexec`	Send string for execution
`ddeinit`	Initiate DDE conversation
`ddepoke`	Send data to application
`ddereq`	Request data from application
`ddeterm`	Terminate DDE conversation
`ddeunadv`	Release advisory link

Purpose

Send string for execution

Syntax

```
rc = ddeexec(channel,'command')
rc = ddeexec(channel,'command','item')
rc = ddeexec(channel,'command','item',timeout)
```

Description

`ddeexec` sends a string for execution to another application via an established DDE conversation. Specify the string as the `command` argument.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

Arguments

`rc`	Return code: `0` indicates failure, `1` indicates success.
`channel`	Conversation channel from `ddeinit`.
`command`	String specifying the command to be executed.
`item` (*optional*)	String specifying the DDE item name for execution. This argument is not used for many applications. If your application requires this argument, it provides additional information for `command`. Consult your server documentation for more information.
`timeout` (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Examples

Given the channel assigned to a conversation, send a command to Excel:

```
rc = ddeexec(channel,'[formula.goto("r1c1")]')
```

Communication with Excel must have been established previously with a `ddeinit` command.

See Also

`ddeadv`	Set up advisory link
`ddeinit`	Initiate DDE conversation
`ddepoke`	Send data to application
`ddereq`	Request data from application
`ddeterm`	Terminate DDE conversation
`ddeunadv`	Release advisory link

ddeinit

Purpose Initiate DDE conversation

Syntax

```
channel = ddeinit('service','topic')
```

Description

`channel = ddeinit('`*`service`*`','`*`topic`*`')` returns a channel handle assigned to the conversation, which is used with other MATLAB DDE functions. `'`*`service`*`'` is a string specifying the service or application name for the conversation. `'`*`topic`*`'` is a string specifying the topic for the conversation.

Examples

To initiate a conversation with Excel for the spreadsheet `'stocks.xls'`:

```
channel = ddeinit('excel','stocks.xls')

channel =
         0.00
```

See Also

`ddeadv`	Set up advisory link
`ddeexec`	Send string for execution
`ddepoke`	Send data to application
`ddereq`	Request data from application
`ddeterm`	Terminate DDE conversation
`ddeunadv`	Release advisory link

Purpose Send data to application

Syntax

```
rc = ddepoke(channel,'item',data)
rc = ddepoke(channel,'item',data,format)
rc = ddepoke(channel,'item',data,format,timeout)
```

Description `ddepoke` sends data to an application via an established DDE conversation. `ddepoke` formats the data matrix as follows before sending it to the server application:

- String matrices are converted, element by element, to characters and the resulting character buffer is sent.
- Numeric matrices are sent as tab-delimited columns and carriage-return, line-feed delimited rows of numbers. Only the real part of nonsparse matrices are sent.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

Arguments

`rc`	Return code: `0` indicates failure, `1` indicates success.
`channel`	Conversation channel from `ddeinit`.
item	String specifying the DDE item for the data sent. Item is the server data entity that is to contain the data sent in the `data` argument.
`data`	Matrix containing the data to send.
`format` (*optional*)	Scalar specifying the format of the data requested. The value indicates the Windows clipboard format to use for the data transfer. The only format currently supported is `cf_text`, which corresponds to a value of `1`.
`timeout` (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Examples

Assume that a conversation channel with Excel has previously been established with `ddeinit`. To send a 5-by-5 identity matrix to Excel, placing the data in Row 1, Column 1 through Row 5, Column 5:

```
rc = ddepoke(channel, 'r1c1:r5c5', eye(5));
```

See Also

`ddeadv`	Set up advisory link
`ddeexec`	Send string for execution
`ddeinit`	Initiate DDE conversation
`ddereq`	Request data from application
`ddeterm`	Terminate DDE conversation
`ddeunadv`	Release advisory link

Purpose

Request data from application

Syntax

```
data = ddereq(channel,'item')
data = ddereq(channel,'item',format)
data = ddereq(channel,'item',format,timeout)
```

Description

`ddereq` requests data from a server application via an established DDE conversation. `ddereq` returns a matrix containing the requested data or an empty matrix if the function is unsuccessful.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

Arguments

`data`	Matrix containing requested data, empty if function fails.
`channel`	Conversation channel from `ddeinit`.
`item`	String specifying the server application's DDE item name for the data requested.
`format` (*optional*)	Two-element array specifying the format of the data requested. The first element specifies the Windows clipboard format to use. The only currently supported format is `cf_text`, which corresponds to a value of 1. The second element specifies the type of the resultant matrix. Valid types are `numeric` (the default, which corresponds to `0`) and `string` (which corresponds to a value of `1`). The default format array is `[1 0]`.
`timeout` (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Examples

Assume that we have an Excel spreadsheet `stocks.xls`. This spreadsheet contains the prices of three stocks in row 3 (columns 1 through 3) and the number of shares of these stocks in rows 6 through 8 (column 2). Initiate conversation with Excel with the command:

```
channel = ddeinit('excel','stocks.xls')
```

DDE functions require the `r`*x*`c`*y* reference style for Excel worksheets. In Excel terminology the prices are in `r3c1:r3c3` and the shares in `r6c2:r8c2`.

ddereq

To request the prices from Excel:

```
prices = ddereq(channel,'r3c1:r3c3')

prices =
         42.50      15.00      78.88
```

To request the number of shares of each stock:

```
shares = ddereq(channel, 'r6c2:r8c2')

shares =
        100.00
        500.00
        300.00
```

See Also

ddeadv	Set up advisory link
ddeexec	Send string for execution
ddeinit	Initiate DDE conversation
ddepoke	Send data to application
ddeterm	Terminate DDE conversation
ddeunadv	Release advisory link

Purpose Terminate DDE conversation

Syntax

```
rc = ddeterm(channel)
```

Description

`rc = ddeterm(channel)` accepts a channel handle returned by a previous call to `ddeinit` that established the DDE conversation. `ddeterm` terminates this conversation. `rc` is a return code where `0` indicates failure and `1` indicates success.

Examples

To close a conversation channel previously opened with `ddeinit`:

```
rc = ddeterm(channel)

rc =

    1.00
```

See Also

`ddeadv`	Set up advisory link
`ddeexec`	Send string for execution
`ddeinit`	Initiate DDE conversation
`ddepoke`	Send data to application
`ddereq`	Request data from application
`ddeunadv`	Release advisory link

ddeunadv

Purpose

Release advisory link

Syntax

```
rc = ddeunadv(channel,'item')
rc = ddeunadv(channel,'item',format)
rc = ddeunadv(channel,'item',format,timeout)
```

Description

`ddeunadv` releases the advisory link between MATLAB and the server application established by an earlier `ddeadv` call. The `channel`, *item*, and `format` must be the same as those specified in the call to `ddeadv` that initiated the link. If you include the `timeout` argument but accept the default `format`, you must specify `format` as an empty matrix.

Arguments

`rc`	Return code: `0` indicates failure, `1` indicates success.
`channel`	Conversation channel from `ddeinit`.
item	String specifying the DDE item name for the advisory link. Changing the data identified by `item` at the server triggers the advisory link.
`format` (*optional*)	Two-element array. This must be the same as the `format` argument for the corresponding `ddeadv` call.
`timeout` (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Example

To release an advisory link established previously with `ddeadv`:

```
rc = ddeadv(channel, 'r1c1:r5c5')
rc =

    1.00
```

See Also

`ddeadv`	Set up advisory link
`ddeexec`	Send string for execution
`ddeinit`	Initiate DDE conversation
`ddepoke`	Send data to application
`ddereq`	Request data from application
`ddeterm`	Release advisory link

Purpose Deal inputs to outputs

Syntax

```
[Y1,Y2,Y3,...] = deal(X)
[Y1,Y2,Y3,...] = deal(X1,X2,X3,...)
```

Description `[Y1,Y2,Y3,...] = deal(X)` copies the single input to all the requested outputs. It is the same as `Y1 = X`, `Y2 = X`, `Y3 = X`, ...

`[Y1,Y2,Y3,...] = deal(X1,X2,X3,...)` is the same as `Y1 = X1`; `Y2 = X2`; `Y3 = X3`; ...

Remarks `deal` is most useful when used with cell arrays and structures via comma separated list expansion. Here are some useful constructions:

`[S.field] = deal(X)` sets all the fields with the name `field` in the structure array `S` to the value `X`. If `S` doesn't exist, use `[S(1:m).field] = deal(X)`.

`[X{:}] = deal(A.field)` copies the values of the field with name `field` to the cell array `X`. If `X` doesn't exist, use `[X{1:m}] = deal(A.field)`.

`[Y1,Y2,Y3,...] = deal(X{:})` copies the contents of the cell array `X` to the separate variables `Y1,Y2,Y3,`...

`[Y1,Y2,Y3,...] = deal(S.field)` copies the contents of the fields with the name `field` to separate variables `Y1,Y2,Y3,`...

Examples

Use deal to copy the contents of a 4-element cell array into four separate output variables.

```
C = {rand(3) ones(3,1) eye(3) zeros(3,1)};
[a,b,c,d] = deal(C{:})

a =

    0.9501   0.4860   0.4565
    0.2311   0.8913   0.0185
    0.6068   0.7621   0.8214

b =

    1
    1
    1

c =

    1   0   0
    0   1   0
    0   0   1

d =

    0
    0
    0
```

Use `deal` to obtain the contents of all the `name` fields in a structure array:

```
A.name = 'Pat'; A.number = 176554;
A(2).name = 'Tony'; A(2).number = 901325;
[name1,name2] = deal(A(:).name)

name1 =

Pat

name2 =

Tony
```

deblank

Purpose Strip trailing blanks from the end of a string

Syntax

```
str = deblank(str)
c = deblank(c)
```

Description The `deblank` function is useful for cleaning up the rows of a character array.

`str = deblank(`*str*`)` removes the trailing blanks from the end of a character string *str*.

`c = deblank(c),` when `c` is a cell array of strings, applies `deblank` to each element of `c`.

Examples

```
A{1,1} = 'MATLAB    ';
A{1,2} = 'SIMULINK    ';
A{2,1} = 'Toolboxes    ';
A{2,2} = 'The MathWorks    ';

A =


    'MATLAB    '       'SIMULINK    '
    'Toolboxes    '    'The MathWorks    '

deblank(A)

ans =


    'MATLAB'        'SIMULINK'
    'Toolboxes'    'The MathWorks'
```

Purpose

Decimal number to base conversion

Syntax

```
str = dec2base(d,base)
str = dec2base(d,base,n)
```

Description

`str = dec2base(d,base)` converts the nonnegative integer `d` to the specified `base`. `d` must be a nonnegative integer smaller than 2^52, and `base` must be an integer between 2 and 36. The returned argument `str` is a string.

`str = dec2base(d,base,n)` produces a representation with at least `n` digits.

Examples

The expression `dec2base(23,2)` converts 23_{10} to base 2, returning the string `'10111'`.

See Also

`base2dec`

dec2bin

Purpose Decimal to binary number conversion

Syntax

```
str = dec2bin(d)
str = dec2bin(d,n)
```

Description

`str = dec2bin(d)` returns the binary representation of `d` as a string. `d` must be a nonnegative integer smaller than 2^{52}.

`str = dec2bin(d,n)` produces a binary representation with at least `n` bits.

Examples

`dec2bin(23)` returns `'10111'`.

See Also

`bin2dec`	Binary to decimal number conversion
`dec2hex`	Decimal to hexadecimal number conversion

Purpose

Decimal to hexadecimal number conversion

Syntax

```
str = dec2hex(d)
str = dec2hex(d,n)
```

Description

`str = dec2hex(d)` converts the decimal integer `d` to its hexadecimal representation stored in a MATLAB string. `d` must be a nonnegative integer smaller than 2^{52}.

`str = dec2hex(d,n)` produces a hexadecimal representation with at least `n` digits.

Examples

`dec2hex(1023)` is the string `'3ff'`.

See Also

`dec2bin`	Decimal to binary number conversion
`format`	Control the output display format
`hex2dec`	IEEE hexadecimal to decimal number conversion
`hex2num`	Hexadecimal to double number conversion

deconv

Purpose Deconvolution and polynomial division

Syntax

```
[q,r] = deconv(v,u)
```

Description `[q,r] = deconv(v,u)` deconvolves vector `u` out of vector `v`, using long division. The quotient is returned in vector `q` and the remainder in vector `r` such that `v = conv(u,q)+r`.

If `u` and `v` are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials, and deconvolution is polynomial division. The result of dividing `v` by `u` is quotient `q` and remainder `r`.

Examples If

```
u = [1   2   3   4]
v = [10   20   30]
```

the convolution is

```
c = conv(u,v)
c =
    10     40     100     160     170     120
```

Use deconvolution to recover `u`:

```
[q,r] = deconv(c,u)
q =
    10    20    30
r =
    0      0      0      0      0      0
```

This gives a quotient equal to `v` and a zero remainder.

Algorithm `deconv` uses the `filter` primitive.

See Also `convmtx`, `conv2`, and `filter` in the Signal Processing Toolbox, and:

`conv`	Convolution and polynomial multiplication
`residue`	Convert between partial fraction expansion and polynomial coefficients

Purpose Discrete Laplacian

Syntax

```
L = del2(U)
L = del2(U,h)
L = del2(U,hx,hy)
L = del2(U,hx,hy,hz,...)
```

Definition If the matrix `U` is regarded as a function $u(x,y)$ evaluated at the point on a square grid, then `4*del2(U)` is a finite difference approximation of Laplace's differential operator applied to u, that is:

$$l = \frac{\nabla^2 u}{4} = \frac{1}{4}\left(\frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2}\right)$$

where:

$$l_{ij} = \frac{1}{4}(u_{i+1,\,j} + u_{i-1,\,j} + u_{i,\,j+1} + u_{i,\,j-1}) - u_{i,\,j}$$

in the interior. On the edges, the same formula is applied to a cubic extrapolation.

For functions of more variables $u(x,y,z,\ldots)$, `del2(U)` is an approximation,

$$l = \frac{\nabla^2 u}{2N} = \frac{1}{2N}\left(\frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} + \frac{d^2 u}{dz^2} + \ldots\right)$$

where N is the number of variables in u.

Description `L = del2(U)` where `U` is a rectangular array is a discrete approximation of

$$l = \frac{\nabla^2 u}{4} = \frac{1}{4}\left(\frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2}\right)$$

The matrix `L` is the same size as `U` with each element equal to the difference between an element of `U` and the average of its four neighbors.

`L = del2(U)` when `U` is an multidimensional array, returns an approximation of

$$\frac{\nabla^2 u}{2N}$$

where N is `ndims(u)`.

`L = del2(U,h)` where `H` is a scalar uses `H` as the spacing between points in each direction (`h=1` by default).

`L = del2(U,hx,hy)` when `U` is a rectangular array, uses the spacing specified by `hx` and `hy`. If `hx` is a scalar, it gives the spacing between points in the x-direction. If `hx` is a vector, it must be of length `size(u,2)` and specifies the x-coordinates of the points. Similarly, if `hy` is a scalar, it gives the spacing between points in the y-direction. If `hy` is a vector, it must be of length `size(u,1)` and specifies the y-coordinates of the points.

`L = del2(U,hx,hy,hz,...)` where `U` is multidimensional uses the spacing given by `hx`, `hy`, `hz`, ...

Examples

The function

$$u(x, y) = x^2 + y^2$$

has

$$\nabla^2 u = 4$$

For this function, `4*del2(U)` is also 4.

```
[x,y] = meshgrid(-4:4,-3:3);
U = x.*x+y.*y
U =
    25    18    13    10     9    10    13    18    25
    20    13     8     5     4     5     8    13    20
    17    10     5     2     1     2     5    10    17
    16     9     4     1     0     1     4     9    16
    17    10     5     2     1     2     5    10    17
    20    13     8     5     4     5     8    13    20
    25    18    13    10     9    10    13    18    25
```

```
V = 4*del2(U)
V =
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
```

See Also

`diff`	Differences and approximate derivatives
`gradient`	Numerical gradient

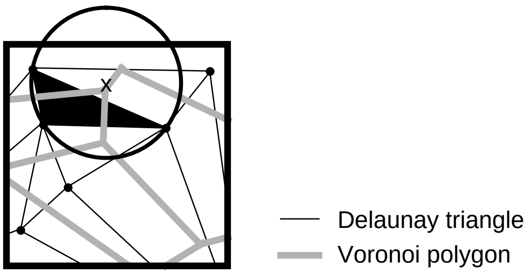
delaunay

Purpose Delaunay triangulation

Syntax

```
TRI = delaunay(x,y)
TRI = delaunay(x,y,'sorted')
```

Definition Given a set of data points, the *Delaunay triangulation* is a set of lines connecting each point to its natural neighbors. The Delaunay triangulation is related to the Voronoi diagram— the circle circumscribed about a Delaunay triangle has its center at the vertex of a Voronoi polygon.



Description `TRI = delaunay(x,y)` returns a set of triangles such that no data points are contained in any triangle's circumscribed circle. Each row of the `m`-by-3 matrix `TRI` defines one such triangle and contains indices into the vectors `x` and `y`.

`TRI = delaunay(x,y,'sorted')` assumes that the points `x` and `y` are sorted first by `y` and then by `x` and that duplicate points have already been eliminated.

Remarks The Delaunay triangulation is used with: `griddata` (to interpolate scattered data), `convhull`, `voronoi` (to compute the `voronoi` diagram), and is useful by itself to create a triangular grid for scattered data points.

The functions `dsearch` and `tsearch` search the triangulation to find nearest neighbor points or enclosing triangles, respectively.

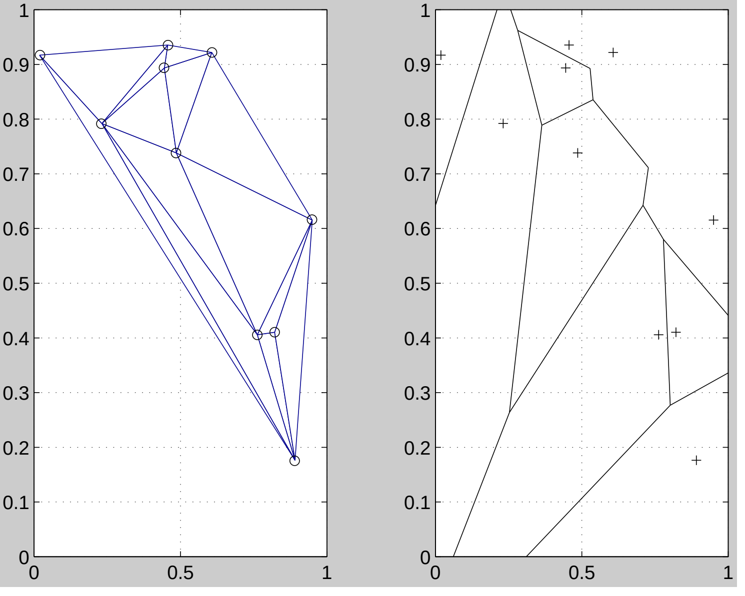
Examples

This code plots the Delaunay triangulation for 10 randomly generated points.

```
rand('state',0);
x = rand(1,10);
y = rand(1,10);
TRI = delaunay(x,y);
subplot(1,2,1),...
trimesh(TRI,x,y,zeros(size(x))); view(2),...
axis([0 1 0 1]); hold on;
plot(x,y,'o');
set(gca,'box','on');
```

Compare the Voronoi diagram of the same points:

```
[vx, vy] = voronoi(x,y,TRI);
subplot(1,2,2),...
plot(x,y,'r+',vx,vy,'b-'),...
axis([0 1 0 1])
```



See Also

`convhull` Convex hull

`dsearch`	Search for nearest point
`griddata`	Data gridding
`tsearch`	Search for enclosing Delaunay triangle
`voronoi`	Voronoi diagram

Purpose Delete files and graphics objects

Syntax

```
delete filename
delete(h)
```

Description `delete` *filename* deletes the named file. Wildcards may be used.

`delete(h)` deletes the graphics object with handle `h`. The function deletes the object without requesting verification even if the object is a window.

Use the functional form of delete, such as `delete('`*filename*`')`, when the file-name is stored in a string.

See Also

`!`	Operating system command
`dir`	Directory listing
`type`	List file

det

Purpose

Matrix determinant

Syntax

```
d = det(X)
```

Description

`d = det(X)` returns the determinant of the square matrix `X`. If `X` contains only integer entries, the result `d` is also an integer.

Remarks

Using `det(X) == 0` as a test for matrix singularity is appropriate only for matrices of modest order with small integer entries. Testing singularity using `abs(det(X)) <= tolerance` is not recommended as it is difficult to choose the correct tolerance. The function `cond(X)` can check for singular and nearly singular matrices.

Algorithm

The determinant is computed from the triangular factors obtained by Gaussian elimination

```
[L,U] = lu(A)
s =  det(L)        % This is always +1 or -1
det(A) = s*prod(diag(U))
```

Examples

The statement `A = [1   2   3;  4   5   6;  7   8   9]`

produces

```
A =
     1       2       3
     4       5       6
     7       8       9
```

This happens to be a singular matrix, so `d = det(A)` produces `d = 0`. Changing `A(3,3)` with `A(3,3) = 0` turns `A` into a nonsingular matrix. Now `d = det(A)` produces `d = 27`.

See Also

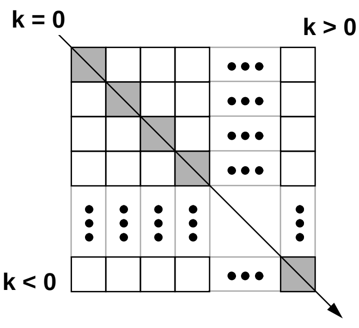
`\`	Matrix left division (backslash)
`/`	Matrix right division (slash)
`cond`	Condition number with respect to inversion
`condest`	1-norm matrix condition number estimate
`inv`	Matrix inverse
`lu`	LU matrix factorization
`rref`	Reduced row echelon form

Purpose Diagonal matrices and diagonals of a matrix

Syntax

```
X = diag(v,k)
X = diag(v)
v = diag(X,k)
v = diag(X)
```

Description `X = diag(v,k)` when `v` is a vector of `n` components, returns a square matrix `X` of order `n+abs(k)`, with the elements of `v` on the `k`th diagonal. `k = 0` represents the main diagonal, `k > 0` above the main diagonal, and `k < 0` below the main diagonal.



`X = diag(v)` puts `v` on the main diagonal, same as above with `k = 0`.

`v = diag(X,k)` for matrix `X`, returns a column vector `v` formed from the elements of the `k`th diagonal of `X`.

`v = diag(X)` returns the main diagonal of `X`, same as above with `k = 0`.

Examples `diag(diag(X))` is a diagonal matrix.

`sum(diag(X))` is the trace of `X`.

The statement

```
diag(-m:m)+diag(ones(2*m,1),1)+diag(ones(2*m,1),-1)
```

produces a tridiagonal matrix of order `2*m+1`.

See Also `spdiags`, `tril`, `triu`

diary

Purpose

Save session in a disk file

Syntax

```
diary
diary filename
diary off
diary on
```

Description

The `diary` command creates a log of keyboard input and system responses. The output of `diary` is an ASCII file, suitable for printing or for inclusion in reports and other documents.

`diary`, by itself, toggles `diary` mode on and off.

`diary` *filename* writes a copy of all subsequent keyboard input and most of the resulting output (but not graphs) to the named file. If the file already exists, output is appended to the end of the file.

`diary off` suspends the diary.

`diary on` resumes diary mode using the current filename, or the default filename `diary` if none has yet been specified.

Remarks

The function form of the syntax, `diary('filename')`, is also permitted.

Limitations

You cannot put a diary into the files named `off` and on.

Purpose Differences and approximate derivatives

Syntax

```
Y = diff(X)
Y = diff(X,n)
Y = diff(X,n,dim)
```

Description `Y = diff(X)` calculates differences between adjacent elements of `X`.

If `X` is a vector, then `diff(X)` returns a vector, one element shorter than `X`, of differences between adjacent elements:

```
[X(2)-X(1) X(3)-X(2) ... X(n)-X(n-1)]
```

If `X` is a matrix, then `diff(X)` returns a matrix of column differences:

```
[X(2:m,:)-X(1:m-1,:)]
```

In general, `diff(X)` returns the differences calculated along the first non-singleton (`size(X,dim) > 1`) dimension of `X`.

`Y = diff(X,`*n*`)` applies `diff` recursively *n* times, resulting in the nth difference. Thus, `diff(X,2)` is the same as `diff(diff(X))`.

`Y = diff(X,`*n*`,dim)` is the nth difference function calculated along the dimension specified by scalar `dim`. If order *n* equals or exceeds the length of dimension `dim`, `diff` returns an empty array.

Remarks Since each iteration of `diff` reduces the length of `X` along dimension `dim`, it is possible to specify an order *n* sufficiently high to reduce `dim` to a singleton (`size(X,dim) = 1`) dimension. When this happens, `diff` continues calculating along the next nonsingleton dimension.

Examples

The quantity `diff(y)./diff(x)` is an approximate derivative.

```
x = [1 2 3 4 5];
y = diff(x)
y =
     1     1     1     1

z = diff(x,2)
z =
     0     0     0
```

Given,

```
A = rand(1,3,2,4);
```

`diff(A)` is the first-order difference along dimension `2`.

`diff(A,3,4)` is the third-order difference along dimension `4`.

See Also

`gradient`	Approximate gradient.
`int`	Integrate (see Symbolic Toolbox).
`prod`	Product of array elements
`sum`	Sum of array elements

Purpose Directory listing

Syntax

```
dir
dir dirname
names = dir
names = dir('dirname')
```

Description

`dir`, by itself, lists the files in the current directory.

`dir` *`dirname`* lists the files in the specified directory. Use pathnames, wildcards, and any options available in your operating system.

`names = dir('`*`dirname`*`')` or `names = dir` returns the results in an `m`-by-`1` structure with the fields:

`name`	Filename
`date`	Modification date
`bytes`	Number of bytes allocated to the file
`isdir`	`1` if name is a directory; `0` if not

Examples

```
cd /Matlab/Toolbox/Local; dir

Contents.m matlabrc.m siteid.m userpath.m

names = dir

names =

4x1 struct array with fields:
    name
    date
    bytes
    isdir
```

See Also `!`, `cd`, `delete`, `type`, `what`

disp

Purpose

Display text or array

Syntax

```
disp(X)
```

Description

`disp(X)` displays an array, without printing the array name. If `X` contains a text string, the string is displayed.

Another way to display an array on the screen is to type its name, but this prints a leading "`X =`," which is not always desirable.

Examples

One use of `disp` in an M-file is to display a matrix with column labels:

```
disp('         Corn          Oats          Hay')
disp(rand(5,3))
```

which results in

```
         Corn          Oats          Hay
        0.2113        0.8474        0.2749
        0.0820        0.4524        0.8807
        0.7599        0.8075        0.6538
        0.0087        0.4832        0.4899
        0.8096        0.6135        0.7741
```

See Also

`format`	Control the output display format
`int2str`	Integer to string conversion
`num2str`	Number to string conversion
`rats`	Rational fraction approximation
`sprintf`	Write formatted data to a string

Purpose

Read an ASCII delimited file into a matrix

Syntax

```
M = dlmread(filename,delimiter)
M = dlmread(filename,delimiter,r,c)
M = dlmread(filename,delimiter,r,c,range)
```

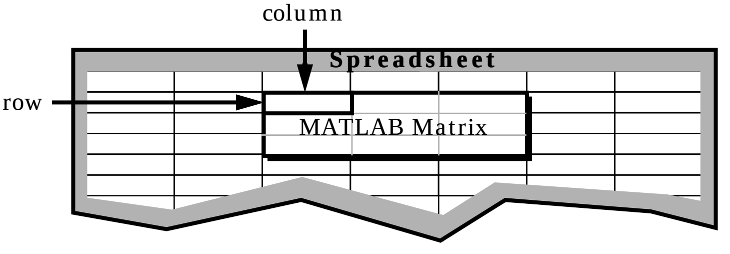
Description

`M = dlmread(`*`filename`*`,`*`delimiter`*`)` reads data from the ASCII delimited format *filename*, using the delimiter *delimiter*. Use `'\t'` to specify a tab.

`M = dlmread(`*`filename`*`,`*`delimiter`*`,r,c)` reads data from the ASCII delimited format *filename*, using the delimiter *delimiter*, starting at file offset `r` and `c`. `r` and `c` are zero based so that `r=0`, `c=0` specifies the first value in the file.

`M = dlmread(`*`filename`*`,`*`delimiter`*`,r,c,range)` imports an indexed or named range of ASCII-delimited data. To use the cell range, specify `range` by:

```
range = [UpperLeftRow UpperLeftColumn LowerRightRow
LowerRightColumn]
```



Arguments

delimiter	The character separating individual matrix elements in the ASCII- format spreadsheet file. A comma (,) is the default delimiter.
`r, c`	The spreadsheet cell from which the upper-left-most matrix element is taken.
`range`	A vector specifying a range of spreadsheet cells.

See Also

`dlmwrite`	Write a matrix to an ASCII delimited file
`wk1read`	Read a Lotus123 WK1 spreadsheet file into a matrix
`wk1write`	Write a matrix to a Lotus123 WK1 spreadsheet file

dlmwrite

Purpose

Write a matrix to an ASCII delimited file

Syntax

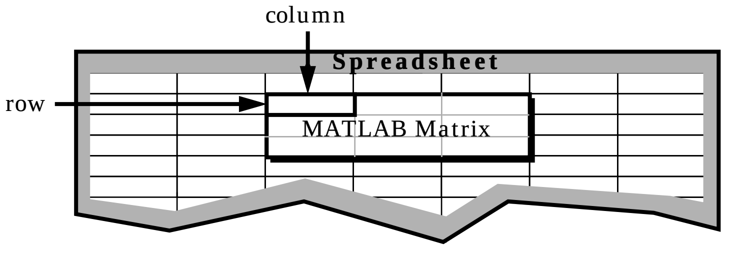
```
dlmwrite(filename,A,delimiter)
dlmwrite(filename,A,delimiter,r,c)
```

Description

The `dlmwrite` command converts a MATLAB matrix into an ASCII-format file readable by spreadsheet programs.

`dlmwrite(`*`filename`*`,A,`*`delimiter`*`)` writes matrix `A` into the upper left-most cell of the ASCII-format spreadsheet file *filename*, and uses the delimiter to separate matrix elements. Specify `'\t'` to produce tab-delimited files. Any elements whose value is 0 will be omitted. For example, the array `[1 0 2]` will appear in a file as `'1,,2'` when the delimiter is a comma.

`dlmwrite(`*`filename`*`,A,`*`delimiter`*`,r,c)` writes `A` into *filename*, starting at spreadsheet cell `r` and `c`, with *delimiter* used to separate matrix elements.



Arguments

delimiter	The character separating individual matrix elements in the ASCII- format spreadsheet file. A comma (,) is the default delimiter.
`r,c`	The spreadsheet cell into which the upper-left-most matrix element is written.

See Also

`dlmread`	Read an ASCII delimited file into a matrix
`wk1read`	Read a Lotus123 WK1 spreadsheet file into a matrix
`wk1write`	Write a matrix to a Lotus123 WK1 spreadsheet file

Purpose Dulmage-Mendelsohn decomposition

Syntax

```
p = dmperm(A)
[p,q,r] = dmperm(A)
[p,q,r,s] = dmperm(A)
```

Description If `A` is a reducible matrix, the linear system $Ax = b$ can be solved by permuting `A` to a block upper triangular form, with irreducible diagonal blocks, and then performing block backsubstitution. Only the diagonal blocks of the permuted matrix need to be factored, saving fill and arithmetic in the blocks above the diagonal.

`p = dmperm(A)` returns a row permutation `p` so that if `A` has full column rank, `A(p,:)` is square with nonzero diagonal. This is also called a *maximum matching*.

`[p,q,r] = dmperm(A)` where `A` is a square matrix, finds a row permutation `p` and a column permutation `q` so that `A(p,q)` is in block upper triangular form. The third output argument `r` is an integer vector describing the boundaries of the blocks: The kth block of `A(p,q)` has indices `r(k):r(k+1)–1`.

`[p,q,r,s] = dmperm(A)`, where `A` is not square, finds permutations `p` and `q` and index vectors `r` and `s` so that `A(p,q)` is block upper triangular. The blocks have indices `(r(i):r(i+1)–1, s(i):s(i+1)–1)`.

In graph theoretic terms, the diagonal blocks correspond to strong Hall components of the adjacency graph of `A`.

doc

Purpose Load hypertext documentation

Syntax

```
doc
doc command
```

Description `doc`, by itself, loads hypertext-based reference documentation. You'll be presented with an index of MATLAB's main categories of functions.

`doc` *command* loads documentation about a specific command or function.

See Also

`help`	Online help for MATLAB functions and M-files
`type`	List file

Purpose Convert to double precision

Syntax `double(X)`

Description `double(x)` returns the double precision value for `X`. If `X` is already a double precision array, `double` has no effect.

Remarks `double` is called for the expressions in `for`, `if`, and `while` loops if the expression isn't already double precision. `double` should be overloaded for any object when it makes sense to convert it to a double precision value.

dsearch

Purpose

Search for nearest point

Syntax

```
K = dsearch(x,y,TRI,xi,yi)
K = dsearch(x,y,TRI,xi,yi,S)
```

Description

`K = dsearch(x,y,TRI,xi,yi)` returns the index of the nearest (x,y) point to the point (xi,yi). `dsearch` requires a triangulation `TRI` of the points `x,y` obtained from `delaunay`.

`K = dsearch(x,y,TRI,xi,yi,S)` uses the sparse matrix `S` instead of computing it each time:

```
S = sparse(TRI(:,[1 1 2 2 3 3]),TRI(:,[2 3 1 3 1 2]),1,nxy,nxy)
```

where `nxy = prod(size(x))`.

See Also

`delaunay`	Delaunay triangulation
`tsearch`	Search for enclosing Delaunay triangle
`voronoi`	Voronoi diagram

echo

Purpose

Echo M-files during execution

Syntax

```
echo on
echo off
echo
echo fcnname on
echo fcnname off
echo fcnname
echo on all
echo off all
```

Description

The echo command controls the echoing of M-files during execution. Normally, the commands in M-files do not display on the screen during execution. Command echoing is useful for debugging or for demonstrations, allowing the commands to be viewed as they execute.

The echo command behaves in a slightly different manner for script files and function files. For script files, the use of echo is simple; echoing can be either on or off, in which case any script used is affected:

`echo on`	Turns on the echoing of commands in all script files.
`echo off`	Turns off the echoing of commands in all script files.
`echo`	Toggles the echo state.

With function files, the use of echo is more complicated. If echo is enabled on a function file, the file is interpreted, rather than compiled. Each input line is then displayed as it is executed. Since this results in inefficient execution, use echo only for debugging.

`echo` *fcnname* `on`	Turns on echoing of the named function file.
`echo` *fcnname* `off`	Turns off echoing of the named function file.
`echo` *fcnname*	Toggles the echo state of the named function file.
`echo on all`	Set echoing on for all function files.
`echo off all`	Set echoing off for all function files.

See Also

`function`

edit

Purpose

Edit an M-file

Syntax

```
edit
edit fun
edit file.ext
edit class/fun
edit private/fun
edit class/private/fun
```

Description

`edit` opens a new editor window.

`edit` *fun* opens the M-file `fun.m` in a text editor.

`edit` *file.ext* opens the specified text file.

`edit` *class/fun*, `edit private/`*fun*, or `edit` *class*`/private/`*fun* can be used to edit a method, private function, or private method (for the class named *class*.)

Purpose Eigenvalues and eigenvectors

Syntax

```
d = eig(A)
[V,D] = eig(A)
[V,D] = eig(A,'nobalance')
d = eig(A,B)
[V,D] = eig(A,B)
```

Description `d = eig(A)` returns a vector of the eigenvalues of matrix `A`.

`[V,D] = eig(A)` produces matrices of eigenvalues (`D`) and eigenvectors (`V`) of matrix `A`, so that `A*V = V*D`. Matrix `D` is the *canonical form* of `A`—a diagonal matrix with `A`'s eigenvalues on the main diagonal. Matrix `V` is the *modal matrix*—its columns are the eigenvectors of `A`.

The eigenvectors are scaled so that the norm of each is 1.0. Use `[W,D] = eig(A'); W = W'` to compute the *left eigenvectors*, which satisfy `W*A = D*W`.

`[V,D] = eig(A,'nobalance')` finds eigenvalues and eigenvectors without a preliminary balancing step. Ordinarily, balancing improves the conditioning of the input matrix, enabling more accurate computation of the eigenvectors and eigenvalues. However, if a matrix contains small elements that are really due to roundoff error, balancing may scale them up to make them as significant as the other elements of the original matrix, leading to incorrect eigenvectors. Use the `nobalance` option in this event. See the `balance` function for more details.

`d = eig(A,B)` returns a vector containing the generalized eigenvalues, if `A` and `B` are square matrices.

`[V,D] = eig(A,B)` produces a diagonal matrix `D` of generalized eigenvalues and a full matrix `V` whose columns are the corresponding eigenvectors so that `A*V = B*V*D`. The eigenvectors are scaled so that the norm of each is 1.0.

Remarks The eigenvalue problem is to determine the nontrivial solutions of the equation:

$$Ax = \lambda x$$

where A is an n-by-n matrix, x is a length n column vector, and λ is a scalar. The n values of λ that satisfy the equation are the *eigenvalues*, and the corresponding values of x are the *right eigenvectors*. In MATLAB, the function eig solves for the eigenvalues λ, and optionally the eigenvectors x.

The *generalized* eigenvalue problem is to determine the nontrivial solutions of the equation

$$Ax = \lambda Bx$$

where both A and B are n-by-n matrices and λ is a scalar. The values of λ that satisfy the equation are the *generalized eigenvalues* and the corresponding values of x are the *generalized right eigenvectors*.

If B is nonsingular, the problem could be solved by reducing it to a standard eigenvalue problem

$$B^{-1}Ax = \lambda x$$

Because B can be singular, an alternative algorithm, called the QZ method, is necessary.

When a matrix has no repeated eigenvalues, the eigenvectors are always independent and the eigenvector matrix V *diagonalizes* the original matrix A if applied as a similarity transformation. However, if a matrix has repeated eigenvalues, it is not similar to a diagonal matrix unless it has a full (independent) set of eigenvectors. If the eigenvectors are not independent then the original matrix is said to be *defective*. Even if a matrix is defective, the solution from eig satisfies A*X = X*D.

Examples

The matrix

```
B = [3 –2 –.9 2*eps;–2 4 –1 –eps;–eps/4 eps/2 –1 0;–.5  –.5 .1 1];
```

has elements on the order of roundoff error. It is an example for which the nobalance option is necessary to compute the eigenvectors correctly. Try the statements

```
[VB,DB] = eig(B)
B*VB – VB*DB
[VN,DN] = eig(B,'nobalance')
B*VN – VN*DN
```

Algorithm

For real matrices, `eig(X)` uses the EISPACK routines `BALANC`, `BALBAK`, `ORTHES`, `ORTRAN`, and `HQR2`. `BALANC` and `BALBAK` balance the input matrix. `ORTHES` converts a real general matrix to Hessenberg form using orthogonal similarity transformations. `ORTRAN` accumulates the transformations used by `ORTHES`. `HQR2` finds the eigenvalues and eigenvectors of a real upper Hessenberg matrix by the QR method. The EISPACK subroutine `HQR2` is modified to make computation of eigenvectors optional.

When `eig` is used with two input arguments, the EISPACK routines `QZHES`, `QZIT`, `QZVAL`, and `QZVEC` solve for the generalized eigenvalues via the QZ algorithm. Modifications handle the complex case.

When `eig` is used with one complex argument, the solution is computed using the QZ algorithm as `eig(X,eye(X))`. Modifications to the QZ routines handle the special case $B = I$.

For detailed descriptions of these algorithms, see the *EISPACK Guide*.

Diagnostics

If the limit of 30n iterations is exhausted while seeking an eigenvalue:

```
Solution will not converge.
```

See Also

`balance`	Improve accuracy of computed eigenvalues
`condeig`	Condition number with respect to eigenvalues
`hess`	Hessenberg form of a matrix
`qz`	QZ factorization for generalized eigenvalues
`schur`	Schur decomposition

References

[1] Smith, B. T., J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide*, Lecture Notes in Computer Science, Vol. 6, second edition, Springer-Verlag, 1976.

[2] Garbow, B. S., J. M. Boyle, J. J. Dongarra, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide Extension*, Lecture Notes in Computer Science, Vol. 51, Springer-Verlag, 1977.

[3] Moler, C. B. and G.W. Stewart, "An Algorithm for Generalized Matrix Eigenvalue Problems", *SIAM J. Numer. Anal.*, Vol. 10, No. 2, April 1973.

Purpose Find a few eigenvalues and eigenvectors

Syntax

```
[V,D] = eigs(A)
[V,D] = eigs('Afun',n)
[V,D] = eigs(A,B,k,sigma,options)
[V,D] = eigs('Afun',n,B,k,sigma,options)
```

Description `eigs` solves the eigenvalue problem `A*v = lambda*v` or the generalized eigenvalue problem `A*v = lambda*B*v`. Only a few selected eigenvalues, or eigenvalues and eigenvectors, are computed.

`[V,D] = eigs(A)` or `[V,D] = eigs('Afun',n)` solves the eigenvalue problem where the first input argument is either a square matrix (which can be full or sparse, symmetric or nonsymmetric, real or complex), or a string containing the name of an M-file which applies a linear operator to the columns of a given matrix. In the latter case, the second input argument must be `n`, the order of the problem. For example, `eigs('fft', ...)` is much faster than `eigs(F, ...)` where `F` is the explicit FFT matrix.

With one output argument, `D` is a vector containing `k` eigenvalues.With two output arguments, `D` is a `k`-by-`k` diagonal matrix and `V` is a matrix with `k` columns so that `A*V = V*D` or `A*V = B*V*D`.

The remaining input arguments are optional and can be given in practically any order:

Argument	Value
`B`	A matrix the same size as `A`. If `B` is not specified, `B = eye(size(A))` is used.
`k`	An integer, the number of eigenvalues desired. If `k` is not specified, `k = 6` eigenvalues are computed.
`sigma`	A scalar shift or a two letter string. If `sigma` is not specified, the `k`-th eigenvalues largest in magnitude are computed. If `sigma` is `0`, the `k`-th eigenvalues smallest in magnitude are computed. If `sigma` is a real or complex scalar, the *shift*, the `k`-th eigenvalues nearest `sigma`, are computed. If `sigma` is one of the following strings, it specifies the desired eigenvalues: `'lm'` Largest Magnitude (the default) `'sm'` Smallest Magnitude (same as sigma = 0) `'lr'` Largest Real part `'sr'` Smallest Real part `'be'` Both Ends. Computes `k/2` eigenvalues from each end of the spectrum (one more from the high end if `k` is odd.)

The `options` structure specifies certain parameters in the algorithm.

Parameter	Description	Value
options.tol	Convergence tolerance norm(A*V-V*D) <= tol	1e-10 (symmetric) 1e-6 (nonsymmetric)
options.p	Dimension of the Arnoldi basis	2*k
options.maxit	Maximum number of iterations	300
options.disp	Number of eigenvalues displayed at each iteration. Set to 0 for no intermediate output.	20
options.issym	Positive if Afun is symmetric	0
options.cheb	Positive if A is a string, sigma is 'lr','sr', or a shift, and polynomial acceleration should be applied.	0
options.v0	Starting vector for the Arnoldi factorization	rand(n,1)-.5

See Also

eig Eigenvalues and eigenvectors
svds Singular value decomposition

Purpose Jacobi elliptic functions

Syntax

```
[SN,CN,DN] = ellipj(U,M)
[SN,CN,DN] = ellipj(U,M,tol)
```

Definition The Jacobi elliptic functions are defined in terms of the integral:

$$u = \int_0^{\phi} \frac{d\theta}{(1 - m\sin^2\theta)^{\frac{1}{2}}}$$

Then

$$sn(u) = \sin\phi,\ \ cn(u) = \cos\phi,\ \ dn(u) = (1 - \sin^2\phi)^{\frac{1}{2}},\ \ am(u) = \phi$$

Some definitions of the elliptic functions use the modulus k instead of the parameter m. They are related by:

$$k^2 = m = \sin^2\alpha$$

The Jacobi elliptic functions obey many mathematical identities; for a good sample, see [1].

Description `[SN,CN,DN] = ellipj(U,M)` returns the Jacobi elliptic functions `SN`, `CN`, and `DN`, evaluated for corresponding elements of argument `U` and parameter `M`. Inputs `U` and `M` must be the same size (or either can be scalar).

`[SN,CN,DN] = ellipj(U,M,tol)` computes the Jacobi elliptic functions to accuracy *tol*. The default is `eps`; increase this for a less accurate but more quickly computed answer.

Algorithm `ellipj` computes the Jacobi elliptic functions using the method of the arithmetic-geometric mean [1]. It starts with the triplet of numbers:

$$a_0 = 1, b_0 = (1 - m)^{\frac{1}{2}}, c_0 = (m)^{\frac{1}{2}}$$

`ellipj` computes successive iterates with:

$$a_i = \frac{1}{2}(a_{i-1} + b_{i-1})$$
$$b_i = (a_{i-1}b_{i-1})^{\frac{1}{2}}$$
$$c_i = \frac{1}{2}(a_{i-1} - b_{i-1})$$

Next, it calculates the amplitudes in radians using:

$$\sin(2\phi_{n-1} - \phi_n) = \frac{c_n}{a_n}\sin(\phi_n)$$

being careful to unwrap the phases correctly. The Jacobian elliptic functions are then simply:

$$sn(u) = \sin\phi_0$$
$$cn(u) = \cos\phi_0$$
$$dn(u) = (1 - m \cdot sn(u)^2)^{\frac{1}{2}}$$

Limitations

The `ellipj` function is limited to the input domain $0 \le m \le 1$. Map other values of `M` into this range using the transformations described in [1], equations 16.10 and 16.11. `U` is limited to real values.

See Also

`ellipke`	Complete elliptic integrals of the first and second kind

References

[1] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, 17.6.

Purpose

Complete elliptic integrals of the first and second kind

Syntax

```
K = ellipke(M)
[K,E] = ellipke(M)
[K,E] = ellipke(M,tol)
```

Definition

The *complete* elliptic integral of the first kind [1] is:

$$K(m) = F(\pi/2|m),$$

where F, the elliptic integral of the first kind, is:

$$K(m) = \int_0^1 [(1-t^2)(1-mt^2)]^{\frac{-1}{2}}\,dt = \int_0^{\frac{\pi}{2}} (1-m\sin^2\theta)^{\frac{-1}{2}}\,d\theta$$

The complete elliptic integral of the second kind,

$$E(m) = E(K(m)) = E\langle\pi/2|m\rangle,$$

is:

$$E(m) = \int_0^1 (1-t^2)^{\frac{-1}{2}}(1-mt^2)^{\frac{1}{2}}\,dt = \int_0^{\frac{\pi}{2}} (1-m\sin^2\theta)^{\frac{1}{2}}d\theta$$

Some definitions of `K` and `E` use the modulus k instead of the parameter m. They are related by:

$$k^2 = m = \sin^2\alpha$$

Description

`K = ellipke(M)` returns the complete elliptic integral of the first kind for the elements of `M`.

`[K,E] = ellipke(M)` returns the complete elliptic integral of the first and second kinds.

`[K,E] = ellipke(M,tol)` computes the Jacobian elliptic functions to accuracy *tol*. The default is `eps`; increase this for a less accurate but more quickly computed answer.

Algorithm

ellipke computes the complete elliptic integral using the method of the arithmetic-geometric mean described in [1], section 17.6. It starts with the triplet of numbers:

$$a_0 = 1,\ b_0 = (1-m)^{\frac{1}{2}},\ c_0 = (m)^{\frac{1}{2}}$$

ellipke computes successive iterations of a_i, b_i, and c_i with:

$$\begin{aligned} a_i &= \frac{1}{2}(a_{i-1} + b_{i-1}) \\ b_i &= (a_{i-1}b_{i-1})^{\frac{1}{2}} \\ c_i &= \frac{1}{2}(a_{i-1} - b_{i-1}) \end{aligned}$$

stopping at iteration n when $cn \approx 0$, within the tolerance specified by eps. The complete elliptic integral of the first kind is then:

$$K(m) = \frac{\pi}{2a_n}$$

Limitations

ellipke is limited to the input domain $0 \le m \le 1$.

See Also

ellipj	Jacobi elliptic functions

References

[1] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, 17.6.

Purpose

Conditionally execute statements

Syntax

```
if expression
    statements
else
    statements
end
```

Description

The `else` command is used to delineate an alternate block of statements.

```
if expression
    statements
else
    statements
end
```

The second set of *statements* is executed if the *expression* has any zero elements. The expression is usually the result of

expression rop expression

where *rop* is ==, <, >, <=, >=, or ~=.

See Also

`break`	Break out of flow control structures
`elseif`	Conditionally execute statements
`end`	Terminate `for`, `while`, and `if` statements and indicate the last index
`for`	Repeat statements a specific number of times
`if`	Conditionally execute statements
`return`	Return to the invoking function
`switch`	Switch among several cases based on expression
`while`	Repeat statements an indefinite number of times

elseif

Purpose Conditionally execute statements

Syntax

```
if expression
     statements
elseif expression
     statements
end
```

Description The `elseif` command conditionally executes statements.

```
if expression
     statements
elseif expression
     statements
end
```

The second block of *statements* executes if the first *expression* has any zero elements and the second *expression* has all nonzero elements. The expression is usually the result of

expression rop expression

where *rop* is ==, <, >, <=, >=, or ~=.

`else if`, with a space between the `else` and the `if`, differs from `elseif`, with no space. The former introduces a new, nested, `if`, which must have a matching `end`. The latter is used in a linear sequence of conditional statements with only one terminating `end`.

The two segments

```
if A                       if A
   x = a                       x = a
else                       elseif B
   if B                        x = b
      x = b                elseif C
   else                        x = c
       if C                else
          x = c                x = d
       else                end
           x = d
       end
   end
end
```

produce identical results. Exactly one of the four assignments to `x` is executed, depending upon the values of the three logical expressions, `A`, `B`, and `C`.

See Also

`break`	Break out of flow control structures
`else`	Conditionally execute statements
`end`	Terminate `for`, `while`, and `if` statements and indicate the last index
`for`	Repeat statements a specific number of times
`if`	Conditionally execute statements
`return`	Return to the invoking function
`switch`	Switch among several cases based on expression
`while`	Repeat statements an indefinite number of times

end

Purpose

Terminate `for`, `while`, `switch`, and `if` statements or indicate last index

Syntax

```
while expression% (or if or for)
    statements
end
```

```
B = A(index:end,index)
```

Description

`end` is used to terminate `for`, `while`, `switch`, and `if` statements. Without an `end` statement, `for`, `while`, `switch`, and `if` wait for further input. Each `end` is paired with the closest previous unpaired `for`, `while`, `switch`, or `if` and serves to delimit its scope.

The `end` command also serves as the last index in an indexing expression. In that context, `end = (size(x,k))` when used as part of the kth index.

Examples

This example shows `end` used with `for` and `if`. Indentation provides easier readability.

```
for i = 1:n
    if a(i) == 0
        a(i) = a(i) + 2;
     end
end
```

Here, `end` is used in an indexing expression:

```
A = rand(5,4)
B = A(end,2:end)
```

In this example, `B` is a 1-by-3 vector equal to `[A(5,2) A(5,3) A(5,4)]`.

See Also

`break`	Break out of flow control structures
`for`	Repeat statements a specific number of times
`if`	Conditionally execute statements
`return`	Return to the invoking function
`switch`	Switch among several cases based on expression
`while`	Repeat statements an indefinite number of times

Purpose

End of month

Syntax

```
E = eomday(Y,M)
```

Description

`E = eomday(Y,M)` returns the last day of the year and month given by corresponding elements of arrays `Y` and `M`.

Examples

Because 1996 is a leap year, the statement `eomday(1996,2)` returns `29`.

To show all the leap years in this century, try:

```
y = 1900:1999;
E = eomday(y,2*ones(length(y),1)');
y(find(E==29))'

ans =
  Columns 1 through 6
       1904      1908      1912      1916      1920      1924

  Columns 7 through 12
       1928      1932      1936      1940      1944      1948

  Columns 13 through 18
       1952      1956      1960      1964      1968      1972

  Columns 19 through 24
       1976      1980      1984      1988      1992      1996
```

See Also

`datenum`	Serial date number
`datevec`	Date components
`weekday`	Day of the week

eps

Purpose Floating-point relative accuracy

Syntax

```
eps
```

Description `eps` returns the distance from 1.0 to the next largest floating-point number.

The value `eps` is a default tolerance for `pinv` and `rank`, as well as several other MATLAB functions. On machines with IEEE floating-point arithmetic, `eps = 2^(-52)`, which is roughly `2.22e-16`.

See Also

`realmax`	Largest positive floating-point number
`realmin`	Smallest positive floating-point number

Purpose

Error functions

Syntax

`Y = erf(X)`	Error function
`Y = erfc(X)`	Complementary error function
`Y = erfcx(X)`	Scaled complementary error function
`X = erfinv(Y)`	Inverse of the error function

Definition

The error function `erf(X)` is defined as the integral of the Gaussian distribution function from 0 to `x`:

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

The complementary error function `erfc(X)` is defined as:

$$erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt = 1 - erf(x)$$

The scaled complementary error function `erfcx(X)` is defined as:

$$erfcx(x) = e^{x^2} erfc(x)$$

For large `X`, `erfcx(X)` is approximately $\left(\frac{1}{\sqrt{\pi}}\right)\frac{1}{x}$.

Description

`Y = erf(X)` returns the value of the error function for each element of real array `X`.

`Y = erfc(X)` computes the value of the complementary error function.

`Y = erfcx(X)` computes the value of the scaled complementary error function.

`X = erfinv(Y)` returns the value of the inverse error function for each element of `Y`. The elements of `Y` must fall within the domain $-1 < Y < 1$.

Examples

`erfinv(1)` is `Inf`

`erfinv(-1)` is `-Inf`.

For `abs(Y) > 1`, `erfinv(Y)` is `NaN`.

Remarks

The relationship between the error function and the standard normal probability distribution is:

```
x = –5:0.1:5;
standard_normal_cdf = (1 + (erf(x*sqrt(2)))))./2;
```

Algorithms

For the error functions, the MATLAB code is a translation of a Fortran program by W. J. Cody, Argonne National Laboratory, NETLIB/SPECFUN, March 19, 1990. The main computation evaluates near-minimax rational approximations from [1].

For the inverse of the error function, rational approximations accurate to approximately six significant digits are used to generate an initial approximation, which is then improved to full accuracy by two steps of Newton's method. The M-file is easily modified to eliminate the Newton improvement. The resulting code is about three times faster in execution, but is considerably less accurate.

References

[1] Cody, W. J., "Rational Chebyshev Approximations for the Error Function," *Math. Comp.*, pgs. 631-638, 1969

Purpose Display error messages

Syntax

```
error('error_message')
```

Description `error('`*`error_message`*`')` displays an error message and returns control to the keyboard. The error message contains the input string *error_message*.

The `error` command has no effect if *error_message* is a null string.

Examples The `error` command provides an error return from M-files.

```
function foo(x,y)
if nargin ~= 2
     error('Wrong number of input arguments')
end
```

The returned error message looks like:

```
» foo(pi)
??? Error using ==> foo
Wrong number of input arguments
```

See Also

`dbstop`	Set breakpoints in an M-file function
`disp`	Display text or array
`lasterr`	Last error message
`warning`	Display warning message

errortrap

Purpose Continue execution after errors during testing

Syntax

```
errortrap on
errortrap off
```

Description `errortrap on` continues execution after errors when they occur. Execution continues with the next statement in a top level script.

`errortrap off` (the default) stops execution when an error occurs.

Purpose Elapsed time

Syntax

```
e = etime(t2,t1)
```

Description

`e = etime(t2,t1)` returns the time in seconds between vectors `t1` and `t2`. The two vectors must be six elements long, in the format returned by `clock`:

```
T = [Year Month Day Hour Minute Second]
```

Examples

Calculate how long a 2048-point real FFT takes.

```
x = rand(2048,1);
t = clock; fft(x); etime(clock,t)
ans =
      0.4167
```

Limitations

As currently implemented, the `etime` function fails across month and year boundaries. Since `etime` is an M-file, you can modify the code to work across these boundaries if needed.

See Also

`clock`	Current time as a date vector
`cputime`	Elapsed CPU time
`tic`, `toc`	Stopwatch timer

eval

Purpose Interpret strings containing MATLAB expressions

Syntax

```
a = eval('expression')
[a1,a2,a3...] = eval('expression')
eval(string,catchstring)
```

Description

`a = eval('`*expression*`')` returns the value of *expression*, a MATLAB expression, enclosed in single quotation marks. Create `'`*expression*`'` by concatenating substrings and variables inside square brackets.

`[a1,a2,a3...] = eval('`*expression*`')` evaluates and returns the results in separate variables. Use of this syntax is recommended over:

```
eval('[a1,a2,a3...] = expression')
```

which hides information from the MATLAB parser and can produce unexpected behavior.

`eval(`*string*`,`*catchstring*`)` provides the ability to catch errors. It executes *string* and returns if the operation was successful. If the operation generates an error, *catchstring* is evaluated before returning. Use `lasterr` to obtain the error string produced by *string*.

Examples

```
A = '1+4'; eval(A)
ans =
     5
```

```
P = 'pwd'; eval(P)
ans =
/home/myname
```

The loop

```
for n = 1:12
        eval(['M',int2str(n),' = magic(n)'])
end
```

generates a sequence of 12 matrices named `M1` through `M12`.

The next example runs a selected M-file script. Note that the strings making up the rows of matrix D must all have the same length.

```
D = ['odedemo '
     'quaddemo'
     'zerodemo'
     'fitdemo '];
n = input('Select a demo number: ');
eval(D(n,:))
```

See Also

feval	Function evaluation
lasterr	Last error message.

evalin

Purpose Evaluate expression in workspace.

Syntax

```
evalin(ws,'expression')
evalin(ws,'try','catch')
```

Description `evalin(ws,'`*`expression`*`')` evaluates *expression* in the context of the workspace `ws`. `ws` can be either `'caller'` or `'base'`.

`evalin(ws,'`*`try`*`','`*`catch`*`')` tries to evaluate the *try* expression and if that fails it evaluates the *catch* expression in the specified workspace.

`evalin` is useful for getting values from another workspace while `assigin` is useful for depositing values into another workspace.

See Also

`assignin`	Assign variable in workspace.
`evalin`	Interpret strings containing MATLAB expressions

Purpose Check if a variable or file exists

Syntax

```
a = exist('item')
ident = exist('item',kind)
```

Description

`a = exist('`*`item`*`')` returns the status of the variable or file *item*:

`0`	If *item* does not exist.
`1`	If the variable *item* exists in the workspace.
`2`	If *item* is an M-file or a file of unknown type.
`3`	If *item* is a MEX-file.
`4`	If *item* is a MDL-file.
`5`	If *item* is a built-in MATLAB function.
`6`	If *item* is a P-file.
`7`	If *item* is a directory.

`exist('`*`item`*`')` or `exist('`*`item.ext`*`')`returns `2` if *item* is on the MATLAB search path but the filename extension (*ext*) is not `m`, `p`, or `mex`. *item* may be a `MATLABPATH` relative partial pathname.

`ident = exist('`*`item`*`','`*`kind'`*`)` returns logical true (`1`) if an item of the specified *kind* is found, and returns `0` otherwise. *kind* may be:

`'var'`	Checks only for variables.
`'builtin'`	Checks only for built-in functions.
`'file'`	Checks only for files.
`'dir'`	Checks only for directories.

Examples

exist can check whether a MATLAB function is built-in or a file:

```
ident = exist('plot')
ident =
     5
```

plot is a built-in function.

See Also

dir	Directory listing
help	Online help for MATLAB functions and M-files
lookfor	Keyword search through all help entries
what	Directory listing of M-files, MAT-files, and MEX-files
which	Locate functions and files
who	List directory of variables in memory

See also partialpath.

Purpose Exponential

Syntax

```
Y = exp(X)
```

Description The `exp` function is an elementary function that operates element-wise on arrays. Its domain includes complex numbers.

`Y = exp(X)` returns the exponential for each element of `X`. For complex $z = x + i*y$, it returns the complex exponential: $e^z = e^x(\cos(y) + i\sin(y))$.

Remark Use `expm` for matrix exponentials.

See Also

`expm`	Matrix exponential
`log`	Natural logarithm
`log10`	Common (base 10) logarithm
`expint`	Exponential integral

expint

Purpose

Exponential integral

Syntax

```
Y = expint(X)
```

Definitions

The exponential integral is defined as:

$$\int_x^\infty \frac{e^{-t}}{t}\,dt$$

Another common definition of the exponential integral function is the Cauchy principal value integral:

$$E_i(x) = \int_{-\infty}^{x} e^{-t}\,dt$$

which, for real positive `x`, is related to `expint` as follows:

```
expint(-x+i*0) = -Ei(x) - i*pi
Ei(x) = real(-expint(-x))
```

Description

`Y = expint(X)` evaluates the exponential integral for each element of `X`.

Algorithm

For elements of `X` in the domain $[-38, 2]$, `expint` uses a series expansion representation (equation 5.1.11 in [1]):

$$E_i(x) = -\gamma - \ln x - \sum_{n=1}^{\infty} \frac{(-1)^n x^n}{n\ n!}$$

For all other elements of `X`, `expint` uses a continued fraction representation (equation 5.1.22 in [1]):

$$E_n(z) = e^{-z}\left(\frac{1}{z+}\,\frac{n}{1+}\,\frac{1}{z+}\,\frac{n+1}{1+}\,\frac{2}{z+}\ldots\right),\ |angle(z)| < \pi$$

References

[1] Abramowitz, M. and I. A. Stegun. *Handbook of Mathematical Functions*. Chapter 5, New York: Dover Publications, 1965.

Purpose Matrix exponential

Syntax

```
Y = expm(X)
```

Description `Y = expm(X)` raises the constant *e* to the matrix power `X`. Complex results are produced if `X` has nonpositive eigenvalues.

Use `exp` for the element-by-element exponential.

Algorithm The `expm` function is built-in, but it uses the Padé approximation with scaling and squaring algorithm expressed in the file `expm1.m`.

A second method of calculating the matrix exponential uses a Taylor series approximation. This method is demonstrated in the file `expm2.m`. The Taylor series approximation is not recommended as a general-purpose method. It is often slow and inaccurate.

A third way of calculating the matrix exponential, found in the file `expm3.m`, is to diagonalize the matrix, apply the function to the individual eigenvalues, and then transform back. This method fails if the input matrix does not have a full set of linearly independent eigenvectors.

References [1] and [2] describe and compare many algorithms for computing `expm(X)`. The built-in method, `expm1`, is essentially method 3 of [2].

Examples Suppose `A` is the 3-by-3 matrix

```
1        1        0
0        0        2
0        0       -1
```

then `expm(A)` is

```
2.7183   1.7183        1.0862
0        1.0000        1.2642
0             0        0.3679
```

while `exp(A)` is

```
2.7183        2.7183        1.0000
1.0000        1.0000        7.3891
1.0000        1.0000        0.3679
```

Notice that the diagonal elements of the two results are equal; this would be true for any triangular matrix. But the off-diagonal elements, including those below the diagonal, are different.

See Also

`exp`	Exponential
`funm`	Evaluate functions of a matrix
`logm`	Matrix logarithm
`sqrtm`	Matrix square root

References

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

[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 Identity matrix

Syntax

```
Y = eye(n)
Y = eye(m,n)
Y = eye(size(A))
```

Description `Y = eye(n)` returns the `n`-by-`n` identity matrix.

`Y = eye(m,n)` or `eye([m n])` returns an `m`-by-`n` matrix with `1`'s on the diagonal and `0`'s elsewhere.

`Y = eye(size(A))` returns an identity matrix the same size as `A`.

Limitations The identity matrix is not defined for higher-dimensional arrays. The assignment `y = eye([2,3,4])` results in an error.

See Also

`ones`	Create an array of all ones
`rand`	Uniformly distributed random numbers and arrays
`randn`	Normally distributed random numbers and arrays
`zeros`	Create an array of all zeros

factor

Purpose

Prime factors

Syntax

```
f = factor(n)
f = factor(symb)
```

Description

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

Examples

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

See Also

`isprime`	True for prime numbers
`primes`	Generate list of prime numbers

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).

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

See Also

`ferror`	Query MATLAB about errors in file input or output
`fopen`	Open a file or obtain information about open files
`fprintf`	Write formatted data to file
`fread`	Read binary data from file
`fscanf`	Read formatted data from file
`fseek`	Set file position indicator
`ftell`	Get file position indicator
`fwrite`	Write binary data from a MATLAB matrix to a file

feof

Purpose Test for end-of-file

Syntax

```
eofstat = feof(fid)
```

Description

`eofstat = feof(fid)` tests whether the end-of-file indicator is set for the file with identifier `fid`. It returns `1` if the end-of-file indicator is set, or `0` if it is not. (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`	Open a file or obtain information about open files

Purpose

Query MATLAB about errors in file input or output

Syntax

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

Description

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

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

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

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

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

See Also

`fclose`	Close one or more open files
`fopen`	Open a file or obtain information about open files
`fprintf`	Write formatted data to file
`fread`	Read binary data from file
`fscanf`	Read formatted data from file
`fseek`	Set file position indicator
`ftell`	Get file position indicator
`fwrite`	Write binary data from a MATLAB matrix to a file

feval

Purpose

Function evaluation

Syntax

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

Description

`[y1,y2...] = feval(`*function*`,x1, ...,xn)` If *function* is a 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.

Examples

The statements:

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

are equivalent. `feval` is useful in functions that accept string arguments specifying function names. For example, the function:

```
function plotf(fun,x)
y = feval(fun,x);
plot(x,y)
```

can be used to graph other functions.

See Also

`assignin`	Assign value to variable in workspace
`builtin`	Execute builtin function from overloaded method
`eval`	Interpret strings containing MATLAB expressions
`evalin`	Evaluate expression in workspace.

Purpose

One-dimensional fast Fourier transform

Syntax

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

Definition

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

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

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

where

is an nth root of unity.

Description

`Y = fft(X)` returns the discrete Fourier transform 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 FFT. 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)` apply the FFT operation across the dimension `dim`.

Remarks

The `fft` function employs a radix-2 fast Fourier transform algorithm if the length of the sequence is a power of two, and a slower mixed-radix algorithm if it is not. See "Algorithm."

Examples

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

```
t = 0:0.001:0.6;
x = sin(2*pi*50*t)+sin(2*pi*120*t);
y = x + 2*randn(size(t));
plot(y(1:50))
```

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 spectral density, a measurement of the energy 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))
```

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

When the sequence length is a power of two, a high-speed radix-2 fast Fourier transform algorithm is employed. The radix-2 FFT routine is optimized to perform a real FFT if the input sequence is purely real, otherwise it computes the complex FFT. This causes a real power-of-two FFT to be about 40% faster than a complex FFT of the same length.

When the sequence length is not an exact power of two, an alternate algorithm finds the prime factors of the sequence length and computes the mixed-radix discrete Fourier transforms of the shorter sequences.

The time it takes to compute an FFT varies greatly depending upon the sequence length. The FFT of sequences whose lengths have many prime factors is computed quickly; the FFT of those that have few is not. Sequences whose lengths are prime numbers are reduced to the raw (and slow) discrete Fourier transform (DFT) algorithm. For this reason it is generally better to stay with power-of-two FFTs unless other circumstances dictate that this cannot be done. For example, on one machine a 4096-point real FFT takes 2.1 seconds and a complex FFT of the same length takes 3.7 seconds. The FFTs of neighboring sequences of length 4095 and 4097, however, take 7 seconds and 58 seconds, respectively.

See Also

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

`fft2`	Two-dimensional fast Fourier transform
`fftshift`	Rearrange the outputs of `fft` and `fft2`
`ifft`	Inverse one-dimensional fast Fourier transform

Purpose

Two-dimensional fast Fourier transform

Syntax

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

Description

`Y = fft2(X)` performs the two-dimensional FFT. 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 FFT of each column `X`, then of each row of the result. The time required to compute `fft2(X)` depends strongly on the number of prime factors in `[m,n] = size(X)`. It is fastest when `m` and `n` are powers of 2.

See Also

`fft`	One-dimensional fast Fourier transform
`fftshift`	Rearrange the outputs of `fft` and `fft2`
`ifft2`	Inverse two-dimensional fast Fourier transform

Purpose Multidimensional fast Fourier transform

Syntax

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

Description `Y = fftn(X)` performs the N-dimensional fast Fourier transform. 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 time required to compute `fftn(X)` depends strongly on the number of prime factors of the dimensions of `X`. It is fastest when all of the dimensions are powers of 2.

See Also

`fft`	One-dimensional fast Fourier transform
`fft2`	Two-dimensional fast Fourier transform
`ifftn`	Inverse multidimensional fast Fourier transform

fftshift

Purpose Move zero'th lag to center of spectrum.

Syntax

```
Y = fftshift(X)
```

Description `Y = fftshift(X)` rearranges the outputs of `fft` and `fft2` by moving the zero frequency component to the center of the spectrum, which is sometimes a more convenient form.

If `X` is a vector, `Y` is a vector with the left and right halves swapped.

If `X` is a matrix, `Y` is a matrix with quadrants one and three swapped with quadrants two and four.

Examples For any matrix `X`

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

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

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

this DC component is near the center of the matrix.

See Also

`fft`	One-dimensional fast Fourier transform
`fft2`	Two-dimensional fast Fourier transform

Purpose Return the next line of a file as a string without line terminator(s)

Syntax

```
line = fgetl(fid)
```

Description

`line = fgetl(fid)` returns the next line of the file with identifier `fid`. If `fgetl` encounters the end of a file, it returns `-1`. (See `fopen` for a complete description of `fid`.)

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

See Also

`fgets`	Return the next line of a file as a string with line terminator(s)

fgets

Purpose Return the next line of a file as a string with line terminator(s)

Syntax

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

Description

`line = fgets(fid)` returns the next line for the file with identifier `fid`. If `fgets` encounters the end of a file, it returns `–1`. (See `fopen` for a complete description of `fid`.)

The returned string `line` includes the line terminator(s) associated with the text line (to obtain the string without the line terminator(s), use `fgetl`).

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

See Also

`fgetl`	Return the next line of a file as a string without line terminator(s)

Purpose Field names of a structure

Syntax

```
names = fieldnames(s)
```

Description

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

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 `n = fieldnames(mystr)` yields

```
n =

    'name'
    'ID'
```

See Also

`getfield`	Get field of structure array
`setfield`	Set field of structure array

filter

Purpose

Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter

Syntax

```
y = filter(b,a,X)
[y,zf] = filter(b,a,X)
[y,zf] = filter(b,a,X,zi)
y = filter(b,a,X,zi,dim)
[...] = filter(b,a,X,[],dim)
```

Description

The `filter` function filters a data sequence using a digital filter which works for both real and complex inputs. The filter is a *direct form II transposed* implementation of the standard difference equation (see “Algorithm”).

`y = filter(b,a,X)` filters the data in vector `X` with the filter described by numerator coefficient vector `b` and denominator coefficient vector `a`. If `a(1)` is not equal to `1`, `filter` normalizes the filter coefficients by `a(1)`. If `a(1)` equals `0`, `filter` returns an error.

If `X` is a matrix, `filter` operates on the columns of `X`. If `X` is a multidimensional array, `filter` operates on the first nonsingleton dimension.

`[y,zf] = filter(b,a,X)` returns the final conditions, `zf`, of the filter delays. Output `zf` is a vector of `max(size(a),size(b))` or an array of such vectors, one for each column of `X`.

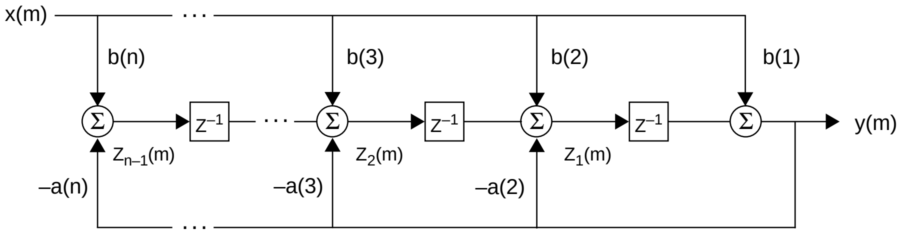
`[y,zf] = filter(b,a,X,zi)` accepts initial conditions and returns the final conditions, `zi` and `zf` respectively, of the filter delays. Input `zi` is a vector (or an array of vectors) of length `max(length(a),length(b))-1`.

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

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

Algorithm

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



or

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

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

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

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

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

filtfilt in the Signal Processing Toolbox, and:

`filter2`	Two-dimensional digital filtering

References

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

filter2

Purpose Two-dimensional digital filtering

Syntax

```
Y = filter2(B,X)
Y = filter2(B,X,'shape')
```

Description `Y = filter2(B,X)` filters the data in `X` with the two-dimensional FIR filter `B`. The result, `Y`, is computed using two-dimensional convolution and is the same size as `X`.

`Y = filter2(B,X,'`*shape*`')` returns `Y` computed via two-dimensional convolution with size specified by *shape*—one of three strings which determines the size of the output matrix:

- `same` returns the central part of the convolution that is the same size as `X` (default).
- `full` returns the full two-dimensional convolution, `size(Y)` > `size(X)`.
- `valid` returns only those parts of the convolution that are computed without the zero-padded edges, `size(Y)` < `size(X)`.

Algorithm The `filter2` function uses `conv2` to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, `filter2` extracts and returns the central part of the convolution that is the same size as the input matrix. Use the *shape* parameter to specify an alternate part of the convolution for return.

See Also

`conv2`	Two-dimensional convolution
`filter`	Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter

Purpose

Find indices and values of nonzero elements

Syntax

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

Description

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

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

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

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

Examples

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

Some operations on a vector

```
x = [11  0  33  0  55]';
find(x)

ans =

     1
     3
     5

find(x == 0)

ans =

     2
     4
```

```
find(0 < x & x < 10*pi)

ans =

     1
```

And on a matrix

```
M = magic(3)

M =

     8     1     6
     3     5     7
     4     9     2

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

i =            j =               m =

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

See Also

The relational operators <, <=,>,>=,==, ~=, and:

`nonzeros`	Nonzero matrix elements
`sparse`	Create sparse matrix

Purpose Find one string within another

Syntax

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

Description

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

Examples

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

ans =
     6    30
```

See Also

`strcmp`	Compare strings
`strmatch`	Find possible matches for a string
`strncmp`	Compare the first `n` characters of two strings

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 =

  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`	Round toward infinity
`floor`	Round towards minus infinity
`round`	Round to nearest integer

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`	Flip matrices left-right
`flipud`	Flip matrices up-down
`permute`	Rearrange the dimensions of a multidimensional array
`rot90`	Rotate matrix 90°

fliplr

Purpose Flip matrices left-right

Syntax

```
B = fliplr(A)
```

Description `B = fliplr(A)` returns `A` with columns flipped in the left-right direction, that is, about a vertical axis.

Examples

```
A =
     1    4
     2    5
     3    6
```

produces

```
     4    1
     5    2
     6    3
```

Limitations Array `A` must be two dimensional.

See Also

`flipdim`	Flip array along a specified dimension
`flipud`	Flip matrices up-down
`rot90`	Rotate matrix 90°

flipud

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.

Examples

```
A =
     1    4
     2    5
     3    6
```

produces

```
     3    6
     2    5
     1    4
```

Limitations Array `A` must be two dimensional.

See Also

`flipdim`	Flip array along a specified dimension
`fliplr`	Flip matrices left-right
`rot90`	Rotate matrix 90°

floor

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 =

  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`	Round toward infinity
`fix`	Round towards zero
`round`	Round to nearest integer

Purpose

Count floating-point operations

Syntax

```
f = flops
flops(0)
```

Description

`f = flops` returns the cumulative number of floating-point operations.

`flops(0)` resets the count to zero.

Examples

If `A` and `B` are real `n`-by-`n` matrices, some typical flop counts for different operations are:

Operation	Flop Count
`A+B`	`n^2`
`A*B`	`2*n^3`
`A^100`	`99*(2*n^3)`
`lu(A)`	`(2/3)*n^3`

MATLAB's version of the LINPACK benchmark is:

```
n = 100;
A = rand(n,n);
b = rand(n,1);
flops(0)
tic;
x = A\b;
t = toc
megaflops = flops/t/1.e6
```

Algorithm

It is not feasible to count all the floating-point operations, but most of the important ones are counted. Additions and subtractions are each one flop if real and two if complex. Multiplications and divisions count one flop each if the result is real and six flops if it is complex. Elementary functions count one if real and more if complex.

fmin

Purpose

Minimize a function of one variable

Syntax

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

Description

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

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

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

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

Arguments

`x1,x2`	Interval over which *function* is minimized.
`P1,P2...`	Arguments to be passed to *function*.
fun	A string containing the name of the function to be minimized.
`options`	A vector of control parameters. Only three of the 18 components of `options` are referenced by `fmin`; Optimization Toolbox functions use the others. The three control `options` used by `fmin` are: • `options(1)` — If this is nonzero, intermediate steps in the solution are displayed. The default value of `options(1)` is 0. • `options(2)` — This is the termination tolerance. The default value is `1.e–4`. • `options(14)` — This is the maximum number of steps. The default value is `500`.

Examples

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

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

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

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

Then invoke `fmin` with

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

The result is

```
x =
    0.8165
```

The value of the function at the minimum is

```
y = f(x)

y =
    -6.0887
```

Algorithm

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

See Also

`fmins`	Minimize a function of several variables
`fzero`	Zero of a function of one variable

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

References

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

Purpose

Minimize a function of several variables

Syntax

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

Description

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

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

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

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

Arguments

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

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

Examples

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

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

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

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

The statements

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

produce

```
x =

    1.0000    1.0000

ans =

    165
```

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

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

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

Then the statement

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

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

Algorithm

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

At each step of the search, a new point in or near the current simplex is generated. The function value at the new point is compared with the function's values at the vertices of the simplex and, usually, one of the vertices is replaced by the new point, giving a new simplex. This step is repeated until the diameter of the simplex is less than the specified tolerance.

See Also

fmin Minimize a function of one variable
foptions in the Optimization Toolbox (or type help foptions).

References

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

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

fopen

Purpose

Open a file or obtain information about open files

Syntax

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

Description

If `fopen` successfully opens a file, it returns a file identifier `fid`, and the value of `message` is empty. The file identifier can be used as the first argument to other file input/output routines. If `fopen` does not successfully open the file, it returns a `-1` value for `fid`. In that case, the value of `message` is a string that helps you determine the type of error that occurred.

Three `fids` are predefined and cannot be explicitly opened or closed:

- 0 — Standard input, which is always open for reading (`permission` set to `'r'`),
- 1— Standard output, which is always open for appending (`permission` set to `'a'`), and
- 2 — Standard error, which is always open for appending (`permission` set to `'a'`).

`fid = fopen(`*`filename`*`,`*`permission`*`)` opens the file *filename* in the mode specified by *permission* and returns `fid`, the file identifier. *filename* may a `MATLABPATH` relative partial pathname. If the file is opened for reading and it is not found in the current working directory, `fopen` searches down MATLAB's search path.

permission is one of the strings:

`'r'`	Open the file for reading (default).
`'r+'`	Open the file for reading and writing.
`'w'`	Delete the contents of an existing file or create a new file, and open it for writing.
`'w+'`	Delete the contents of an existing file or create new file, and open it for reading and writing.
`'W'`	Write without automatic flushing; used with tape drives

`'a'`	Create and open a new file or open an existing file for writing, appending to the end of the file.
`'a+'`	Create and open new file or open an existing file for reading and writing, appending to the end of the file.
`'A'`	Append without automatic flushing; used with tape drives

Add a `'t'` to these strings, for example, `'rt'`, on systems that distinguish between text and binary files, to force the file to be opened in text mode. Under DOS and VMS, for example, you cannot read a text file unless you set the permission to `'rt'`. Similarly, use a `'b'` to force the file to be opened in binary mode (the default).

`[fid,message] = fopen(`*`filename`*`,`*`permission`*`,`*`format`*`)` opens a file as above, returning file identifier and message. In addition, you specify the numeric format with *`format`*, a string defining the numeric format of the file, allowing you to share files between machines of different formats. If you omit the *`format`* argument, the numeric format of the local machine is used. Individual calls to `fread` or `fwrite` can override the numeric format specified in a call to `fopen`. Permitted format strings are:

`'native'` or `'n'`	The numeric format of the machine you are currently running
`'ieee–le'` or `'l'`	IEEE floating point with little-endian byte ordering
`'ieee–be'` or `'b'`	IEEE floating point with big-endian byte ordering
`'vaxd'` or `'d'`	VAX D floating point and VAX ordering
`'vaxg'` or `'g'`	VAX G floating point and VAX ordering
`'cray'` or `'c'`	Cray floating point with big-endian byte ordering
`'ieee-le.l64'` or `'a'`	IEEE floating point with little-endian byte ordering and 64-bit long data type
`'ieee-be.l64'` or `'s'`	IEEE floating point with big-endian byte ordering and 64-bit long data type

`fids = fopen('all')` returns a row vector containing the file identifiers of all open files, not including 0, 1, and 2 (standard input, output, and error). The number of elements in the vector is equal to the number of open files.

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

See Also

`fclose`	Close one or more open files
`ferror`	Query MATLAB about errors in file input or output
`fprintf`	Write formatted data to file
`fread`	Read binary data from file
`fscanf`	Read formatted data from file
`fseek`	Set file position indicator
`ftell`	Get file position indicator
`fwrite`	Write binary data from a MATLAB matrix to a file

See also partialpath.

Purpose

Repeat statements a specific number of times

Syntax

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

Description

The general format is

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

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

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

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

Examples

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

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

Step `s` with increments of `–0.1`

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

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

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

The line

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

has the same effect as

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

except `j` is also set here.

See Also

`break`	Break out of flow control structures
`end`	Terminate `for`, `while`, `switch`, and `if` statements and indicate the last index
`if`	Conditionally execute statements
`return`	Return to the invoking function
`switch`	Switch among several cases based on expression
`while`	Repeat statements an indefinite number of times

Purpose Control the output display format

Syntax MATLAB performs all computations in double precision.The `format` command described below switches among different display formats.

Description

Command	Result	Example
`format`	Default. Same as `short`.	
`format short`	5 digit scaled fixed point	`3.1416`
`format long`	15 digit scaled fixed point	`3.14159265358979`
`format short e`	5 digit floating-point	`3.1416e+00`
`format long e`	15 digit floating-point	`3.141592653589793e+00`
`format short g`	Best of 5 digit fixed or floating	`3.1416`
`format long g`	Best of 15 digit fixed or floating	`3.14159265358979`
`format hex`	Hexadecimal	`400921fb54442d18`
`format bank`	Fixed dollars and cents	`3.14`
`format rat`	Ratio of small integers	`355/113`
`format +`	+,–, blank	`+`
`format compact`	Suppresses excess line feeds.	
`format loose`	Add line feeds.	

Algorithms The command `format +` displays +, –, and blank characters for positive, negative, and zero elements. `format hex` displays the hexadecimal representation of a binary double-precision number. `format rat` uses a continued fraction algorithm to approximate floating-point values by ratios of small integers. See `rat.m` for the complete code.

See Also `fprintf`, `num2str`, `rat`, `sprintf`, `spy`

fprintf

Purpose Write formatted data to file

Syntax

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

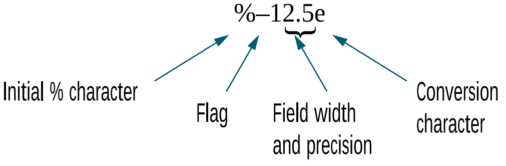
Description

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

Argument `fid` is an integer file identifier obtained from `fopen`. (It may also be `1` for standard output (the screen) or `2` for standard error. See `fopen` for more information.) Omitting `fid` from `fprintf`'s argument list causes output to appear on the screen, and is the same as writing to standard output (`fid = 1`)

`fprintf(format,A,...)` writes to standard output—the screen.

The *format* string specifies notation, alignment, significant digits, field width, and other aspects of output format. It can contain ordinary alphanumeric characters; along with escape characters, conversion specifiers, and other characters, organized as shown below:



For more information see "Tables" and "References".

Remarks

The `fprintf` function behaves like its ANSI C language `fprintf()` namesake with certain exceptions and extensions. These include:

1 The following non-standard subtype specifiers are supported for conversion specifiers `%o`, `%u`, `%x`, and `%X`.

- `t` The underlying C data type is a float rather than an unsigned integer.
- `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`'.

2 The `fprintf` function is *vectorized* for the case when input matrix `A` is non-scalar. The format string is cycled through the elements of `A` (columnwise) until all the elements are used up. It is then cycled in a similar manner, without reinitializing, through any additional matrix arguments.

Tables

The following tables describe the non-alphanumeric characters found in format specification strings.

Escape Characters

Character	Description
`\n`	New line
`\t`	Horizontal tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Form feed
`\\`	Backslash
\'' or ''	Single quotation mark
`%%`	Percent character

Conversion characters specify the notation of the output.

Conversion Specifiers

Specifier	Description
`%c`	Single character
`%d`	Decimal notation (signed)
`%e`	Exponential notation (using a lowercase `e` as in `3.1415e+00`)
`%E`	Exponential notation (using an uppercase `E` as in `3.1415E+00`)
`%f`	Fixed-point notation
`%g`	The more compact of `%e` or `%f`, as defined in [2]. Insignificant zeros do not print.
`%G`	Same as `%g`, but using an uppercase `E`
`%o`	Octal notation (unsigned)
`%s`	String of characters
`%u`	Decimal notation (unsigned)
`%x`	Hexadecimal notation (using lowercase letters `a`–`f`)
`%X`	Hexadecimal notation (using uppercase letters `A`–`F`)

Other characters can be inserted into the conversion specifier between the `%` and the conversion character.

Other Characters

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`
Digits (field width)	A digit string specifying the minimum number of digits to be printed.	`%6f`
Digits (precision)	A digit string including a period (.) specifying the number of digits to be printed to the right of the decimal point.	`%6.2f`

For more information about format strings, refer to the `printf()` and `fprintf()` routines in the documents listed in “References”.

Examples

The statements

```
x = 0:.1:1;
y = [x; exp(x)];
fid = fopen('exp.txt','w');
fprintf(fid,'%6.2f %12.8f\n',y);
fclose(fid)
```

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

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

The command

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

displays a line on the screen:

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

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

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

displays on the screen:

```
It's Friday.
```

The commands

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

display the lines:

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

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

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

See Also

`fclose`	Close one or more open files
`ferror`	Query MATLAB about errors in file input or output
`fopen`	Open a file or obtain information about open files
`fscanf`	Read formatted data from file
`fseek`	Set file position indicator
`ftell`	Get file position indicator

References

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

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

Purpose Read binary data from file

Syntax

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

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

`size` is an optional argument that determines how much data is read. If `size` is not specified, `fread` reads to the end of the file. Valid options are:

`n`	Reads `n` elements into a column vector.
`inf`	Reads to the end of the file, resulting in a column vector containing the same number of elements as are in the file.
`[m,n]`	Reads enough elements to fill an `m–by–n` matrix, filling in elements in column order, padding with zeros if the file is too small to fill the matrix.

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

`precision` is a string representing the numeric precision of the values read, *precision* controls the number of bits read for each value and the interpretation of those bits as an integer, a floating-point value, or a character. The *precision* string may contain a positive integer repetition factor of the form `'n*'` which prepends one of the strings above, like `'40*uchar'`. If *precision* is not specified, the default is `'uchar'` (8-bit unsigned character) is assumed. See “Remarks” for more information.

`[A,count] = fread(fid,size,`*`precision`*`,skip)` includes an optional `skip` argument that specifies the number of bytes to skip after each read. This is useful for extracting data in noncontiguous fields from fixed length records. If `precision` is a bit format like `'bitN'` or `'ubitN'`, `skip` is specified in bits.

Remarks

Numeric precisions can differ depending on how numbers are represented in your computer's architecture, as well as by the type of compiler used to produce executable code for your computer.

The tables below give C-compliant, platform-independent numeric precision string formats that you should use whenever you want your code to be portable.

For convenience, MATLAB accepts some C and Fortran data type equivalents for the MATLAB precisions listed. If you are a C or Fortran programmer, you may find it more convenient to use the names of the data types in the language with which you are most familiar.

MATLAB	C or Fortran	Interpretation
'char'	'char*1'	Character; 8 bits
'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
'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

If you always work on the same platform and don't care about portability, these platform-dependent numeric precision string formats are also available:

MATLAB	C or Fortran	Interpretation
'short'	'short'	Integer; 16 bits
'int'	'int'	Integer; 32 bits
'long'	'long'	Integer; 32 or 64 bits
'ushort'	'usigned 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
'double'	'double'	Floating-point; 64 bits

Two formats map to an input steam of bits rather than bytes:

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

See Also

fclose	Close one or more open files
ferror	Query MATLAB about errors in file input or output
fopen	Open a file or obtain information about open files
fprintf	Write formatted data to file
fscanf	Read formatted data from file
fseek	Set file position indicator
ftell	Get file position indicator
fwrite	Write binary data from a MATLAB matrix to a file

frewind

Purpose

Rewind an open file

Syntax

```
frewind(fid)
```

Description

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

Remarks

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

See Also

`fclose`	Close one or more open files
`ferror`	Query MATLAB about errors in file input or output
`fopen`	Open a file or obtain information about open files
`fprintf`	Write formatted data to file
`fread`	Read binary data from file
`fscanf`	Read formatted data from file
`fseek`	Set file position indicator
`ftell`	Get file position indicator
`fwrite`	Write binary data from a MATLAB matrix to a file

Purpose

Read formatted data from file

Syntax

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

Description

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

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

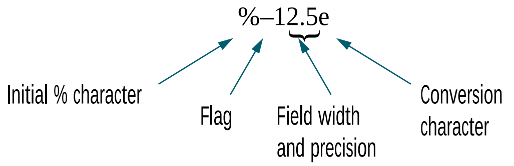
`n`	Read `n` elements into a column vector.
`inf`	Read to the end of the file, resulting in a column vector containing the same number of elements as are in the file.
`[m,n]`	Read enough elements to fill an `m`-by-`n` matrix, filling the matrix in column order. `n` can be `Inf`, but not `m`.

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

Remarks

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

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



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

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

Valid conversion characters are:

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

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

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

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

Examples

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

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

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

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

See Also

`fclose`	Close one or more open files
`ferror`	Query MATLAB about errors in file input or output
`fopen`	Open a file or obtain information about open files
`fprintf`	Write formatted data to file
`fread`	Read binary data from file
`fseek`	Set file position indicator
`ftell`	Get file position indicator
`fwrite`	Write binary data from a MATLAB matrix to a file

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*.

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 about the nature of the error.

See Also

`fopen`	Open a file or obtain information about open files
`ftell`	Get file position indicator

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`	Close one or more open files
`ferror`	Query MATLAB about errors in file input or output
`fopen`	Open a file or obtain information about open files
`fprintf`	Write formatted data to file
`fread`	Read binary data from file
`fscanf`	Read formatted data from file
`fseek`	Set file position indicator
`fwrite`	Write binary data from a MATLAB matrix to a file

full

Purpose

Convert sparse matrix to full matrix

Syntax

```
A = full(S)
```

Description

`A = full(S)` converts a sparse matrix `S` to full storage organization. If `S` is a full matrix, it is left unchanged. If `A` is full, `issparse(A)` is `0`.

Remarks

Let `X` be an `m`-by-`n` matrix with `nz = nnz(X)` nonzero entries. Then `full(X)` requires space to store `m*n` real numbers while `sparse(X)` requires space to store `nz` real numbers and `(nz+n)` integers.

On most computers, a real number requires twice as much storage as an integer. On such computers, `sparse(X)` requires less storage than `full(X)` if the density, `nnz/prod(size(X))`, is less than one third. Operations on sparse matrices, however, require more execution time per element than those on full matrices, so density should be considerably less than two-thirds before sparse storage is used.

Examples

Here is an example of a sparse matrix with a density of about two-thirds. `sparse(S)` and `full(S)` require about the same number of bytes of storage.

```
S = sparse(rand(200,200) < 2/3);
A = full(S);
whos
Name     Size      Bytes   Class
  A    200X200   320000   double array (logical)
  S    200X200   318432   sparse array (logical)
```

See Also

`sparse`	Create sparse matrix

Purpose Build full filename from parts

Syntax `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. Specify the filename as `''` to build a pathname from parts. On VMS, care is taken to handle the cases involving [or].

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

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

produce the same result on UNIX, but only the second one works on all platforms.

function

Purpose

Function M-files

Description

You add new functions to MATLAB's vocabulary by expressing them in terms of existing functions. The existing commands and functions that compose the new function reside in a text file called an *M-file*.

M-files can be either *scripts* or *functions*. Scripts are simply files containing a sequence of MATLAB statements. Functions make use of their own local variables and accept input arguments.

The name of an M-file begins with an alphabetic character, and has a filename extension of `.m`. The M-file name, less its extension, is what MATLAB searches for when you try to use the script or function.

A line at the top of a function M-file contains the syntax definition. The name of a function, as defined in the first line of the M-file, should be the same as the name of the file without the `.m` extension. For example, the existence of a file on disk called `stat.m` with

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

defines a new function called `stat` that calculates the mean and standard deviation of a vector. The variables within the body of the function are all local variables.

A *subfunction*,visible only to the other functions in the same file, is created by defining a new function with the `function` keyword after the body of the preceding function or subfunction. For example, `avg` is a subfunction within the file `stat.m`:

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

function mean = avg(x,n)
mean = sum(x)/n;
```

Subfunctions are not visible outside the file where they are defined. Functions normally return when the end of the function is reached. Use a `return` statement to force an early return.

When MATLAB does not recognize a function by name, it searches for a file of the same name on disk. If the function is found, MATLAB compiles it into memory for subsequent use. In general, if you input the name of something to MATLAB, the MATLAB interpreter:

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

At execution MATLAB:

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

When you call an M-file function from the command line or from within another M-file, MATLAB parses the function and stores it in memory. The parsed function remains in memory until cleared with the clear command or you `quit` MATLAB. The `pcode` command performs the parsing step and stores the result on the disk as a P-file to be loaded later.

See Also

`nargin`	Number of function arguments (input)
`nargout`	Number of function arguments(output)
`pcode`	Create preparsed pseudocode file (P-file)
`varargin`	Pass or return variable numbers of arguments (input)
`varargout`	Pass or return variable numbers of arguments (output)
`what`	Directory listing of M-files, MAT-files, and MEX-files

funm

Purpose

Evaluate functions of a matrix

Syntax

```
Y = funm(X,'function')
[Y,esterr] = funm(X,'function')
```

Description

`Y = funm(X,'function')` evaluates *function* using Parlett's method [1]. `X` must be a square matrix, and *function* any element-wise function.

The commands `funm(X,'sqrt')` and `funm(X,'log')` are equivalent to the commands `sqrtm(X)` and `logm(X)`. The commands `funm(X,'exp')` and `expm(X)` compute the same function, but by different algorithms. `expm(X)` is preferred.

`[Y,esterr] = funm(X,'function')` does not print any message, but returns a very rough estimate of the relative error in the computer result. If `X` is symmetric or Hermitian, then its Schur form is diagonal, and `funm` is able to produce an accurate result.

Examples

The statements

```
S = funm(X,'sin');
C = funm(X,'cos');
```

produce the same results to within roundoff error as

```
E = expm(i*X);
C = real(E);
S = imag(E);
```

In either case, the results satisfy `S*S+C*C = I`, where `I = eye(size(X))`.

Algorithm

The matrix functions are evaluated using Parlett's algorithm, 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`	Matrix exponential
`logm`	Matrix logarithm
`sqrtm`	Matrix square root

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.

fwrite

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 numeric *precision*. (See "Remarks" for more information.)

The data are written to the file in column order, and a `count` is kept of the number of elements written successfully. Argument `fid` is an integer file identifier obtained from `fopen`.

`count = fwrite(fid,A,`*`precision`*`,skip)` includes an optional `skip` argument that specifies the number of bytes to skip before each write. This is useful for inserting data into noncontiguous fields in fixed-length records. If *precision* is a bit format like `'bitN'` or `'ubitN'`, `skip` is specified in bits.

Remarks

Numeric precisions can differ depending on how numbers are represented in your computer's architecture, as well as by the type of compiler used to produce executable code for your computer.

The tables below give C-compliant, platform-independent numeric precision string formats that you should use whenever you want your code to be portable.

For convenience, MATLAB accepts some C and Fortran data type equivalents for the MATLAB precisions listed. If you are a C or Fortran programmer, you may find it more convenient to use the names of the data types in the language with which you are most familiar.

MATLAB	C or Fortran	Interpretation
'char'	'char*1'	Character; 8 bits
'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
'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

If you always work on the same platform and don't care about portability, these platform-dependent numeric precision string formats are also available:

MATLAB	C or Fortran	Interpretation
'short'	'short'	Integer; 16 bits
'int'	'int'	Integer; 32 bits
'long'	'long'	Integer; 32 or 64 bits
'ushort'	'usigned 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
'double'	'double'	Floating-point; 64 bits

Two formats map to an input steam 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 ($1 \le N \le 64$)

Examples

```
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	Close one or more open files
ferror	Query MATLAB about errors in file input or output
fopen	Open a file or obtain information about open files
fprintf	Write formatted data to file
fread	Read binary data from file
fscanf	Read formatted data from file
fseek	Set file position indicator
ftell	Get file position indicator

Purpose Zero of a function of one variable

Syntax

```
z = fzero('fun',x)
z = fzero('fun',x,tol)
z = fzero('fun',x,tol,trace)
z = fzero('fun',x,tol,trace,P1,P2,...)
```

Description

`fzero('`*fun*`',x)` finds a zero of *fun*. *fun* is a string containing the name of a real-valued function of a single real variable. The value returned is near a point where *fun* changes sign, or `NaN` if the search fails.

`fzero('`*fun*`',x)` where `x` is a vector of length 2, assumes `x` is an interval where the sign of `f(x(1))` differs from the sign of `f(x(2))`. An error occurs if this is not true. Calling `fzero` with an interval guarantees `fzero` will return a value near a point where *fun* changes sign.

`fzero('`*fun*`',x)` where `x` is a scalar value, uses `x` as a starting point. `fzero` looks for an interval containing a sign change for *fun* and containing `x`. If no such interval is found, `NaN` is returned. In this case, the search terminates when the search interval is expanded until an `Inf`, `NaN`, or complex value is found.

`fzero('`*fun*`',x,tol)` returns an answer accurate to within a relative error of `tol`.

`z = fzero('`*fun*`',x,tol,trace)` displays information at each iteration.

`z = fzero('`*fun*`',x,tol,trace,P1,P2,...)` provides for additional arguments passed to the function `fun(x,P1,P2,...)`. Pass an empty matrix for `tol` or `trace` to use the default value, for example: `fzero('`*fun*`',x,[],[],P1)`

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	A string containing the name of a file in which an arbitrary function of one variable is defined.
`x`	Your initial estimate of the *x*-coordinate of a zero of the function or an interval in which you think a zero is found.

`tol`	The relative error tolerance. By default, tol is eps.
`trace`	A nonzero value that causes the `fzero` command to display information at each iteration of its calculations.
`P1,P2`	Additional arguments passed to the function

Examples

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

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

To find the zero of `cosine` between 1 and 2:

```
x = fzero('cos',[1 2])
x =
    1.5708
```

Note that `cos(1)` and `cos(2)` differ in sign.

To find a zero of the function:

$$f(x) = x^3 - 2x - 5$$

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

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

To find the zero near 2

```
z = fzero('f',2)
z =
    2.0946
```

Since 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
```

`fzero('abs(x)+1', 1)` returns `NaN` since this function does not change sign anywhere on the real axis (and does not have a zero as well).

Algorithm

The `fzero` command is an M-file. The algorithm, which was originated by T. Dekker, uses a combination of bisection, secant, and inverse quadratic interpolation methods. An Algol 60 version, with some improvements, is given in [1]. A Fortran version, upon which the `fzero` M-file is based, is in [2].

Limitations

The `fzero` command defines a *zero* as a point where the function crosses the *x*-axis. Points where the function touches, but does not cross, the *x*-axis are not valid zeros. For example, `y = x.^2` is a parabola that touches the *x*-axis at (0,0). Since the function never crosses the *x*-axis, however, no zero is found. For functions with no valid zeros, `fzero` executes until `Inf`, `NaN`, or a complex value is detected.

See Also

`eps`	Floating-point relative accuracy
`fmin`	Minimize a function of one variable
`roots`	Polynomial roots

References

[1] Brent, R., *Algorithms for Minimization Without Derivatives*, Prentice-Hall, 1973.

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

Purpose Test matrices

Syntax

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

Description

`[A,B,C,...] = gallery('`*`tmfun`*`',P1,P2,...)` returns the test matrices specified by string *tmfun*. *tmfun* is the name of a matrix family selected from the table below. `P1, P2,...` are input parameters required by the individual matrix family. The number of optional parameters `P1,P2,...` used in the calling syntax varies from matrix to matrix.The exact calling syntaxes are detailed in the individual matrix descriptions below.

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

Test Matrices			
cauchy	chebspec	chebvand	chow
circul	clement	compar	condex
cycol	dorr	dramadah	fiedler
forsythe	frank	gearmat	grcar
hanowa	house	invhess	invol
ipjfact	jordbloc	kahan	kms
krylov	lauchli	lehmer	lesp
lotkin	minij	moler	neumann
orthog	parter	pei	poisson
prolate	rando	randhess	randsvd
redheff	riemann	ris	rosser
smoke	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 `V` of the Chebyshev spectral differentiation matrix is ill-conditioned.

chebvand—Vandermonde-like matrix for the Chebyshev polynomials

`C = gallery('chebvand',p)` produces the (primal) Chebyshev Vandermonde matrix based on the vector of points `p`, which define where the Chebyshev polynomial is calculated.

`C = gallery('chebvand',m,p)` where `m` is scalar, produces a rectangular version of the above, with `m` rows.

If `p` is a vector, then: $C(i, j) = T_{i-1}(p(j))$ where T_{i-1} is the Chebyshev polynomial of degree `i–1`. If `p` is a scalar, then `p` equally spaced points on the interval `[0,1]` are used to calculate `C`.

chow—Singular Toeplitz lower Hessenberg matrix

`A = gallery('chow',n,alpha,delta)` returns `A` such that `A = H(alpha) + delta*eye(n)`, where $H_{i,j}(\alpha) = \alpha^{(i-j+1)}$. Argument `n` is the order of the Chow matrix, while `alpha` and `delta` are scalars with default values `1` and `0`, respectively.

`H(alpha)` has `p = floor(n/2)` eigenvalues that are equal to zero. The rest of the eigenvalues are equal to `4*alpha*cos(k*pi/(n+2))^2, k=1:n–p`.

circul—Circulant matrix

`C = gallery('circul',v)` returns the circulant matrix whose first row is the vector `v`.

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. It is a special Toeplitz matrix in which the diagonals “wrap around.”

If `v` is a scalar, then `C = gallery('circul',1:v)`.

The eigensystem of `C` (n-by-n) is known explicitly: If `t` is an `n`th root of unity, then the inner product of `v` with $w = [1\ t\ t^2\ \ldots\ t^n]$ is an eigenvalue of `C` and `w(n:-1:1)` is an eigenvector.

clement—Tridiagonal matrix with zero diagonal entries

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

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

compar—Comparison matrices

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

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

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

condex—Counter-examples to matrix condition number estimators

`A = gallery('condex',n,k,theta)` returns a “counter-example” matrix to a condition estimator. It has order `n` and scalar parameter `theta` (default 100).

The matrix, its natural size, and the estimator to which it applies are specified by `k` as follows:

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

If `n` is not equal to the natural size of the matrix, then the matrix is padded out with an identity matrix to order `n`.

cycol—Matrix whose columns repeat cyclically

`A = gallery('cycol',[m n],k)` returns an `m`-by-`n` matrix with cyclically repeating columns, where one “cycle” consists of `randn(m,k)`. Thus, the rank of matrix `A` cannot exceed `k`. `k` must be a scalar.

Argument `k` defaults to `round(n/4)`, and need not evenly divide `n`.

`A = gallery('cycol',n,k)`, where `n` is a scalar, is the same as `gallery('cycol',[n n],k)`.

dorr—Diagonally dominant, ill-conditioned, tridiagonal matrix

`[c,d,e] = gallery('dorr',n,theta)` returns the vectors defining a row diagonally dominant, tridiagonal order `n` matrix that is ill-conditioned for small nonnegative values of `theta`. The default value of `theta` is `0.01`. The Dorr matrix itself is the same as `gallery('tridiag',c,d,e)`.

`A = gallery('dorr',n,theta)` returns the matrix itself, rather than the defining vectors.

dramadah—Matrix of zeros and ones whose inverse has large integer entries

`A = gallery('dramadah',n,k)` returns an `n`-by-`n` matrix of `0`'s and `1`'s for which `mu(A) = norm(inv(A),'fro')` is relatively large, although not necessarily maximal. An anti-Hadamard matrix `A` is a matrix with elements `0` or `1` for which `mu(A)` is maximal.

`n` and `k` must both be scalars. Argument `k` determines the character of the output matrix:

`k = 1`	Default. `A` is Toeplitz, with `abs(det(A)) = 1`, and `mu(A) > c(1.75)^n`, where `c` is a constant. The inverse of `A` has integer entries.
`k = 2`	`A` is upper triangular and Toeplitz. The inverse of `A` has integer entries.
`k = 3`	`A` has maximal determinant among lower Hessenberg (0,1) matrices. `det(A)` = the `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+2a), ..., sin(w+Na)]`, where `a` and `w` are given in Gear, C. W., “A Simple Set of Test Matrices for Eigenvalue Programs”, *Math. Comp.*, Vol. 23 (1969), pp. 119–125.

grcar—Toeplitz matrix with sensitive eigenvalues

`A = gallery('grcar',n,k)` returns an `n`-by-`n` Toeplitz matrix with `–1`s on the subdiagonal, `1`s on the diagonal, and `k` superdiagonals of `1`s. The default is `k = 3`. The eigenvalues are sensitive.

hanowa—Matrix whose eigenvalues lie on a vertical line in the complex plane

`A = gallery('hanowa',n,d)` returns an `n`-by-`n` block `2`-by-`2` matrix of the form:

```
[d*eye(m)   -diag(1:m)
diag(1:m)   d*eye(m)]
```

Argument `n` is an even integer `n=2*m`. Matrix `A` has complex eigenvalues of the form `d` ± `k*i`, for `1 <= k <= m`. The default value of `d` is `-1`.

house—Householder matrix

`[v, beta] = gallery('house',x)` takes `x`, a scalar or `n`-element column vector, and returns `v` and `beta` such that `eye(n,n) - beta*v*v'` is a Householder matrix.

A Householder matrix `H` satisfies the relationship

```
H*x = -sign(x(1))*norm(x)*e1
```

where `e1` is the first column of `eye(n,n)`. Note that if `x` is complex, then `sign(x) = exp(i*arg(x))` (which equals `x./abs(x)` when `x` is nonzero).

If `x = 0`, then `v = 0` and `beta = 1`.

invhess—Inverse of an upper Hessenberg matrix

`A = gallery('invhess',x,y)`, where `x` is a length `n` vector and `y` a length `n-1` vector, returns the matrix whose lower triangle agrees with that of `ones(n,1)*x'` and whose strict upper triangle agrees with that of `[1 y]*ones(1,n)`.

The matrix is nonsingular if `x(1) ~= 0` and `x(i+1) ~= y(i)` for all `i`, and its inverse is an upper Hessenberg matrix. Argument `y` defaults to `-x(1:n-1)`.

If `x` is a scalar, `invhess(x)` is the same as `invhess(1:x)`.

invol—Involutory matrix

`A = gallery('invol',n)` returns an n-by-n involutory (`A*A = eye(n)`) and ill-conditioned matrix. It is a diagonally scaled version of `hilb(n)`.

`B = (eye(n)-A)/2` and `B = (eye(n)+A)/2` are idempotent (`B*B = B`).

ipjfact—Hankel matrix with factorial elements

`[A,d] = gallery('ipjfact',n,k)` returns `A`, an n-by-n Hankel matrix, and `d`, the determinant of `A`, which is known explicitly. If `k = 0` (the default), then the elements of `A` are `A(i,j) = (i+j)!` If `k = 1`, then the elements of `A` are `A(i,j) = 1/(i+j)`.

Note that the inverse of `A` is also known explicitly.

jordbloc—Jordan block

`A = gallery('jordbloc',n,lambda)` returns the n-by-n Jordan block with eigenvalue `lambda`. The default value for `lambda` is `1`.

kahan—Upper trapezoidal matrix

`A = gallery('kahan',n,theta,pert)` returns an upper trapezoidal matrix that has interesting properties regarding estimation of condition and rank.

If `n` is a two-element vector, then `A` is `n(1)`-by-`n(2)`; otherwise, `A` is n-by-n. The useful range of `theta` is `0 < theta < pi`, with a default value of `1.2`.

To ensure that the QR factorization with column pivoting does not interchange columns in the presence of rounding errors, the diagonal is perturbed by `pert*eps*diag([n:-1:1])`. The default `pert` is `25`, which ensures no interchanges for `gallery('kahan',n)` up to at least `n = 90` in IEEE arithmetic.

kms—Kac-Murdock-Szego Toeplitz matrix

`A = gallery('kms',n,rho)` returns the n-by-n Kac-Murdock-Szego Toeplitz matrix such that `A(i,j) = rho^(abs(i-j))`, for real `rho`.

For complex `rho`, the same formula holds except that elements below the diagonal are conjugated. `rho` defaults to 0.5.

The KMS matrix `A` has these properties:

- An LDL' factorization with `L = inv(triw(n,-rho,1)')`, and `D(i,i) = (1-abs(rho)^2)*eye(n)`, except `D(1,1) = 1`.
- Positive definite if and only if `0 < abs(rho) < 1`.
- The inverse `inv(A)` is tridiagonal.

krylov—Krylov matrix

`B = gallery('krylov',A,x,j)` returns the Krylov matrix

```
[x, Ax, A^2x, ..., A^(j-1)x]
```

where `A` is an `n`-by-`n` matrix and `x` is a length `n` vector. The defaults are `x = ones(n,1)`, and `j = n`.

`B = gallery('krylov',n)` is the same as `gallery('krylov',(randn(n))`.

lauchli—Rectangular matrix

`A = gallery('lauchli',n,mu)` returns the `(n+1)`-by-`n` matrix

```
[ones(1,n); mu*eye(n)]
```

The Lauchli matrix is a well-known example in least squares and other problems that indicates the dangers of forming `A'*A`. Argument `mu` defaults to `sqrt(eps)`.

lehmer—Symmetric positive definite matrix

`A = gallery('lehmer',n)` returns the symmetric positive definite `n`-by-`n` matrix such that `A(i,j) = i/j` for `j >= i`.

The Lehmer matrix `A` has these properties:

- `A` is totally nonnegative.
- The inverse `inv(A)` is tridiagonal and explicitly known.
- The order `n <= cond(A) <= 4*n*n`.

lesp—Tridiagonal matrix with real, sensitive eigenvalues

A = gallery('lesp',n) returns an n-by-n matrix whose eigenvalues are real and smoothly distributed in the interval approximately [-2*N-3.5, -4.5].

The sensitivities of the eigenvalues increase exponentially as the eigenvalues grow more negative. The matrix is similar to the symmetric tridiagonal matrix with the same diagonal entries and with off-diagonal entries 1, via a similarity transformation with D = diag(1!,2!,...,n!).

lotkin—Lotkin matrix

A = gallery('lotkin',n) returns the Hilbert matrix with its first row altered to all ones. The Lotkin matrix A is nonsymmetric, ill-conditioned, and has many negative eigenvalues of small magnitude. Its inverse has integer entries and is known explicitly.

minij—Symmetric positive definite matrix

A = gallery('minij',n) returns the n-by-n symmetric positive definite matrix with A(i,j) = min(i,j).

The minij matrix has these properties:

- The inverse inv(A) is tridiagonal and equal to -1 times the second difference matrix, except its (n,n) element is 1.
- Givens' matrix, 2*A-ones(size(A)), has tridiagonal inverse and eigenvalues 0.5*sec((2*r-1)*pi/(4*n))^2, where r=1:n.
- (n+1)*ones(size(A))-A has elements that are max(i,j) and a tridiagonal inverse.

moler—Symmetric positive definite matrix

A = gallery('moler',n,alpha) returns the symmetric positive definite n-by-n matrix U'*U, where U = triw(n,alpha).

For the default alpha = -1, A(i,j) = min(i,j)-2, and A(i,i) = i. One of the eigenvalues of A is small.

neumann—Singular matrix from the discrete Neumann problem (sparse)

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

orthog—Orthogonal and nearly orthogonal matrices

`Q = gallery('orthog',n,k)` returns the kth type of matrix of order `n`, where `k > 0` selects exactly orthogonal matrices, and `k < 0` selects diagonal scalings of orthogonal matrices. Available types are:

`k = 1`	`Q(i,j) = sqrt(2/(n+1)) * sin(i*j*pi/(n+1))` Symmetric eigenvector matrix for second difference matrix. This is the default.
`k = 2`	`Q(i,j) = 2/(sqrt(2*n+1)) * sin(2*i*j*pi/(2*n+1))` Symmetric.
`k = 3`	`Q(r,s) = exp(2*pi*i*(r–1)*(s–1)/n) / sqrt(n)` Unitary, the Fourier matrix. Q^4 is the identity. This is essentially the same matrix as `fft(eye(n))/sqrt(n)`!
`k = 4`	Helmert matrix: a permutation of a lower Hessenberg matrix, whose first row is `ones(1:n)/sqrt(n)`.
`k = 5`	`Q(i,j) = sin(2*pi*(i–1)*(j–1)/n) + cos(2*pi*(i–1)*(j–1)/n)` Symmetric matrix arising in the Hartley transform.
`k = –1`	`Q(i,j) = cos((i–1)*(j–1)*pi/(n–1))` Chebyshev Vandermonde-like matrix, based on extrema of `T(n–1)`.
`k = –2`	`Q(i,j) = cos((i–1)*(j–1/2)*pi/n))` Chebyshev Vandermonde-like matrix, based on zeros of `T(n)`.

parter—Toeplitz matrix with singular values near pi

C = gallery('parter',n) returns the matrix C such that
C(i,j) = 1/(i–j+0.5).

C is a Cauchy matrix and a Toeplitz matrix. Most of the singular values of C are very close to pi.

pei—Pei matrix

A = gallery('pei',n,alpha), where alpha is a scalar, returns the symmetric matrix alpha*eye(n) + ones(n). The default for alpha is 1. The matrix is singular for alpha equal to either 0 or –n.

poisson—Block tridiagonal matrix from Poisson's equation (sparse)

A = gallery('poisson',n) returns the block tridiagonal (sparse) matrix of order n^2 resulting from discretizing Poisson's equation with the 5-point operator on an n-by-n mesh.

prolate—Symmetric, ill-conditioned Toeplitz matrix

A = gallery('prolate',n,w) returns the n-by-n prolate matrix with parameter w. It is a symmetric Toeplitz matrix.

If 0 < w < 0.5 then A is positive definite

- The eigenvalues of A are distinct, lie in (0,1), and tend to cluster around 0 and 1.
- The default value of w is 0.25.

randhess—Random, orthogonal upper Hessenberg matrix

`H = gallery('randhess',n)` returns an `n`-by-`n` real, random, orthogonal upper Hessenberg matrix.

`H = gallery('randhess',x)` if `x` is an arbitrary, real, length `n` vector with `n > 1`, constructs `H` nonrandomly using the elements of `x` as parameters.

Matrix `H` is constructed via a product of `n–1` Givens rotations.

rando—Random matrix composed of elements –1, 0 or 1

`A = gallery('rando',n,k)` returns a random `n`-by-`n` matrix with elements from one of the following discrete distributions:

`k = 1`	`A(i,j)` = `0` or `1` with equal probability (default)
`k = 2`	`A(i,j)` = `–1` or `1` with equal probability
`k = 3`	`A(i,j)` = `–1`, `0` or `1` with equal probability

Argument `n` may be a two-element vector, in which case the matrix is `n(1)`-by-`n(2)`.

randsvd—Random matrix with preassigned singular values

`A = gallery('randsvd',n,kappa,mode,kl,ku)` returns a banded (multidiagonal) random matrix of order `n` with `cond(A) = kappa` and singular values from the distribution mode. If `n` is a two-element vector, `A` is `n(1)`-by-`n(2)`.

Arguments `kl` and `ku` specify the number of lower and upper off-diagonals, respectively, in `A`. If they are omitted, a full matrix is produced. If only `kl` is present, `ku` defaults to `kl`.

Distribution `mode` may be:

1	One large singular value
2	One small singular value
3	Geometrically distributed singular values (default)
4	Arithmetically distributed singular values

`1`	One large singular value
`5`	Random singular values with uniformly distributed logarithm
`< 0`	If `mode` is `-1`, `-2`, `-3`, `-4`, or `-5`, then `randsvd` treats `mode` as `abs(mode)`, except that in the original matrix of singular values the order of the diagonal entries is reversed: small to large instead of large to small.

Condition number `kappa` defaults to `sqrt(1/eps)`. In the special case where `kappa < 0`, `A` is a random, full, symmetric, positive definite matrix with `cond(A) = -kappa` and eigenvalues distributed according to `mode`. Arguments `kl` and `ku`, if present, are ignored.

redheff—Redheffer's matrix of 1s and 0s

`A = gallery('redheff',n)` returns an `n`-by-`n` matrix of `0`'s and `1`'s defined by `A(i,j) = 1`, if `j = 1` or if `i` divides `j`, and `A(i,j) = 0` otherwise.

The Redheffer matrix has these properties:

- `(n-floor(log2(n)))-1` eigenvalues equal to `1`
- A real eigenvalue (the spectral radius) approximately `sqrt(n)`
- A negative eigenvalue approximately `-sqrt(n)`
- The remaining eigenvalues are provably "small."
- The Riemann hypothesis is true if and only if `det(A)` = O(n^(1/2+epsilon)) for every epsilon > `0`.

Barrett and Jarvis conjecture that "the small eigenvalues all lie inside the unit circle `abs(Z) = 1`," and a proof of this conjecture, together with a proof that some eigenvalue tends to zero as `n` tends to infinity, would yield a new proof of the prime number theorem.

riemann—Matrix associated with the Riemann hypothesis

`A = gallery('riemann',n)` returns an `n`-by-`n` matrix for which the Riemann hypothesis is true if and only if `det(A)` = O(`n`! `n`^(`-1/2`+epsilon)) for every epsilon > `0`.

The Riemann matrix is defined by:

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

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

The Riemann matrix has these properties:

- Each eigenvalue `e(i)` satisfies `abs(e(i)) <= m–1/m`, where `m = n+1`.
- `i <= e(i) <= i+1` with at most `m–sqrt(m)` exceptions.
- All integers in the interval `(m/3, m/2]` are eigenvalues.

ris—Symmetric Hankel matrix

`A = gallery('ris',n)` returns a symmetric `n`-by-`n` Hankel matrix with elements

```
A(i,j) = 0.5/(n–i–j+1.5)
```

The eigenvalues of `A` cluster around $\pi/2$ and $-\pi/2$. This matrix was invented by F.N. Ris.

rosser—Classic symmetric eigenvalue test matrix

`A = rosser` returns the Rosser matrix. This matrix was a challenge for many matrix eigenvalue algorithms. But the Francis QR algorithm, as perfected by Wilkinson and implemented in EISPACK and MATLAB, has no trouble with it. The matrix is 8-by-8 with integer elements. It has:

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

smoke—Complex matrix with a 'smoke ring' pseudospectrum

`A = gallery('smoke',n)` returns an `n`-by-`n` matrix with `1`'s on the superdiagonal, `1` in the `(n,1)` position, and powers of roots of unity along the diagonal.

`A = gallery('smoke',n,1)` returns the same except that element `A(n,1)` is zero.

The eigenvalues of `smoke(n,1)` are the nth roots of unity; those of `smoke(n)` are the nth roots of unity times `2^(1/n)`.

toeppd—Symmetric positive definite Toeplitz matrix

`A = gallery('toeppd',n,m,w,theta)` returns an `n`-by-`n` symmetric, positive semi-definite (SPD) Toeplitz matrix composed of the sum of `m` rank 2 (or, for certain `theta`, rank 1) SPD Toeplitz matrices. Specifically,

```
T = w(1)*T(theta(1)) + ... + w(m)*T(theta(m))
```

where `T(theta(k))` has `(i,j)` element `cos(2*pi*theta(k)*(i–j))`.

By default: `m = n`, `w = rand(m,1)`, and `theta = rand(m,1)`.

toeppen—Pentadiagonal Toeplitz matrix (sparse)

`P = gallery('toeppen',n,a,b,c,d,e)` returns the `n`-by-`n` sparse, pentadiagonal Toeplitz matrix with the diagonals: `P(3,1) = a`, `P(2,1) = b`, `P(1,1) = c`, `P(1,2) = d`, and `P(1,3) = e`, where `a`, `b`, `c`, `d`, and `e` are scalars.

By default, `(a,b,c,d,e) = (1,–10,0,10,1)`, yielding a matrix of Rutishauser. This matrix has eigenvalues lying approximately on the line segment `2*cos(2*t) + 20*i*sin(t)`.

tridiag—Tridiagonal matrix (sparse)

`A = gallery('tridiag',c,d,e)` returns the tridiagonal matrix with subdiagonal `c`, diagonal `d`, and superdiagonal `e`. Vectors `c` and `e` must have `length(d)–1`.

`A = gallery('tridiag',n,c,d,e)`, where `c`, `d`, and `e` are all scalars, yields the Toeplitz tridiagonal matrix of order `n` with subdiagonal elements `c`, diagonal elements `d`, and superdiagonal elements `e`. This matrix has eigenvalues

```
d + 2*sqrt(c*e)*cos(k*pi/(n+1))
```

where `k = 1:n`. (see [1].)

`A = gallery('tridiag',n)` is the same as `A = gallery('tridiag',n,–1,2,–1)`, which is a symmetric positive definite M-matrix (the negative of the second difference matrix).

triw—Upper triangular matrix discussed by Wilkinson and others

`A = gallery('triw',n,alpha,k)` returns the upper triangular matrix with ones on the diagonal and `alpha`s on the first `k >= 0` superdiagonals.

Order `n` may be a 2-vector, in which case the matrix is `n(1)`-by-`n(2)` and upper trapezoidal.

Ostrowski ["On the Spectrum of a One-parametric Family of Matrices, *J. Reine Angew. Math.*, 1954] shows that

```
cond(gallery('triw',n,2)) = cot(pi/(4*n))^2,
```

and, for large `abs(alpha)`, `cond(gallery('triw',n,alpha))` is approximately `abs(alpha)^n*sin(pi/(4*n–2))`.

Adding `–2^(2–n)` to the `(n,1)` element makes `triw(n)` singular, as does adding `–2^(1–n)` to all the elements in the first column.

vander—Vandermonde matrix

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

wathen—Finite element matrix (sparse, random entries)

`A = gallery('wathen',nx,ny)` returns a sparse, random, `n`-by-`n` finite element matrix where

```
n = 3*nx*ny + 2*nx + 2*ny + 1.
```

Matrix `A` is precisely the "consistent mass matrix" for a regular `nx`-by-`ny` grid of 8-node (serendipity) elements in two dimensions. `A` is symmetric, positive definite for any (positive) values of the "density," `rho(nx,ny)`, which is chosen randomly in this routine.

A = gallery('wathen',nx,ny,1) returns a diagonally scaled matrix such that

```
0.25 <= eig(inv(D)*A) <= 4.5
```

where D = diag(diag(A)) for any positive integers nx and ny and any densities rho(nx,ny).

wilk—Various matrices devised or discussed by Wilkinson

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

n	MATLAB Code	Result
n = 3	[A,b] = gallery('wilk',3)	Upper triangular system Ux=b illustrating inaccurate solution.
n = 4	[A,b] = gallery('wilk',4)	Lower triangular system Lx=b, ill-conditioned.
n = 5	A = gallery('wilk',5)	hilb(6)(1:5,2:6)*1.8144. A symmetric positive definite matrix.
n = 21	A = gallery('wilk',21)	W21+, tridiagonal matrix. Eigenvalue problem.

See Also

`hadamard`	Hadamard matrix
`hilb`	Hilbert matrix
`invhilb`	Inverse of the Hilbert matrix
`magic`	Magic square
`wilkinson`	Wilkinson's eigenvalue test matrix

References

The MATLAB gallery of test matrices is based upon the work of Nicholas J. Higham at the Department of Mathematics, University of Manchester, Manchester, England. Additional detail on these matrices is documented in *The Test Matrix Toolbox for MATLAB (Version 3.0)* by N. J. Higham, September, 1995. To obtain this report in pdf format, enter the doc command at the MATLAB prompt and select the item `Related Papers` > `Test Matrix Toolbox` under the `Full Documentation Set` entry on the Help Desk main screen. This report is also available via anonymous ftp from The MathWorks at `/pub/contrib/linalg/testmatrix/testmatrix.ps` or World Wide Web (`ftp://ftp.ma.man.ac.uk/pub/narep` or `http://www.ma.man.ac.uk/MCCM/MCCM.html`). Further background may be found in the book *Accuracy and Stability of Numerical Algorithms*, Nicholas J. Higham, SIAM, 1996.

Purpose Gamma functions

Syntax

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

Definition The gamma function is defined by the integral:

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

The gamma function interpolates the factorial function. For integer `n`:

```
gamma(n+1) = n! = prod(1:n)
```

The incomplete gamma function is:

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

Description `Y = gamma(A)` returns the gamma function at the elements of `A`. `A` must be real.

`Y = gammainc(X,A)` returns the incomplete gamma function of corresponding elements of `X` and `A`. Arguments `X` and `A` must be real and the same size (or either can be scalar).

`Y = gammaln(A)` returns the logarithm of the gamma function, `gammaln(A) = log(gamma(A))`. The `gammaln` command avoids the underflow and overflow that may occur if it is computed directly using `log(gamma(A))`.

Algorithm The computations of `gamma` and `gammaln` are based on algorithms outlined in [1]. Several different minimax rational approximations are used depending upon the value of `A`. Computation of the incomplete gamma function is based on the algorithm in [2].

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

gcd

Purpose Greatest common divisor

Syntax

```
G = gcd(A,B)
[G,C,D] = gcd(A,B)
```

Description

`G = gcd(A,B)` returns an array containing the greatest common divisors of the corresponding elements of integer arrays `A` and `B`. By convention, `gcd(0,0)` returns a value of `0`; all other inputs return positive integers for `G`.

`[G,C,D] = gcd(A,B)` returns both the greatest common divisor array `G`, and the arrays `C` and `D`, which satisfy the equation: `A(i).*C(i) + B(i).*D(i) = G(i)`. These are useful for solving Diophantine equations and computing elementary Hermite transformations.

Examples

The first example involves elementary Hermite transformations.

For any two integers `a` and `b` there is a 2-by-2 matrix `E` with integer entries and determinant = `1` (a *unimodular* matrix) such that:

```
E * [a;b] = [g,0],
```

where `g` is the greatest common divisor of `a` and `b` as returned by the command `[g,c,d] = gcd(a,b)`.

The matrix `E` equals:

```
c       d
-b/g    a/g
```

In the case where `a = 2` and `b = 4`:

```
[g,c,d] = gcd(2,4)
g =
     2
c =
     1
d =
     0
```

So that:

```
E =
    1     0
   -2     1
```

In the next example, we solve for x and y in the Diophantine equation 30x + 56y = 8.

```
[g,c,d] = gcd(30,56)
g =
     2
c =
   -13
d =
     7
```

By the definition, for scalars c and d:

```
30(-13) + 56(7) = 2,
```

Multiplying through by 8/2:

```
30(-13*4) + 56(7*4) = 8
```

Comparing this to the original equation, a solution can be read by inspection:

```
x = (-13*4) = -52; y = (7*4) = 28
```

See Also

`lcm` Least common multiple

References

[1] Knuth, Donald, *The Art of Computer Programming*, Vol. 2, Addison-Wesley: Reading MA, 1973. Section 4.5.2, Algorithm X.

getfield

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`.

`f = getfield(s,{i,j},'field',{k})` returns the contents of the specified field. This is equivalent to the syntax `f = s(i,j).field(k)`. All subscripts must be passed as cell arrays—that is, they must be enclosed in curly braces (similar to`{i,j}` and `{k}` above). Pass field references as strings.

Examples

Given the structure:

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

Then the command `f = getfield(mystr,{2,1},'name')` yields

```
f =

gertrude
```

To list the contents of all `name` (or other) fields, embed `getfield` in a loop:

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

name =

    'alice'    'gertrude'
```

See Also

`fields`	Field names of a structure
`setfield`	Set field of structure array

Purpose Define global variables

Syntax

```
global X Y Z
```

Description `global X Y Z` defines `X`, `Y`, and `Z` as global in scope.

Ordinarily, each MATLAB function, defined by an M-file, has its own local variables, which are separate from those of other functions, and from those of the base workspace and nonfunction scripts. 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 initialiizied to the empty matrix. By convention, global variable names are often long with all capital letters (not required).

It is an error to declare a variable global if:

- in the current workspace, a variable with the same name exists.
- in an M-file, it has been referenced previously.

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')
```

Examples Here is the code for the functions `tic` and `toc` (some comments abridged), which 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,restart)
gmres(A,b,restart,tol)
gmres(A,b,restart,tol,maxit)
gmres(A,b,restart,tol,maxit,M)
gmres(A,b,restart,tol,maxit,M1,M2)
gmres(A,b,restart,tol,maxit,M1,M2,x0)
x = gmres(A,b,restart,tol,maxit,M1,M2,x0)
[x,flag] = gmres(A,b,restart,tol,maxit,M1,M2,x0)
[x,flag,relres] = gmres(A,b,restart,tol,maxit,M1,M2,x0)
[x,flag,relres,iter] = gmres(A,b,restart,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec] =
   gmres(A,b,restart,tol,maxit,M1,M2,x0)
```

Description

`x = gmres(A,b,restart)` attempts to solve the system of linear equations `A*x = b` for `x`. The coefficient matrix `A` must be square and the right hand side (column) vector `b` must have length `n`, where `A` is `n`-by-`n`. `gmres` will start iterating from an initial estimate that by default is an all zero vector of length `n`. `gmres` will restart itself every `restart` iterations using the last iterate from the previous outer iteration as the initial guess for the next outer iteration. Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate `x` has relative residual `norm(b-A*x)/norm(b)` less than or equal to the tolerance of the method. The default tolerance is `1e–6`. The default maximum number of iterations is the minimum of `n/restart` and 10. No preconditioning is used.

`gmres(A,b,restart,tol)` specifies the tolerance of the method, `tol`.

`gmres(A,b,restart,tol,maxit)` additionally specifies the maximum number of iterations, `maxit`.

`gmres(A,b,restart,tol,maxit,M)` and `gmres(A,b,restart,tol,maxit,M1,M2)` use left preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M1` or `M2` is given as the empty matrix (`[]`), it is considered to be the identity matrix, equivalent to no preconditioning at all. Since systems of equations of the form `M*y = r` are solved using backslash within `gmres`, it is wise to factor preconditioners into their LU factors first. For example, replace `gmres(A,b,restart,tol,maxit,M)` with:

```
[M1,M2] = lu(M);
gmres(A,b,restart,tol,maxit,M1,M2).
```

`gmres(A,b,restart,tol,maxit,M1,M2,x0)` specifies the first initial estimate `x0`. If `x0` is given as the empty matrix (`[]`), the default all zero vector is used.

`x = gmres(A,b,restart,tol,maxit,M1,M2,x0)` returns a solution `x`. If `gmres` converged, a message to that effect is displayed. If `gmres` failed to converge after the maximum number of iterations or halted 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.

`[x,flag] = gmres(A,b,restart,tol,maxit,M1,M2,x0)` returns a solution `x` and a flag which describes the convergence of `gmres`:

Flag	Convergence
0	`gmres` converged to the desired tolerance `tol` within `maxit` iterations without failing for any reason.
1	`gmres` iterated `maxit` times but did not converge.
2	One of the systems of equations of the form `M*y = r` involving the preconditioner was ill-conditioned and did not return a useable result when solved by \ (backslash).
3	The method 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,restart,tol,maxit,M1,M2,x0)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, then `relres` ≤ `tol`.

`[x,flag,relres,iter] = gmres(A,b,restart,tol,maxit,M1,M2,x0)` also returns both the outer and inner iteration numbers at which `x` was computed. The outer iteration number `iter(1)` is an integer between 0 and `maxit`. The inner iteration number `iter(2)` is an integer between `0` and `restart`.

`[x,flag,relres,iter,resvec] = gmres(A,b,restart,tol,maxit,M1,M2,x0)` also returns a vector of the residual norms at each inner iteration, starting from `resvec(1) = norm(b-A*x0)`. If `flag` is `0` and `iter = [i j]`, `resvec` is of length `(i-1)*restart+j+1` and `resvec(end)` ≤ `tol*norm(b)`.

Examples

```
load west0479
A = west0479
b = sum(A,2)
[x,flag] = gmres(A,b,5)
```

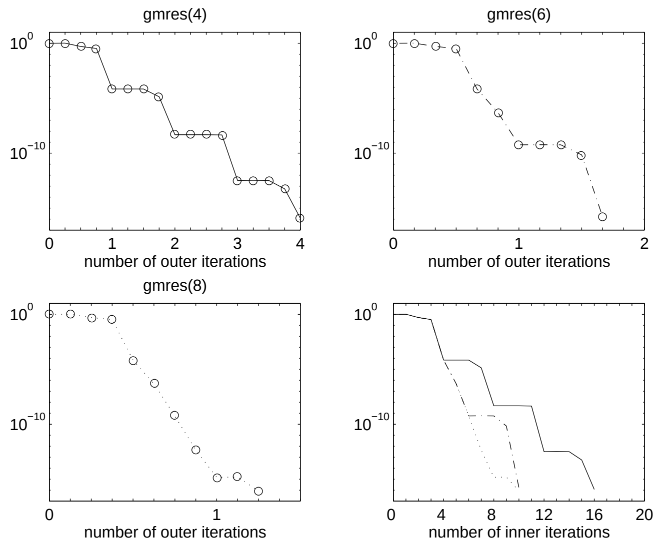
`flag` is `1` since `gmres(5)` will not converge to the default tolerance `1e-6` within the default 10 outer iterations.

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

`flag1` is `2` since the upper triangular `U1` has a zero on its diagonal so `gmres(5)` fails in the first iteration when it tries to solve a system such as `U1*y = r` for `y` with 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` since `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`	BiConjugate Gradients method
`bicgstab`	BiConjugate Gradients Stabilized method
`cgs`	Conjugate Gradients Squared method
`luinc`	Incomplete LU matrix factorizations
`pcg`	Preconditioned Conjugate Gradients method
`qmr`	Quasi-Minimal Residual method
`\`	Matrix left division

References

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

Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM, Philadelphia, 1994.

Purpose Numerical gradient

Syntax

```
FX = gradient(F)
[FX,FY] = gradient(F)
[Fx,Fy,Fz,...] = gradient(F)
[...] = gradient(F,h)
[...] = gradient(F,h1,h2,...)
```

Definition The *gradient* of a function of two variables, *F(x,y)*, is defined as:

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

and can be thought of as a collection of vectors pointing in the direction of increasing values of F. In MATLAB, numerical gradients (differences) can be computed for functions with any number of variables. For a function of *N* variables, *F(x,y,z,...)*,

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

Description `FX = gradient(F)` where `F` is a vector returns the one-dimensional numerical gradient of `F`. `FX` corresponds to $\partial F / \partial x$, the differences in the x direction.

`[FX,FY] = gradient(F)` where `F` is a matrix returns the x and y components of the two-dimensional numerical gradient. `FX` corresponds to $\partial F / \partial x$, the differences in the `x` (column) direction. `FY` corresponds to $\partial F / \partial y$, the differences in the y (row) direction. The spacing between points in each direction is assumed to be one.

`[FX,FY,FZ,...] = gradient(F)` where `F` has `N dimensions` returns the `N` components of the gradient of `F`.

There are two ways to control the spacing between values in `F`:

A single spacing value, `h`, specifies the spacing between points in every direction.

`N` spacing values (`h1,h2,...`) specify the spacing for each dimension of `F`. Scalar spacing parameters specify a constant spacing for each dimension. Vector parameters specify the coordinates of the values along corresponding dimensions of `F`. In this case, the length of the vector must match the size of the corresponding dimension.

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

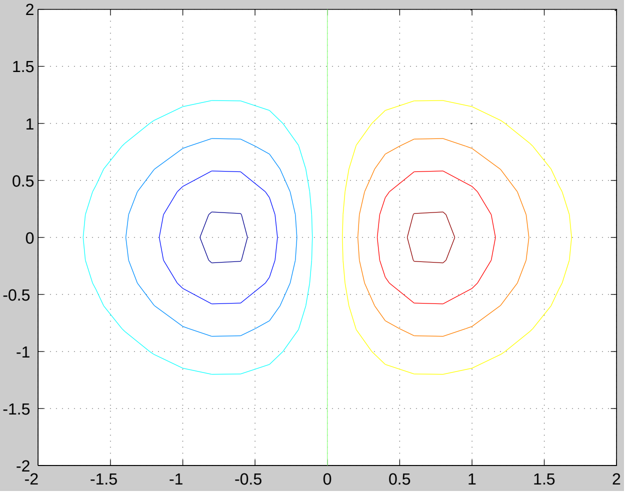
`[...] = gradient(F,h1,h2,...)` with `N` spacing parameters specifies the spacing for each dimension of `F`.

Examples

The statements

```
v = -2:0.2:2;
[x,y] = meshgrid(v);
z = x .* exp(-x.^2 - y.^2);
[px,py] = gradient(z,.2,.2);
contour(v,v,z), hold on, quiver(px,py), hold off
```

produce



Given,

`F(:,:,1) = magic(3); F(:,:,2) = pascal(3);`
`gradient(F)` takes `dx = dy = dz = 1`.
`[PX,PY,PZ] = gradient(F,0.2,0.1,0.2)` takes `dx = 0.2, dy = 0.1,` and `dz = 0.2`.

See Also

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

griddata

Purpose Data gridding

Syntax

```
ZI = griddata(x,y,z,XI,YI)
[XI,YI,ZI] = griddata(x,y,z,xi,yi)
[...] = griddata(...,method)
```

Description

`ZI = griddata(x,y,z,XI,YI)` fits a surface of the form `z = f(x,y)` to the data in the (usually) nonuniformly spaced vectors `(x,y,z)`. `griddata` interpolates this surface at the points specified by `(XI,YI)` to produce `ZI`. The surface always passes through the data points. `XI` and `YI` usually form a uniform grid (as produced by `meshgrid`).

`XI` can be a row vector, in which case it specifies a matrix with constant columns. Similarly, `YI` can be a column vector, and it specifies a matrix with constant rows.

`[XI,YI,ZI] = griddata(x,y,z,xi,yi)` returns the interpolated matrix `ZI` as above, and also returns the matrices `XI` and `YI` formed from row vector `xi` and column vector `yi`. These latter are the same as the matrices returned by `meshgrid`.

`[...] = griddata(...,`*method*`)` uses the specified interpolation method:

`'linear'`	Triangle-based linear interpolation (default)
`'cubic'`	Triangle-based cubic interpolation
`'nearest'`	Nearest neighbor interpolation
`'invdist'`	Inverse distance method

The *method* defines the type of surface fit to the data. The `'cubic'` and `'invdist'` methods produce smooth surfaces while `'linear'` and `'nearest'` have discontinuities in the first and zero'th derivatives, respectively. All the methods except `'invdist'` are based on a Delaunay triangulation of the data.

Remarks

`XI` and `YI` can be matrices, in which case `griddata` returns the values for the corresponding points `(XI(i,j),YI(i,j))`. Alternatively, you can pass in the row and column vectors `xi` and `yi`, respectively. In this case, `griddata` interprets these vectors as if they were matrices produced by the command `meshgrid(xi,yi)`.

Algorithm

The `griddata(...,'invdist')` command uses the inverse distance method of [1]. The other methods are based on Delaunay triangulation (see `delaunay`).

Examples

Sample a function at 100 random points between ±`2.0`:

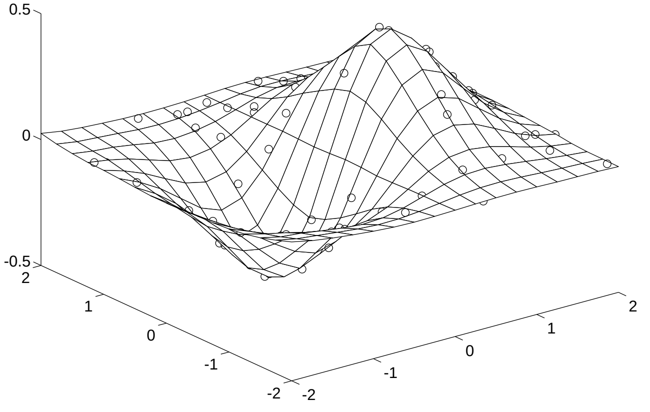
```
rand('seed',0)
x = rand(100,1)*4–2; y = rand(100,1)*4–2;
z = x.*exp(–x.^2–y.^2);
```

`x`, `y`, and `z` are now vectors containing nonuniformly sampled data. Define a regular grid, and grid the data to it:

```
ti = –2:.25:2;
[XI,YI] = meshgrid(ti,ti);
ZI = griddata(x,y,z,XI,YI);
```

Plot the gridded data along with the nonuniform data points used to generate it:

```
mesh(XI,YI,ZI), hold
plot3(x,y,z,'o'), hold off
```



See Also

`delaunay`, `interp2`, `meshgrid`

References

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

[2] Watson, David E., *Contouring: A Guide to the Analysis and Display of Spatial Data*, Tarrytown, NY: Pergamon (Elsevier Science, Inc.): 1992.

Purpose

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`	Companion matrix
`hankel`	Hankel matrix
`toeplitz`	Toeplitz matrix

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`	Hadamard matrix
`toeplitz`	Toeplitz matrix

Purpose

Online help for MATLAB functions and M-files

Syntax

```
help
help topic
```

Description

`help`, by itself, lists all primary help topics. Each main help topic corresponds to a directory name on MATLAB's search path.

`help` *topic* gives help on the specified topic. The topic can be a function name, a directory name, or a `MATLABPATH` relative partial pathname If it is a function name, `help` displays information on that function. If it is a directory name, `help` displays the contents file for the specified directory. It is not necessary to give the full pathname of the directory; the last component, or the last several components, is sufficient.

It's possible to write help text for your own M-files and toolboxes; see Remarks.

Remarks

MATLAB's Help system, like MATLAB itself, is highly extensible. This allows you to write help descriptions for your own M-files and toolboxes – using the same self-documenting method that MATLAB's M-files and toolboxes use.

The command `help`, by itself, lists all help topics by displaying the first line (the H1 line) of the contents files in each directory on MATLAB's search path. The contents files are the M-files named `Contents.m` within each directory.

The command `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.

The command `help` *topic*, where *topic* is a function name, displays help on the function by listing the first contiguous comment lines in the M-file *topic.m*.

Creating Online Help for Your Own M-Files

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). (See *Applying MATLAB* for information about creating M-files.) For example, an abridged version of the M-file `angle.m` provided with MATLAB could contain:

```
function p = angle(h)
% ANGLE Polar angle.
% ANGLE(H) returns the phase angles, in radians, of a matrix
% with complex elements. Use ABS for the magnitudes.
p = atan2(imag(h),real(h));
```

When you execute `help angle`, lines 2, 3, and 4 display. These lines are the first block of contiguous comment lines. The help system ignores comment lines that appear later in an M-file, after any executable statements, or after a blank line.

The first comment line in any M-file (the H1 line) is special. It should contain the function name and a brief description of the function. The `lookfor` command searches and displays this line, and `help` displays these lines in directories that do not contain a `Contents.m` file.

Creating Contents Files for Your Own M-File Directories

A `Contents.m` file is provided for each M-file directory included with the MATLAB software. If you create directories in which to store your own M-files, you should create `Contents.m` files for them too. To do so, simply follow the format used in an existing `Contents.m` file.

Examples

The command

```
help datafun
```

gives help on the `datafun` directory.

To prevent long descriptions from scrolling off the screen before you have time to read them, enter `more on`; then enter the `help` command.

See Also

`dir`	Directory listing
`lookfor`	Keyword search through all help entries
`more`	Control paged output for the command window
`path`	Control MATLAB's directory search path
`what`	Directory listing of M-files, MAT-files, and MEX-files
`which`	Locate functions and files

See also partialpath.

Purpose Hessenberg form of a matrix

Syntax

```
[P,H] = hess(A)
H = hess(A)
```

Description `H = hess(A)` finds `H`, the Hessenberg form of matrix `A`.

`[P,H] = hess(A)` produces a Hessenberg matrix `H` and a unitary matrix `P` so that `A = P*H*P'` and `P'*P = eye(size(A))`.

Definition A Hessenberg matrix is zero below the first subdiagonal. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original, but less computation is needed to reveal them.

Examples `H` is a 3-by-3 eigenvalue test matrix:

```
H =
   –149    –50   –154
    537    180    546
    –27     –9    –25
```

Its Hessenberg form introduces a single zero in the (3,1) position:

```
hess(H) =
   –149.0000    42.2037   –156.3165
   –537.6783   152.5511   –554.9272
           0     0.0728      2.4489
```

Algorithm For real matrices, `hess` uses the EISPACK routines `ORTRAN` and `ORTHES`. `ORTHES` converts a real general matrix to Hessenberg form using orthogonal similarity transformations. `ORTRAN` accumulates the transformations used by `ORTHES`.

When `hess` is used with a complex argument, the solution is computed using the QZ algorithm by the EISPACK routines `QZHES`. It has been modified for complex problems and to handle the special case $B = I$.

For detailed write-ups on these algorithms, see the *EISPACK Guide*.

See Also

`eig`	Eigenvalues and eigenvectors
`qz`	QZ factorization for generalized eigenvalues
`schur`	Schur decomposition

References

[1] Smith, B. T., J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide*, Lecture Notes in Computer Science, Vol. 6, second edition, Springer-Verlag, 1976.

[2] Garbow, B. S., J. M. Boyle, J. J. Dongarra, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide Extension*, Lecture Notes in Computer Science, Vol. 51, Springer-Verlag, 1977.

[3] Moler, C.B. and G. W. Stewart, "An Algorithm for Generalized Matrix Eigenvalue Problems," *SIAM J. Numer. Anal.*, Vol. 10, No. 2, April 1973.

Purpose IEEE 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. If *hex_value* is a character array, each row is interpreted as a hexadecimal string.

Examples

`hex2dec('3ff')` is 1023.

For a character array `S`

```
S =
0FF
2DE
123

hex2dec(S)
ans =
    255
    734
    291
```

See Also

`dec2hex`	Decimal to hexadecimal number conversion
`format`	Control the output display format
`hex2num`	Hexadecimal to double number conversion
`sprintf`	Write formatted data to a string

hex2num

Purpose Hexadecimal to double number conversion

Syntax

```
f = hex2num('hex_value')
```

Description

`f = hex2num('`*hex_value*`')` converts *hex_value* to the IEEE double precision floating-point number it represents. `NaN`, `Inf`, and denormalized numbers are all handled correctly. Fewer than 16 characters are padded on the right with zeros.

Examples

```
f = hex2num('400921fb54442d18')
f =
    3.14159265358979
```

Limitations

`hex2num` only works for IEEE numbers; it does not work for the floating-point representation of the VAX or other non-IEEE computers.

See Also

`format`	Control the output display format
`hex2dec`	IEEE hexadecimal to decimal number conversion
`sprintf`	Write formatted data to a string

Purpose

Hilbert matrix

Syntax

```
H = hilb(n)
```

Description

`H = hilb(n)` returns the Hilbert matrix of order `n`.

Definition

The Hilbert matrix is a notable example of a poorly conditioned matrix [1]. The elements of the Hilbert matrices are $H(i, j) = 1/(i + j - 1)$.

Examples

Even the fourth-order Hilbert matrix shows signs of poor conditioning.

```
cond(hilb(4)) =
    1.5514e+04
```

Algorithm

See the M-file for a good example of efficient MATLAB programming where conventional `for` loops are replaced by vectorized statements.

See Also

`invhilb`	Inverse of the Hilbert matrix

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

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`	Complex conjugate
`imag`	Imaginary part of a complex number
`j`	Imaginary unit
`real`	Real part of complex number

Purpose

Conditionally execute statements

Syntax

```
if expression
    statements
end
if expression1
    statements
elseif expression2
    statements
else
    statements
end
```

Description

`if` conditionally executes statements.

The simple form is:

```
if expression
    statements
end
```

More complicated forms use `else` or `elseif`. Each `if` must be paired with a matching `end`.

Arguments

expression — A MATLAB expression, usually consisting of smaller expressions or variables joined by relational operators (==, <, >, <=, >=, or ~=). Two examples are: `count < limit` and `(height - offset) >= 0`.
Expressions may also include logical functions, as in: `isreal(A)`.
Simple expressions can be combined by logical operators (&,|,~) into compound expressions such as: `(count < limit) & ((height - offset) >= 0)`.

statements — One or more MATLAB statements to be executed only if the *expression* is *true* (or nonzero). See Examples for information about how nonscalar variables are evaluated.

Examples

Here is an example showing `if`, `else`, and `elseif`:

```
for i = 1:n
    for j = 1:n
        if i == j
            a(i,j) = 2;
        elseif abs([i j]) == 1
            a(i,j) = 1;
        else
            a(i,j) = 0;
        end
    end
end
```

Such expressions are evaluated as *false* unless every element-wise comparison evaluates as *true*. Thus, given matrices `A` and `B`:

```
A =                   B =
     1     0               1     1
     2     3               3     4
```

The expression:

`A < B`	Evaluates as *false*	Since `A(1,1)` is not less than `B(1,1)`.
`A < (B+1)`	Evaluates as *true*	Since no element of `A` is greater than the corresponding element of `B`.
`A & B`	Evaluates as *false*	Since `A(1,2)	B(1,2)` is *false*.
`5 > B`	Evaluates as *true*	Since every element of `B` is less than `5`.

See Also

`break`	Break out of flow control structures
`else`	Conditionally execute statements
`end`	Terminate for, while, switch, and if statements or indicate last index
`for`	Repeat statements a specific number of times
`return`	Return to the invoking function
`switch`	Switch among several cases based on expression
`while`	Repeat statements an indefinite number of times

Purpose

Inverse one-dimensional fast Fourier transform

Syntax

```
y = ifft(X)
y = ifft(X,n)
y = ifft(X,[],dim)
y = ifft(X,n,dim)
```

Description

`y = ifft(X)` returns the inverse fast Fourier transform of vector `X`.

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

If `X` is a multidimensional array, `ifft` operates on the first non-singleton dimension.

`y = ifft(X,n)` returns the `n`-point inverse fast Fourier transform of vector `X`.

`y = ifft(X,[],`*`dim`*`)` and `y = ifft(X,n,`*`dim`*`)` return the inverse discrete Fourier transform of `X` across the dimension *`dim`*.

Examples

For any `x`, `ifft(fft(x))` equals `x` to within roundoff error. If `x` is real, `ifft(fft(x))` may have small imaginary parts.

Algorithm

The algorithm for `ifft(x)` is the same as the algorithm for `fft(x)`, except for a sign change and a scale factor of `n = length(x)`. So the execution time is fastest when `n` is a power of 2 and slowest when `n` is a large prime.

See Also

`dftmtx`, `freqz`, `specplot`, and `spectrum` in the Signal Processing Toolbox, and:

`fft`	One-dimensional fast Fourier transform
`fft2`	Two-dimensional fast Fourier transform
`fftshift`	Move zero'th lag to center of spectrum.

ifft2

Purpose

Inverse two-dimensional fast Fourier transform

Syntax

```
Y = ifft2(X)
Y = ifft2(X,m,n)
```

Description

`Y = ifft2(X)` returns the two-dimensional inverse fast Fourier transform of matrix `X`.

`Y = ifft2(X,m,n)` returns the `m`-by-`n` inverse fast Fourier transform of matrix `X`.

Examples

For any `X`, `ifft2(fft2(X))` equals `X` to within roundoff error. If `X` is real, `ifft2(fft2(X))` may have small imaginary parts.

Algorithm

The algorithm for `ifft2(X)` is the same as the algorithm for `fft2(X)`, except for a sign change and scale factors of `[m,n] = size(X)`. The execution time is fastest when `m` and `n` are powers of 2 and slowest when they are large primes.

See Also

`dftmtx`, `freqz`, `specplot`, and `spectrum` in the Signal Processing Toolbox, and:

`fft2`	Two-dimensional fast Fourier transform
`fftshift`	Move zero'th lag to center of spectrum.
`ifft`	Inverse one-dimensional fast Fourier transform

Purpose

Inverse multidimensional fast Fourier transform

Syntax

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

Description

`Y = ifftn(X)` performs the N-dimensional inverse fast Fourier transform. The result `Y` is the same size as `X`.

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

Remarks

For any `X`, `ifftn(fftn(X))` equals `X` within roundoff error. If `X` is real, `ifftn(fftn(X))` may have small imaginary parts.

Algorithm

`ifftn(X)` is equivalent to

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

This computes in-place the one-dimensional inverse fast Fourier transform along each dimension of `X`. The time required to compute `ifftn(X)` depends strongly on the number of prime factors of the dimensions of `X`. It is fastest when all of the dimensions are powers of 2.

See Also

`fft`	One-dimensional fast Fourier transform
`fft2`	Two-dimensional fast Fourier transform
`fftn`	Multidimensional fast Fourier transform

imag

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`	Complex conjugate
`i`, `j`	Imaginary unit ($\sqrt{-1}$)
`real`	Real part of complex number

Purpose

Return information about a graphics file

Synopsis

```
info = imfinfo(filename,fmt)
info = imfinfo(filename)
```

Description

`info = imfinfo(filename,fmt)` returns a structure whose fields contain information about an image in a graphics file. `filename` is a string that specifies the name of the graphics file, and `fmt` is a string that specifies the format of the file. The file must be in the current directory or in a directory on the MATLAB path. If `imfinfo` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

This table lists the possible values for `fmt`:

Format	File type
'bmp'	Windows Bitmap (BMP)
'hdf'	Hierarchical Data Format (HDF)
'jpg' or 'jpeg'	Joint Photographic Experts Group (JPEG)
'pcx'	Windows Paintbrush (PCX)
'tif' or 'tiff'	Tagged Image File Format (TIFF)
'xwd'	X Windows Dump (XWD)

If `filename` is a TIFF or HDF file containing more than one image, `info` is a structure array with one element (i.e., an individual structure) for each image in the file. For example, `info(3)` would contain information about the third image in the file.

The set of fields in `info` depends on the individual file and its format. However, the first nine fields are always the same. This table lists these fields and describes their values:

Field	Value
`Filename`	A string containing the name of the file; if the file is not in the current directory, the string contains the full pathname of the file
`FileModDate`	A string containing the date when the file was last modified
`FileSize`	An integer indicating the size of the file in bytes
`Format`	A string containing the file format, as specified by `fmt`; for JPEG and TIFF files, the three-letter variant is returned
`FormatVersion`	A string or number describing the version of the format
`Width`	An integer indicating the width of the image in pixels
`Height`	An integer indicating the height of the image in pixels
`BitDepth`	An integer indicating the number of bits per pixel
`ColorType`	A string indicating the type of image; either `'truecolor'` for a truecolor RGB image, `'grayscale'` for a grayscale intensity image, or `'indexed'` for an indexed image

`info = imfinfo(filename)` attempts to infer the format of the file from its content.

Example

```
info = imfinfo('flowers.bmp')

info =

                 Filename: 'flowers.bmp'
              FileModDate: '16-Oct-1996 11:41:38'
                 FileSize: 182078
                   Format: 'bmp'
            FormatVersion: 'Version 3 (Microsoft Windows 3.x)'
                    Width: 500
                   Height: 362
                 BitDepth: 8
                ColorType: 'indexed'
          FormatSignature: 'BM'
       NumColormapEntries: 256
                 Colormap: [256x3  double]
                  RedMask: []
                GreenMask: []
                 BlueMask: []
          ImageDataOffset: 1078
         BitmapHeaderSize: 40
                NumPlanes: 1
          CompressionType: 'none'
               BitmapSize: 181000
           HorzResolution: 0
           VertResolution: 0
            NumColorsUsed: 256
       NumImportantColors: 0
```

See Also

`imread`	Read image from graphics file
`imwrite`	Write an image to a graphics file

imread

Purpose

Read image from graphics file

Synopsis

```
A = imread(filename,fmt)
[X,map] = imread(filename,fmt)
[...] = imread(filename)
[...] = imread(...,idx)   (TIFF only)
[...] = imread(...,ref)   (HDF only)
```

Description

`A = imread(filename,fmt)` reads the image in `filename` into `A`, whose class is `uint8`. If the file contains a grayscale intensity image, `A` is a two-dimensional array. If the file contains a truecolor (RGB) image, `A` is a three-dimensional (m-by-n-by-3) array. `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 in the MATLAB path. If `imread` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

This table lists the possible values for `fmt`:

Format	File type
`'bmp'`	Windows Bitmap (BMP)
`'hdf'`	Hierarchical Data Format (HDF)
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)
`'pcx'`	Windows Paintbrush (PCX)
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF)
`'xwd'`	X Windows Dump (XWD)

`[X,map] = imread(filename,fmt)` reads the indexed image in `filename` into `X` and its associated colormap into `map`. `X` is of class `uint8`, and `map` is of class `double`. The colormap values are rescaled to the range [0, 1].

`[...] = imread(filename)` attempts to infer the format of the file from its content.

`[...] = imread(...,idx)` reads in one image from a multi-image TIFF file. `idx` is an integer value that specifies the order in which the image appears in the file. For example, if `idx` is 3, `imread` reads the third image in the file. If you omit this argument, `imread` reads the first image in the file.

`[...] = imread(...,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.) If you omit this argument, `imread` reads the first image in the file.

This table summarizes the types of images that `imread` can read:

Format	Variants
BMP	1-bit, 4-bit, 8-bit, and 24-bit uncompressed images; 4-bit and 8-bit run-length encoded (RLE) images
HDF	8-bit raster image datasets, with or without associated colormap; 24-bit raster image datasets
JPEG	Any baseline JPEG image; JPEG images with some commonly used extensions
PCX	1-bit, 8-bit, and 24-bit images
TIFF	Any baseline TIFF image, including 1-bit, 8-bit, and 24-bit uncompressed images; 1-bit, 8-bit, and 24-bit images with packbit compression; 1-bit images with CCITT compression
XWD	1-bit and 8-bit ZPixmaps; XYBitmaps; 1-bit XYPixmaps

Examples

This example reads the sixth image in a TIFF file:

```
[X,map] = imread('flowers.tif',6);
```

This example reads the fourth image in an HDF file:

```
info = imfinfo('skull.hdf');
[X,map] = imread('skull.hdf',info(4).Reference);
```

See Also

`imfinfo`	Return information about a graphics file
`imwrite`	Write an image to a graphics file

Purpose Write an image to a graphics file

Synopsis

```
imwrite(A,filename,fmt)
imwrite(X,map,filename,fmt)
imwrite(...,filename)
imwrite(...,Parameter,Value,...)
```

Description

`imwrite(A,filename,fmt)` writes the image in `A` to `filename`. `filename` is a string that specifies the name of the output file, and `fmt` is a string that specifies the format of the file. If `A` is a grayscale intensity image or a truecolor (RGB) image of class `uint8`, `imwrite` writes the actual values in the array to the file. If `A` is of class `double`, `imwrite` rescales the values in the array before writing, using `uint8(round(255*A))`. This operation converts the floating-point numbers in the range [0, 1] to 8-bit integers in the range [0, 255].

This table lists the possible values for `fmt`:

Format	File type
`'bmp'`	Windows Bitmap (BMP)
`'hdf'`	Hierarchical Data Format (HDF)
`'jpg'` or `'jpeg'`	Joint Photographers Expert Group (JPEG)
`'pcx'`	Windows Paintbrush (PCX)
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF)
`'xwd'`	X Windows Dump (XWD)

`imwrite(X,map,filename,fmt)` writes the indexed image in `X`, and its associated colormap `map`, to `filename`. If `X` is of class `uint8`, `imwrite` writes the actual values in the array to the file. If `X` is of class `double`, `imwrite` offsets the values in the array before writing, using `uint8(X–1)`. `map` must be of class `double`; `imwrite` rescales the values in `map` using `uint8(round(255*map))`.

`imwrite(...,filename)` writes the image to `filename`, inferring the format to use from the filename's extension. The extension must be one of the legal values for `fmt`.

`imwrite(...,Parameter,Value,...)` specifies parameters that control various characteristics of the output file. Parameters are currently supported for HDF, JPEG, and TIFF files.

This table describes the available parameters for HDF files:

Parameter	Values	Default
`'Compression'`	One of these strings: `'none'`, `'rle'`, `'jpeg'`	`'rle'`
`'Quality'`	A number between 0 and 100; parameter applies only if `'Compression'` is `'jpeg'`; higher numbers mean quality is better (less image degradation due to compression), but the resulting file size is larger	75
`'WriteMode'`	One of these strings: `'overwrite'`, `'append'`	`'overwrite'`

This table describes the available parameters for JPEG files:

Parameter	Values	Default
`'Quality'`	A number between 0 and 100; higher numbers mean quality is better (less image degradation due to compression), but the resulting file size is larger	75

This table describes the available parameters for TIFF files:

Parameter	Values	Default
`'Compression'`	One of these strings: `'none'`, `'packbits'`, `'ccitt'`; `'ccitt'` is valid for binary images only	`'ccitt'` for binary images; `'packbits'` for all other images
`'Description'`	Any string; fills in the `ImageDescription` field returned by `imfinfo`	empty

This table summarizes the types of images that `imwrite` can write:

Format	Variants
BMP	8-bit uncompressed images with associated colormap; 24-bit uncompressed images
HDF	8-bit raster image datasets, with or without associated colormap; 24-bit raster image datasets
JPEG	Baseline JPEG images
PCX	8-bit images
TIFF	Baseline TIFF images, including 1-bit, 8-bit, and 24-bit uncompressed images; 1-bit, 8-bit, and 24-bit images with packbit compression; 1-bit images with CCITT compression
XWD	8-bit ZPixmaps

Example

```
imwrite(X,map,'flowers.hdf','Compression','none',...
'WriteMode','append')
```

See Also

`imfinfo`	Return information about a graphics file
`imread`	Read image from graphics file

Purpose Subscripts from linear index

Syntax

```
[I,J] = ind2sub(siz,IND)
[I1,I2,I3,...,In] = ind2sub(siz,IND)
```

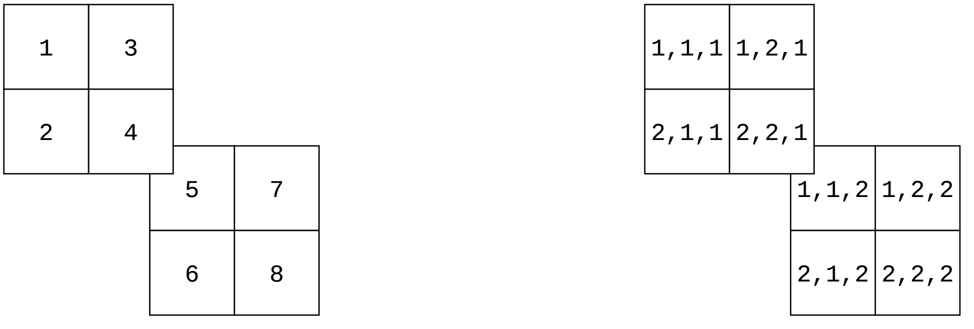
Description The `ind2sub` command determines the equivalent subscript values corresponding to a single index into an array.

`[I,J] = ind2sub(`*siz*`,IND)` returns the arrays `I` and `J` containing the equivalent row and column subscripts corresponding to the index matrix `IND` for a matrix of size *siz*.

For matrices, `[I,J] = ind2sub(size(A),find(A>5))` returns the same values as
`[I,J] = find(A>5)`.

`[I1,I2,I3,...,In] = ind2sub(`*siz*`,IND)` returns `n` subscript arrays `I1,I2,..,In` containing the equivalent multidimensional array subscripts equivalent to `IND` for an array of size *siz*.

Examples The mapping from linear indexes to subscript equivalents for a 2-by-2-by-2 array is:



See Also

`sub2ind`	Single index from subscripts
`find`	Find indices and values of nonzero elements

Inf

Purpose Infinity

Syntax `Inf`

Description `Inf` returns the IEEE arithmetic representation for positive infinity. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values.

Examples `1/0`, `1.e1000`, `2^1000`, and `exp(1000)` all produce `Inf`.

`log(0)` produces `-Inf`.

`Inf-Inf` and `Inf/Inf` both produce `NaN`, Not-a-Number.

See Also

`is*`	Detect state
`NaN`	Not-a-Number

Purpose Inferior class relationship

Syntax

```
inferiorto('class1','class2',...)
```

Description The `inferiorto` function establishes a hierarchy which determines the order in which MATLAB calls object methods.

`inferiorto('class1','class2',...)` invoked within a class constructor method (say `myclass.m`) indicates that `myclass`'s method should not be invoked if a function is called with an object of class `myclass` and one or more objects of class `class1`, `class2`, and so on.

Remarks Suppose `A` is of class `'class_a'`, `B` is of class `'class_b'` and `C` is of class `'class_c'`. Also suppose the constructor `class_c.m` contains the statement: `inferiorto('class_a')`. Then `e = fun(a,c)` or `e = fun(c,a)` invokes `class_a/fun`.

If a function is called with two objects having an unspecified relationship, the two objects are considered to have equal precedence, and the leftmost object's method is called. So, `fun(b,c)` calls `class_b/fun`, while `fun(c,b)` calls `class_c/fun`.

See Also

`superiorto`	Superior class relationship

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 there is a tie, 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*`)` returns the string that can be used to recreate the inline function object. This is the opposite of the constructor `inline`.

`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*.

Examples

Create a simple inline function to square a number:

```
g = inline('t^2')
g =
    Inline function:
    g(t) = t^2

char(g)
ans =
    inline('t^2', 't')
```

Create an inline function to compute the formula $f = 3\sin(2x^2)$:

```
g = inline('3*sin(2*x.^2)')

g =
   Inline function:
   g(x) = 3*sin(2*x.^2)

argnames(g)
ans =
   'x'

formula(g)

ans =
3*sin(2*x.^2)

g(pi)
ans =

    2.3306

g(2*pi)
ans =

   -1.2151

fmin(g,pi,2*pi)
ans =

    3.8630
```

inmem

Purpose Functions in memory

Syntax

```
M = inmem
[M,mex] = inmem
```

Description `M = inmem` returns a cell array of strings containing the names of the M-files that are in the P-code buffer.

`[M,mex] = inmem` returns a cell array containing the names of the MEX-files that have been loaded.

Examples

```
clear all % start with a clean slate
erf(.5)
M = inmem
```

lists the M-files that were required to run `erf`.

Purpose

Detect points inside a polygonal region

Syntax

`IN = inpolygon(X,Y,xv,yv)`

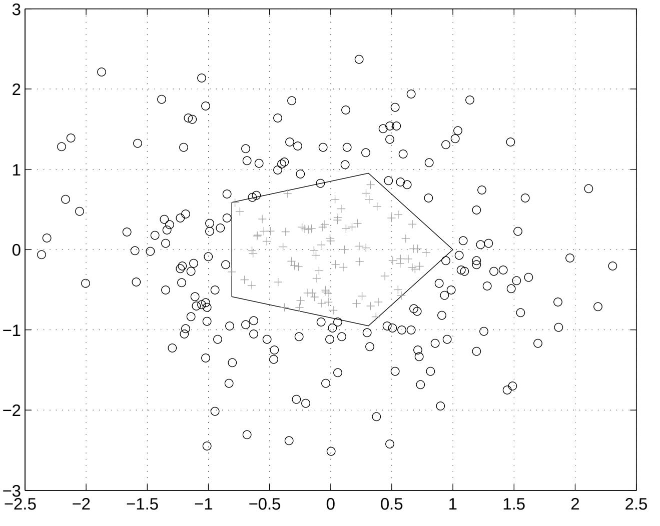
Description

`IN = inpolygon(X,Y,xv,yv)` returns a matrix `IN` the same size as `X` and `Y`. Each element of `IN` is assigned one of the values 1, 0.5 or 0, depending on whether the point `(X(p,q),Y(p,q))` is inside the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

`IN(p,q) = 1`	If `(X(p,q),Y(p,q))` is inside the polygonal region
`IN(p,q) = 0.5`	If `(X(p,q),Y(p,q))` is on the polygon boundary
`IN(p,q) = 0`	If `(X(p,q),Y(p,q))` is outside the polygonal region

Examples

```
L = linspace(0,2.*pi,6); xv = cos(L)';yv = sin(L)';
xv = [xv ; xv(1)]; yv = [yv ; yv(1)];
x = randn(250,1); y = randn(250,1);
in = inpolygon(x,y,xv,yv);
plot(xv,yv,x(in),y(in),'r+',x(~in),y(~in),'bo')
```



input

Purpose

Request user input

Syntax

```
user_entry = input('prompt')
user_entry = input('prompt','s')
```

Description

The response to the `input` prompt can be any MATLAB expression, which is evaluated using the variables in the current workspace.

`user_entry = input('`*`prompt`*`')` displays *prompt* as a prompt on the screen, waits for input from the keyboard, and returns the value entered in `user_entry`.

`user_entry = input('`*`prompt`*`','s')` returns the entered string as a text variable rather than as a variable name or numerical value.

Remarks

If you press the **Return** key without entering anything, `input` returns an empty matrix.

The text string for the prompt may contain one or more `'\n'` characters. The `'\n'` means to skip to the next line. This allows the prompt string to span several lines. To display just a backslash, use `'\\'`.

Examples

Press **Return** to select a default value by detecting an empty matrix:

```
i = input('Do you want more? Y/N [Y]: ','s');
if isempty(i)
    i = 'Y';
end
```

See Also

The `ginput` and `uicontrol` commands in the *MATLAB Graphics Guide*, and:

`keyboard`	Invoke the keyboard in an M-file
`menu`	Generate a menu of choices for user input

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`	Number of function arguments
`nargchk`	Check number of input arguments

int2str

Purpose Integer to string conversion

Syntax

```
str = int2str(N)
```

Description

`str = int2str(N)` converts an integer to a string with integer format. The input `N` can be a single integer or a vector or matrix of integers. Noninteger inputs are rounded before conversion.

Examples

`int2str(2+3)` is the string `'5'`.

One way to label a plot is

```
title(['case number ' int2str(n)])
```

For matrix or vector inputs, `int2str` returns a string matrix:

```
int2str(eye(3))

ans =

1  0  0
0  1  0
0  0  1
```

See Also

`fprintf`	Write formatted data to file
`num2str`	Number to string conversion
`sprintf`	Write formatted data to a string

Purpose

One-dimensional data interpolation (table lookup)

Syntax

```
yi = interp1(x,Y,xi)
yi = interp1(x,Y,xi,method)
```

Description

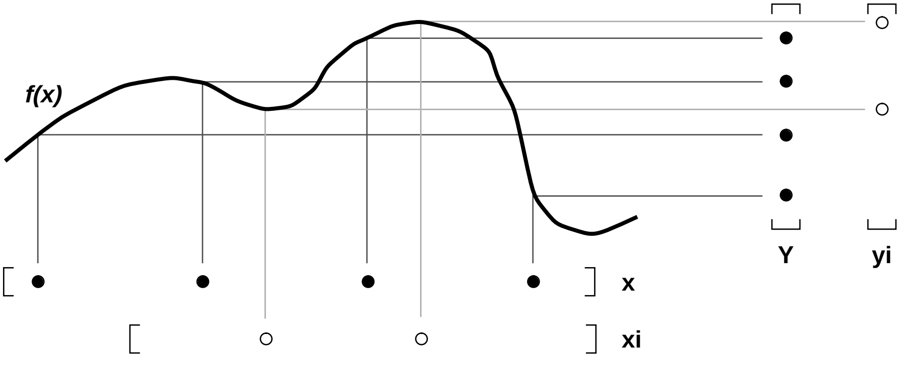
`yi = interp1(x,Y,xi)` returns vector `yi` containing elements corresponding to the elements of `xi` and determined by interpolation within vectors `x` and `Y`. The vector `x` specifies the points at which the data `Y` is given. If `Y` is a matrix, then the interpolation is performed for each column of `Y` and `yi` will be `length(xi)`-by-`size(Y,2)`. Out of range values are returned as `NaN`s.

`yi = interp1(x,Y,xi,`*`method`*`)` interpolates using alternative methods:

- `'nearest'` for nearest neighbor interpolation
- `'linear'` for linear interpolation
- `'spline'` for cubic spline interpolation
- `'cubic'` for cubic interpolation

All the interpolation methods require that `x` be monotonic. For faster interpolation when `x` is equally spaced, use the methods `'*linear'`, `'*cubic'`, `'*nearest'`, or `'*spline'`.

The `interp1` command interpolates between data points. It finds values of a one-dimensional function *f(x)* underlying the data at intermediate points. This 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 `tab = [x,y]` and `interp1` *looks up* the elements of `xi` in `x`, and, based upon their locations, returns values yi interpolated within the elements of y.

Examples

Here are two vectors representing the census years from 1900 to 1990 and the corresponding United States population in millions of people.

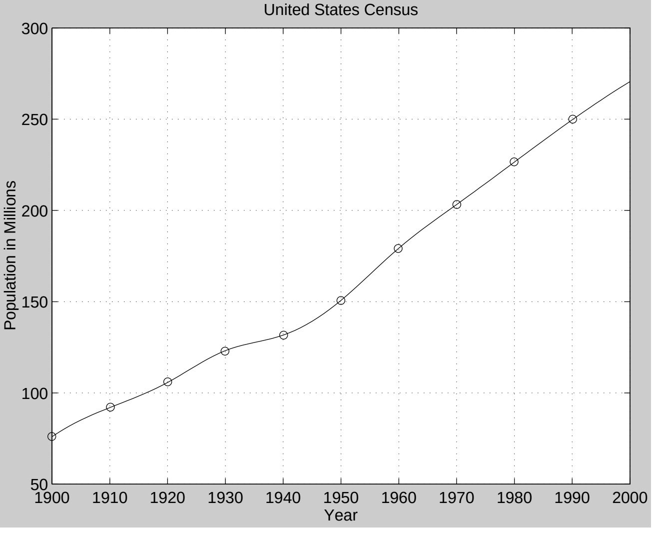
```
t = 1900:10:1990;
p = [75.995  91.972  105.711  123.203  131.669...
     150.697  179.323  203.212  226.505  249.633];
```

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'`, `'linear'` and `'cubic'` methods have fairly straightforward implementations. For the `'spline'` method, `interp1` calls a function `spline` that uses the M-files `ppval`, `mkpp`, and `unmkpp`. These routines form a small suite of functions for working with piecewise polynomials. `spline` uses them in a fairly simple fashion to perform cubic spline interpolation. For access to the more advanced features, see these M-files and the Spline Toolbox.

See Also

`interpft`	One-dimensional interpolation using the FFT method.
`interp2`	Two-dimensional data interpolation (table lookup)
`interp3`	Three-dimensional data interpolation (table lookup)
`interpn`	Multidimensional data interpolation (table lookup)
`spline`	Cubic spline interpolation

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)
```

Description

`ZI = interp2(X,Y,Z,XI,YI)` returns matrix `ZI` containing elements corresponding to the elements of `XI` and `YI` and determined by interpolation within the two-dimensional function specified by matrices `X`, `Y`, and `Z`. `X` and `Y` must be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. Matrices `X` and `Y` specify the points at which the data `Z` is given. Out of range values are returned as `NaN`s.

`XI` and `YI` can be matrices, in which case `interp2` returns the values of `Z` corresponding to the points `(XI(i,j),YI(i,j))`. Alternatively, you can pass in the row and column vectors `xi` and `yi`, respectively. In this case, `interp2` interprets these vectors as if you issued the command `meshgrid(xi,yi)`.

`ZI = interp2(Z,XI,YI)` assumes that `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`.

`ZI = interp2(Z,ntimes)` expands `Z` by interleaving interpolates between every element, working recursively for `ntimes`. `interp2(Z)` is the same as `interp2(Z,1)`.

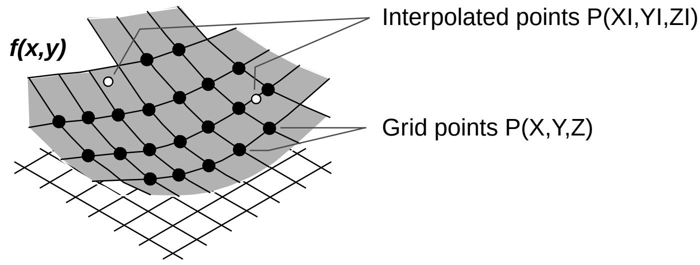
`ZI = interp2(X,Y,Z,XI,YI,`*`method`*`)` specifies an alternative interpolation method:

- `'linear'` for bilinear interpolation (default)
- `'nearest'` for nearest neighbor interpolation
- `'cubic'` for bicubic interpolation

All interpolation methods require that `X` and `Y` be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. Variable spacing is handled by mapping the given values in `X`, `Y`, `XI`, and `YI` to an equally spaced domain before interpolating. For faster interpolation when `X` and `Y` are equally spaced and monotonic, use the methods `'*linear'`, `'*cubic'`, or `'*nearest'`.

Remarks

The `interp2` command interpolates between data points. It finds values of a two-dimensional function *f(x,y)* underlying the data at intermediate points.

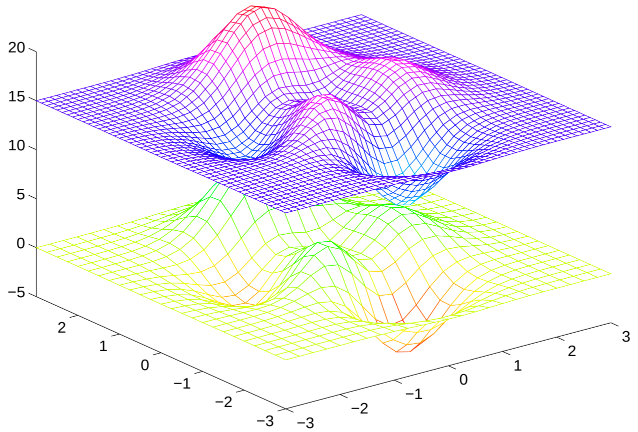


Interpolation is the same operation as table lookup. Described in table lookup terms, the table is `tab = [NaN,Y; X,Z]` and `interp2` looks up the elements of `XI` in `X`, `YI` in `Y`, and, based upon their location, returns values `ZI` interpolated within the elements of `Z`.

Examples

Interpolate the peaks function over a finer grid:

```
[X,Y] = meshgrid(-3:.25:3);
Z = peaks(X,Y);
[XI,YI] = meshgrid(-3:.125:3);
ZI = interp2(X,Y,Z,XI,YI);
mesh(X,Y,Z), hold, mesh(XI,YI,ZI+15)
hold off
axis([-3 3 -3 3 -5 20])
```



Given this set of employee data,

```
years = 1950:10:1990;
service = 10:10:30;
wage = [150.697 199.592 187.625
         179.323 195.072 250.287
         203.212 179.092 322.767
         226.505 153.706 426.730
         249.633 120.281 598.243];
```

it is possible to interpolate to find the wage earned in 1975 by an employee with 15 years' service:

```
w = interp2(service,years,wage,15,1975)
w =
  190.6287
```

See Also

`griddata`	Data gridding
`interp1`	One-dimensional data interpolation (table lookup)
`interp3`	Three-dimensional data interpolation (table lookup)
`interpn`	Multidimensional data interpolation (table lookup)
`meshgrid`	Generation of `X` and `Y` arrays for three-dimensional plots.

interp3

Purpose

Three-dimensional data interpolation (table lookup)

Syntax

```
VI = interp3(X,Y,Z,V,XI,YI,ZI)
VI = interp3(V,XI,YI,ZI)
VI = interp3(V,ntimes)
VI = interp3(...,method)
```

Description

`VI = interp3(X,Y,Z,V,XI,YI,ZI)` interpolates to find `VI`, the values of the underlying three-dimensional function `V` at the points in matrices `XI`,`YI` and `ZI`. Matrices `X`,`Y` and `Z` specify the points at which the data `V` is given. Out of range values are returned as `NaN`.

`XI`, `YI`, and `ZI` can be matrices, in which case `interp3` returns the values of `Z` corresponding to the points `(XI(i,j),YI(i,j),ZI(i,j))`. Alternatively, you can pass in the vectors `xi`, `yi`, and `zi`. Vector arguments that are not the same size are interpreted as if you called `meshgrid`.

`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,1)` is the same as `interp3(V)`.

`VI = interp3(...,`*method*`)` specifies alternative methods:

- `'linear'` for linear interpolation (default)
- `'cubic'` for cubic interpolation
- `'nearest'` for nearest neighbor interpolation

Discussion

All the interpolation methods require that `X`,`Y` and `Z` be monotonic and have the same format (“plaid”) as if they were produced by `meshgrid`. Variable spacing is handled by mapping the given values in `X`,`Y`,`Z`,`XI`,`YI` and `ZI` to an equally spaced domain before interpolating. For faster interpolation when `X`,`Y`, and `Z` are equally spaced and monotonic, use the methods '∗`linear`', '∗`cubic`', or '∗`nearest`'.

Examples

To generate a course approximation of `flow` and interpolate over a finer mesh:

```
[x,y,z,v] = flow(10);
[xi,yi,zi] = meshgrid(.1:.25:10, –3:.25:3, –3:.25:3);
vi = interp3(x,y,z,v,xi,yi,zi); % V is 31-by-41-by-27
slice(xi,yi,zi,vi,[6 9.5],2,[–2 .2]) shading flat
```

See Also

`interp1`	One-dimensional data interpolation (table lookup)
`interp2`	Two-dimensional data interpolation (table lookup)
`interpn`	Multidimensional data interpolation (table lookup).
`meshgrid`	Generate X and Y matrices for three-dimensional plots

interpft

Purpose

One-dimensional interpolation using the FFT method

Syntax

```
y = interpft(x,n)
y = interpft(x,n,dim)
```

Description

`y = interpft(x,n)` returns the vector `y` that contains the value of the periodic function `x` resampled to `n` equally spaced points.

If `length(x) = m`, and `x` has sample interval `dx`, then the new sample interval for `y` is
`dy = dx*m/n`. Note that `n` cannot be smaller than `m`.

If `X` is a matrix, `interpft` operates on the columns of `X`, returning a matrix `Y` with the same number of columns as `X`, but with `n` rows.

`y = interpft(x,n,dim)` operates along the specified dimension.

Algorithm

The `interpft` command uses the FFT method. The original vector `x` is transformed to the Fourier domain using `fft` and then transformed back with more points.

See Also

`interp1` One-dimensional data interpolation (table lookup)

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 a multidimensional `V`, you should call `interpn` with `2*N+1` arguments, where `N` is the number of dimensions in `V`. Arrays `X1`,`X2`,`X3`,... specify the points at which the data `V` is given. Out of range values are returned as `NaN`.

`Y1`, `Y2`, `Y3`,`...` can be matrices, in which case `interpn` returns the values of `VI` corresponding to the points `(Y1(i,j),Y2(i,j),Y3(i,j),...)`. Alternatively, you can pass in the vectors `y1`, `y2`, `y3`,... In this case, `interpn` interprets these vectors as if you issued the command `ndgrid(y1,y2,y3,...)`.

`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)`, and so on.

`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'` for linear interpolation (default)
- `'cubic'` for cubic interpolation
- `'nearest'` for nearest neighbor interpolation

Discussion

All the interpolation methods require that `X`,`Y` and `Z` be monotonic and have the same format (“plaid”) as if they were produced by `ndgrid`. Variable spacing is handled by mapping the given values in `X1`,`X2`,`X3`,... and `Y1`,`Y2`,`Y3`,... to an equally spaced domain before interpolating. For faster interpolation when `X1`,`X2`,`Y3`, and so on are equally spaced and monotonic, use the methods '*`linear`', '*`cubic`', or '*`nearest`'.

See Also

`interp1`	One-dimensional data interpolation (table lookup)
`interp2`	Two-dimensional data interpolation (table lookup)
`ndgrid`	Generate arrays for multidimensional functions and interpolation

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 $a \cap b$. Non-vector input arrays are regarded as column vectors `a = A(:)` or `b = B(:)`.

`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`	True for a set member
`setdiff`	Return the set difference of two vectors
`setxor`	Set exclusive-or of two vectors
`union`	Set union of two vectors
`unique`	Unique elements of a vector

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 matrix `A` of order 100 has been 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 100 and the right-hand side is `b = A*x`. Thus the system of linear equations is badly conditioned, but consistent.

On a 20 MHz 386SX notebook computer, the statements

```
tic, y = inv(A)*b, toc
err = norm(y–x)
res = norm(A*y–b)
```

produce

```
elapsed_time =
    9.6600
err =
    2.4321e–07
res =
    1.8500e–09
```

while the statements

```
tic, z = A\b, toc
err = norm(z–x)
res = norm(A*z–b)
```

produce

```
elapsed_time =
    3.9500
err =
    6.6161e–08
res =
    9.1103e–16
```

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–7`, 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

The `inv` command uses the subroutines `ZGEDI` and `ZGEFA` from LINPACK. For more information, see the *LINPACK Users' Guide*.

Diagnostics

From `inv`, if the matrix is singular,

```
Matrix is singular to working precision.
```

On machines with IEEE arithmetic, this is only a warning message. `inv` then returns a matrix with each element set to `Inf`. On machines without IEEE arithmetic, like the VAX, this is treated as an error.

If the inverse was found, but is not reliable, this message is displayed.

```
Warning: Matrix is close to singular or badly scaled.
    Results may be inaccurate. RCOND = xxx
```

See Also

`\`	Matrix left division (backslash)
`/`	Matrix right division (slash)
`det`	Matrix determinant
`lu`	LU matrix factorization
`rref`	Reduced row echelon form

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

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`	Hilbert matrix

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

ipermute

Purpose Inverse permute the dimensions of a multidimensional array

Syntax `A = ipermute(B,order)`

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

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

Examples Consider the 2-by-2-by-3 array `a`:

```
a = cat(3,eye(2),2*eye(2),3*eye(2))

a(:,:,1) =                a(:,:,2) =
     1     0                   2     0
     0     1                   0     2

a(:,:,3) =
     3     0
     0     3
```

Permuting and inverse permuting `a` in the same fashion restores the array to its original form:

```
B = permute(a,[3 2 1]);
C = ipermute(B,[3 2 1]);
isequal(a,C)
ans=

   1
```

See Also `permute` Rearrange the dimensions of a multidimensional array

Purpose Detect state

Syntax

```
k = iscell(C)
k = iscellstr(S)
k = ischar(S)
k = isempty(A)
k = isequal(A,B,...)
k = isfield(S,'field')
TF = isfinite(A)
k = isglobal(NAME)
TF = ishandle(H)
k = ishold
k = isieee
TF = isinf(A)
TF = isletter('str')
k = islogical(A)
TF = isnan(A)
k = isnumeric(A)
k = isobject(A)
k = isppc
TF = isprime(A)
k = isreal(A)
TF = isspace('str')
k = issparse(S)
k = isstruct(S)
k = isstudent
k = isunixt
k = isvms
```

Description

`k = iscell(C)` returns logical true (`1`) if `C` is a cell array and logical false (`0`) otherwise.

`k = iscellstr(S)` returns logical true (`1`) if `S` is a cell array of strings and logical false (`0`) otherwise. A cell array of strings is a cell array where every element is a character array.

`k = ischar(S)` returns logical true (`1`) if `S` is a character array and logical false (`0`) otherwise.

`k = isempty(A)` returns logical true (`1`) if `A` is an empty array and logical false (`0`) otherwise. An empty array has at least one dimension of size zero, for example, 0-by-0 or 0-by-5.

`k = isequal(A,B,...)` returns logical true (`1`) if the input arrays are the same type and size and hold the same contents, and logical false (`0`) otherwise.

`k = isfield(S,'`*`field`*`')` returns logical true (`1`) if *field* is the name of a field in the structure array `S`.

`TF = isfinite(A)` returns an array the same size as `A` containing logical true (`1`) where the elements of the array `A` are finite and logical false (`0`) where they are infinite or `NaN`.

For any `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

`k = isglobal(NAME)` returns logical true (1) if `NAME` has been declared to be a global variable, and logical false (0) if it has not been so declared.

`TF = ishandle(H)` returns an array the same size as `H` that contains logical true (1) where the elements of `H` are valid graphics handles and logical false (0)where they are not.

`k = ishold` returns logical true (1) if `hold` is `on`, and logical false (0) if it is `off`. When `hold` is `on`, the current plot and all axis properties are held so that subsequent graphing commands add to the existing graph. `hold on` means the `NextPlot` property of both figure and axes is set to `add`.

`k = isieee` returns logical true (1) on machines with IEEE arithmetic (e.g., IBM PC, most UNIX workstations, Macintosh) and logical false (0) on machines without IEEE arithmetic (e.g., VAX, Cray).

`TF = isinf(A)` returns an array the same size as `A` containing logical true (1) where the elements of `A` are `+Inf` or `–Inf` and logical false (0) where they are not.

`TF = isletter('`*str*`')` returns an array the same size as `'`*str*`'` containing logical true (1) where the elements of *str* are letters of the alphabet and logical false (0) where they are not.

`k = islogical(A)` returns logical true (1) if `A` is a logical array and logical false (0) otherwise.

`TF = isnan(A)` returns an array the same size as `A` containing logical true (1) where the elements of `A` are `NaN`s and logical false (0) where they are not.

`k = isnumeric(A)` returns logical true (1) if A is a numeric array and logical false (0) otherwise. For example, sparse arrays, and double precision arrays are numeric while strings, cell arrays, and structure arrays are not.

`k = isobject(A)` returns logical true (1) if A is an object and logical false (0) otherwise.

`k = isppc` returns logical true (`1`) if the computer running MATLAB is a Macintosh Power PC and logical false (`0`) otherwise.

`TF = isprime(A)` returns an array the same size as `A` containing logical true (`1`) for the elements of `A` which are prime, and logical false (`0`) otherwise.

`k = isreal(A)` returns logical true (`1`) if all elements of `A` are real numbers, and logical false (`0`) if either `A` is not a numeric array, or if any element of `A` has a nonzero imaginary component. Since strings are a subclass of numeric arrays, `isreal` always returns `1` for a string input.

Because MATLAB supports complex arithmetic, certain of its functions can introduce significant imaginary components during the course of calculations that appear to be limited to real numbers. Thus, you should use `isreal` with discretion.

`TF = isspace('`*str*`')` returns an array the same size as `'`*str*`'` containing logical true (`1`) where the elements of *str* are ASCII white spaces and logical false (`0`) where they are not. White spaces in ASCII are space, newline, carriage return, tab, vertical tab, or formfeed characters.

`k = issparse(S)` returns logical true (`1`) if the storage class of `S` is sparse and logical false (`0`) otherwise.

`k = isstruct(S)` returns logical true (`1`) if `S` is a structure and logical false (`0`) otherwise.

`k = isstudent` returns logical true (`1`) for student editions of MATLAB and logical false (`0`) for commercial editions.

`k = isunix` returns logical true (`1`) for UNIX versions of MATLAB and logical false (`0`) otherwise.

`k = isvms` returns logical true (`1`) for VMS versions of MATLAB and logical false (`0`) otherwise.

Examples

```
s = 'A1,B2,C3';

isletter(s)
ans =
     1   0   0   1   0   0   1   0

B = rand(2,2,2);
B(:,:,:) = [];

isempty(B)
ans =
     1
```

Given,

```
A =            B =              C =
     1     0      1     0     1     0
     0     1      0     1     0     0
```

`isequal(A,B,C)` returns `0`, and `isequal(A,B)` returns `1`.

Let

```
a = [-2  -1  0  1  2]
```

Then

```
isfinite(1./a) =  [1  1  0  1  1]
isinf(1./a)  =  [0  0  1  0  0]
isnan(1./a)  =  [0  0  0  0  0]
```

and

```
isfinite(0./a) =  [1  1  0  1  1]
isinf(0./a)  =  [0  0  0  0  0]
isnan(0./a)  =  [0  0  1  0  0]
```

Purpose Detect an object of a given 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 *class_name* is the name of a user-defined or pre-defined class of objects. Predefined MATLAB classes include:

`cell`	Multidimensional cell array
`double`	Multidimensional double precision array
`sparse`	Two-dimensional real (or complex) sparse array
`char`	Array of alphanumeric characters
`struct`	Structure
`'`*`class_name`*`'`	User-defined object class

Examples

`isa(rand(3,4),'double')` returns `1`.

See Also

`class`	Create object or return class of object

ismember

Purpose

Detect members of a set

Syntax

```
k = ismember(a,S)
k = ismember(A,S,'rows')
```

Description

`k = ismember(a,S)` returns an vector the same length as `a` containing logical true (`1`) where the elements of `a` are in the set `S`, and logical false (`0`) elsewhere. In set theoretic terms, `k` is 1 where $a \in S$.

`k = ismember(A,S,'rows')` when `A` and `S` are matrices with the same number of columns returns a vector containing `1` where the rows of `A` are also rows of `S` and `0` otherwise.

Examples

```
set = [0 2 4 6 8 10 12 14 16 18 20];
a = reshape(1:5,[5 1])

a =

     1
     2
     3
     4
     5
ismember(a,set)

ans =

     0
     1
     0
     1
     0
```

See Also

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

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`	Complex conjugate
`i`	Imaginary unit
`imag`	Imaginary part of a complex number
`real`	Real part of complex number

keyboard

Purpose Invoke the keyboard in an M-file

Syntax

```
keyboard
```

Description `keyboard` , when placed in an M-file, stops execution of the file and gives control to the keyboard. The special status is indicated by a `K` appearing before the prompt. You can examine or change variables; all MATLAB commands are valid. This keyboard mode is useful for debugging your M-files.

To terminate the keyboard mode, type the command:

```
return
```

then press the **Return** key.

See Also

`dbstop`	Set breakpoints in an M-file function
`input`	Request user input
`quit`	Terminate MATLAB
`return`	Terminate keyboard mode

Purpose

Kronecker tensor product

Syntax

```
K = kron(X,Y)
```

Description

`K = kron(X,Y)` returns the Kronecker tensor product of `X` and `Y`. The result is a large array formed by taking all possible products between the elements of `X` and those of `Y`. If `X` is `m`-by-`n` and `Y` is `p`-by-`q`, then `kron(X,Y)` is `m*p`-by-`n*q`.

Examples

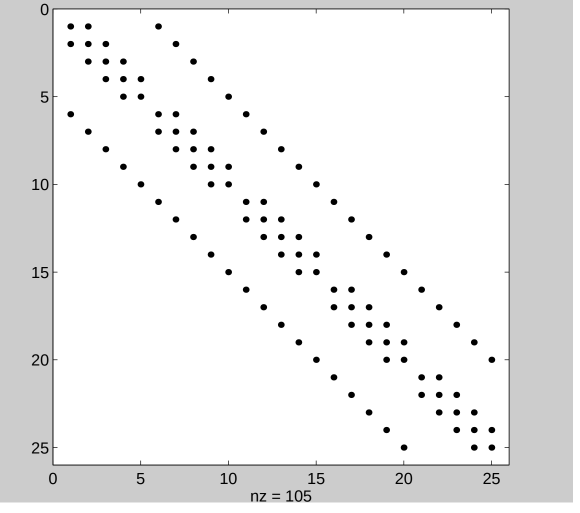
If `X` is 2-by-3, then `kron(X,Y)` is

```
[ X(1,1)*Y X(1,2)*Y X(1,3)*Y
  X(2,1)*Y X(2,2)*Y X(2,3)*Y ]
```

The matrix representation of the discrete Laplacian operator on a two-dimensional, `n`-by-`n` grid is a `n^2`-by-`n^2` sparse matrix. There are at most five nonzero elements in each row or column. The matrix can be generated as the Kronecker product of one-dimensional difference operators with these statements:

```
I = speye(n,n);
E = sparse(2:n,1:n–1,1,n,n);
D = E+E'–2*I;
A = kron(D,I)+kron(I,D);
```

Plotting this with the spy function for `n = 5` yields:



lasterr

Purpose

Last error message

Syntax

```
str = lasterr
lasterr('')
```

Description

`str = lasterr` returns the last error message generated by MATLAB.

`lasterr('')` resets `lasterr` so it returns an empty matrix until the next error occurs.

Examples

Here is a function that examines the `lasterr` string and displays its own message based on the error that last occurred. This example deals with two cases, each of which is an error that can result from a matrix multiply.

```
function catch
l = lasterr;
j = findstr(l,'Inner matrix dimensions');
if j~=[]
    disp('Wrong dimensions for matrix multiply')
else
    k = findstr(l,'Undefined function or variable')
    if (k~=[])
        disp('At least one operand does not exist')
    end
end
```

The `lasterr` function is useful in conjunction with the two-argument form of the `eval` function:

`eval('`*str*`','`*catchstr*`')`

where *catchstr* examines the `lasterr` string to determine the cause of the error and take appropriate action. The `eval` function evaluates the string `str` and returns if no error occurs. If an error occurs, `eval` executes *catchstr*. Using `eval` with the `catch` function above:

```
clear
A = [1  2  3; 6  7  2; 0  -1  5];
B = [9  5  6; 0  4  9];
eval('A*B','catch')
```

MATLAB responds with `Wrong dimensions for matrix multiply`.

See Also

`error`	Display error messages
`eval`	Interpret strings containing MATLAB expressions

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` Greatest common divisor

Purpose

Associated Legendre functions

Syntax

```
P = legendre(n,X)
S = legendre(n,X,'sch')
```

Definition

The Legendre functions are defined by:

$$P_n^m(x) = (-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}P_n(x)$$

where $P_n(x)$ is the Legendre polynomial of degree n:

$$P_n(x) = \frac{1}{2^n n!}\left[\frac{d^n}{dx}(x^2-1)^n\right]$$

The Schmidt seminormalized associated Legendre functions are related to the nonnormalized associated Legendre functions $P_n^m(x)$ by:

$$S_n^m(x) = \sqrt{\frac{2(n-m)!}{(n+m)!}}P_n^m(x)$$

Description

`P = legendre(n,X)` computes the associated Legendre functions of degree `n` and order `m = 0,1,...,n`, evaluated at `X`. Argument `n` must be a scalar integer less than 256, and `X` must contain real values in the domain $-1 \le x \le 1$.

The returned array `P` has one more dimension than `X`, and each element `P(m+1,d1,d2...)` contains the associated Legendre function of degree `n` and order `m` evaluated at `X(d1,d2...)`.

If `X` is a vector, then `P` is a matrix of the form:

$$\begin{matrix} P_2^0(x(1)) & P_2^0(x(2)) & P_2^0(x(3)) & \ldots \\ P_2^1(x(1)) & P_2^1(x(2)) & P_2^1(x(3)) & \ldots \\ P_2^2(x(1)) & P_2^2(x(2)) & P_2^2(x(3)) & \ldots \end{matrix}$$

`S = legendre(...,'sch')` computes the Schmidt seminormalized associated Legendre functions $S_n^m(x)$.

Examples

The statement `legendre(2,0:0.1:0.2)` returns the matrix:

	x = 0	**x = 0.1**	**x = 0.2**
m = 0	0.5000	0.4850	0.4400
m = 1	0	0.2985	0.5879
m = 2	3.0000	2.9700	2.8800

Note that this matrix is of the form shown at the bottom of the previous page.

Given,

```
X = rand(2,4,5); N = 2;
P = legendre(N,X)
```

Then `size(P)` is 3-by-2-by-4-by-5, and `P(:,1,2,3)` is the same as `legendre(n,X(1,2,3))`.

Purpose Length of vector

Syntax

```
n = length(X)
```

Description The statement `length(X)` is equivalent to `max(size(X))` for nonempty arrays and `0` for empty arrays.

`n = length(X)` returns the size of the longest dimension of `X`. If `X` is a vector, this is the same as its length.

Examples

```
x = ones(1,8);
n = length(x)
n =
     8

x = rand(2,10,3);
n = length(x)
n =
    10
```

See Also

`ndims`	Number of array dimensions
`size`	Array dimensions

lin2mu

Purpose Linear to mu-law conversion.

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 $0 \le mu \le 255$.

See Also

`auwrite`	Write NeXT/SUN (.au) sound file
`mu2lin`	Linear to mu-law conversion

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 `a` and `b`.

`y = linspace(a,b,n)` generates `n` points.

See Also

`:` (Colon)	Create vectors, matrix subscripting, and `for` iterations
`logspace`	Generate logarithmically spaced vectors

load

Purpose Retrieve variables from disk

Syntax

```
load
load filename
load (filename)
load filename.ext
load filename -ascii
load filename -mat
```

Description The `load` and `save` commands retrieve and store MATLAB variables on disk.

`load` by itself, loads all the variables saved in the file `'matlab.mat'`.

`load` *filename* retrieves the variables from `'filename.mat'` given a full pathname or a `MATLABPATH` relative partial pathname .

`load` *(filename)* loads a file whose name is stored in *filename*. The statements:

```
str = 'filename.mat'; load (str)
```

retrieve the variables from the binary file `'filename.mat'`.

`load filename.ext` reads ASCII files that contain rows of space separated values. The resulting data is placed into an variable with the same name as the file (without the extension). ASCII files may contain MATLAB comments (lines that begin with `%`).

`load filename -ascii` or `load filename -mat` can be used to force `load` to treat the file as either an ASCII file or a MAT file.

Remarks MAT-files are double-precision binary MATLAB format files created by the `save` command and readable by the `load` command. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the disparate formats allow. They can also be manipulated by other programs, external to MATLAB.

Algorithm

The *Application Program Interface Guide* discusses the structure of MAT-files in detail. The Application Program Interface Libraries contain C and Fortran callable routines to read and write MAT-files from external programs.

See Also

`fprintf`	Write formatted data to file
`fscanf`	Read formatted data from file
`save`	Save workspace variables on disk
`spconvert`	Import matrix from sparse matrix external format

See also partialpath.

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`	Exponential
`log10`	Common (base 10) logarithm
`log2`	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
`logm`	Matrix logarithm

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`	Natural logarithm
`pow2`	Base 2 power and scale floating-point numbers

log10

Purpose Common (base 10) logarithm

Syntax

```
Y = log10(X)
```

Description The `log10` function operates element-by-element on arrays. Its domain includes complex numbers, which may lead to unexpected results if used unintentionally.

`Y = log10(X)` returns the base 10 logarithm of the elements of `X`.

Examples On a computer with IEEE arithmetic

`log10(realmax)` is `308.2547`

and

`log10(eps)` is `–15.6536`

See Also

`exp`	Exponential
`log`	Natural logarithm
`log2`	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
`logm`	Matrix logarithm

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.The array `K` is the same size as `A` and is displayed using `1` where corresponding elements of `A` are nonzero, and `0` where corresponding elements of `A` are zero.

Remarks Logical arrays are also created by the relational operators (==,<,>,~, etc.) and functions like `any`, `all`, `isnan`, `isinf`, and `isfinite`.

Examples Given `A = [1 2 3; 4 5 6; 7 8 9]`, the statement `B = logical(eye(3))` returns a logical array

```
B =
   1   0   0
   0   1   0
   0   0   1
```

which can be used in logical indexing that returns `A`'s diagonal elements:

```
A(B)

ans =
     1
     5
     9
```

However, attempting to index into `A` using the *numeric* array `eye(3)` results in:

```
A(eye(3))
??? Index into matrix is negative or zero.
```

logm

Purpose

Matrix logarithm

Syntax

```
Y = logm(X)
[Y,esterr] = logm(X)
```

Description

`Y = logm(X)` returns the matrix logarithm: the inverse function of `expm(X)`. Complex results are produced if `X` has negative eigenvalues. A warning message is printed if the computed `expm(Y)` is not close to `X`.

`[Y,esterr] = logm(X)` does not print any warning message, but returns an estimate of the relative residual, `norm(expm(Y)-X)/norm(X)`.

Remarks

If `X` is real symmetric or complex Hermitian, then so is `logm(X)`.

Some matrices, like `X = [0 1; 0 0]`, do not have any logarithms, real or complex, and `logm` cannot be expected to produce one.

Limitations

For most matrices:

```
logm(expm(X)) = X = expm(logm(X))
```

These identities may fail for some `X`. For example, if the computed eigenvalues of `X` include an exact zero, then `logm(X)` generates infinity. Or, if the elements of `X` are too large, `expm(X)` may overflow.

Examples

Suppose `A` is the 3-by-3 matrix

```
          1         1         0
          0         0         2
          0         0        -1
```

and `X = expm(A)` is

```
X =

    2.7183    1.7183    1.0862
         0    1.0000    1.2642
         0         0    0.3679
```

Then `A = logm(X)` produces the original matrix `A`.

```
A =
```

```
   1.0000    1.0000    0.0000
        0         0    2.0000
        0         0   -1.0000
```

But `log(X)` involves taking the logarithm of zero, and so produces

```
ans =

   1.0000    0.5413    0.0826
     -Inf         0    0.2345
     -Inf      -Inf   -1.0000
```

Algorithm

The matrix functions are evaluated using an algorithm due to Parlett, which is described in [1]. The algorithm uses the Schur factorization of the matrix and may give poor results or break down completely when the matrix has repeated eigenvalues. A warning message is printed when the results may be inaccurate.

See Also

`expm`	Matrix exponential
`funm`	Evaluate functions of a matrix
`sqrtm`	Matrix square root

References

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

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

logspace

Purpose Generate logarithmically spaced vectors

Syntax

```
y = logspace(a,b)
y = logspace(a,b,n)
y = logspace(a,pi)
```

Description The `logspace` function generates logarithmically spaced vectors. Especially useful for creating frequency vectors, it is a logarithmic equivalent of `linspace` and the ":" or colon operator.

`y = logspace(a,b)` generates a row vector `y` of 50 logarithmically spaced points between decades `10^a` and `10^b`.

`y = logspace(a,b,n)` generates `n` points between decades `10^a` and `10^b`.

`y = logspace(a,pi)` generates the points between `10^a` and `pi`, which is useful for digital signal processing where frequencies over this interval go around the unit circle.

Remarks All the arguments to `logspace` must be scalars.

See Also

`:` (Colon)	Create vectors, matrix subscripting, and `for` iterations
`linspace`	Generate linearly spaced vectors

Purpose

Keyword search through all help entries

Syntax

```
lookfor topic
lookfor topic -all
```

Description

`lookfor topic` searches for the string *topic* in the first comment line (the H1 line) of the help text in all M-files found on MATLAB's search path. For all files in which a match occurs, `lookfor` displays the H1 line.

`lookfor topic -all` searches the entire first comment block of an M-file looking for *topic*.

Examples

For example

```
lookfor inverse
```

finds at least a dozen matches, including H1 lines containing "inverse hyperbolic cosine," "two-dimensional inverse FFT," and "pseudoinverse." Contrast this with

```
which inverse
```

or

```
what inverse
```

These commands run more quickly, but probably fail to find anything because MATLAB does not ordinarily 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.

See Also

`dir`	Directory listing
`help`	Online help for MATLAB functions and M-files
`what`	Directory listing of M-files, MAT-files, and MEX-files
`which`	Locate functions and files
`who`	List directory of variables in memory

lower

Purpose Convert string to lower case

Syntax

```
t = lower('str')
```

Description `t = lower('str')` returns the string formed by converting any upper-case characters in *str* to the corresponding lower-case characters and leaving all other characters unchanged.

Examples `lower('MathWorks')` is `mathworks`.

Remarks Character sets supported:

- Mac: Standard Roman
- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

See Also

`upper`	Convert string to upper case

Purpose Least squares solution in the presence of known covariance

Syntax

```
x = lscov(A,b,V)
[x,dx] = lscov(A,b,V)
```

Description `x = lscov(A,b,V)` returns the vector `x` that solves `A*x = b + e` where `e` is normally distributed with zero mean and covariance `V`. Matrix `A` must be `m`-by-`n` where `m > n`. This is the over-determined least squares problem with covariance `V`. The solution is found without inverting `V`.

`[x,dx] = lscov(A,b,V)` returns the standard errors of `x` in `dx`. The standard statistical formula for the standard error of the coefficients is:

```
mse = B'*(inv(V)-inv(V)*A*inv(A'*inv(V)*A)*A'*inv(V))*B./(m-n)
dx = sqrt(diag(inv(A'*inv(V)*A)*mse))
```

Algorithm The vector `x` minimizes the quantity `(A*x-b)'*inv(V)*(A*x-b)`. The classical linear algebra solution to this problem is

```
x = inv(A'*inv(V)*A)*A'*inv(V)*b
```

but the `lscov` function instead computes the QR decomposition of `A` and then modifies `Q` by `V`.

See Also

`\`	Matrix left division (backslash)
`nnls`	Nonnegative least squares
`qr`	Orthogonal-triangular decomposition

Reference Strang, G., *Introduction to Applied Mathematics*, Wellesley-Cambridge, 1986, p. 398.

Purpose LU matrix factorization

Syntax

```
[L,U] = lu(X)
[L,U,P] = lu(X)
lu(X)
```

Description The `lu` function expresses any square matrix `X` as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The factorization is often called the *LU*, or sometimes the *LR*, factorization.

`[L,U] = lu(X)` returns an upper triangular matrix in `U` and a psychologically lower triangular matrix (i.e., a product of lower triangular and permutation matrices) in `L`, so that `X = L*U`.

`[L,U,P] = lu(X)` returns an upper triangular matrix in `U`, a lower triangular matrix in `L`, and a permutation matrix in `P`, so that `L*U = P*X`.

`lu(X)` returns the output from the LINPACK routine `ZGEFA`.

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

`L`	A factor of `X`. Depending on the form of the function, `L` is either lower triangular, or else the product of a lower triangular matrix with a permutation matrix `P`.
`U`	An upper triangular matrix that is a factor of `X`.
`P`	The permutation matrix satisfying the equation `L*U = P*X`.

Examples Start with

```
A =
     1     2     3
     4     5     6
     7     8     0
```

To see the LU factorization, call `lu` with two output arguments:

```
[L,U] = lu(A)
L =
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
    1.0000         0         0
U =
    7.0000    8.0000    0.0000
         0    0.8571    3.0000
         0         0    4.5000
```

Notice that `L` is a permutation of a lower triangular matrix that has `1`'s on the permuted diagonal, and that `U` is upper triangular. To check that the factorization does its job, compute the product:

```
L*U
```

which returns the original `A`. Using three arguments on the left-hand side to get the permutation matrix as well

```
[L,U,P] = lu(A)
```

returns the same value of `U`, but `L` is reordered:

```
L =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000

U =

    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000

P =

    0    0    1
    1    0    0
    0    1    0
```

To verify that `L*U` is a permuted version of `A`, compute `L*U` and subtract it from `P*A`:

```
P*A - L*U
```

The inverse of the example matrix, `X = inv(A)`, is actually computed from the inverses of the triangular factors:

```
X = inv(U)*inv(L)
```

The determinant of the example matrix is

```
d = det(A)
```

which gives

```
d =
    27
```

It is computed from the determinants of the triangular factors:

```
d = det(L)*det(U)
```

The solution to $Ax = b$ is obtained with matrix division:

```
x = A\b
```

The solution is actually computed by solving two triangular systems:

```
y = L\b, x = U\y
```

Algorithm

`lu` uses the subroutines `ZGEDI` and `ZGEFA` from LINPACK. For more information, see the *LINPACK Users' Guide*.

See Also

`\`	Matrix left division (backslash)
`/`	Matrix right division (slash)
`cond`	Condition number with respect to inversion
`det`	Matrix determinant
`inv`	Matrix inverse
`qr`	Orthogonal-triangular decomposition
`rref`	Reduced row echelon form

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

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 permutated `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 1's 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.

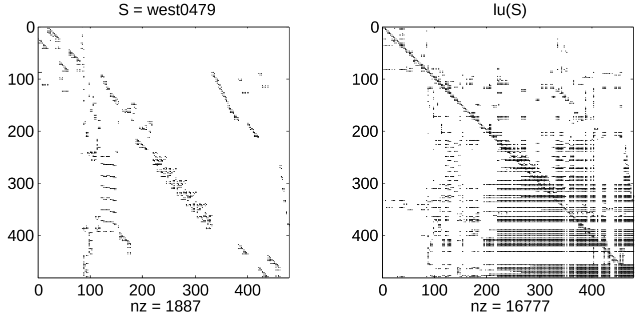
Limitations

`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);
```

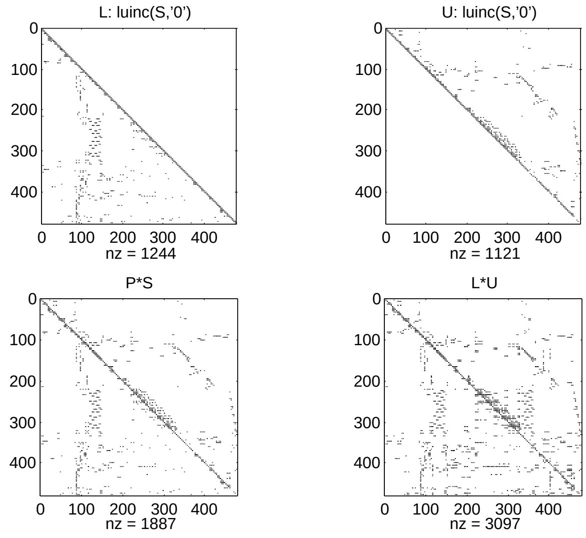


Compute the incomplete LU factorization of level 0.

```
[L,U,P] = luinc(S,'0');
D = (L*U).*spones(P*S)-P*S;
```

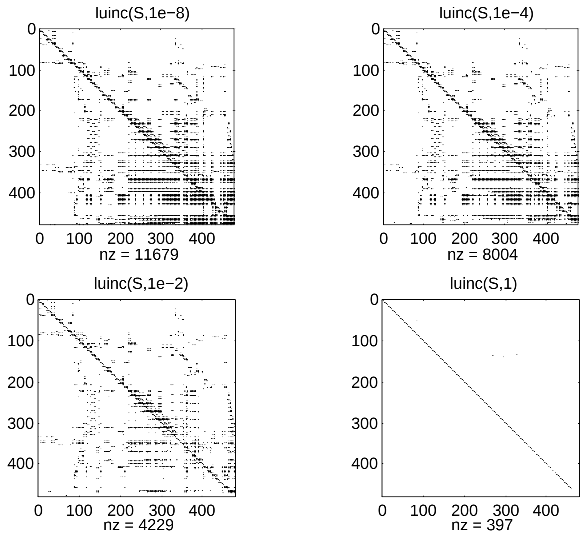
`spones(U)` and `spones(triu(P*S))` are identical.

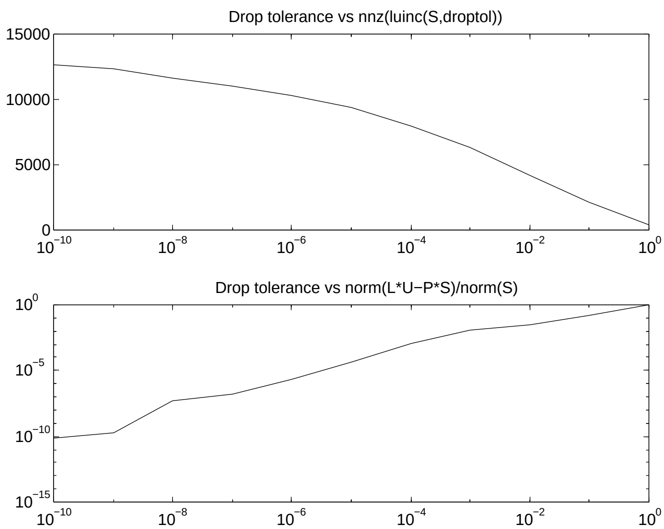
spones(L) and spones(tril(P*S)) disagree at 73 places on the diagonal, where L is 1 and P*S is 0, and also at position (206,113), where L is 0 due to cancellation, and P*S is –1. D has entries of the order of eps.




```
[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 second figure below.





Algorithm

`luinc(X,'0')` is based on the “KJI” variant of the LU factorization with partial pivoting. Updates are made only to positions which are nonzero in `X`.

`luinc(X,droptol)` and `luinc(X,options)` are based on the column-oriented `lu` for sparse matrices.

See Also

`lu`	LU matrix factorization
`cholinc`	Incomplete Cholesky factorizations
`bicg`	BiConjugate Gradients method

References

Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

Purpose Magic square

Syntax

```
M = magic(n)
```

Description `M = magic(n)` returns an `n`-by-`n` matrix constructed from the integers `1` through `n^2` with equal row and column sums. The order `n` must be a scalar greater than or equal to `3`.

Remarks A magic square, scaled by its magic sum, is doubly stochastic.

Examples The magic square of order 3 is

```
M = magic(3)
M =
    8    1    6
    3    5    7
    4    9    2
```

This is called a magic square because the sum of the elements in each column is the same.

```
sum(M) =
    15    15    15
```

And the sum of the elements in each row, obtained by transposing twice, is the same.

```
sum(M')' =
    15
    15
    15
```

This is also a special magic square because the diagonal elements have the same sum.

```
sum(diag(M)) =
     15
```

The value of the characteristic sum for a magic square of order `n` is

```
sum(1:n^2)/n
```

which, when `n = 3`, is `15`.

Algorithm

There are three different algorithms: one for odd `n`, one for even `n` not divisible by four, and one for even `n` divisible by four.

To make this apparent, type:

```
for n = 3:20
    A = magic(n);
    plot(A,'-')
    r(n) = rank(A);
end
r
```

Limitations

If you supply `n` less than `3`, `magic` returns either a nonmagic square, or else the degenerate magic squares `1` and `[]`.

See Also

`ones`	Create an array of all ones
`rand`	Uniformly distributed random numbers and arrays

Purpose

Convert a matrix into a string

Syntax

```
str = mat2str(A)
str = mat2str(A,n)
```

Description

`str = mat2str(A)` converts matrix `A` into a string, suitable for input to the `eval` function, using full precision.

`str = mat2str(A,n)` converts matrix `A` using `n` digits of precision.

Limitations

The `mat2str` function is intended to operate on scalar, vector, or rectangular array inputs only. An error will result if `A` is a multidimensional array.

Examples

Consider the matrix:

```
A =
     1     2
     3     4
```

The statement

```
b = mat2str(A)
```

produces:

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

where `b` is a string of 11 characters, including the square brackets, spaces, and a semicolon.

`eval(mat2str(A))` reproduces `A`.

See Also

`int2str`	Integer to string conversion
`sprintf`	Write formatted data to a string
`str2num`	String to number conversion

matlabrc

Purpose MATLAB startup M-file

Syntax

```
matlabrc
startup
```

Description

At startup time, MATLAB automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`. On multiuser or networked systems, `matlabrc.m` is reserved for use by the system manager. The file `matlabrc.m` invokes the file `startup.m` if it exists on MATLAB's search path.

As an individual user, you can create a startup file in your own MATLAB directory. Use these files 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.

See Also

`!`	Operating system command
`exist`	Check if a variable or file exists
`path`	Control MATLAB's directory search path

`quit` Terminate MATLAB

Purpose Root directory of MATLAB installation

Syntax

```
rd = matlabroot
```

Description

`rd = matlabroot` returns the name of the directory in which the MATLAB software is installed.

Example

```
fullfile(matlabroot,'toolbox','matlab','general','')
```

produces a full path to the `toolbox/matlab/general` directory that is correct for the platform it is executed on.

Purpose Maximum elements of an array

Syntax

```
C = max(A)
C = max(A,B)
C = max(A,[],dim)
[C,I] = max(...)
```

Description

`C = max(A)` returns the largest elements along different dimensions of an array.

If `A` is a vector, `max(A)` returns the largest element in `A`.

If `A` is a matrix, `max(A)` treats the columns of `A` as vectors, returning a row vector containing the maximum element from each column.

If `A` is a multidimensional array, `max(A)` treats the values along the first non-singleton dimension as vectors, returning the maximum value of each vector.

`C = max(A,B)` returns an array the same size as `A` and `B` with the largest elements taken from `A` or `B`.

`C = max(A,[],`*`dim`*`)` returns the largest elements along the dimension of `A` specified by scalar *`dim`*. For example, `max(A,[],1)` produces the maximum values along the first dimension (the rows) of `A`.

`[C,I] = max(...)` finds the indices of the maximum values of `A`, and returns them in output vector `I`. If there are several identical maximum values, the index of the first one found is returned.

Remarks

For complex input `A`, `max` returns the complex number with the largest modulus, computed with `max(abs(A))`. The `max` function ignores `NaN`s.

See Also

`isnan`	Detect Not-A-Number (NaN)
`mean`	Average or mean values of array
`median`	Median values of array
`min`	Minimum elements of an array
`sort`	Sort elements in ascending order

Purpose

Average or mean value of arrays

Syntax

```
M = mean(A)
M = mean(A,dim)
```

Description

`M = mean(A)` returns the mean values of the elements along different dimensions of an array.

If `A` is a vector, `mean(A)` returns the mean value of `A`.

If `A` is a matrix, `mean(A)` treats the columns of `A` as vectors, returning a row vector of mean values.

If `A` is a multidimensional array, `mean(A)` treats the values along the first non-singleton dimension as vectors, returning an array of mean values.

`M = mean(A,`*`dim`*`)` returns the mean values for elements along the dimension of `A` specified by scalar *`dim`*.

Examples

```
A = [1 2 4 4; 3 4 6 6; 5 6 8 8; 5 6 8 8];
mean(A)
ans =
    3.5000    4.5000    6.5000    6.5000

mean(A,2)
ans =
    2.7500
    4.7500
```

```
6.7500
6.7500
```

See Also

`corrcoef`	Correlation coefficients
`cov`	Covariance matrix
`max`	Maximum elements of an array
`median`	Median value of arrays
`min`	Minimum elements of an array
`std`	Standard deviation

median

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`	Correlation coefficients
`cov`	Covariance matrix
`max`	Maximum elements of an array
`mean`	Average or mean value of arrays
`min`	Minimum elements of an array
`std`	Standard deviation

Purpose

Generate a menu of choices for user input

Syntax

`k = menu('`*`mtitle`*`','`*`opt1`*`','`*`opt2`*`',...,'`*`optn`*`')`

Description

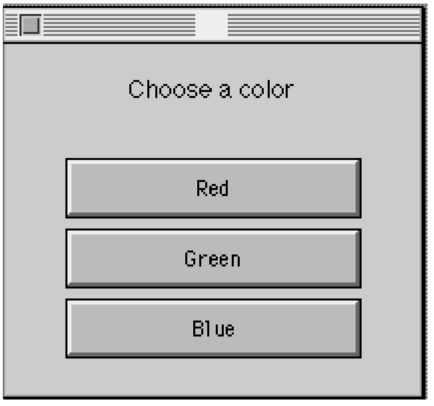
`k = menu('mtitle','`*`opt1`*`','`*`opt2`*`',...,'`*`optn`*`')` displays the menu whose title is in the string variable `'`*`mtitle`*`'` and whose choices are string variables *'opt1'*, *'opt2'*, and so on. `menu` returns the value you entered.

Remarks

To call `menu` from another ui-object, set that object's `Interruptible` property to `'yes'`. For more information, see the *MATLAB Graphics Guide*.

Examples

`k = menu('Choose a color','Red','Green','Blue')` displays



After input is accepted, use `k` to control the color of a graph.

```
color = ['r','g','b']
plot(t,s,color(k))
```

See Also

The `uicontrol` command in the *MATLAB Graphics Guide*, and:

`input` Request user input

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

The function

```
[X,Y] = meshgrid(1:3,10:14)
```

produces two output arrays, `X` and `Y`:

```
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 `mesh`, `slice`, and `surf` in the *MATLAB Graphics Guide*, `griddata`, `ndgrid`

Purpose Display method names

Syntax

```
methods class_name
n = methods('class_name')
```

Description `methods` *`class_name`* displays the names of the methods for the class with the name *class_name*.

`n = methods('`*`class_name`*`')` returns the method names in a cell array of strings.

See Also

`help`	Online help for MATLAB functions and M-files
`what`	List M-, MAT- and MEX-files
`which`	Locate functions and files

Purpose Return the MEX-filename extension

Syntax

```
ext = mexext
```

Description `ext = mexext` returns the filename extension for the current platform.

Purpose The name of the currently running M-file

Syntax

```
mfilename
```

Description `mfilename` returns a string containing the name of the most recently invoked M-file. When called from within an M-file, it returns the name of that M-file, allowing an M-file to determine its name, even if the filename has been changed.

When called from the command line, `mfilename` returns an empty matrix.

Purpose

Minimum elements of an array

Syntax

```
C = min(A)
C = min(A,B)
C = min(A,[],dim)
[C,I] = min(...)
```

Description

`C = min(A)` returns the smallest elements along different dimensions of an array.

If `A` is a vector, `min(A)` returns the smallest element in `A`.

If `A` is a matrix, `min(A)` treats the columns of `A` as vectors, returning a row vector containing the minimum element from each column.

If `A` is a multidimensional array, `min` operates along the first nonsingleton dimension.

`C = min(A,B)` returns an array the same size as `A` and `B` with the smallest elements taken from `A` or `B`.

`C = min(A,[],dim)` returns the smallest elements along the dimension of `A` specified by scalar *dim*. For example, `min(A,[],1)` produces the minimum values along the first dimension (the rows) of `A`.

`[C,I] = min(...)` finds the indices of the minimum values of `A`, and returns them in output vector `I`. If there are several identical minimum values, the index of the first one found is returned.

Remarks

For complex input `A`, `min` returns the complex number with the smallest modulus, computed with `min(abs(A))`. The `min` function ignores `NaN`s.

See Also

`max`	Maximum elements of an array
`mean`	Average or mean values of array
`median`	Median values of array
`sort`	Sort elements in ascending order

mod

Purpose

Modulus (signed remainder after division)

Syntax

```
M = mod(X,Y)
```

Definition

`mod(x,y)` is *x* mod *y*.

Description

`M = mod(X,Y)` returns the remainder `X - Y.*floor(X./Y)` for nonzero `Y`, and returns `X` otherwise. `mod(X,Y)` always differs from `X` by a multiple of `Y`.

Remarks

So long as operands `X` and `Y` are of the same sign, the function `mod(X,Y)` returns the same result as does `rem(X,Y)`. However, for positive `X` and `Y`,

```
mod(-x,y) = rem(-x,y)+y
```

The `mod` function is useful for congruence relationships:
x and y are congruent (mod m) if and only if `mod(x,m) == mod(y,m)`.

Examples

```
mod(13,5)

ans =
     3

mod([1:5],3)

ans =
     1     2     0     1     2

mod(magic(3),3)

ans =
     2     1     0
     0     2     1
     1     0     2
```

Limitations

Arguments `X` and `Y` should be integers. Due to the inexact representation of floating-point numbers on a computer, real (or complex) inputs may lead to unexpected results.

See Also

`rem`	Remainder after division

Purpose Control paged output for the command window

Syntax

```
more off
more on
more(n)
```

Description `more off` disables paging of the output in the MATLAB command window.

`more on` enables paging of the output in the MATLAB command window.

`more(n)` displays `n` lines per page.

When you've enabled `more` and are examining output:

Press the...	To...
Return key	Advance to the next line of output.
Space bar	Advance to the next page of output.
q (for quit) key	Terminate display of the text.

By default, `more` is disabled. When enabled, `more` defaults to displaying `23` lines per page.

See Also `diary` Save session in a disk file

Purpose Mu-law to linear conversion.

Syntax

```
y = mu2lin(mu)
```

Description `y = mu2lin(mu)` converts mu-law encoded 8-bit audio signals, stored as "flints" in the range $0 \le mu \le 255$, to linear signal amplitude in the range `–s < Y < s` where `s = 32124/32768 ~= .9803`. The input `mu` is often obtained using `fread(...,'uchar')` to read byte-encoded audio files. "Flints" are MATLAB's integers – floating-point numbers whose values are integers.

See Also

`auread` Read NeXT/SUN (.au) sound file
`lin2mu` Linear to mu-law conversion

Purpose Not-a-Number

Syntax `NaN`

Description `NaN` returns the IEEE arithmetic representation for Not-a-Number (`NaN`). These result from operations which have undefined numerical results.

Examples These operations produce `NaN`:

- Any arithmetic operation on a `NaN`, such as `sqrt(NaN)`
- Addition or subtraction, such as magnitude subtraction of infinities as `(+Inf)+(-Inf)`
- Multiplication, such as `0*Inf`
- Division, such as `0/0` and `Inf/Inf`
- Remainder, such as `rem(x,y)` where `y` is zero or `x` is infinity

Remarks Logical operations involving `NaN`s always return false, except ~= (not equal). Consequently, the statement `NaN` ~= `NaN` is true while the statement `NaN` == `NaN` is false.

See Also

`Inf` Infinity

Purpose Check number of input arguments

Syntax

`msg = nargchk(`*low*`,`*high*`,number)`

Description The `nargchk` function often is used inside an M-file to check that the correct number of arguments have been passed.

`msg = nargchk(`*low*`,`*high*`,number)` returns an error message if `number` is less than *low* or greater than *high*. If `number` is between *low* and *high* (inclusive), `nargchk` returns an empty matrix.

Arguments

low, *high*	The minimum and maximum number of input arguments that should be passed.
`number`	The number of arguments actually passed, as determined by the `nargin` function.

Examples

Given the function `foo`:

```
function f = foo(x,y,z)
error(nargchk(2,3,nargin))
```

Then typing `foo(1)` produces:

```
Not enough input arguments.
```

See Also

`nargin`, `nargout` Number of function arguments

Purpose

Number of function arguments

Syntax

```
n = nargin
n = nargin('fun')
n = nargout
n = nargout('fun')
```

Description

In the body of a function M-file, `nargin` and `nargout` indicate how many input or output arguments, respectively, a user has supplied. Outside the body of a function M-file, `nargin` and `nargout` indicate the number of input or output arguments, respectively, for a given function. The number of arguments is negative if the function has a variable number of arguments.

`nargin` returns the number of input arguments specified for a function.

`nargin('`*fun*`')` returns the number of declared inputs for the M-file function *fun* or `–1` if the function has a variable of input arguments.

`nargout` returns the number of output arguments specified for a function.

`nargout('`*fun*`')` returns the number of declared outputs for the M-file function *fun*.

Examples

This example shows portions of the code for a function called `myplot`, which accepts an optional number of input and output arguments:

```
function [x0,y0] = myplot(fname,lims,npts,angl,subdiv)
% MYPLOT  Plot a function.
% MYPLOT(fname,lims,npts,angl,subdiv)
%     The first two input arguments are
%     required; the other three have default values.
```

```
 ...
if nargin < 5, subdiv = 20; end
if nargin < 4, angl = 10; end
if nargin < 3, npts = 25; end
 ...
if nargout == 0
     plot(x,y)
else
     x0 = x;
     y0 = y;
end
```

See Also

`inputname`	Input argument name
`nargchk`	Check number of input arguments

Purpose All combinations of the *n* elements in `v` taken *k* at a time

Syntax

```
C = nchoosek(v,k)
```

Description

`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`	All possible permutations

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 ith dimension of the output array X*i* are copies of elements of the vector x*i*.

`[X1,X2,...] = ndgrid(x)` is the same as `[X1,X2,...] = ndgrid(x,x,...)`.

Examples To 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`	Generate X and Y matrices for three-dimensional plots
`interpn`	Multidimensional data interpolation (table lookup).

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`	Array dimensions

nextpow2

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`	One-dimensional fast Fourier transform
`log2`	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
`pow2`	Base 2 power and scale floating-point numbers

Purpose

Nonnegative least squares

Syntax

```
x = nnls(A,b)
x = nnls(A,b,tol)
[x,w] = nnls(A,b)
[x,w] = nnls(A,b,tol)
```

Description

`x = nnls(A,b)` solves the system of equations $Ax = b$ in a least squares sense, subject to the constraint that the solution vector `x` has nonnegative elements: $x_j \geq 0, \quad j = 1, 2, \ldots n$. The solution `x` minimizes $\|(Ax = b)\|$ subject to $x \geq 0$.

`x = nnls(A,b,tol)` solves the system of equations, and specifies a tolerance `tol`. By default, `tol` is: `max(size(A))*norm(A,1)*eps`.

`[x,w] = nnls(A,b)` also returns the dual vector `w`, where $w_i \leq 0$ when $x_i = 0$ and $w_i \cong 0$ when $x_i > 0$.

`[x,w] = nnls(A,b,tol)` solves the system of equations, returns the dual vector `w`, and specifies a tolerance `tol`.

Examples

Compare the unconstrained least squares solution to the `nnls` solution for a 4-by-2 problem:

```
A =
    0.0372    0.2869
    0.6861    0.7071
    0.6233    0.6245
    0.6344    0.6170

b =

    0.8587
    0.1781
    0.0747
    0.8405

[A\b nnls(A,b)] =

    -2.5627        0
     3.1108   0.6929
```

```
[norm(A*(a\b)-b) norm(A*nnls(a,b)-b)] =

     0.6674 0.9118
```

The solution from `nnls` does not fit as well, but has no negative components.

Algorithm

The `nnls` function uses the algorithm described in [1], Chapter 23. The algorithm starts with a set of possible basis vectors, computes the associated dual vector `w`, and selects the basis vector corresponding to the maximum value in `w` to swap out of the basis in exchange for another possible candidate, until $w \leq 0$.

See Also

`\`	Matrix left division (backslash)

References

[1] Lawson, C. L. and R. J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, 1974, Chapter 23.

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`	Find indices and values of nonzero elements
`nonzeros`	Nonzero matrix elements
`nzmax`	Amount of storage allocated for nonzero matrix elements
`size`	Array dimensions
`whos`	List directory of variables in memory
`isa`	Detect an object of a given class

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`	Find indices and values of nonzero elements
`nnz`	Number of nonzero matrix elements
`nzmax`	Amount of storage allocated for nonzero matrix elements
`size`	Array dimensions
`whos`	List directory of variables in memory
`isa`	Detect an object of a given class

Purpose

Vector and matrix norms

Syntax

```
n = norm(A)
n = norm(A,p)
```

Description

The *norm* of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. The `norm` function calculates several different types of matrix norms:

`n = norm(A)` returns the largest singular value of `A`, `max(svd(A))`.

`n = norm(A,p)` returns a different kind of norm, depending on the value of *p*:

If *p* is...	Then `norm` returns...
`1`	The 1-norm, or largest column sum of `A`, `max(sum(abs((A)))`.
`2`	The largest singular value (same as `norm(A)`).
`inf`	The infinity norm, or largest row sum of `A`, `max(sum(abs(A')))`.
`'fro'`	The Frobenius-norm of matrix `A`, `sqrt(sum(diag(A'*A)))`.

When `A` is a vector, slightly different rules apply:

`norm(A,p)`	Returns `sum(abs(A).^p)^(1/p)`, for any $1 \leq p \leq \infty$.
`norm(A)`	Returns `norm(A,2)`.
`norm(A,inf)`	Returns `max(abs(A))`.
`norm(A,-inf)`	Returns `min(abs(A))`.

Remarks

To obtain the root-mean-square (RMS) value, use `norm(A)/sqrt(n)`.

Note that `norm(A)`, where `A` is an `n`-element vector, is the length of `A`.

See Also

`cond`	Condition number with respect to inversion
`normest`	2-norm estimate

`svd`	Singular value decomposition

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`	Condition number with respect to inversion
`condest`	1-norm matrix condition number estimate
`norm`	Vector and matrix norms
`svd`	Singular value decomposition

now

Purpose

Current date and time

Syntax

```
t = now
```

Description

`t = now` returns the current date and time as a serial date number. To return the time only, use `rem(now,1)`. To return the date only, use `floor(now)`.

Examples

```
t1 = now, t2 = rem(now,1)

t1 =

   7.2908e+05


t2 =

    0.4013
```

See Also

`clock`	Current time as a date vector
`date`	Current date string
`datenum`	Serial date number

Purpose Null space of a matrix

Syntax

```
B = null(A)
```

Description `B = null(A)` returns an orthonormal basis for the null space of `A`.

Remarks `B'*B = I`, `A*B` has negligible elements, and (if `B` is not equal to the empty matrix) the number of columns of `B` is the nullity of `A`.

See Also

`orth`	Range space of a matrix
`qr`	Orthogonal-triangular decomposition
`svd`	Singular value decomposition

num2cell

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`	Concatenate arrays

Purpose Number to string conversion

Syntax

```
str = num2str(A)
str = num2str(A,precision)
str = num2str(A,format)
```

Description The `num2str` function converts numbers to their string representations. This function is useful for labeling and titling plots with numeric values.

`str = num2str(a)` converts array `A` into a string representation `str` with roughly four digits of precision and an exponent if required.

`str = num2str(a,`*`precision`*`)` converts the array `A` into a string representation `str` with maximum precision specified by *precision*. Argument *precision* specifies the number of digits the output string is to contain. The default is four.

`str = num2str(A,`*`format`*`)` converts array `A` using the supplied *format*. By default, this is `'%11.4g'`, which signifies four significant digits in exponential or fixed-point notation, whichever is shorter. (See `fprintf` for format string details).

Examples `num2str(pi)` is `3.142`.

`num2str(eps)` is `2.22e–16`.

`num2str(magic(2))` produces the string matrix

```
1  3
4  2
```

See Also

`fprintf`	Write formatted data to file
`int2str`	Integer to string conversion
`sprintf`	Write formatted data to a string

nzmax

Purpose

Amount of storage allocated for nonzero matrix elements

Syntax

```
n = nzmax(S)
```

Description

`n = nzmax(S)` returns the amount of storage allocated for nonzero elements.

If `S` is a sparse matrix...	`nzmax(S)` is the number of storage locations allocated for the nonzero elements in `S`.
If `S` is a full matrix...	`nzmax(S) = prod(size(S))`.

Often, `nnz(S)` and `nzmax(S)` are the same. But if `S` is created by an operation which produces fill-in matrix elements, such as sparse matrix multiplication or sparse LU factorization, more storage may be allocated than is actually required, and `nzmax(S)` reflects this. Alternatively, `sparse(i,j,s,m,n,nzmax)` or its simpler form, `spalloc(m,n,nzmax)`, can set `nzmax` in anticipation of later fill-in.

See Also

`find`	Find indices and values of nonzero elements
`nnz`	Number of nonzero matrix elements
`nonzeros`	Nonzero matrix elements
`size`	Array dimensions
`whos`	List directory of variables in memory
`isa`	Detect an object of a given class

Purpose

Solve differential equations

Syntax

```
[T,Y] = solver('F',tspan,y0)
[T,Y] = solver('F',tspan,y0,options)
[T,Y] = solver('F',tspan,y0,options,p1,p2...)
[T,Y,TE,YE,IE] = solver('F',tspan,y0,options)
[T,X,Y] = solver('model',tspan,y0,options,ut,p1,p2,...)
```

Arguments

`F`	Name of the ODE file, a MATLAB function of `t` and `y` returning a column vector. All solvers can solve systems of equations in the form $y' = F(t, y)$. `ode15s` and `ode23s` can both solve equations of the form $My' = F(t, y)$. Only `ode15s` can solve equations in the form $M(t)y' = F(t, y)$. For information about ODE file syntax, see the `odefile` reference page.
`tspan`	A vector specifying the interval of integration `[t0 tfinal]`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1, ..., tfinal]`.
`y0`	A vector of initial conditions.
`options`	Optional integration argument created using the `odeset` function. See `odeset` for details.
`p1,p2...`	Optional parameters to be passed to `F`.
`T,Y`	Solution matrix `Y`, where each row corresponds to a time returned in column vector `T`.

Description

`[T,Y] = `*`solver`*`('F',tspan,y0)` with `tspan = [t0 tfinal]` integrates the system of differential equations $y' = F(t,y)$ from time `t0` to `tfinal` with initial conditions `y0`. `'F'` is a string containing the name of an ODE file. Function `F(t,y)` must return a column vector. Each row in solution array `y` corresponds to a time returned in column vector `t`. To obtain solutions at the specific times `t0`, `t1`, ..., `tfinal` (all increasing or all decreasing), use `tspan = [t0 t1 ... tfinal]`.

`[T,Y] = `*`solver`*`('F',tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function (see `odeset` for details). Commonly used properties 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).

`[T,Y] = `*solver*`('F',tspan,y0,options,p1,p2...)` solves as above, passing the additional parameters `p1,p2...` to the M-file `F`, whenever it is called. Use `options = []` as a place holder if no options are set.

`[T,Y,TE,YE,IE] = `*solver*`('F',tspan,y0,options)` with the `Events` property in `options` set to `'on'`, solves as above while also locating zero crossings of an event function defined in the ODE file. The ODE file must be coded so that `F(t,y,'events')` returns appropriate information. See `odefile` for details. 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.

When called with no output arguments, the solvers call the default output function `odeplot` to plot the solution as it is computed. An alternate method is to set the `OutputFcn` property to `'odeplot'`. Set the `OutputFcn` property to `'odephas2'` or `'odephas3'` for two- orthree-dimensional phase plane plotting. See `odefile` for details.

For the stiff solvers `ode15s` and `ode23s`, the Jacobian matrix $\partial F / \partial y$ is critical to reliability and efficiency so there are special options. Set `JConstant` to `'on'` if $\partial F / \partial y$ is constant. Set `Vectorized` to `'on'` if the ODE file is coded so that `F(t,[y1 y2 ...])` returns `[F(t,y1) F(t,y2) ...]`. Set `Jattern` to `'on'` if $\partial F / \partial y$ is a sparse matrix and the ODE file is coded so that `F([],[],'jpattern')` returns a sparsity pattern matrix of `1`'s and `0`'s showing the nonzeros of $\partial F / \partial y$. Set `Jacobian` to `'on'` if the ODE file is coded so that `F(t,y,'jacobian')` returns $\partial F / \partial y$.

Both `ode15s` and `ode23s` can solve problems $My' = F(t, y)$ with a constant mass matrix M that is nonsingular and (usually) sparse. Set `Mass` to `'on'` if the ODE file is coded so that `F([],[],'mass')` returns M (see `fem2ode`). Only `ode15s` can solve problems $M(t)y' = F(t, y)$ with a time-dependent mass matrix *M(t)* that is nonsingular and (usually) sparse. Set `Mass` to `'on'` if the ODE file is coded so that `F(t,[],'mass')` returns $M(t)$ (see `fem1ode`). For `ode15s` set `MassConstant` to `'on'` if M is constant.

`[T,X,Y] = `*`solver`*`('model',tspan,y0,options,ut,p1,p2,...)` solves a SIMULINK model by calling the corresponding solver in SIMULINK:

```
[T,X,Y] = sim(solver,'model',...)
```

The `options` argument is created with the `odeset` function. See `sim`.

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	If using crude error tolerances or solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	If using stringent error tolerances or solving a computationally intensive ODE file.
`ode15s`	Stiff	Low to medium	If `ode45` is slow (stiff systems) or there is a mass matrix.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems or there is a constant mass matrix.

The algorithms used in the ODE solvers vary according to order of accuracy [5] and the type of systems (stiff or nonstiff) they are designed to solve. See Algorithms on page 2-459 for more details.

It is possible to specify `tspan`, `y0` and `options` in the ODE file (see `odefile`). If `tspan` or `y0` is empty, then the solver calls the ODE file:

```
[tspan,y0,options] = F([],[],'init')
```

to obtain any values not supplied in the solver's argument list. Empty arguments at the end of the call list may be omitted. This permits you to call the solvers with other syntaxes such as:

```
[T,Y] = solver('F')
[T,Y] = solver('F',[],y0)
[T,Y] = solver('F',tspan,[],options)
[T,Y] = solver('F',[],[],options)
```

Integration parameters (options) can be specified both in the ODE file and on the command line. If an option is specified in both places, the command line specification takes precedence. For information about constructing an ODE file, see the odefile reference page.

Options

Different solvers accept different parameters in the options list. For more information, see odeset and *Applying MATLAB* .

Parameters	ode45	ode23	ode113	ode115s	ode23s
RelTol, AbsTol	√	√	√	√	√
OutputFcn, OutputSel, Refine, Stats	√	√	√	√	√
Events	√	√	√	√	√
MaxStep, InitialStep	√	√	√	√	√
JConstant, Jacobian, JPattern, Vectorized	—	—	—	√	√
Mass MassConstant	— —	— —	— —	√ √	√ —
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 M-file `rigid` containing the equations:

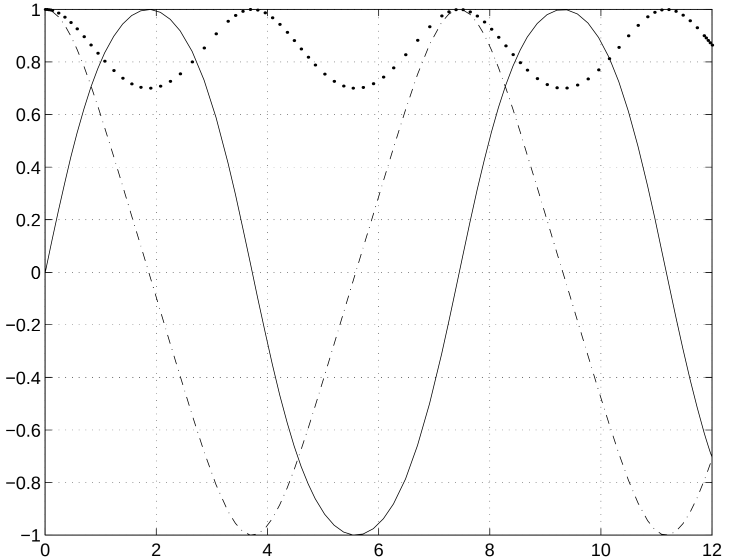
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we will change the error tolerances with the `odeset` command and solve on a time interval of `[0 12]` with 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 governing relaxation oscillation. The limit cycle has portions where the

solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y'_1 = y_2 \qquad\qquad y_1(0) = 0$$
$$y'_2 = 1000(1-y_1^2)y_2 - y_1 \quad y_2(0) = 1$$

To simulate this system, create a function M-file `vdp1000` containing the equations:

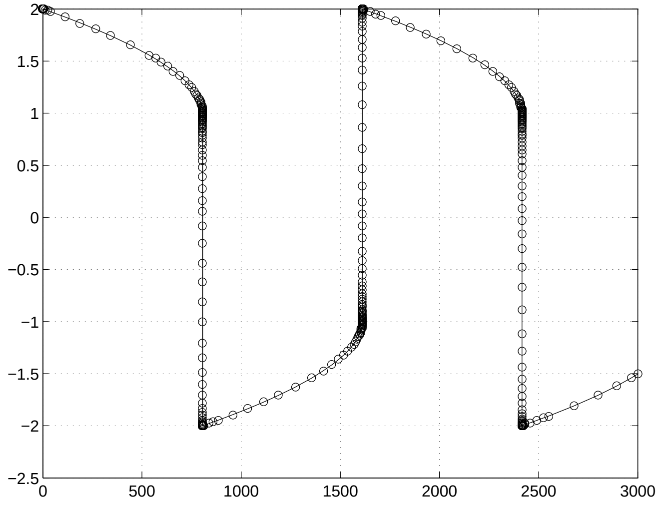
```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 – y(1)^2)*y(2) – y(1);
```

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. [1]

`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. [3]

The above algorithms are intended to solve non-stiff systems. If they appear to be unduly slow, try using one of the stiff solvers (`ode15s` and `ode23s`) instead.

`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. If you suspect that a problem is stiff or if `ode45` has failed or was very inefficient, try `ode15s`. [7]

`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. [7]

See Also

`odeset`, `odeget`, `odefile`

References

[1] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp 19–26.

[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] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[6] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[7] Shampine, L. F. and M. W. Reichelt, “The MATLAB ODE Suite,” (to appear in *SIAM Journal on Scientific Computing*, Vol. 18-1, 1997).

Purpose

Define a differential equation problem for 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 MATLAB's ODE solvers. In MATLAB documentation, this M-file is referred to as `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:

$$y' = F(t, y)$$

$$My' = F(t, y)$$

$$M(t)y' = F(t, y)$$

where:

- t is a scalar independent variable, typically representing time.
- y is a vector of dependent variables.
- F is a function of t and y returning a column vector the same length as y.
- M and $M(t)$ represent nonsingular constant or time dependent mass matrices.

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`.

Only the stiff solver `ode15s` can solve $M(t)y' = F(t, y)$. Both `ode15s` and `ode23s` can solve equations of the form $My' = F(t, y)$.

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 2-464).

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. (See item 2 as well as Examples on page 2-464). Here is an annotated version of the result:

```
function [out1,out2,out3] = odefile(t,y,flag,p1,p2)      1
% ODEFILE The template for ODE files.
%
if nargin < 3 | isempty(flag)  % Return dy/dt = F(t,y)   2
  out1 = < Insert a function of t and/or y, p1, and p2 here >;

else
  switch(flag)                                           3
case 'init'          % Return default [tspan, y0, and options]
    out1 = < Insert tspan here >;                        4
    out2 = < Insert y0 here >;
    out3 = < Insert options = odeset(...) or [] here >;

  case 'jacobian'   % Return matrix J(t,y) = dF/dy       5
    out1 = < Insert Jacobian matrix here >;

  case 'jpattern'  % Return sparsity pattern matrix S    6
    out1 = < Insert Jacobian matrix sparsity pattern here >

  case 'mass'      % Return mass matrix M(t) or M        7
    out1 = < Insert mass matrix here >;

  case 'events'    % Return event vector and info
    out1 = < Insert event function vector here >;
    out2 = < Insert logical isterminal vector here >;    8
    out3 = < Insert direction vector here >;

  otherwise                                              9
    error(['Unknown flag''' flag '''.']);
  end
end
```

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 steps 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 wish to improve the performance of the stiff solvers `ode15s` and `ode23s`.
6. In the `'jpattern'` case, the ODE file returns the Jacobian sparsity pattern matrix to the solver. You need 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 provide this case only when you want to solve a system in either of the forms $My' = F(t, y)$ or $M(t)y' = F(t, y)$.
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 `1`, 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. See the *Applying MATLAB* and also the examples `ballode` and `orbitode`.
9. An unrecognized `flag` generates an error.

Examples

The van der Pol equation, $y''_1 - \mu(1 - y_1^{\,2})y'_1 + y_1 =$ is equivalent to a system of coupled first-order differential equations:

$$y'_1 = y_2$$

$$y'_2 = \mu(1 - y_1^{\,2})y_2 - y_1$$

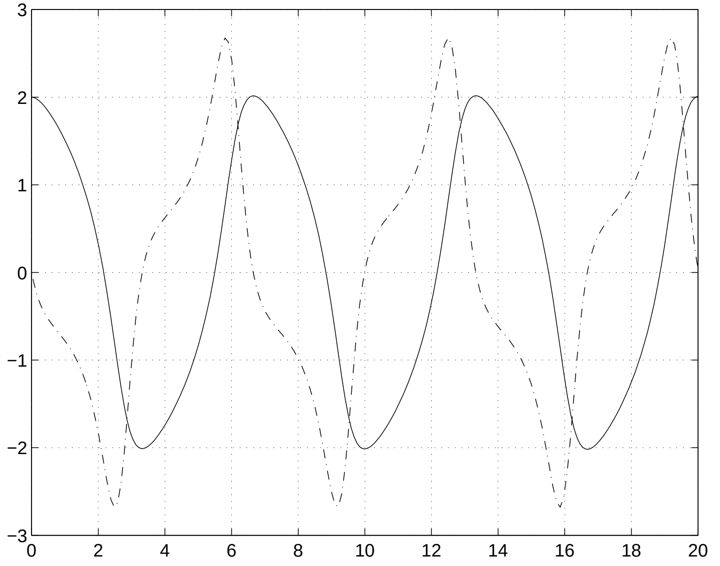
The M-file

```
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).

Some example ODE files we have provided include b5ode, brussode, vdpode, orbitode, and rigidode. Use type *filename* from the MATLAB command line to see the coding for a specific ODE file.

See Also

The *Applying MATLAB* and the reference entries for the ODE solvers and their associated functions:

ode23, ode45, ode113, ode15s, ode23s, odeget, odeset

odeget

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 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. 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. The `odeset` function sets any unspecified properties to the empty matrix `[]`.

It is sufficient to type only the leading characters that uniquely identify the property name. Case is ignored for property names.

`options = odeset(oldopts,'name1',value1,...)` alters an existing options structure with the values supplied.

`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`. For example:

`oldopts`

F	1	[]	4	's'	's'	[]	[]	[]	...

`newopts`

T	3	F	[]	''	[]	[]	[]	[]	...

`odeset(oldopts,newopts)`

T	3	F	4	''	's'	[]	[]	[]	...

`odeset` by itself, displays all property names and their possible values:

```
odeset
        AbsTol: [ positive scalar or vector {1e–6}]
        BDF: [ on | {off} ]
        Events: [ on | {off} ]
        InitialStep: [ positive scalar ]
        Jacobian: [ on | {off} ]
        JConstant: [ on | {off} ]
        JPattern: [ on | {off} ]
        Mass: [ on | {off} ]
        MassConstant: [ on | off]
        MaxOrder: [ 1 | 2 | 3 | 4 | {5} ]
        MaxStep: [ positive scalar ]
        OutputFcn: [ string ]
        OutputSel: [ vector of integers ]
        Refine: [ positive integer ]
        RelTol: [ positive scalar {1e–3} ]
        Stats: [ on | {off} ]
        Vectorized: [ on | {off} ]
```

Properties

The available properties depend upon the ODE solver used. There are seven principal categories of properties:

- Error tolerance
- Solver output
- Jacobian
- Event location
- Mass matrix
- Step size
- ode15s

Table 1-1: Error Tolerance Properties

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	A relative error tolerance that applies to all components of the solution vector.
`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.

Table 1-2: Solver Output Properties

Property	Value	Description
`OutputFcn`	String	The name of an installable output function (for example, `odeplot`, `odephas2`, `odephas3`, and `odeprint`). The ODE solvers call `outputfcn(TSPAN,Y0,'init')` before beginning the integration, to initialize the output function. Subsequently, the solver calls `status = outputfcn(T,Y)` after computing each output point `(T,Y)`. The `status` return value should be `1` if integration should be halted (e.g., a **STOP** button has been pressed) and 0 otherwise. When the integration is complete, the solver calls `outputfcn([],[],'done')`.
`OutputSel`	Vector of indices	Specifies which components of the solution vector are to be passed to the output function.

Table 1-2: Solver Output Properties

Property	Value	Description
`Refine`	Positive Integer	Produces smoother output, increasing the number of output points by a factor of `n`. In most solvers, the default value is `1`. However, within `ode45`, `Refine` is `4` by default to compensate for the solver's large step sizes. To override this and see only the time steps chosen by `ode45`, set `Refine` to `1`.
`Stats`	on | {off}	Specifies whether statistics about the computational cost of the integration should be displayed.

Table 1-3: Jacobian Matrix Properties (for ode15s and ode23s)

Property	Value	Description
`JConstant`	on | {off}	Specifies whether the Jacobian matrix $\partial F / \partial y$ is constant (see `b5ode`).
`Jacobian`	on | {off}	Informs the solver that the ODE file responds to the arguments `(t,y,'jacobian')` by returning $\partial F / \partial y$ (see `odefile`).
`JPattern`	on | {off}	Informs the solver that the ODE file responds to the arguments `([],[],'jpattern')` by returning a sparse matrix containing `1`'s showing the nonzeros of $\partial F / \partial y$ (see `brussode`).

Table 1-3: Jacobian Matrix Properties (for ode15s and ode23s)

Property	Value	Description
`Vectorized`	`on | {off}`	Informs the solver that the ODE file `F(t,y)` has been vectorized so that `F(t,[y1 y2 ...])` returns `[F(t,y1) F(t,y2) ...]`. That is, your ODE file can pass to the solver a whole array of column vectors at once. Your ODE file will be called by a stiff solver in a vectorized manner only if generating Jacobians numerically (the default behavior) and `odeset` has been used to set `Vectorized` to `'on'`.

Table 1-4: Event Location Property

Property	Value	Description
`Events`	`on | {off}`	Instructs the solver to locate events. The ODE file must respond to the arguments `(t,y,'events')` by returning the appropriate values. See `odefile`.

Table 1-5: Mass Matrix Properties (for ode15s and ode23s)

Property	Value	Description
`Mass`	`on | {off}`	Informs the solver that the ODE file is coded so that `F(t,[],'mass')` returns M or $M(t)$ (see `odefile`).
`MassConstant`	`on | {off}`	Informs the solver that the mass matrix $M(t)$ is constant.

Table 1-6: Step Size Properties

Property	Value	Description
`MaxStep`	Positive scalar	An upper bound on the magnitude of the step size that the solver uses.
`InitialStep`	Positive scalar	Suggested initial step size. The solver tries this first, but if too large an error results, the solver uses a smaller step size.

In addition there are two options that apply only to the `ode15s` solver.

Table 1-7: ode15s Properties

Property	Value	Description
`MaxOrder`	`1 | 2 | 3 | 4 | {5}`	The maximum order formula used.
`BDF`	`on | {off}`	Specifies whether the Backward Differentiation Formulas (BDF's) are to be used instead of the default Numerical Differentiation Formulas (NDF's).

See Also

`odefile`, `odeget`, `ode45`, `ode23`, `ode113`, `ode15s`,`ode23s`

Purpose Create an array of all ones

Syntax

```
Y = ones(n)
Y = ones(m,n)
Y = ones([m n])
Y = ones(d1,d2,d3...)
Y = ones([d1 d2 d3...])
Y = ones(size(A))
```

Description

`Y = ones(n)` returns an `n`-by-`n` matrix of `1`s. An error message appears if `n` is not a scalar.

`Y = ones(m,n)` or `Y = ones([m n])` returns an `m`-by-`n` matrix of ones.

`Y = ones(d1,d2,d3...)` or `Y = ones([d1 d2 d3...])` returns an array of `1`s with dimensions `d1`-by-`d2`-by-`d3`-by-`...`.

`Y = ones(size(A))` returns an array of `1`s that is the same size as `A`.

See Also

`eye`	Identity matrix
`rand`	Uniformly distributed random numbers and arrays
`randn`	Normally distributed random numbers and arrays
`zeros`	Create an array of all zeros

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`	Null space of a matrix
`svd`	Singular value decomposition
`rank`	Rank of a matrix

Purpose

Default part of switch statement

Description

`otherwise` is part of the `switch` statement syntax, which allows for conditional execution. The statements following `otherwise` are executed only if none of the preceding case expressions (`case_expr`) match the switch expression (`sw_expr`).

Examples

The general form of the `switch` statement is:

```
switch sw_expr
    case case_expr
        statement
        statement
    case {case_expr1,case_expr2,case_expr3}
        statement
        statement
   otherwise
        statement
        statement
end
```

See `switch` for more details.

See Also

`switch`	Switch among several cases based on expression

Purpose Consolidate workspace memory

Syntax

```
pack
pack filename
```

Description `pack`, by itself, frees up needed space by compressing information into the minimum memory required.

`pack` *filename* accepts an optional *filename* for the temporary file used to hold the variables. Otherwise it uses the file named `pack.tmp`.

Remarks The `pack` command doesn't affect the amount of memory allocated to the MATLAB process.You must quit MATLAB to free up this memory.

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

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

The `pack` command frees space by:

- Saving all variables on disk in a temporary file called `pack.tmp`.
- Clearing all variables and functions from memory.
- Reloading the variables back from `pack.tmp`.
- Deleting the temporary file `pack.tmp`.

If you use pack and there is still not enough free memory to proceed, you must clear some variables. If you run out of memory often, here are some system-specific tips:

- **MS-Windows:** Increase the swap space by opening the Control Panel, double-clicking on the 386 Enhanced icon, and pressing the **Virtual Memory** button.
- **Macintosh**: Change the application memory size by using **Get Info** on the program icon. You may also want to turn on virtual memory via the Memory Control Panel.
- **VAX/VMS**: Ask your system manager to increase your working set and/or pagefile quota.
- **UNIX**: Ask your system manager to increase your swap space.

See Also

`clear`	Remove items from memory

partialpath

Purpose Partial pathname

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

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

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

Purpose Pascal matrix

Syntax

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

Description

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

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

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

Examples

`pascal(4)` returns

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

`A = pascal(3,2)` produces

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

See Also

`chol` Cholesky factorization

path

Purpose

Control MATLAB's directory search path

Syntax

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

Description

`path` prints out the current setting of MATLAB's search path. On all platforms except the Macintosh, the path resides in `pathdef.m` (in `toolbox/local`). The Macintosh stores its path in the `Matlab Settings File` (usually in the `Preferences` folder).

`p = path` returns the current search path in string variable `p`.

`path('`*newpath*`')` changes the path to the string `'`*newpath*`'`.

`path(path,'`*newpath*`')` appends a new directory to the current path.

`path('`*newpath*`',path)` prepends a new directory to the current path.

Remarks

MATLAB has a *search path*. If you enter a name, such as `fox`, the MATLAB interpreter:

1. Looks for `fox` as a variable.
2. Checks for `fox` as a built-in function.
3. Looks in the current directory for `fox.mex` and `fox.m`.
4. Searches the directories specified by `path` for `fox.mex` and `fox.m`.

Examples

Add a new directory to the search path on various operating systems:

UNIX:	`path(path,'/home/myfriend/goodstuff')`
VMS:	`path(path,'DISKS1:[MYFRIEND.GOODSTUFF]')`
MS-DOS:	`path(path,'TOOLS\GOODSTUFF')`
Macintosh:	`path(path,'Tools:GoodStuff')`

See Also

`addpath`	Add directories to MATLAB's search path
`cd`	Change working directory
`dir`	Directory listing
`rmpath`	Remove directories from MATLAB's search path
`what`	Directory listing of M-files, MAT-files, and MEX-files

Purpose Halt execution temporarily

Syntax

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

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

`pause(n)` pauses execution for `n` seconds before continuing.

`pause on` allows subsequent pause commands to pause execution.

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

See Also The `drawnow` command in the *MATLAB Graphics Guide*.

Purpose Preconditioned Conjugate Gradients method

Syntax

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

Description

`x = pcg(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The coefficient matrix `A` must be symmetric and positive definite and the right hand side (column) vector `b` must have length `n`, where `A` is `n`-by-`n`. `pcg` will start iterating from an initial estimate that by default is an all zero vector of length `n`. Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate `x` has relative residual `norm(b–A*x)/norm(b)` less than or equal to the tolerance of the method.The default tolerance is `1e–6`. The default maximum number of iterations is the minimum of `n` and 20. No preconditioning is used.

`pcg(A,b,tol)` specifies the tolerance of the method, `tol`.

`pcg(A,b,tol,maxit)` additionally specifies the maximum number of iterations, `maxit`.

`pcg(A,b,tol,maxit,M)` and `pcg(A,b,tol,maxit,M1,M2)` use left preconditioner `M` or `M=M1*M2` and effectively solve the system `inv(M)*A*x =inv(M)*b` for `x`. If `M1` or `M2` is given as the empty matrix (`[]`), it is considered to be the identity matrix, equivalent to no preconditioning at all. Since systems of equations of the form `M*y=r` are solved using backslash within `pcg`, it is wise to factor preconditioners into their Cholesky factors first. For example, replace `pcg(A,b,tol,maxit,M)` with:

```
R = chol(M);
pcg(A,b,tol,maxit,R',R).
```

The preconditioner `M` must be symmetric and positive definite.

`pcg(A,b,tol,maxit,M1,M2,x0)` specifies the initial estimate `x0`. If `x0` is given as the empty matrix (`[]`), the default all zero vector is used.

`x = pcg(A,b,tol,maxit,M1,M2,x0)` returns a solution `x`. If `pcg` converged, a message to that effect is displayed. If `pcg` failed to converge after the maximum number of iterations or halted 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.

`[x,flag] = pcg(A,b,tol,maxit,M1,M2,x0)` returns a solution `x` and a flag which describes the convergence of `pcg`:

Flag	Convergence
`0`	`pcg` converged to the desired tolerance `tol` within `maxit` iterations without failing for any reason.
`1`	`pcg` iterated `maxit` times but did not converge.
`2`	One of the systems of equations of the form `M*y = r` involving the preconditioner was ill-conditioned and did not return a useable result when solved by \ (backslash).
`3`	The method stagnated. (Two consecutive iterates were the same.)
`4`	One of the scalar quantities calculated during `pcg` became too small or too large to continue computing

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

[x,flag,relres] = pcg(A,b,tol,maxit,M1,M2,x0) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, then relres ≤ tol.

[x,flag,relres,iter] = pcg(A,b,tol,maxit,M1,M2,x0) also returns the iteration number at which x was computed. This always satisfies 0 ≤ iter ≤ maxit.

[x,flag,relres,iter,resvec] = pcg(A,b,tol,maxit,M1,M2,x0) also returns a vector of the residual norms at each iteration, starting from resvec(1) = norm(b-A*x0). If flag is 0, resvec is of length iter+1 and resvec(end) ≤ tol*norm(b).

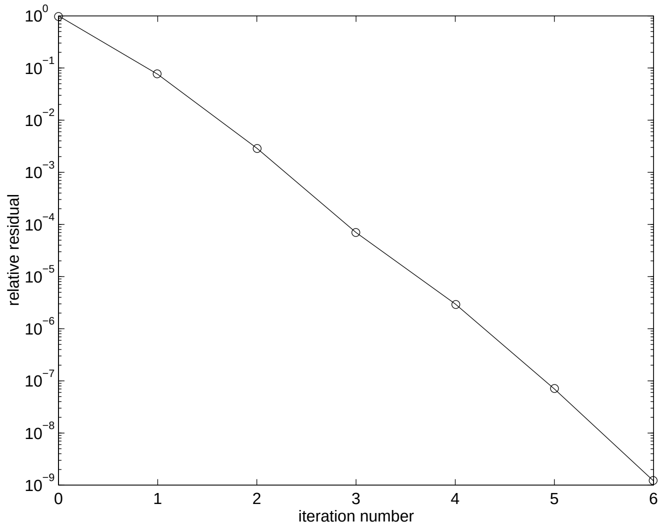
Examples

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

flag is 1 since pcg will not converge to the default tolerance of 1e-6 within the default 20 iterations.

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

flag2 is 0 since pcg will converge to the tolerance of 1.2e-9 (the value of relres2) at the sixth iteration (the value of iter2) when preconditioned by the incomplete Cholesky factorization with a drop tolerance of 1e-3. resvec2(1) = norm(b) and resvec2(7) = norm(b-A*x2).You may follow the progress of pcg by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0) with `semilogy(0:iter2,resvec2/norm(b),'-o')`.



See Also

`bicg`	BiConjugate Gradients method
`bicgstab`	BiConjugate Gradients Stabilized method
`cgs`	Conjugate Gradients Squared method
`cholinc`	Incomplete Cholesky factorizations
`gmres`	Generalized Minimum Residual method (with restarts)
`qmr`	Quasi-Minimal Residual method
`\`	Matrix left division

References

Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM, Philadelphia, 1994.

Purpose Create preparsed pseudocode file (P-file)

Syntax

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

Description

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

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

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

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

perms

Purpose

All possible permutations

Syntax

```
P = perms(v)
```

Description

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

Examples

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

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

Limitations

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

See Also

`nchoosek`	All combinations of the n elements in v taken k at a time
`permute`	Rearrange the dimensions of a multidimensional array
`randperm`	Random permutation

Purpose Rearrange the dimensions of a multidimensional array

Syntax

```
B = permute(A,order)
```

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

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

Examples Given any matrix `A`, the statement

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

is the same as `A'`.

For example:

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

The following code permutes a three-dimensional array:

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

See Also

`ipermute`	Inverse permute the dimensions of a multidimensional array

pi

Purpose

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

Syntax

```
pi
```

Description

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

Examples

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

```
sin(pi)
ans =
   1.2246e-16
```

See Also

`ans`	The most recent answer
`eps`	Floating-point relative accuracy
`i`	Imaginary unit
`Inf`	Infinity
`j`	Imaginary unit
`NaN`	Not-a-Number

Purpose

Moore-Penrose pseudoinverse of a matrix

Syntax

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

Definition

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

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

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

Description

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

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

Examples

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

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

minimize `norm(A*x-b)`

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

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

and

```
y = A\b
```

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

For example, the matrix generated by

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

is an 8-by-6 matrix that happens to have `rank(A) = 3`.

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

The right-hand side is `b = 260*ones(8,1)`,

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

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

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

which is

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

and

```
y = A\b
```

which is

```
y =
    3.0000
    4.0000
         0
         0
    1.0000
         0
```

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

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

is smaller than the norm of any other solution, including

```
norm(y) = 5.0990
```

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

See Also

`inv`	Matrix inverse
`qr`	Orthogonal-triangular decomposition
`rank`	Rank of a matrix
`svd`	Singular value decomposition

pol2cart

Purpose Transform polar or cylindrical coordinates to Cartesian

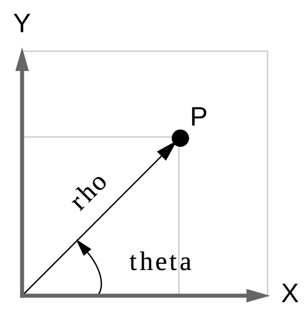
Syntax

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

Description `[X,Y] = pol2cart(THETA,RHO)` transforms the polar coordinate data stored in corresponding elements of `THETA` and `RHO` to two-dimensional Cartesian, or *xy*, coordinates. The arrays `THETA` and `RHO` must be the same size (or either can be scalar). The values in `THETA` must be in radians.

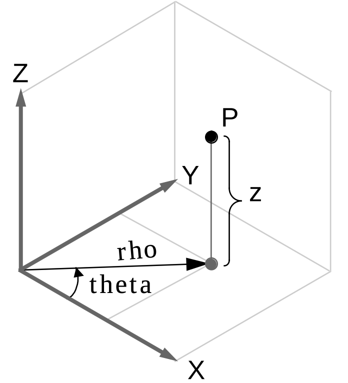
`[X,Y,Z] = pol2cart(THETA,RHO,Z)` transforms the cylindrical coordinate data stored in corresponding elements of `THETA`, `RHO`, and `Z` to three-dimensional Cartesian, or *xyz*, coordinates. The arrays `THETA` , `RHO`, and `Z` must be the same size (or any can be scalar). The values in `THETA` must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

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



Polar to Cartesian Mapping

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



Cylindrical to Cartesian Mapping

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

See Also

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

Purpose

Polynomial with specified roots

Syntax

```
p = poly(A)
p = poly(r)
```

Description

`p = poly(A)` where `A` is an `n`-by-`n` matrix returns an `n+1` element row vector whose elements are the coefficients of the characteristic polynomial, $det(sI - A)$. The coefficients are ordered in descending powers: if a vector `c` has `n+1` components, the polynomial it represents is $c_1 s^n + \ldots + c_n s + c_{n+1}$.

`p = poly(r)` where `r` is a vector returns a row vector whose elements are the coefficients of the polynomial whose roots are the elements of `r`.

Remarks

Note the relationship of this command to

```
r = roots(p)
```

which returns a column vector whose elements are the roots of the polynomial specified by the coefficients row vector `p`. For vectors, `roots` and `poly` are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples

MATLAB displays polynomials as row vectors containing the coefficients ordered by descending powers. The characteristic equation of the matrix

```
A =
    1    2    3
    4    5    6
    7    8    0
```

is returned in a row vector by `poly`:

```
p = poly(A)
p =
    1    –6    –72    –27
```

The roots of this polynomial (eigenvalues of matrix A) are returned in a column vector by roots:

```
r = roots(p)
r =
    12.1229
    -5.7345
    -0.3884
```

Algorithm

The algorithms employed for poly and roots illustrate an interesting aspect of the modern approach to eigenvalue computation. poly(A) generates the characteristic polynomial of A, and roots(poly(A)) finds the roots of that polynomial, which are the eigenvalues of A. But both poly and roots use EISPACK eigenvalue subroutines, which are based on similarity transformations. The classical approach, which characterizes eigenvalues as roots of the characteristic polynomial, is actually reversed.

If A is an n-by-n matrix, poly(A) produces the coefficients c(1) through c(n+1), with c(1) = 1, in

The algorithm is expressed in an M-file:

```
z = eig(A);
c = zeros(n+1,1); c(1) = 1;
for j = 1:n
    c(2:j+1) = c(2:j+1)-z(j)*c(1:j);
end
```

This recursion is easily derived by expanding the product.

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

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

See Also

`conv`	Convolution and polynomial multiplication
`polyval`	Polynomial evaluation
`residue`	Convert between partial fraction expansion and polynomial coefficients
`roots`	Polynomial roots

polyarea

Purpose

Area of polygon

Syntax

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

Description

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

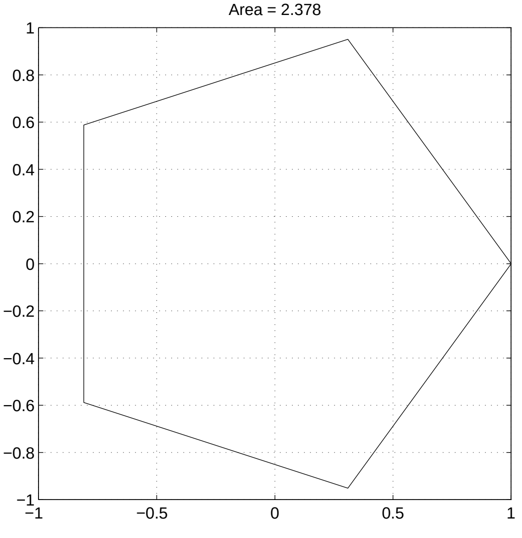
If `X` and `Y` are matrices of the same size, then `polyarea` returns the area of polygons defined by the columns `X` and `Y`.

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

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

Examples

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



See Also

`convhull`	Convex hull
`inpolygon`	Detect points inside a polygonal region

Purpose Polynomial derivative

Syntax

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

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

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

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

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

Examples The derivative of the product

is obtained with

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

This result represents the polynomial

See Also

`conv`	Convolution and polynomial multiplication
`deconv`	Deconvolution and polynomial division

Purpose Polynomial eigenvalue problem

polyeig

Syntax

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

Description

`[X,e] = polyeig(A0,A1,...Ap)` solves the polynomial eigenvalue problem of degree `p`:

$$(A_0 + \lambda A_1 + \ldots + \lambda^P A_p)x = 0$$

where polynomial degree `p` is a non-negative integer, and `A0,A1,...Ap` are input matrices of order `n`. Output matrix `X`, of size `n`-by-`n*p`, contains eigenvectors in its columns. Output vector `e`, of length `n*p`, contains eigenvalues.

Remarks

Based on the values of `p` and `n`, `polyeig` handles several special cases:

- `p = 0`, or `polyeig(A)` is the standard eigenvalue problem: `eig(A)`.
- `p = 1`, or `polyeig(A,B)` is the generalized eigenvalue problem: `eig(A,-B)`.
- `n = 1`, or `polyeig(a0,a1,...ap)` for scalars `a0`, `a1` ..., `ap` is the standard polynomial problem: `roots([ap ... a1 a0])`.

Algorithm

If both `A0` and `Ap` are singular, the problem is potentially ill posed; solutions might not exist or they might not be unique. In this case, the computed solutions may be inaccurate. `polyeig` attempts to detect this situation and display an appropriate warning message. If either one, but not both, of `A0` and `Ap` is singular, the problem is well posed but some of the eigenvalues may be zero or infinite (`Inf`).

The `polyeig` function uses the QZ factorization to find intermediate results in the computation of generalized eigenvalues. It uses these intermediate results to determine if the eigenvalues are well-determined. See the descriptions of `eig` and `qz` for more on this, as well as the *EISPACK Guide*.

See Also

`eig`	Eigenvalues and eigenvectors
`qz`	QZ factorization for generalized eigenvalues

Purpose Polynomial curve fitting

Syntax

```
p = polyfit(x,y,n)
[p,s] = polyfit(x,y,n)
```

Description `p = polyfit(x,y,n)` finds the coefficients of a polynomial `p(x)` of degree `n` that fits the data, `p(x(i))` to `y(i)`, in a least squares sense. The result `p` is a row vector of length `n+1` containing the polynomial coefficients in descending powers:

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

`[p,s] = polyfit(x,y,n)` returns the polynomial coefficients `p` and a structure `S` for use with `polyval` to obtain error estimates or predictions. If the errors in the data `Y` are independent normal with constant variance; `polyval` will produce error bounds that contain at least 50% of the predictions.

Examples This example involves fitting the error function, `erf(x)`, by a polynomial in `x`. This is a risky project because `erf(x)` is a bounded function, while polynomials are unbounded, so the fit might not be very good.

First generate a vector of `x`-points, equally spaced in the interval $[0, 2.5]$; then evaluate `erf(x)` at those points.

```
x = (0: 0.1: 2.5)';
y = erf(x);
```

The coefficients in the approximating polynomial of degree 6 are

```
p = polyfit(x,y,6)
p =
 0.0084  -0.0983   0.4217   -0.7435  0.1471   1.1064  0.0004
```

There are seven coefficients and the polynomial is

To see how good the fit is, evaluate the polynomial at the data points with

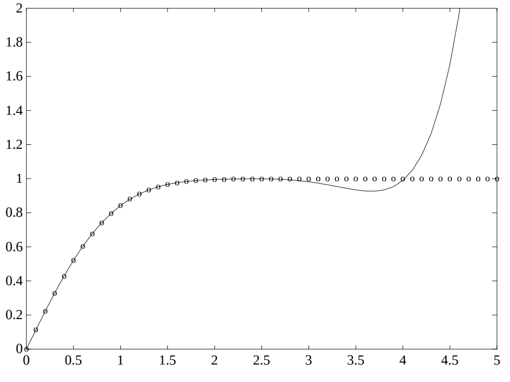
```
f = polyval(p,x);
```

A table showing the data, fit, and error is

```
table = [x y f y-f]
table =
   0          0          0.0004    -0.0004
   0.1000     0.1125     0.1119     0.0006
   0.2000     0.2227     0.2223     0.0004
   0.3000     0.3286     0.3287    -0.0001
   0.4000     0.4284     0.4288    -0.0004
   ...
   2.1000     0.9970     0.9969     0.0001
   2.2000     0.9981     0.9982    -0.0001
   2.3000     0.9989     0.9991    -0.0003
   2.4000     0.9993     0.9995    -0.0002
   2.5000     0.9996     0.9994     0.0002
```

So, on this interval, the fit is good to between three and four digits. Beyond this interval the graph shows that the polynomial behavior takes over and the approximation quickly deteriorates.

```
x = (0: 0.1: 5)';
y = erf(x);
f = polyval(p,x);
plot(x,y,'o',x,f,'-')
axis([0  5  0  2])
```



Algorithm

The M-file forms the Vandermonde matrix, V, whose elements are powers of x.

It then uses the backslash operator, \, to solve the least squares problem

$$V_p \cong y$$

The M-file can be modified to use other functions of x as the basis functions.

See Also

`polyval`	Polynomial evaluation
`roots`	Polynomial roots

polyval

Purpose

Polynomial evaluation

Syntax

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

Description

`y = polyval(p,x)` returns the value of the polynomial `p` evaluated at `x`. Polynomial `p` is a vector whose elements are the coefficients of a polynomial in descending powers.

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

`[y,delta] = polyval(p,x,S)` uses the optional output structure `S` generated by `polyfit` to generate error estimates, `y±delta`. If the errors in the data input to `polyfit` are independent normal with constant variance, `y±delta` contains at least 50% of the predictions.

Remarks

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

Examples

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

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

which results in

```
ans =

     86   162   262
```

For another example, see `polyfit`.

See Also

`polyfit`	Polynomial curve fitting
`polyvalm`	Matrix polynomial evaluation

Purpose Matrix polynomial evaluation

Syntax

```
Y = polyvalm(p,X)
```

Description

`Y = polyvalm(p,X)` evaluates a polynomial in a matrix sense. This is the same as substituting matrix `X` in the polynomial `p`.

Polynomial `p` is a vector whose elements are the coefficients of a polynomial in descending powers, and `X` must be a square matrix.

Examples

The Pascal matrices are formed from Pascal's triangle of binomial coefficients. Here is the Pascal matrix of order 4.

```
X = pascal(4)
X =
    1    1    1    1
    1    2    3    4
    1    3    6   10
    1    4   10   20
```

Its characteristic polynomial can be generated with the `poly` function.

```
p = poly(X)
p =
    1    –29    72    –29    1
```

This represents the polynomial $x^4 - 29x^3 + 72x^2 - 29x + 1$.

Pascal matrices have the curious property that the vector of coefficients of the characteristic polynomial is palindromic; it is the same forward and backward.

Evaluating this polynomial at each element is not very interesting.

```
polyval(p,X)
ans =
    16       16       16       16
    16       15     –140     –563
    16     –140    –2549   –12089
    16     –563   –12089   –43779
```

But evaluating it in a matrix sense is interesting.

```
polyvalm(p,X)
ans =
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

The result is the zero matrix. This is an instance of the Cayley-Hamilton theorem: a matrix satisfies its own characteristic equation.

See Also

`polyfit`	Polynomial curve fitting
`polyval`	Polynomial evaluation

Purpose

Base 2 power and scale floating-point numbers

Syntax

```
X = pow2(Y)
X = pow2(F,E)
```

Description

`X = pow2(Y)` returns an array `X` whose elements are 2 raised to the power `Y`.

`X = pow2(F,E)` computes $x = f \cdot 2^e$ for corresponding elements of `F` and `E`.The result is computed quickly by simply adding `E` to the floating-point exponent of `F`. Arguments `F` and `E` are real and integer arrays, respectively.

Remarks

This function corresponds to the ANSI C function `ldexp()` and the IEEE floating-point standard function `scalbn()`.

Examples

For IEEE arithmetic, the statement `X = pow2(F,E)` yields the values:

```
F          E         X
1/2        1         1
pi/4       2         pi
-3/4       2         -3
1/2        -51       eps
1-eps/2    1024      realmax
1/2        -1021     realmin
```

See Also

`log2`	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
`^`	Matrix power
`.^`	Array power
`exp`	Exponential
`hex2num`	Hexadecimal to double number conversion
`realmax`	Largest positive floating-point number
`realmin`	Smallest positive floating-point number

primes

Purpose

Generate list of prime numbers

Syntax

```
p = primes(n)
```

Description

`p = primes(n)` returns a row vector of the prime numbers less than or equal to `n`. A prime number is one that has no factors other than `1` and itself.

Examples

```
p = primes(37)

p =

     2     3     5     7    11    13    17    19    23    29    31    37
```

See Also

`factor`	Prime factors

Purpose Product of array elements

Syntax

```
B = prod(A)
B = prod(A,dim)
```

Description

`B = prod(A)` returns the products along different dimensions of an array.

If `A` is a vector, `prod(A)` returns the product of the elements.

If `A` is a matrix, `prod(A)` treats the columns of `A` as vectors, returning a row vector of the products of each column.

If `A` is a multidimensional array, `prod(A)` treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

`B = prod(A,`*`dim`*`)` takes the products along the dimension of `A` specified by scalar *dim*.

Examples

The magic square of order 3 is

```
M = magic(3)
M =
    8    1    6
    3    5    7
    4    9    2
```

The product of the elements in each column is

```
prod(M) =
    96    45    84
```

The product of the elements in each row can be obtained by:

```
prod(M,2) =
     48
    105
     72
```

See Also

`cumprod`	Cumulative product
`diff`	Difference
`sum`	Sum of array elements

profile

Purpose

Measure and display M-file execution profiles

Syntax

```
profile function
profile report
profile report n
profile report frac
profile on
profile off
profile done
profile reset
info = profile
```

Description

The profiler utility helps you debug and optimize M-files by tracking the cumulative execution time of each line of code. The utility creates a vector of “bins,” one bin for every line of code in the M-file being profiled. As MATLAB executes the M-file code, the profiler updates each bin with running counts of the time spent executing the corresponding line.

`profile` *function* starts the profiler for *function*. *function* must be the name of an M-file function or a `MATLABPATH` relative partial pathname.

`profile report` displays a profile summary report for the M-file currently being profiled.

`profile report n`, where `n` is an integer, displays a report showing the `n` lines that take the most time.

`profile report frac`, where `frac` is a number between 0.0 and 1.0, displays a report of each line that accounts for more than `frac` of the total time.

`profile on` and `profile off` enable and disable profiling, respectively.

`profile done` turns off the profiler and clears its data.

`profile reset` erases the bin contents without disabling profiling or changing the M-file under inspection.

`info = profile` returns a structure with the fields:

`file`	Full path to the function being profiled.
`function`	Name of function being profiled.
`interval`	Sampling interval in seconds.
`count`	Vector of sample counts
`state`	`on` if the profiler is running and `off` otherwise.

Remarks

You can also profile built-in functions. The profiler tracks the number of intervals in which the built-in function was called (an estimate of how much time was spent executing the built-in function).

The profiler's behavior is defined by root object properties and can be manipulated using the `set` and `get` commands. See the *Applying MATLAB* for more details.

Limitations

The profiler utility can accommodate only one M-file at a time.

See Also

See also `partialpath`.

Purpose

Quasi-Minimal Residual method

Syntax

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

Description

`x = qmr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The coefficient matrix `A` must be square and the right hand side (column) vector `b` must have length `n`, where `A` is `n`-by-`n`. `qmr` will start iterating from an initial estimate that by default is an all zero vector of length `n`. Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate `x` has relative residual `norm(b-A*x)/norm(b)` less than or equal to the tolerance of the method. The default tolerance is `1e-6`. The default maximum number of iterations is the minimum of `n` and 20. No preconditioning is used.

`qmr(A,b,tol)` specifies the tolerance of the method, `tol`.

`qmr(A,b,tol,maxit)` additionally specifies the maximum number of iterations, `maxit`.

`qmr(A,b,tol,maxit,M1)` and `qmr(A,b,tol,maxit,M1,M2)` use left and right preconditioners `M1` and `M2` and effectively solve the system `inv(M1)*A*inv(M2)*y = inv(M1)*b` for `y`, where `x = inv(M2)*y`. If `M1` or `M2` is given as the empty matrix (`[]`), it is considered to be the identity matrix, equivalent to no preconditioning at all. Since systems of equations of the form `M1*y = r` are solved using backslash within `qmr`, it is wise to factor preconditioners into their LU factorizations first. For example, replace `qmr(A,b,tol,maxit,M,[])` or `qmr(A,b,tol,maxit,[],M)` with:

```
[M1,M2] = lu(M);
qmr(A,b,tol,maxit,M1,M2).
```

`qmr(A,b,tol,maxit,M1,M2,x0)` specifies the initial estimate `x0`. If `x0` is given as the empty matrix (`[]`), the default all zero vector is used.

`x = qmr(A,b,tol,maxit,M1,M2,x0)` returns a solution `x`. If `qmr` converged, a message to that effect is displayed. If `qmr` failed to converge after the maximum number of iterations or halted 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.

`[x,flag] = qmr(A,b,tol,maxit,M1,M2,x0)` returns a solution `x` and a flag which describes the convergence of `qmr`:

Flag	Convergence
0	`qmr` converged to the desired tolerance `tol` within `maxit` iterations without failing for any reason.
1	`qmr` iterated `maxit` times but did not converge.
2	One of the systems of equations of the form `M*y = r` involving one of the preconditioners was ill-conditioned and did not return a useable result when solved by `\` (backslash).
3	The method stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `qmr` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = qmr(A,b,tol,maxit,M1,M2,x0)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, then `relres` ≤ `tol`.

`[x,flag,relres,iter] = qmr(A,b,tol,maxit,M1,M2,x0)` also returns the iteration number at which `x` was computed. This always satisfies `0` ≤ `iter` ≤ `maxit`.

`[x,flag,relres,iter,resvec] = qmr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of the residual norms at each iteration, starting from `resvec(1) = norm(b-A*x0)`. If `flag` is `0`, `resvec` is of length `iter+1` and `resvec(end)` ≤ `tol*norm(b)`.

Examples

```
load west0479
A = west0479
b = sum(A,2)
[x,flag] = qmr(A,b)
```

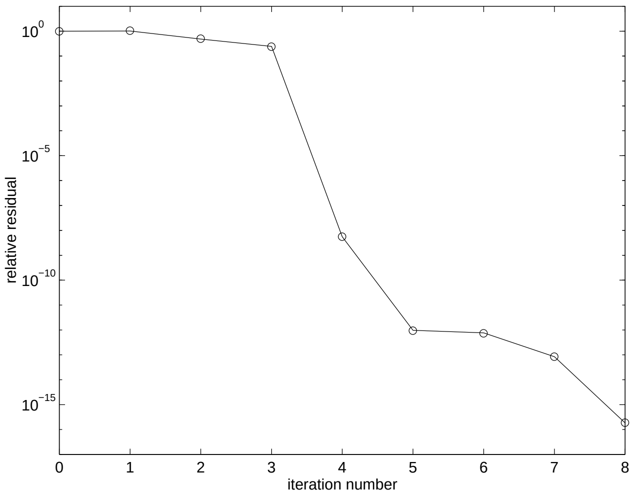
`flag` is `1` since `qmr` will not converge to the default tolerance `1e-6` within the default 20 iterations.

```
[L1,U1] = luinc(A,1e-5)
[x1,flag1] = qmr(A,b,1e-6,20,L1,U1)
```

`flag1` is `2` since the upper triangular `U1` has a zero on its diagonal so `qmr` fails in the first iteration when it tries to solve a system such as `U1*y = r` for `y` with backslash.

```
[L2,U2] = luinc(A,1e-6)
[x2,flag2,relres2,iter2,resvec2] = qmr(A,b,1e-15,10,L2,U2)
```

`flag2` is `0` since `qmr` will converge to the tolerance of `1.9e-16` (the value of `relres2`) at the eighth iteration (the value of `iter2`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. `resvec2(1) = norm(b)` and `resvec2(9) = norm(b-A*x2)`. You may follow the progress of `qmr` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0) with `semilogy(0:iter2,resvec2/norm(b),'-o')`.



See Also

`bicg`	BiConjugate Gradients method
`bicgstab`	BiConjugate Gradients Stabilized method
`cgs`	Conjugate Gradients Squared method
`gmres`	Generalized Minimum Residual method (with restarts)
`luinc`	Incomplete LU matrix factorizations
`pcg`	Preconditioned Conjugate Gradients method
`\`	Matrix left division

References

Freund, Roland W. and Nöel M. Nachtigal, *QMR: A quasi-minimal residual method for non-Hermitian linear systems*, Journal: Numer. Math. 60, 1991, pp. 315-339

Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM, Philadelphia, 1994.

Purpose

Orthogonal-triangular decomposition

Syntax

```
[Q,R] = qr(X)
[Q,R,E] = qr(X)
[Q,R] = qr(X,0)
[Q,R,E] = qr(X,0)
A = qr(X)
```

Description

The qr function performs the orthogonal-triangular decomposition of a matrix. This factorization is useful for both square and rectangular matrices. It expresses the matrix as the product of a real orthonormal or complex unitary matrix and an upper triangular matrix.

`[Q,R] = qr(X)` produces an upper triangular matrix R of the same dimension as X and a unitary matrix Q so that `X = Q*R`.

`[Q,R,E] = qr(X)` produces a permutation matrix E, an upper triangular matrix R with decreasing diagonal elements, and a unitary matrix Q so that `X*E = Q*R`. The column permutation E is chosen so that `abs(diag(R))` is decreasing.

`[Q,R] = qr(X,0)` and `[Q,R,E] = qr(X,0)` produce "economy-size" decompositions in which E is a permutation vector, so that `Q*R = X(:,E)`. The column permutation E is chosen so that `abs(diag(R))` is decreasing.

`A = qr(X)` returns the output of the LINPACK subroutine ZQRDC. `triu(qr(X))` is R.

Examples

Start with

```
A =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

This is a rank-deficient matrix; the middle column is the average of the other two columns. The rank deficiency is revealed by the factorization:

```
[Q,R] = qr(A)
Q =
    -0.0776    -0.8331     0.5444     0.0605
    -0.3105    -0.4512    -0.7709     0.3251
    -0.5433    -0.0694    -0.0913    -0.8317
    -0.7762     0.3124     0.3178     0.4461
R =
    -12.8841    -14.5916    -16.2992
           0     -1.0413     -2.0826
           0           0      0.0000
           0           0           0
```

The triangular structure of R gives it zeros below the diagonal; the zero on the diagonal in R(3,3) implies that R, and consequently A, does not have full rank.

The QR factorization is used to solve linear systems with more equations than unknowns. For example

```
b =
    1
    3
    5
    7
```

The linear system $Ax = b$ represents four equations in only three unknowns. The best solution in a least squares sense is computed by

```
x = A\b
```

which produces

```
Warning: Rank deficient, rank = 2, tol = 1.4594E-014
x =
    0.5000
         0
    0.1667
```

The quantity `tol` is a tolerance used to decide if a diagonal element of `R` is negligible. If `[Q,R,E] = qr(A)`, then

```
tol = max(size(A))*eps*abs(R(1,1))
```

The solution `x` was computed using the factorization and the two steps

```
y = Q'*b;
x = R\y
```

The computed solution can be checked by forming Ax. This equals b to within roundoff error, which indicates that even though the simultaneous equations $Ax = b$ are overdetermined and rank deficient, they happen to be consistent. There are infinitely many solution vectors `x`; the QR factorization has found just one of them.

Algorithm

The `qr` function uses the LINPACK routines `ZQRDC` and `ZQRSL`. `ZQRDC` computes the QR decomposition, while `ZQRSL` applies the decomposition.

See Also

`\`	Matrix left division (backslash)
`/`	Matrix right division (slash)
`lu`	LU matrix factorization
`null`	Null space of a matrix
`orth`	Range space of a matrix
`qrdelete`	Delete column from QR factorization
`qrinsert`	Insert column in QR factorization

References

Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose

Delete column from QR factorization

Syntax

```
[Q,R] = qrdelete(Q,R,j)
```

Description

`[Q,R] = qrdelete(Q,R,j)` changes `Q` and `R` to be the factorization of the matrix `A` with its jth column, `A(:,j)`, removed.

Inputs `Q` and `R` represent the original QR factorization of matrix `A`, as returned by the statement `[Q,R] = qr(A)`. Argument `j` specifies the column to be removed from matrix `A`.

Algorithm

The `qrdelete` function uses a series of Givens rotations to zero out the appropriate elements of the factorization.

See Also

`qr`	Orthogonal-triangular decomposition
`qrinsert`	Insert column in QR factorization

qrinsert

Purpose Insert column in QR factorization

Syntax

```
[Q,R] = qrinsert(Q,R,j,x)
```

Description

`[Q,R] = qrinsert(Q,R,j,x)` changes `Q` and `R` to be the factorization of the matrix obtained by inserting an extra column, `x`, before `A(:,j)`. If `A` has `n` columns and `j = n+1`, then `qrinsert` inserts `x` after the last column of `A`.

Inputs `Q` and `R` represent the original QR factorization of matrix `A`, as returned by the statement `[Q,R] = qr(A)`. Argument `x` is the column vector to be inserted into matrix `A`. Argument `j` specifies the column before which `x` is inserted.

Algorithm

The `qrinsert` function inserts the values of `x` into the jth column of `R`. It then uses a series of Givens rotations to zero out the nonzero elements of `R` on and below the diagonal in the jth column.

See Also

`qr`	Orthogonal-triangular decomposition
`qrdelete`	Delete column from QR factorization

Purpose Numerical evaluation of integrals

Syntax

```
q = quad('fun',a,b)
q = quad('fun',a,b,tol)
q = quad('fun',a,b,tol,trace)
q = quad('fun',a,b,tol,trace,P1,P2,...)
q = quad8(...)
```

Description *Quadrature* is a numerical method of finding the area under the graph of a function, that is, computing a definite integral.

$$q = \int_a^b f(x)dx$$

`q = quad('fun',a,b)` returns the result of numerically integrating *'fun'* between the limits `a` and `b`. *'fun'* must return a vector of output values when given a vector of input values.

`q = quad('fun',a,b,tol)` iterates until the relative error is less than `tol`. The default value for `tol` is `1.e-3`. Use a two element tolerance vector, `tol = [rel_tol abs_tol]`, to specify a combination of relative and absolute error.

`q = quad('fun',a,b,tol,trace)` integrates to a relative error of `tol`, and for non-zero `trace`, plots a graph showing the progress of the integration.

`q = quad('fun',a,b,tol,trace,P1,P2,...)` allows coefficients `P1`, `P2`, ... to be passed directly to the specified function: `G = fun(X,P1,P2,...)`. To use default values for `tol` or `trace`, pass in the empty matrix, for example: `quad('fun',a,b,[],[],P1)`.

Remarks `quad8`, a higher-order method, has the same calling sequence as `quad`.

Examples Integrate the sine function from 0 to π:

```
a = quad('sin',0,pi)
a =
    2.0000
```

Algorithm

quad and quad8 implement two different quadrature algorithms. quad implements a low order method using an adaptive recursive Simpson's rule. quad8 implements a higher order method using an adaptive recursive Newton-Cotes 8 panel rule. quad8 is better than quad at handling functions with soft singularities, for example:

$$\int_0^1 \sqrt{x}\ dx$$

Diagnostics

quad and quad8 have recursion level limits of 10 to prevent infinite recursion for a singular integral. Reaching this limit in one of the integration intervals produces the warning message:

```
Recursion level limit reached in quad. Singularity likely.
```

and sets `q = inf`.

Limitations

Neither quad nor quad8 is set up to handle integrable singularities, such as:

$$\int_0^1 \frac{1}{\sqrt{x}}\ dx$$

If you need to evaluate an integral with such a singularity, recast the problem by transforming the problem into one in which you can explicitly evaluate the integrable singularities and let quad or quad8 take care of the remainder.

References

[1] Forsythe, G.E., M.A. Malcolm and C.B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1977.

Purpose Terminate MATLAB

Syntax

```
quit
```

Description `quit` terminates MATLAB without saving the workspace. To save your workspace variables, use the `save` command before quitting.

See Also

`save`	Save workspace variables on disk
`startup`	MATLAB startup M-file

qz

Purpose

QZ factorization for generalized eigenvalues

Syntax

```
[AA,BB,Q,Z,V] = qz(A,B)
```

Description

The qz function gives access to what are normally only intermediate results in the computation of generalized eigenvalues.

`[AA,BB,Q,Z,V] = qz(A,B)` produces upper triangular matrices `AA` and `BB`, and matrices `Q` and `Z` containing the products of the left and right transformations, such that

```
Q*A*Z = AA
Q*B*Z = BB
```

The `qz` function also returns the generalized eigenvector matrix `V`.

The generalized eigenvalues are the diagonal elements of `AA` and `BB` so that

```
A*V*diag(BB) = B*V*diag(AA)
```

Arguments

`A,B`	Square matrices.
`AA,BB`	Upper triangular matrices.
`Q,Z`	Transformation matrices.
`V`	Matrix whose columns are eigenvectors.

Algorithm

Complex generalizations of the EISPACK routines `QZHES`, `QZIT`, `QZVAL`, and `QZVEC` implement the QZ algorithm.

See Also

`eig`	Eigenvalues and eigenvectors

References

[1] Moler, C. B. and G.W. Stewart, “An Algorithm for Generalized Matrix Eigenvalue Problems”, *SIAM J. Numer. Anal.*, Vol. 10, No. 2, April 1973.

Purpose

Uniformly distributed random numbers and arrays

Syntax

```
Y = rand(n)
Y = rand(m,n)
Y = rand([m n])
Y = rand(m,n,p,...)
Y = rand([m n p...])
Y = rand(size(A))
rand
s = rand('state')
```

Description

The `rand` function generates arrays of random numbers whose elements are uniformly distributed in the interval (`0`,`1`).

`Y = rand(n)` returns an `n`-by-`n` matrix of random entries. An error message appears if `n` is not a scalar.

`Y = rand(m,n)` or `Y = rand([m n])` returns an `m`-by-`n` matrix of random entries.

`Y = rand(m,n,p,...)` or `Y = rand([m n p...])` generates random arrays.

`Y = rand(size(A))` returns an array of random entries that is the same size as `A`.

`rand`, by itself, returns a scalar whose value changes each time it's referenced.

`s = rand('state')` returns a 35-element vector containing the current state of the uniform generator. To change the state of the generator:

`rand('state',s)`	Resets the state to `s`.
`rand('state',0)`	Resets the generator to its initial state.
`rand('state',j)`	For integer `j`, resets the generator to its `j`-th state.
`rand('state',sum(100*clock))`	Resets it to a different state each time.

Remarks

MATLAB 5 uses a new multiseed random number generator that can generate all the floating-point numbers in the closed interval $[2^{-53}, 1-2^{-53}]$. Theoretically, it can generate over 2^{1492} values before repeating itself. MATLAB 4 used random number generators with a single seed. `rand('seed',0)` and `rand('seed',j)` use the MATLAB 4 generator. `rand('seed')` returns the current seed of the MATLAB 4 uniform generator. `rand('state',j)` and `rand('state',s)` use the MATLAB 5 generator.

Examples

`R = rand(3,4)` may produce

```
R =
   0.2190    0.6793    0.5194    0.0535
   0.0470    0.9347    0.8310    0.5297
   0.6789    0.3835    0.0346    0.6711
```

This code makes a random choice between two equally probable alternatives.

```
if rand < .5
   'heads'
else
   'tails'
end
```

See Also

`randn`	Normally distributed random numbers and arrays
`randperm`	Random permutation
`sprand`	Sparse uniformly distributed random matrix
`sprandn`	Sparse normally distributed random matrix

Purpose

Normally distributed random numbers and arrays

Syntax

```
Y = randn(n)
Y = randn(m,n)
Y = randn([m n])
Y = randn(m,n,p,...)
Y = randn([m n p...])
Y = randn(size(A))
randn
s = randn('state')
```

Description

The `randn` function generates arrays of random numbers whose elements are normally distributed with mean 0 and variance 1.

`Y = randn(n)` returns an `n`-by-`n` matrix of random entries. An error message appears if `n` is not a scalar.

`Y = randn(m,n)` or `Y = randn([m n])` returns an `m`-by-`n` matrix of random entries.

`Y = randn(m,n,p,...)` or `Y = randn([m n p...])` generates random arrays.

`Y = randn(size(A))` returns an array of random entries that is the same size as `A`.

`randn`, by itself, returns a scalar whose value changes each time it's referenced.

`s = randn('state')` returns a 2-element vector containing the current state of the normal generator. To change the state of the generator:

`randn('state',s)`	Resets the state to `s`.
`randn('state',0)`	Resets the generator to its initial state.

`randn('state',j)`	For integer `j`, resets the generator to its `j`th state.
`randn('state',sum(100*clock))`	Resets it to a different state each time.

Remarks

MATLAB 5 uses a new multiseed random number generator that can generate all the floating-point numbers in the closed interval $[2^{-53}, 1-2^{-53}]$. Theoretically, it can generate over 2^{1492} values before repeating itself. MATLAB 4 used random number generators with a single seed. `randn('seed',0)` and `randn('seed',j)` use the MATLAB 4 generator. `randn('seed')` returns the current seed of the MATLAB 4 normal generator. `randn('state',j)` and `randn('state',s)` use the MATLAB 5 generator.

Examples

`R = randn(3,4)` may produce

```
R =
   1.1650    0.3516    0.0591    0.8717
   0.6268   -0.6965    1.7971   -1.4462
   0.0751    1.6961    0.2641   -0.7012
```

For a histogram of the `randn` distribution, see `hist`.

See Also

`rand`	Uniformly distributed random numbers and arrays
`randperm`	Random permutation
`sprand`	Sparse uniformly distributed random matrix
`sprandn`	Sparse normally distributed random matrix

randperm

Purpose Random permutation

Syntax `p = randperm(n)`

Description `p = randperm(n)` returns a random permutation of the integers `1:n`.

Remarks The `randperm` function calls `rand` and therefore changes `rand`'s seed value.

Examples `randperm(6)` might be the vector

```
[3  2  6  4  1  5]
```

or it might be some other permutation of `1:6`.

See Also

`permute`	Rearrange the dimensions of a multidimensional array

Purpose Rank of a matrix

Syntax

```
k = rank(A)
k = rank(A,tol)
```

Description The `rank` function provides an estimate of the number of linearly independent rows or columns of a matrix.

`k = rank(A)` returns the number of singular values of `A` that are larger than the default tolerance, `max(size(A))*norm(A)*eps`.

`k = rank(A,tol)` returns the number of singular values of `A` that are larger than `tol`.

Algorithm There are a number of ways to compute the rank of a matrix. MATLAB uses the method based on the singular value decomposition, or SVD, described in Chapter 11 of the *LINPACK Users' Guide*. The SVD algorithm is the most time consuming, but also the most reliable.

The `rank` algorithm is

```
s = svd(A);
tol = max(size(A))*s(1)*eps;
r = sum(s > tol);
```

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

rat, rats

Purpose Rational fraction approximation

Syntax

```
[N,D] = rat(X)
[N,D] = rat(X,tol)
rat(...)
S = rats(X,strlen)
S = rats(X)
```

Description Even though all floating-point numbers are rational numbers, it is sometimes desirable to approximate them by simple rational numbers, which are fractions whose numerator and denominator are small integers. The `rat` function attempts to do this. Rational approximations are generated by truncating continued fraction expansions. The `rats` function calls `rat`, and returns strings.

`[N,D] = rat(X)` returns arrays `N` and `D` so that `N./D` approximates `X` to within the default tolerance, `1.e–6*norm(X(:),1)`.

`[N,D] = rat(X,tol)` returns `N./D` approximating `X` to within `tol`.

`rat(X)`, with no output arguments, simply displays the continued fraction.

`S = rats(X,strlen)` returns a string containing simple rational approximations to the elements of `X`. Asterisks are used for elements that cannot be printed in the allotted space, but are not negligible compared to the other elements in `X`. `strlen` is the length of each string element returned by the `rats` function. The default is `strlen = 13`, which allows 6 elements in 78 spaces.

`S = rats(X)` returns the same results as those printed by MATLAB with `format rat`.

Examples Ordinarily, the statement

```
s = 1 – 1/2 + 1/3 – 1/4 + 1/5 – 1/6 + 1/7
```

produces

```
s =
   0.7595
```

However, with

```
format rat
```

or with

```
rats(s)
```

the printed result is

```
s =
   319/420
```

This is a simple rational number. Its denominator is 420, the least common multiple of the denominators of the terms involved in the original expression. Even though the quantity `s` is stored internally as a binary floating-point number, the desired rational form can be reconstructed.

To see how the rational approximation is generated, the statement `rat(s)`

produces

```
1 + 1/(-4 + 1/(-6 + 1/(-3 + 1/(-5))))
```

And the statement

```
[n,d] = rat(s)
```

produces

```
n = 319, d = 420
```

The mathematical quantity π is certainly not a rational number, but the MATLAB quantity `pi` that approximates it is a rational number. With IEEE floating-point arithmetic, `pi` is the ratio of a large integer and 2^{52}:

```
14148475504056880/4503599627370496
```

However, this is not a simple rational number. The value printed for `pi` with `format rat`, or with `rats(pi)`, is

```
355/113
```

This approximation was known in Euclid's time. Its decimal representation is

```
3.14159292035398
```

and so it agrees with `pi` to seven significant figures. The statement

```
rat(pi)
```

produces

```
3 + 1/(7 + 1/(16))
```

This shows how the `355/113` was obtained. The less accurate, but more familiar approximation `22/7` is obtained from the first two terms of this continued fraction.

Algorithm

The `rat(X)` function approximates each element of `X` by a continued fraction of the form:

$$\frac{n}{d} = d_1 + \cfrac{1}{d_2 + \cfrac{1}{\left(d_3 + \ldots + \frac{1}{d_k}\right)}}$$

The *d*'s are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when `X = sqrt(2)`. For `x = sqrt(2)`, the error with `k` terms is about `2.68*(.173)^k`, so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.

Purpose Matrix reciprocal condition number estimate

Syntax

```
c = rcond(A)
```

Description **`c = rcond(A)`** returns an estimate for the reciprocal of the condition of `A` in 1-norm using the LINPACK condition estimator. If `A` is well conditioned, `rcond(A)` is near 1.0. If `A` is badly conditioned, `rcond(A)` is near 0.0. Compared to `cond`, `rcond` is a more efficient, but less reliable, method of estimating the condition of a matrix.

Algorithm The `rcond` function uses the condition estimator from the LINPACK routine `ZGECO`.

See Also

`cond`	Condition number with respect to inversion
`condest`	1-norm matrix condition number estimate
`norm`	Vector and matrix norms
`normest`	2-norm estimate
`rank`	Rank of a matrix
`svd`	Singular value decomposition

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose Real part of complex number

Syntax `X = real(Z)`

Description `X = real(Z)` returns the real part of the elements of the complex array `Z`.

Examples `real(2+3*i)` is `2`.

See Also

`abs`	Absolute value and complex magnitude
`angle`	Phase angle
`conj`	Complex conjugate
`i`, `j`	Imaginary unit ($\sqrt{-1}$)
`imag`	Imaginary part of a complex number

Purpose Largest positive floating-point number

Syntax

```
n = realmax
```

Description `n = realmax` returns the largest floating-point number representable on a particular computer. Anything larger overflows.

Examples On machines with IEEE floating-point format, `realmax` is one bit less than 2^{1024} or about `1.7977e+308`.

Algorithm The `realmax` function is equivalent to `pow2(2-eps,maxexp)`, where `maxexp` is the largest possible floating-point exponent.

Execute `type realmax` to see `maxexp` for various computers.

See Also

`eps`	Floating-point relative accuracy
`realmin`	Smallest positive floating-point number

realmin

Purpose

Smallest positive floating-point number

Syntax

```
n = realmin
```

Description

`n = realmin` returns the smallest positive normalized floating-point number on a particular computer. Anything smaller underflows or is an IEEE “denormal.”

Examples

On machines with IEEE floating-point format, `realmin` is `2^(-1022)` or about `2.2251e-308`.

Algorithm

The `realmin` function is equivalent to `pow2(1,minexp)` where `minexp` is the smallest possible floating-point exponent.

Execute `type realmin` to see `minexp` for various computers.

See Also

`eps`	Floating-point relative accuracy
`realmax`	Largest positive floating-point number

Purpose Remainder after division

Syntax

```
R = rem(X,Y)
```

Description `R = rem(X,Y)` returns `X - fix(X./Y).*Y`, where `fix(X./Y)` is the integer part of the quotient, `X./Y`.

Remarks So long as operands `X` and `Y` are of the same sign, the statement `rem(X,Y)` returns the same result as does `mod(X,Y)`. However, for positive `X` and `Y`,

```
rem(-x,y) = mod(-x,y)-y
```

The `rem` function returns a result that is between 0 and `sign(X)*abs(Y)`. If `Y` is zero, `rem` returns `NaN`.

Limitations Arguments `X` and `Y` should be integers. Due to the inexact representation of floating-point numbers on a computer, real (or complex) inputs may lead to unexpected results.

See Also

`mod`	Modulus (signed remainder after division)

repmat

Purpose

Replicate and tile an array

Syntax

```
B = repmat(A,m,n)
B = repmat(A,[m n])
B = repmat(A,[m n p...])
repmat(A,m,n)
```

Description

`B = repmat(A,m,n)` creates a large matrix `B` consisting of an `m`-by-`n` tiling of copies of `A`. The statement `repmat(A,n)` creates an `n`-by-`n` tiling.

`B = repmat(A,[m n])` accomplishes the same result as `repmat(A,m,n)`.

`B = repmat(A,[m n p...])` produces a multidimensional (`m`-by-`n`-by-`p`-by-...) array composed of copies of `A`. `A` may be multidimensional.

`repmat(A,m,n)` when `A` is a scalar, produces an `m`-by-`n` matrix filled with `A`'s value. This is much faster than `a*ones(m,n)`.

Examples

In this example, `repmat` replicates 12 copies of the second-order identity matrix, resulting in a "checkerboard" pattern.

```
B = repmat(eye(2),3,4)

B =
     1     0     1     0     1     0     1     0
     0     1     0     1     0     1     0     1
     1     0     1     0     1     0     1     0
     0     1     0     1     0     1     0     1
     1     0     1     0     1     0     1     0
     0     1     0     1     0     1     0     1
```

The statement `N = repmat(NaN,[2 3])` creates a 2-by-3 matrix of `NaN`s.

Purpose Reshape array

Syntax

```
B = reshape(A,m,n)
B = reshape(A,m,n,p,...)
B = reshape(A,[m n p...])
B = reshape(A,siz)
```

Description

`B = reshape(A,m,n)` returns the `m`-by-`n` matrix `B` whose elements are taken column-wise from `A`. An error results if `A` does not have `m*n` elements.

`B = reshape(A,m,n,p,...)` or `B = reshape(A,[m n p...])` returns an N-D array with the same elements as `X` but reshaped to have the size `m`-by-`n`-by-`p`-by-... . `m*n*p*`... must be the same as `prod(size(x))`.

`B = reshape(A,siz)` returns an N-D array with the same elements as `A`, but reshaped to `siz`, a vector representing the dimensions of the reshaped array. The quantity `prod(siz)` must be the same as `prod(size(A))`.

Examples

Reshape a 3-by-4 matrix into a 2-by-6 matrix:

```
A =
    1    4    7    10
    2    5    8    11
    3    6    9    12

B = reshape(A,2,6)

B =
    1    3    5    7     9    11
    2    4    6    8    10    12
```

See Also

`:` (colon)	Colon :
`shiftdim`	Shift dimensions
`squeeze`	Remove singleton dimensions

residue

Purpose

Convert between partial fraction expansion and polynomial coefficients

Syntax

```
[r,p,k] = residue(b,a)
[b,a] = residue(r,p,k)
```

Description

The `residue` function converts a quotient of polynomials to pole-residue representation, and back again.

`[r,p,k] = residue(b,a)` finds the residues, poles, and direct term of a partial fraction expansion of the ratio of two polynomials, *b(s)* and *a(s)*, of the form:

$$\frac{b(s)}{a(s)} = \frac{b_1 + b_2 s^{-1} + b_3 s^{-2} + \ldots + b_{m+1} s^{-m}}{a_1 + a_2 s^{-1} + a_3 s^{-2} + \ldots + a_{n+1} s^{-n}}$$

`[b,a] = residue(r,p,k)` converts the partial fraction expansion back to the polynomials with coefficients in `b` and `a`.

Definition

If there are no multiple roots, then:

$$\frac{b(s)}{a(s)} = \frac{r_1}{s - p_1} + \frac{r_2}{s - p_2} + \ldots + \frac{r_n}{s - p_n} + k(s)$$

The number of poles `n` is

```
n = length(a)-1 = length(r) = length(p)
```

The direct term coefficient vector is empty if `length(b) < length(a)`; otherwise

```
length(k) = length(b)-length(a)+1
```

If `p(j) = ... = p(j+m-1)` is a pole of multiplicity `m`, then the expansion includes terms of the form

$$\frac{r_j}{s - p_j} + \frac{r_{j+1}}{(s - p_j)^2} + \ldots + \frac{r_{j+m-1}}{(s - p_j)^m}$$

Arguments

`b,a`	Vectors that specify the coefficients of the polynomials in descending powers of *s*
`r`	Column vector of residues
`p`	Column vector of poles
`k`	Row vector of direct terms

Algorithm

The `residue` function is an M-file. It first obtains the poles with `roots`. Next, if the fraction is nonproper, the direct term `k` is found using `deconv`, which performs polynomial long division. Finally, the residues are determined by evaluating the polynomial with individual roots removed. For repeated roots, the M-file `resi2` computes the residues at the repeated root locations.

Limitations

Numerically, the partial fraction expansion of a ratio of polynomials represents an ill-posed problem. If the denominator polynomial, *a(s)*, is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can make arbitrarily large changes in the resulting poles and residues. Problem formulations making use of state-space or zero-pole representations are preferable.

See Also

`deconv`	Deconvolution and polynomial division
`poly`	Polynomial with specified roots
`roots`	Polynomial roots

References

[1] Oppenheim, A.V. and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975, p. 56.

return

Purpose Return to the invoking function

Syntax

```
return
```

Description `return` causes a normal return to the invoking function or to the keyboard. It also terminates keyboard mode.

Examples If the determinant function were an M-file, it might use a `return` statement in handling the special case of an empty matrix as follows:

```
function d = det(A)
%DET det(A) is the determinant of A.
if isempty(A)
    d = 1;
    return
else
    ...
end
```

See Also

`break`	Break out of flow control structures
`disp`	Display text or array
`end`	Terminate for, while, switch, and if statements or indicate last index
`error`	Display error messages
`for`	Repeat statements a specific number of times
`if`	Conditionally execute statements
`keyboard`	Invoke the keyboard in an M-file
`switch`	Switch among several cases based on expression
`while`	Repeat statements an indefinite number of times

Purpose Remove structure fields

Syntax

```
s = rmfield(s,'field')
s = rmfield(s,FIELDS)
```

Description `s = rmfield(s,'field')` removes the specified field from the structure array `s`.

`s = rmfield(s,FIELDS)` removes more than one field at a time when `FIELDS` is a character array of field names or cell array of strings.

See Also

`fields`	Field names of a structure
`getfield`	Get field of structure array
`setfield`	Set field of structure array
`strvcat`	Vertical concatenation of strings

rmpath

Purpose Remove directories from MATLAB's search path

Syntax

```
rmpath directory
```

Description

`rmpath directory` removes the specified directory from MATLAB's current search path.

Remarks The function syntax form is also acceptable:

```
rmpath('directory')
```

Examples

```
rmpath /usr/local/matlab/mytools
```

See Also

`addpath`	Add directories to MATLAB's search path
`path`	Control MATLAB's directory search path

roots

Purpose

Polynomial roots

Syntax

```
r = roots(c)
```

Description

`r = roots(c)` returns a column vector whose elements are the roots of the polynomial `c`.

Row vector `c` contains the coefficients of a polynomial, ordered in descending powers. If `c` has `n+1` components, the polynomial it represents is $c_1 s^n + \ldots + c_n s + c_{n+1}$.

Remarks

Note the relationship of this function to `p = poly(r)`, which returns a row vector whose elements are the coefficients of the polynomial. For vectors, `roots` and `poly` are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples

The polynomial $s^3 - 6s^2 - 72s - 27$ is represented in MATLAB as

```
p = [1 -6 -72 -27]
```

The roots of this polynomial are returned in a column vector by

```
r = roots(p)
r =
    12.1229
    -5.7345
    -0.3884
```

Algorithm

The algorithm simply involves computing the eigenvalues of the companion matrix:

```
A = diag(ones(n-2,1),-1);
A(1,:) = -c(2:n-1)./c(1);
eig(A)
```

It is possible to prove that the results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix `A`, but this does not mean that they are the exact roots of a polynomial with coefficients within roundoff error of those in `c`.

See Also

`fzero`	Zero of a function of one variable
`poly`	Polynomial with specified roots
`residue`	Convert between partial fraction expansion and polynomial coefficients

Purpose

Rotate matrix 90°

Syntax

```
B = rot90(A)
B = rot90(A,k)
```

Description

`B = rot90(A)` rotates matrix `A` counterclockwise by 90 degrees.

`B = rot90(A,k)` rotates matrix `A` counterclockwise by `k*90` degrees, where `k` is an integer.

Examples

The matrix

```
X =
     1     2     3
     4     5     6
     7     8     9
```

rotated by 90 degrees is

```
Y = rot90(X)
Y =
     3     6     9
     2     5     8
     1     4     7
```

See Also

`flipdim`	Flip array along a specified dimension
`fliplr`	Flip matrices left-right
`flipud`	Flip matrices up-down

round

Purpose

Round to nearest integer

Syntax

```
Y = round(X)
```

Description

`Y = round(X)` rounds the elements of `X` to the nearest integers. For complex `X`, the imaginary and real parts are rounded independently.

Examples

```
a =
  Columns 1 through 4
  -1.9000          -0.2000           3.4000           5.6000
  Columns 5 through 6
   7.0000            2.4000 + 3.6000i

round(a)

ans =
  Columns 1 through 4
  -2.0000                0           3.0000           6.0000
  Columns 5 through 6
   7.0000            2.0000 + 4.0000i
```

See Also

`ceil`	Round toward infinity
`fix`	Round towards zero
`floor`	Round towards minus infinity

Purpose

Reduced row echelon form

Syntax

```
R = rref(A)
[R,jb] = rref(A)
[R,jb] = rref(A,tol)
rrefmovie(A)
```

Description

`R = rref(A)` produces the reduced row echelon form of `A` using Gauss Jordan elimination with partial pivoting. A default tolerance of (`max(size(A))*eps *norm(A,inf)`) tests for negligible column elements.

`[R,jb] = rref(A)` also returns a vector `jb` so that:

- `r = length(jb)` is this algorithm's idea of the rank of `A`,
- `x(jb)` are the bound variables in a linear system `Ax = b`,
- `A(:,jb)` is a basis for the range of `A`,
- `R(1:r,jb)` is the `r`-by-`r` identity matrix.

`[R,jb] = rref(A,tol)` uses the given tolerance in the rank tests.

Roundoff errors may cause this algorithm to compute a different value for the rank than `rank`, `orth` and `null`.

`rrefmovie(A)` shows a movie of the algorithm working.

Examples

Use `rref` on a rank-deficient magic square:

```
A = magic(4), R = rref(A)
A =
   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1
R =
    1    0    0    1
    0    1    0    3
    0    0    1   -3
    0    0    0    0
```

See Also

`inv`	Matrix inverse
`lu`	LU matrix factorization
`rank`	Rank of a matrix

Purpose

Convert real Schur form to complex Schur form

Syntax

```
[U,T] = rsf2csf(U,T)
```

Description

The *complex Schur form* of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The *real Schur form* has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

`[U,T] = rsf2csf(U,T)` converts the real Schur form to the complex form.

Arguments `U` and `T` represent the unitary and Schur forms of a matrix `A`, respectively, that satisfy the relationships: `A = U*T*U'` and `U'*U = eye(size(A))`. See `schur` for details.

Examples

Given matrix `A`,

```
  1     1     1     3
  1     2     1     1
  1     1     3     1
 -2     1     1     4
```

with the eigenvalues

```
1.9202 - 1.4742i   1.9202 + 1.4742i   4.8121    1.3474
```

Generating the Schur form of `A` and converting to the complex Schur form

```
[u,t] = schur(A);
[U,T] = rsf2csf(u,t)
```

yields a triangular matrix `T` whose diagonal consists of the eigenvalues of `A`.

```
U =

-0.4576 + 0.3044i   0.5802 - 0.4934i   -0.0197   -0.3428
 0.1616 + 0.3556i   0.4235 + 0.0051i    0.1666    0.8001
 0.3963 + 0.2333i   0.1718 + 0.2458i    0.7191   -0.4260
-0.4759 - 0.3278i  -0.2709 - 0.2778i     0.6743    0.2466
```

```
T =
1.9202 + 1.4742i    0.7691 - 1.0772i   -1.5895 - 0.9940i  -1.3798 + 0.1864i
0                   1.9202 - 1.4742i    1.9296 + 1.6909i   0.2511 + 1.0844i
0                   0                   4.8121             1.1314
0                   0                   0                  1.3474
```

See Also

schur   Schur decomposition

save

Purpose

Save workspace variables on disk

Syntax

```
save
save filename
save filename variables
save filename options
save filename variables options
```

Description

`save`, by itself, stores all workspace variables in a binary format in the file named `matlab.mat`. The data can be retrieved with `load`.

`save` *filename* stores all workspace variables in `filename.mat` instead of the default `matlab.mat`. If *filename* is the special string `stdio`, the `save` command sends the data as standard output.

`save` *filename variables* saves only the workspace *variables* you list after the *filename*.

Options

The forms of the `save` command that use *options* are:

`save` *filename options*

`save` *filename variables options*,

Each specifies a particular ASCII data format, as opposed to the binary MAT-file format, in which to save data. Valid option combinations are:

With these options...	Data is stored in:
`–ascii`	8-digit ASCII format
`–ascii –double`	16-digit ASCII format
`–ascii –tabs`	8-digit ASCII format, tab-separated
`–ascii –double –tabs`	16-digit ASCII format, tab-separated

Variables saved in ASCII format merge into a single variable that takes the name of the ASCII file. Therefore, loading the file *filename* shown above results in a single workspace variable named *filename*. Use the colon operator to access individual variables.

Limitations

Saving complex data with the `–ascii` keyword causes the imaginary part of the data to be lost, as MATLAB cannot load nonnumeric data (`'i'`).

Remarks

The `save` and `load` commands retrieve and store MATLAB variables on disk. They can also import and export numeric matrices as ASCII data files.

MAT-files are double-precision binary MATLAB format files created by the `save` command and readable by the `load` command. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the disparate formats allow. They can also be manipulated by other programs, external to MATLAB.

Alternative syntax: The function form of the syntax, `save('filename')`, is also permitted.

Algorithm

The binary formats used by `save` depend on the size and type of each array. Arrays with any noninteger entries and arrays with 10,000 or fewer elements are saved in floating-point formats requiring eight bytes per real element. Arrays with all integer entries and more than 10,000 elements are saved in the formats shown, requiring fewer bytes per element.

Element Range	Bytes per Element
0 to 255	1
0 to 65535	2
–32767 to 32767	2
$-2^{31}+1$ to $2^{31}-1$	4
other	8

The structure of MAT-files is discussed in detail in the *Application Program Interface Guide*. The Application Program Interface Libraries contain C and Fortran routines to read and write MAT-files from external programs. It is important to use recommended access methods, rather than rely upon the specific file format, which is likely to change in the future.

See Also

`fprintf`	Write formatted data to file
`fwrite`	Write binary data to a file
`load`	Retrieve variables from disk

Purpose Schur decomposition

Syntax

```
[U,T] = schur(A)
T = schur(A)
```

Description The `schur` command computes the Schur form of a matrix.

`[U,T] = schur(A)` produces a Schur matrix `T`, and a unitary matrix `U` so that `A = U*T*U'` and `U'*U = eye(size(A))`.

`T = schur(A)` returns just the Schur matrix `T`.

Remarks The *complex Schur form* of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The *real Schur form* has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

If the matrix `A` is real, `schur` returns the real Schur form. If `A` is complex, `schur` returns the complex Schur form. The function `rsf2csf` converts the real form to the complex form.

Examples `H` is a 3-by-3 eigenvalue test matrix:

```
H =
   -149    -50   -154
    537    180    546
    -27     -9    -25
```

Its Schur form is

```
schur(H) =
   1.0000    7.1119   815.8706
        0    2.0000   -55.0236
        0         0     3.0000
```

The eigenvalues, which in this case are `1`, `2`, and `3`, are on the diagonal. The fact that the off-diagonal elements are so large indicates that this matrix has poorly conditioned eigenvalues; small changes in the matrix elements produce relatively large changes in its eigenvalues.

Algorithm For real matrices, `schur` uses the EISPACK routines `ORTRAN`, `ORTHES`, and `HQR2`. `ORTHES` converts a real general matrix to Hessenberg form using orthogonal similarity transformations. `ORTRAN` accumulates the transformations used by `ORTHES`. `HQR2` finds the eigenvalues of a real upper Hessenberg matrix by the QR method.

The EISPACK subroutine `HQR2` has been modified to allow access to the Schur form, ordinarily just an intermediate result, and to make the computation of eigenvectors optional.

When `schur` is used with a complex argument, the solution is computed using the `QZ` algorithm by the EISPACK routines `QZHES`, `QZIT`, `QZVAL`, and `QZVEC`. They have been modified for complex problems and to handle the special case $B = I$.

For detailed descriptions of these algorithms, see the *EISPACK Guide*.

See Also

`eig`	Eigenvalues and eigenvectors
`hess`	Hessenberg form of a matrix
`qz`	QZ factorization for generalized eigenvalues
`rsf2csf`	Convert real Schur form to complex Schur form

References

[1] Garbow, B. S., J. M. Boyle, J. J. Dongarra, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide Extension*, Lecture Notes in Computer Science, Vol. 51, Springer-Verlag, 1977.

[2] Moler, C.B. and G. W. Stewart, “An Algorithm for Generalized Matrix Eigenvalue Problems,” *SIAM J. Numer. Anal.*, Vol. 10, No. 2, April 1973.

[3] Smith, B. T., J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide*, Lecture Notes in Computer Science, Vol. 6, second edition, Springer-Verlag, 1976.

script

Purpose

Script M-files

Description

A script file is an external file that contains a sequence of MATLAB statements. By typing the filename, subsequent MATLAB input is obtained from the file. Script files have a filename extension of `.m` and are often called M-files.

Scripts are the simplest kind of M-file. They are useful for automating blocks of MATLAB commands, such as computations you have to perform repeatedly from the command line. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace so you can use them in further computations. In addition, scripts can produce graphical output using commands like `plot`.

Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters.

Like any M-file, scripts can contain comments. Any text following a percent sign (%) on a given line is comment text. Comments can appear on lines by themselves, or you can append them to the end of any executable line.

See Also

`echo`	Echo M-files during execution
`function`	Function M-files
`type`	List file

Purpose

Secant and hyperbolic secant

Syntax

```
Y = sec(X)
Y = sech(X)
```

Description

The sec and sech commands operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

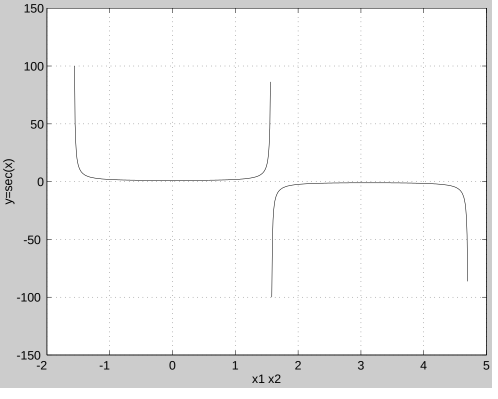
`Y = sec(X)` returns an array the same size as `X` containing the secant of the elements of `X`.

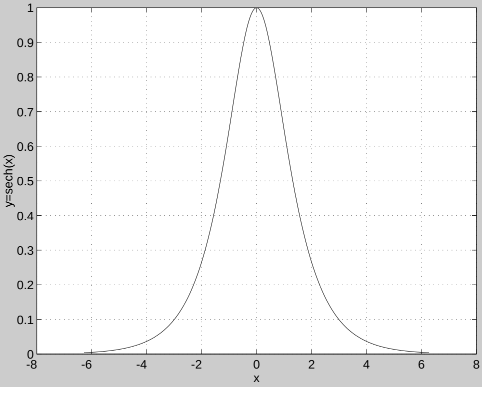
`Y = sech(X)` returns an array the same size as `X` containing the hyperbolic secant of the elements of `X`.

Examples

Graph the secant over the domains $-\pi/2 < x < \pi/2$ and $\pi/2 < x < 3\pi/2$, and the hyperbolic secant over the domain $-2\pi \le x \le 2\pi$.

```
x1 = -pi/2+0.01:0.01:pi/2-0.01;
x2 = pi/2+0.01:0.01:(3*pi/2)-0.01;
plot(x1,sec(x1),x2,sec(x2))
x = -2*pi:0.01:2*pi; plot(x,sech(x))
```





The expression `sec(pi/2)` does not evaluate as infinite but as the reciprocal of the floating-point accuracy `eps`, because `pi` is a floating-point approximation to the exact value of π.

Algorithm

$$\sec(z) = \frac{1}{\cos(z)} \qquad \mathrm{sech}(z) = \frac{1}{\cosh(z)}$$

See Also

`asec`, `asech`	Inverse secant and inverse hyperbolic secant

Purpose Return the set difference of two vectors

Syntax

```
c = setdiff(a,b)
c = setdiff(a,b,'rows')
[c,i] = setdiff(...)
```

Description

`c = setdiff(a,b)` returns the values in `a` that are not in `b`. The resulting vector is sorted is ascending order. In set theoretic terms, `c = a – b`.

`c = (a,b,'rows')` when `a` and `b` are matrices with the same number of columns returns the rows from `a` that are not in `b`.

`[c,i] = setdiff(...)` also returns an index vector `index` such that `c = a(i)` or `c = a(i,:)`.

A non-vector input array `A` is regarded as a column vector `A(:)`.

Examples

```
A = magic(5);
B = magic(4);
[c,i] = setdiff(A,B);
c' =    17    18    19    20    21    22    23    24    25
i' =     1    10    14    18    19    23     2     6    15
```

See Also

`intersect`	Set intersection of two vectors
`ismember`	True for a set member
`setxor`	Set exclusive-or of two vectors
`union`	Set union of two vectors
`unique`	Unique elements of a vector

setfield

Purpose Set field of structure array

Syntax

```
s = setfield(s,'field',v)
s = setfield(s,{i,j},'field',{k},v)
```

Description `s = setfield(s,'field',v)`, where `s` is a 1-by-1 structure, sets the contents of the specified field to the value `v`. This is equivalent to the syntax `s.field = v`.

`s = setfield(s,{i,j},'field',{k},v)` sets the contents of the specified field to the value `v`. This is equivalent to the syntax `s(i,j).field(k) = v`. All subscripts must be passed as cell arrays—that is, they must be enclosed in curly braces (similar to`{i,j}` and `{k}` above). Pass field references as strings.

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 `mystr = setfield(mystr,{2,1},'name','ted')` yields

```
mystr =

2x1 struct array with fields:
    name
    ID
```

See Also

`fields`	Field names of a structure
`getfield`	Get field of structure array

Purpose Set exclusive-or of two vectors

Syntax

```
c = setxor(a,b)
c = setxor(a,b,'rows')
[c,ia,ib] = setxor(...)
```

Description

`c = setxor(a,b)` returns the values that are not in the intersection of `a` and `b`. The resulting vector is sorted.

`c = setxor(a,b,'rows')` when `a` are `b` are matrices with the same number of columns returns the rows that are not in the intersection of `a` and `b`.

`[c,ia,ib] = setxor(...)` also returns index vectors `ia` and `ib` such that `c` is a sorted combination of the elements c = `a(ia)` and c = `b(ib)` or, for row combinations, `c = a(ia,:)` and `c = b(ib,:)`.

A non-vector input array `A` is regarded as a column vector `A(:)`.

Examples

```
a = [-1 0 1 Inf -Inf NaN];
b = [-2 pi 0 Inf];
c = setxor(a,b)

c =
      -Inf   -2.0000   -1.0000    1.0000    3.1416       NaN
```

See Also

`intersect`	Set intersection of two vectors
`ismember`	True for a set member
`setdiff`	Set difference of two vectors
`union`	Set union of two vectors
`unique`	Unique elements of a vector

Purpose

Shift dimensions

Syntax

```
B = shiftdim(X,n)
[B,nshifts] = shiftdim(X)
```

Description

`B = shiftdim(X,n)` shifts the dimensions of `X` by `n`. When `n` is positive, `shiftdim` shifts the dimensions to the left and wraps the `n` leading dimensions to the end. When `n` is negative, `shiftdim` shifts the dimensions to the right and pads with singletons.

`[B,nshifts] = shiftdim(X)` returns the array `B` with the same number of elements as `X` but with any leading singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`. `nshifts` is the number of dimensions that are removed.

If `X` is a scalar, `shiftdim` has no effect.

Examples

The `shiftdim` command is handy for creating functions that, like `sum` or `diff`, work along the first nonsingleton dimension.

```
a = rand(1,1,3,1,2);
[b,n]  = shiftdim(a); % b is 3-by-1-by-2 and n is 2.
c = shiftdim(b,-n);   % c == a.
d = shiftdim(a,3);    % d is 1-by-2-by-1-by-1-by-3.
```

See Also

`reshape`	Reshape array
`squeeze`	Remove singleton dimensions

Purpose

Signum function

Syntax

```
Y = sign(X)
```

Description

`Y = sign(X)` returns an array `Y` the same size as `X`, where each element of `Y` is:

- `1` if the corresponding element of `X` is greater than zero
- `0` if the corresponding element of `X` equals zero
- `-1` if the corresponding element of `X` is less than zero

For nonzero complex `X`, `sign(X) = X./abs(X)`.

See Also

`abs`	Absolute value and complex magnitude
`conj`	Complex conjugate
`imag`	Imaginary part of a complex number
`real`	Real part of complex number

sin, sinh

Purpose Sine and hyperbolic sine

Syntax

```
Y = sin(X)
Y = sinh(X)
```

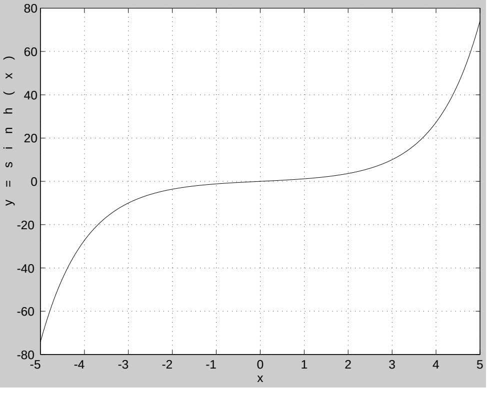
Description The sin and sinh commands operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

`Y = sin(X)` returns the circular sine of the elements of X.

`Y = sinh(X)` returns the hyperbolic sine of the elements of X.

Examples Graph the sine function over the domain $-\pi \le x \le \pi$, and the hyperbolic sine function over the domain $-5 \le x \le 5$.

```
x = -pi:0.01:pi; plot(x,sin(x))
x = -5:0.01:5; plot(x,sinh(x))
```



The expression sin(pi) is not exactly zero, but rather a value the size of the floating-point accuracy eps, because pi is only a floating-point approximation to the exact value of π.

Algorithm

$$\sin(x + iy) = \sin(x)\cos(y) + i\cos(x)\sin(y)$$

$$\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}$$

$$\sinh(z) = \frac{e^{z} - e^{-z}}{2}$$

See Also

`asin`, `asinh` Inverse sine and inverse hyperbolic sine

size

Purpose

Array dimensions

Syntax

```
d = size(X)
[m,n] = size(X)
m = size(X,dim)
[d1,d2,d3,...,dn] = size(X)
```

Description

`d = size(X)` returns the sizes of each dimension of array `X` in a vector `d` with `ndims(X)` elements.

`[m,n] = size(X)` returns the size of matrix `X` in variables `m` and `n`.

`m = size(X,dim)` returns the size of the dimension of `X` specified by scalar `dim`.

`[d1,d2,d3,...,dn] = size(X)` returns the sizes of the various dimensions of array `X` in separate variables.

If the number of output arguments `n` does not equal `ndims(X)`, then:

If `n > ndims(X)`	Ones are returned in the "extra" variables `dndims(X)+1` through `dn`.
If `n < ndims(X)`	The final variable `dn` contains the product of the sizes of all the "remaining" dimensions of `X`, that is, dimensions `n+1` through `ndims(X)`.

Examples

The size of the second dimension of `rand(2,3,4)` is 3.

```
m = size(rand(2,3,4),2)

m =
     3
```

Here the size is output as a single vector.

```
d = size(rand(2,3,4))

d =
     2     3     4
```

Here the size of each dimension is assigned to a separate variable.

```
[m,n,p] = size(rand(2,3,4))
m =
     2

n =
     3

p =
     4
```

If `X = ones(3,4,5)`, then

```
[d1,d2,d3] = size(X)

d1 =        d2 =    d3 =
   3             4      5
```

but when the number of output variables is less than `ndims(X)`:

```
[d1,d2] = size(X)

d1 =        d2 =
   3             20
```

The "extra" dimensions are collapsed into a single product.

If `n > ndims(X)`, the "extra" variables all represent singleton dimensions:

```
[d1,d2,d3,d4,d5,d6] = size(X)

d1 =        d2 =        d3 =
   3             4           5

d4 =        d5 =        d6 =
    1            1           1
```

See Also

`exist`	Check if a variable or file exists
`length`	Length of vector
`whos`	List directory of variables in memory

Purpose

Sort elements in ascending order

Syntax

```
B = sort(A)
[B,INDEX] = sort(A)
B = sort(A,dim)
```

Description

`B = sort(A)` sorts the elements along different dimensions of an array, and arranges those elements in ascending order.

Real, complex, and string elements are permitted. For identical values in `A`, the location in the input array determines location in the sorted list. When `A` is complex, the elements are sorted by magnitude, and where magnitudes are equal, further sorted by phase angle on the interval $[-\pi, \pi]$. If `A` includes any `NaN` elements, `sort` places these at the end.

If `A` is a vector, `sort(A)` arranges those elements in ascending order.

If `A` is a matrix, `sort(A)` treats the columns of `A` as vectors, returning sorted columns.

If `A` is a multidimensional array, `sort(A)` treats the values along the first non-singleton dimension as vectors, returning an array of sorted vectors.

`[B,INDEX] = sort(A)` also returns an array of indices. `INDEX` is an array of `size(A)`, each column of which is a permutation vector of the corresponding column of `A`. If `A` has repeated elements of equal value, indices are returned that preserve the original relative ordering.

`B = sort(A,dim)` sorts the elements along the dimension of `A` specified by scalar `dim`.

If `dim` is a vector, `sort` works iteratively on the specified dimensions. Thus, `sort(A,[1 2])` is equivalent to `sort(sort(A,2),1)`.

See Also

`max`	Maximum elements of an array
`mean`	Average or mean value of arrays
`median`	Median value of arrays
`min`	Minimum elements of an array
`sortrows`	Sort rows in ascending order

Purpose Sort rows in ascending order

Syntax

```
B = sortrows(A)
B = sortrows(A,column)
[B,index] = sortrows(A)
```

Description `B = sortrows(A)` sorts the rows of `A` as a group in ascending order. Argument `A` must be either a matrix or a column vector.

For strings, this is the familiar dictionary sort. When `A` is complex, the elements are sorted by magnitude, and, where magnitudes are equal, further sorted by phase angle on the interval $[-\pi, \pi]$.

`B = sortrows(A,column)` sorts the matrix based on the columns specified in the vector `column`. For example, `sortrows(A,[2 3])` sorts the rows of `A` by the second column, and where these are equal, further sorts by the third column.

`[B,index] = sortrows(A)` also returns an index vector `index`.

If `A` is a column vector, then `B = A(index)`.

If `A` is an m-by-n matrix, then `B = A(index,:)`.

Examples Given the 5-by-5 string matrix,

```
A = ['one  ';'two  ';'three';'four ';'five '];
```

The commands `B = sortrows(A)` and `C = sortrows(A,1)` yield

```
B =             C =
five            four
four            five
one             one
three           two
two             three
```

See Also

`sort`	Sort elements in ascending order

sound

Purpose Convert vector into sound

Syntax

```
sound(y,Fs)
sound(y)
sound(y,Fs,bits)
```

Description `sound(y,Fs)`, sends the signal in vector `y` (with sample frequency `Fs`) to the speaker on PC, Macintosh, and most UNIX platforms. Values in `y` are assumed to be in the range $-1.0 \le y \le 1.0$. Values outside that range are clipped. Stereo sound is played on platforms that support it when `y` is an `n`-by-2 matrix.

`sound(y)` plays the sound at the default sample rate or 8192 Hz.

`sound(y,Fs,bits)` plays the sound using `bits` bits/sample if possible. Most platforms support `bits = 8` or `bits = 16`.

Remarks MATLAB supports all Windows-compatible sound devices.

See Also

`auread`	Read NeXT/SUN (`.au`) sound file
`auwrite`	Write NeXT/SUN (`.au`) sound file
`soundsc`	Scale data and play as sound
`wavread`	Read Microsoft WAVE (`.wav`) sound file
`wavwrite`	Write Microsoft WAVE (`.wav`) sound file

Purpose Scale data and play as sound

Syntax

```
soundsc(y,Fs)
soundsc(y)
soundsc(y,Fs,bits)
soundsc(y,...,slim)
```

Description `soundsc(y,Fs)` sends the signal in vector `y` (with sample frequency `Fs`) to the speaker on PC, Macintosh, and most UNIX platforms. The signal `y` is scaled to the range $-1.0 \le y \le 1.0$ before it is played, resulting in a sound that is played as loud as possible without clipping.

`soundsc(y)` plays the sound at the default sample rate or 8192 Hz.

`soundsc(y,Fs,bits)` plays the sound using `bits` bits/sample if possible. Most platforms support `bits = 8` or `bits = 16`.

`soundsc(y,...,slim)` where `slim = [slow shigh]` maps the values in `y` between `slow` and `shigh` to the full sound range. The default value is `slim = [min(y) max(y)]`.

Remarks MATLAB supports all Windows-compatible sound devices.

See Also

`auread`	Read NeXT/SUN (.au) sound file
`auwrite`	Write NeXT/SUN (.au) sound file
`sound`	Convert vector into sound
`wavread`	Read Microsoft WAVE (.wav) sound file
`wavwrite`	Write Microsoft WAVE (.wav) sound file

Purpose

Allocate space for sparse matrix

Syntax

```
S = spalloc(m,n,nzmax)
```

Description

`S = spalloc(m,n,nzmax)` creates an all zero sparse matrix `S` of size `m`-by-`n` with room to hold `nzmax` nonzeros. The matrix can then be generated column by column without requiring repeated storage allocation as the number of nonzeros grows.

`spalloc(m,n,nzmax)` is shorthand for

```
sparse([],[],[],m,n,nzmax)
```

Examples

To generate efficiently a sparse matrix that has an average of at most three nonzero elements per column

```
S = spalloc(n,n,3*n);
for j = 1:n
    S(:,j) = [zeros(n-3,1)' round(rand(3,1))']';
end
```

Purpose Create sparse matrix

Syntax

```
S = sparse(A)
S = sparse(i,j,s,m,n,nzmax)
S = sparse(i,j,s,m,n)
S = sparse(i,j,s)
S = sparse(m,n)
```

Description The `sparse` function generates matrices in MATLAB's sparse storage organization.

`S = sparse(A)` converts a full matrix to sparse form by squeezing out any zero elements. If `S` is already sparse, `sparse(S)` returns `S`.

`S = sparse(i,j,s,m,n,nzmax)` uses vectors `i`, `j`, and `s` to generate an `m`-by-`n` sparse matrix with space allocated for `nzmax` nonzeros. Any elements of `s` that are zero are ignored, along with the corresponding values of `i` and `j`. Vectors `i`, `j`, and `s` are all the same length.

To simplify this six-argument call, you can pass scalars for the argument `s` and one of the arguments `i` or `j`—in which case they are expanded so that `i`, `j`, and `s` all have the same length.

`S = sparse(i,j,s,m,n)` uses `nzmax = length(s)`.

`S = sparse(i,j,s)` uses `m = max(i)` and `n = max(j)`. The maxima are computed before any zeros in `s` are removed, so one of the rows of `[i j s]` might be `[m n 0]`.

`S = sparse(m,n)` abbreviates `sparse([],[],[],m,n,0)`. This generates the ultimate sparse matrix, an `m`-by-`n` all zero matrix.

Remarks All of MATLAB's built-in arithmetic, logical, and indexing operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices.

In most cases, operations on mixtures of sparse and full matrices return full matrices. The exceptions include situations where the result of a mixed operation is structurally sparse, for example, `A.*S` is at least as sparse as `S`.

Examples

`S = sparse(1:n,1:n,1)` generates a sparse representation of the `n`-by-`n` identity matrix. The same `S` results from `S = sparse(eye(n,n))`, but this would also temporarily generate a full `n`-by-`n` matrix with most of its elements equal to zero.

`B = sparse(10000,10000,pi)` is probably not very useful, but is legal and works; it sets up a `10000`-by-`10000` matrix with only one nonzero element. Don't try `full(B)`; it requires 800 megabytes of storage.

This dissects and then reassembles a sparse matrix:

```
[i,j,s] = find(S);
[m,n] = size(S);
S = sparse(i,j,s,m,n);
```

So does this, if the last row and column have nonzero entries:

```
[i,j,s] = find(S);
S = sparse(i,j,s);
```

See Also

The `sparfun` directory, and:

`diag`	Diagonal matrices and diagonals of a matrix
`find`	Find indices and values of nonzero elements
`full`	Convert sparse matrix to full matrix
`nnz`	Number of nonzero matrix elements
`nonzeros`	Nonzero matrix elements
`nzmax`	Amount of storage allocated for nonzero matrix elements
`spones`	Replace nonzero sparse matrix elements with ones
`sprandn`	Sparse normally distributed random matrix
`sprandsym`	Sparse symmetric random matrix
`spy`	Visualize sparsity pattern

Purpose

Import matrix from sparse matrix external format

Syntax

```
S = spconvert(D)
```

Description

`spconvert` is used to create sparse matrices from a simple sparse format easily produced by non-MATLAB sparse programs. `spconvert` is the second step in the process:

1. Load an ASCII data file containing `[i,j,v]` or `[i,j,re,im]` as rows into a MATLAB variable.
2. Convert that variable into a MATLAB sparse matrix.

`S = spconvert(D)` converts a matrix `D` with rows containing `[i,j,s]` or `[i,j,r,s]` to the corresponding sparse matrix. `D` must have an `nnz` or `nnz+1` row and three or four columns. Three elements per row generate a real matrix and four elements per row generate a complex matrix. A row of the form `[m n 0]` or `[m n 0 0]` anywhere in `D` can be used to specify `size(S)`. If `D` is already sparse, no conversion is done, so `spconvert` can be used after `D` is loaded from either a MAT-file or an ASCII file.

Examples

Suppose the ASCII file `uphill.dat` contains

```
1    1    1.000000000000000
1    2    0.500000000000000
2    2    0.333333333333333
1    3    0.333333333333333
2    3    0.250000000000000
3    3    0.200000000000000
1    4    0.250000000000000
2    4    0.200000000000000
3    4    0.166666666666667
4    4    0.142857142857143
4    4    0.000000000000000
```

Then the statements

```
load uphill.dat
H = spconvert(uphill)
```

recreate `sparse(triu(hilb(4)))`, possibly with roundoff errors. In this case, the last line of the input file is not necessary because the earlier lines already specify that the matrix is at least 4-by-4.

Purpose Extract and create sparse band and diagonal matrices

Syntax

```
[B,d] = spdiags(A)
B = spdiags(A,d)
A = spdiags(B,d,A)
A = spdiags(B,d,m,n)
```

Description The spdiags function generalizes the function diag. Four different operations, distinguished by the number of input arguments, are possible:

`[B,d] = spdiags(A)` extracts all nonzero diagonals from the m-by-n matrix A. B is a min(m,n)-by-p matrix whose columns are the p nonzero diagonals of A. d is a vector of length p whose integer components specify the diagonals in A.

`B = spdiags(A,d)` extracts the diagonals specified by d.

`A = spdiags(B,d,A)` replaces the diagonals specified by d with the columns of B. The output is sparse.

`A = spdiags(B,d,m,n)` creates an m-by-n sparse matrix by taking the columns of B and placing them along the diagonals specified by d.

Remarks If a column of B is longer than the diagonal it's replacing, spdiags takes elements from B's tail.

Arguments The spdiags function deals with three matrices, in various combinations, as both input and output:

A An m-by-n matrix, usually (but not necessarily) sparse, with its nonzero or specified elements located on p diagonals.

B A min(m,n)-by-p matrix, usually (but not necessarily) full, whose columns are the diagonals of A.

d A vector of length p whose integer components specify the diagonals in A.

Roughly, A, B, and d are related by

```
for k = 1:p
    B(:,k) = diag(A,d(k))
end
```

Some elements of B, corresponding to positions outside of A, are not defined by these loops. They are not referenced when B is input and are set to zero when B is output.

Examples

This example generates a sparse tridiagonal representation of the classic second difference operator on n points.

```
e = ones(n,1);
A = spdiags([e -2*e e], -1:1, n, n)
```

Turn it into Wilkinson's test matrix (see gallery):

```
A = spdiags(abs(-(n-1)/2:(n-1)/2)',0,A)
```

Finally, recover the three diagonals:

```
B = spdiags(A)
```

The second example is not square.

```
A = [11    0   13    0
      0   22    0   24
      0    0   33    0
     41    0    0   44
      0   52    0    0
      0    0   63    0
      0    0    0   74]
```

Here m = 7, n = 4, and p = 3.

The statement [B,d] = spdiags(A) produces d = [-3 0 2]' and

```
B = [41   11    0
     52   22    0
     63   33   13
     74   44   24]
```

Conversely, with the above `B` and `d`, the expression `spdiags(B,d,7,4)` reproduces the original `A`.

See Also

`diag`	Diagonal matrices and diagonals of a matrix

speye

Purpose

Sparse identity matrix

Syntax

```
S = speye(m,n)
S = speye(n)
```

Description

`S = speye(m,n)` forms an `m`-by-`n` sparse matrix with `1`s on the main diagonal.

`S = speye(n)` abbreviates `speye(n,n)`.

Examples

`I = speye(1000)` forms the sparse representation of the 1000-by-1000 identity matrix, which requires only about 16 kilobytes of storage. This is the same final result as `I = sparse(eye(1000,1000))`, but the latter requires eight megabytes for temporary storage for the full representation.

See Also

`spalloc`	Allocate space for sparse matrix
`spones`	Replace nonzero sparse matrix elements with ones
`spdiags`	Extract and create sparse band and diagonal matrices
`sprand`	Sparse uniformly distributed random matrix
`sprandn`	Sparse normally distributed random matrix

Purpose

Apply function to nonzero sparse matrix elements

Syntax

`f = spfun('`*`function`*`',S)`

Description

The `spfun` function selectively applies a function to only the *nonzero* elements of a sparse matrix, preserving the sparsity pattern of the original matrix (except for underflow).

`f = spfun('`*`function`*`',S)` evaluates *`function`*`(S)` on the nonzero elements of `S`. *`function`* must be the name of a function, usually defined in an M-file, which can accept a matrix argument, `S`, and evaluate the function at each element of `S`.

Remarks

Functions that operate element-by-element, like those in the `elfun` directory, are the most appropriate functions to use with `spfun`.

Examples

Given the 4-by-4 sparse diagonal matrix

```
S =
   (1,1)        1
   (2,2)        2
   (3,3)        3
   (4,4)        4
```

`f = spfun('exp',S)` has the same sparsity pattern as `S`:

```
f =
   (1,1)       2.7183
   (2,2)       7.3891
   (3,3)      20.0855
   (4,4)      54.5982
```

whereas `exp(S)` has `1`s where `S` has `0`s.

```
full(exp(S))

ans =
    2.7183    1.0000    1.0000    1.0000
    1.0000    7.3891    1.0000    1.0000
    1.0000    1.0000   20.0855    1.0000
    1.0000    1.0000    1.0000   54.5982
```

sph2cart

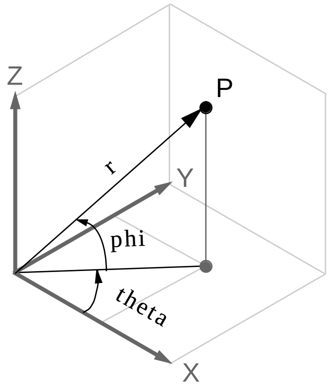
Purpose Transform spherical coordinates to Cartesian

Syntax

```
[x,y,z] = sph2cart(THETA,PHI,R)
```

Description `[x,y,z] = sph2cart(THETA,PHI,R)` transforms the corresponding elements of spherical coordinate arrays to Cartesian, or *xyz*, coordinates. `THETA`, `PHI`, and `R` must all be the same size. `THETA` and `PHI` are angular displacements in radians from the positive *x*-axis and from the *x*-*y* plane, respectively.

Algorithm The mapping from spherical coordinates to three-dimensional Cartesian coordinates is:




```
x = r .* cos(phi) .* cos(theta)
y = r .* cos(phi) .* sin(theta)
      z = r .* sin(phi)
```

See Also

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

Purpose Cubic spline interpolation

Syntax

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

Description The `spline` function interpolates between data points using cubic spline fits.

`yi = spline(x,y,xi)` accepts vectors `x` and `y` that contain coarsely spaced data, and vector `xi` that specifies a new, more finely spaced abscissa. The function uses cubic spline interpolation to find a vector `yi` corresponding to `xi`.

`pp = spline(x,y)` returns the pp-form of the cubic spline interpolant, for later use with `ppval` and other spline functions.

Examples The two vectors

```
t = 1900:10:1990;
p = [ 75.995  91.972  105.711  123.203  131.669 ...
     150.697 179.323  203.212  226.505  249.633 ]';
```

represent the census years from 1900 to 1990 and the corresponding United States population in millions of people. The expression

```
spline(t,p,2000)
```

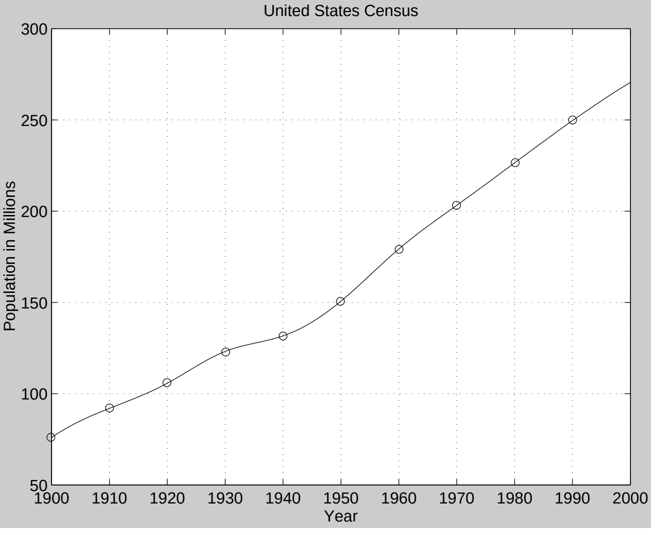
uses the cubic spline to extrapolate and predict the population in the year 2000. The result is

```
ans =
    270.6060
```

The statements

```
x = 1900:1:2000;
y = spline(t,p,x);
plot(t,p,'o',x,y)
```

interpolate the data with a cubic spline, evaluate that spline for each year from 1900 to 2000, and plot the result.



Algorithm

`spline` is a MATLAB M-file. It uses the M-files `ppval`, `mkpp`, and `unmkpp`. These routines form a small suite of functions for working with piecewise polynomials. `spline` uses these functions in a fairly simple fashion to perform cubic spline interpolation. For access to the more advanced features, see the M-files and the Spline Toolbox.

See Also

`interp1`	One-dimensional data interpolation (table lookup)
`ppval`	Evaluate piecewise polynomial

References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

Purpose Replace nonzero sparse matrix elements with ones

Syntax

```
R = spones(S)
```

Description `R = spones(S)` generates a matrix `R` with the same sparsity structure as `S`, but with `1`'s in the nonzero positions.

Examples `c = sum(spones(S))` is the number of nonzeros in each column.

`r = sum(spones(S'))'` is the number of nonzeros in each row.

`sum(c)` and `sum(r)` are equal, and are equal to `nnz(S)`.

See Also

`nnz`	Number of nonzero matrix elements
`spalloc`	Allocate space for sparse matrix
`spfun`	Apply function to nonzero sparse matrix elements

spparms

Purpose

Set parameters for sparse matrix routines

Syntax

```
spparms('key',value)
spparms
values = spparms
[keys,values] = spparms
spparms(values)
value = spparms('key')
spparms('default')
spparms('tight')
```

Description

`spparms('key',value)` sets one or more of the *tunable* parameters used in the sparse linear equation operators, `\` and `/`, and the minimum degree orderings, `colmmd` and `symmmd`. In ordinary use, you should never need to deal with this function.

The meanings of the key parameters are

`'spumoni'`	Sparse Monitor flag. `0` produces no diagnostic output, the default. `1` produces information about choice of algorithm based on matrix structure, and about storage allocation. `2` also produces very detailed information about the minimum degree algorithms.
`'thr_rel'`, `'thr_abs'`	Minimum degree threshold is `thr_rel*mindegree+thr_abs`.
`'exact_d'`	Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.
`'supernd'`	If positive, minimum degree amalgamates the supernodes every `supernd` stages.
`'rreduce'`	If positive, minimum degree does row reduction every `rreduce` stages.
`'wh_frac'`	Rows with `density > wh_frac` are ignored in `colmmd`.

'autommd'	Nonzero to use minimum degree orderings with \ and /.
'aug_rel', 'aug_abs'	Residual scaling parameter for augmented equations is `aug_rel*max(max(abs(A))) + aug_abs`. For example, `aug_rel = 0`, `aug_abs = 1` puts an unscaled identity matrix in the (1,1) block of the augmented matrix.

`spparms`, by itself, prints a description of the current settings.

`values = spparms` returns a vector whose components give the current settings.

`[keys,values] = spparms` returns that vector, and also returns a character matrix whose rows are the keywords for the parameters.

`spparms(values)`, with no output argument, sets all the parameters to the values specified by the argument vector.

`value = spparms('`*`key`*`')` returns the current setting of one parameter.

`spparms('default')` sets all the parameters to their default settings.

`spparms('tight')` sets the minimum degree ordering parameters to their *tight* settings, which can lead to orderings with less fill-in, but which make the ordering functions themselves use more execution time.

The key parameters for `default` and `tight` settings are

	Keyword	Default	Tight
`values(1)`	`'spumoni'`	0.0	
`values(2)`	`'thr_rel'`	1.1	1.0
`values(3)`	`'thr_abs'`	1.0	0.0
`values(4)`	`'exact_d'`	0.0	1.0
`values(5)`	`'supernd'`	3.0	1.0
`values(6)`	`'rreduce'`	3.0	1.0
`values(7)`	`'wh_frac'`	0.5	0.5
`values(8)`	`'autommd'`	1.0	
`values(9)`	`'aug_rel'`	0.001	
`values(10)`	`'aug_abs'`	0.0	

See Also

`\`	Matrix left division (backslash)
`colmmd`	Sparse column minimum degree permutation
`symmmd`	Sparse symmetric minimum degree ordering

References

[1] Gilbert, John R., Cleve Moler and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis and Applications 13*, 1992, pp. 333-356.

Purpose

Sparse uniformly distributed random matrix

Syntax

```
R = sprand(S)
R = sprand(m,n,density)
R = sprand(m,n,density,rc)
```

Description

`R = sprand(S)` has the same sparsity structure as `S`, but uniformly distributed random entries.

`R = sprand(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` uniformly distributed nonzero entries ($0 \le$ `density` ≤ 1).

`R = sprand(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr` ≤ `min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

See Also

`sprandn`	Sparse normally distributed random matrix
`sprandsym`	Sparse symmetric random matrix

sprandn

Purpose

Sparse normally distributed random matrix

Syntax

```
R = sprandn(S)
R = sprandn(m,n,density)
R = sprandn(m,n,density,rc)
```

Description

`R = sprandn(S)` has the same sparsity structure as `S`, but normally distributed random entries with mean `0` and variance `1`.

`R = sprandn(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` normally distributed nonzero entries (`0` ≤ `density` ≤ `1`).

`R = sprandn(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr` ≤ `min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

See Also

`sprand`	Sparse uniformly distributed random matrix
`sprandsym`	Sparse symmetric random matrix

Purpose

Sparse symmetric random matrix

Syntax

```
R = sprandsym(S)
R = sprandsym(n,density)
R = sprandsym(n,density,rc)
R = sprandsym(n,density,rc,kind)
```

Description

`R = sprandsym(S)` returns a symmetric random matrix whose lower triangle and diagonal have the same structure as `S`. Its elements are normally distributed, with mean `0` and variance `1`.

`R = sprandsym(n,density)` returns a symmetric random, `n`-by-`n`, sparse matrix with approximately `density*n*n` nonzeros; each entry is the sum of one or more normally distributed random samples, and (`0 ≤ density ≤ 1`).

`R = sprandsym(n,density,rc)` returns a matrix with a reciprocal condition number equal to `rc`. The distribution of entries is nonuniform; it is roughly symmetric about 0; all are in $[-1, 1]$.

If `rc` is a vector of length `n`, then `R` has eigenvalues `rc`. Thus, if `rc` is a positive (nonnegative) vector then `R` is a positive definite matrix. In either case, `R` is generated by random Jacobi rotations applied to a diagonal matrix with the given eigenvalues or condition number. It has a great deal of topological and algebraic structure.

`R = sprandsym(n,density,rc,kind)` returns a positive definite matrix. Argument `kind` can be:

- `1` to generate `R` by random Jacobi rotation of a positive definite diagonal matrix. `R` has the desired condition number exactly.
- `2` to generate an `R` that is a shifted sum of outer products. `R` has the desired condition number only approximately, but has less structure.
- `3` to generate an `R` that has the same structure as the matrix `S` and approximate condition number `1/rc`. `density` is ignored.

See Also

`sprand`	Sparse uniformly distributed random matrix
`sprandn`	Sparse normally distributed random matrix

Purpose

Write formatted data to a string

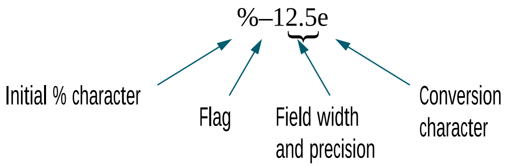
Syntax

```
s = sprintf(format,A,...)
[s,errrmsg] = sprintf(format,A,...)
```

Description

`s = sprintf(`*`format`*`,A,...)` formats the data in matrix `A` (and in any additional matrix arguments) under control of the specified *`format`* string, and returns it in the MATLAB string variable `s`. `sprintf` is the same as `fprintf` except that it returns the data in a MATLAB string variable rather than writing it to a file.

The *`format`* string specifies notation, alignment, significant digits, field width, and other aspects of output format. It can contain ordinary alphanumeric characters; along with escape characters, conversion specifiers, and other characters, organized as shown below:



For more information see “Tables” and “References.”

`[s,errrmsg] = sprintf(`*`format`*`,A,...)` returns an error message string `errmsg` if an error occurred or an empty matrix if an error did not occur.

Remarks

The `sprintf` function behaves like its ANSI C language `sprintf()` namesake with certain exceptions and extensions. These include:

1. The following nonstandard subtype specifiers are supported for conversion specifiers `%o`, `%u`, `%x`, and `%X`.

   `t` The underlying C data type is a float rather than an unsigned integer.

   `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`'.
2. `sprintf` is *vectorized* for the case when input matrix `A` is nonscalar. The format string is cycled through the elements of `A` (columnwise) until all the elements are used up. It is then cycled in a similar manner, without reinitializing, through any additional matrix arguments.

Tables

The following tables describe the nonalphanumeric characters found in format specification strings.

Escape Characters

Character	Description
`\n`	New line
`\t`	Horizontal tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Form feed
`\\`	Backslash
\'' or ''	Single quotation mark
`%%`	Percent character

Conversion characters specify the notation of the output.

Conversion Specifiers

Specifier	Description
`%c`	Single character
`%d`	Decimal notation (signed)
`%e`	Exponential notation (using a lowercase `e` as in `3.1415e+00`)
`%E`	Exponential notation (using an uppercase `E` as in `3.1415E+00`)
`%f`	Fixed-point notation
`%g`	The more compact of `%e` or `%f`, as defined in [2]. Insignificant zeros do not print.
`%G`	Same as `%g`, but using an uppercase `E`
`%o`	Octal notation (unsigned)
`%s`	String of characters
`%u`	Decimal notation (unsigned)
`%x`	Hexadecimal notation (using lowercase letters `a`–`f`)
`%X`	Hexadecimal notation (using uppercase letters `A`–`F`)

Other characters can be inserted into the conversion specifier between the `%` and the conversion character.

Other Characters

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`
Digits (field width)	A digit string specifying the minimum number of digits to be printed.	`%6f`
Digits (precision)	A digit string including a period (.) specifying the number of digits to be printed to the right of the decimal point.	`%6.2f`

Examples

Command	Result
`sprintf('%0.5g',(1+sqrt(5))/2)`	`1.618`
`sprintf('%0.5g',1/eps)`	`4.5036e+15`
`sprintf('%15.5f',1/eps)`	`4503599627370496.00000`
`sprintf('%d',round(pi))`	`3`
`sprintf('%s','hello')`	`hello`
`sprintf('The array is %dx%d.',2,3)`	`The array is 2x3`
`sprintf('\n')`	Line termination character on all platforms

See Also

`int2str`, `num2str`, `sscanf`

References

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

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

Purpose Visualize sparsity pattern

Syntax

```
spy(S)
spy(S,markersize)
spy(S,'LineSpec')
spy(S,'LineSpec',markersize)
```

Description `spy(S)` plots the sparsity pattern of any matrix `S`.

`spy(S,marksize)`, where `markersize` is an integer, plots the sparsity pattern using markers of the specified point size.

`spy(S,'`*`LineSpec`*`')`, where *LineSpec* is a string, uses the specified plot marker type and color.

`spy(S,'`*`LineSpec`*`',markersize)` uses the specified type, color, and size for the plot markers.

`S` is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.

`spy` replaces `format +`, which takes much more space to display essentially the same information.

See Also The `gplot` and `LineSpec` reference entries in the *MATLAB Graphics Guide*, and:

`find`	Find indices and values of nonzero elements
`symmmd`	Sparse symmetric minimum degree ordering
`symrcm`	Sparse reverse Cuthill-McKee ordering

Purpose Square root

Syntax

```
B = sqrt(A)
```

Description

B = sqrt(A) returns the square root of each element of the array X. For the elements of X that are negative or complex, sqrt(X) produces complex results.

Remarks

See sqrtm for the matrix square root.

Examples

```
sqrt((-2:2)')
ans =
      0 + 1.4142i
      0 + 1.0000i
      0
 1.0000
 1.4142
```

See Also

sqrtm Matrix square root

sqrtm

Purpose

Matrix square root

Syntax

```
Y = sqrtm(X)
[Y,esterr] = sqrtm(X)
```

Description

`Y = sqrtm(X)` is the matrix square root of `X`. Complex results are produced if `X` has negative eigenvalues. A warning message is printed if the computed `Y*Y` is not close to `X`.

`[Y,esterr] = sqrtm(X)` does not print any warning message, but returns an estimate of the relative residual, `norm(Y*Y–X)/norm(X)`.

Remarks

If `X` is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.

Some matrices, like `X = [0 1; 0 0]`, do not have any square roots, real or complex, and `sqrtm` cannot be expected to produce one.

Examples

A matrix representation of the fourth difference operator is

```
X =
      5   -4    1    0    0
     -4    6   -4    1    0
      1   -4    6   -4    1
      0    1   -4    6   -4
      0    0    1   -4    5
```

This matrix is symmetric and positive definite. Its unique positive definite square root, `Y = sqrtm(X)`, is a representation of the second difference operator.

```
Y =
      2   -1   -0    0   -0
     -1    2   -1   -0   -0
     -0   -1    2   -1    0
      0   -0   -1    2   -1
     -0   -0    0   -1    2
```

The matrix

```
X =
     7   10
    15   22
```

has four square roots. Two of them are

```
Y1 =
    1.5667    1.7408
    2.6112    4.1779
```

and

```
Y2 =
     1    2
     3    4
```

The other two are -Y1 and -Y2. All four can be obtained from the eigenvalues and vectors of X.

```
[V,D] = eig(X);
D =
    0.1386          0
         0    28.8614
```

The four square roots of the diagonal matrix D result from the four choices of sign in

```
S =
    ±0.3723          0
          0    ±5.3723
```

All four Ys are of the form

```
Y = V*S/V
```

The sqrtm function chooses the two plus signs and produces Y1, even though Y2 is more natural because its entries are integers.

Finally, the matrix

```
 X =
     0    1
     0    0
```

does not have any square roots. There is no matrix `Y`, real or complex, for which `Y*Y = X`. The statement

```
Y = sqrtm(X)
```

produces several warning messages concerning accuracy and the answer

```
Y =

   1.0e+03 *

  0.0000+ 0.0000i   4.9354- 7.6863i
  0.0000+ 0.0000i   0.0000+ 0.0000i
```

Algorithm

The function `sqrtm(X)` is an abbreviation for `funm(X,'sqrt')`. The algorithm used by `funm` is based on a Schur decomposition. It can fail in certain situations where `X` has repeated eigenvalues. See `funm` for details.

See Also

`expm`	Matrix exponential
`funm`	Evaluate functions of a matrix
`logm`	Matrix logarithm

Purpose Remove singleton dimensions

Syntax

```
B = squeeze(A)
```

Description

`B = squeeze(A)` returns an array `B` with the same elements as `A`, but with all singleton dimensions removed. A singleton dimension is any dimension for which
`size(A,dim) = 1`.

Examples

Consider the 2-by-1-by-3 array `Y = rand(2,1,3)`. This array has a singleton column dimension — that is, there's only one column per page.

```
Y =

Y(:,:,1) =     Y(:,:,2) =
    0.5194         0.0346
    0.8310         0.0535

Y(:,:,3) =
    0.5297
    0.6711
```

The command `Z = squeeze(Y)` yields a 2-by-3 matrix:

```
Z =
    0.5194    0.0346    0.5297
    0.8310    0.0535    0.6711
```

See Also

`reshape`	Reshape array
`shiftdim`	Shift dimensions

Purpose

Read string under format control

Syntax

```
A = sscanf(s,format)
A = sscanf(s,format,size)
[A,count,errmsg,nextindex] = sscanf(...)
```

Description

`A = sscanf(s,`*`format`*`)` reads data from the MATLAB string variable `s`, converts it according to the specified *`format`* string, and returns it in matrix `A`. *`format`* is a string specifying the format of the data to be read. See "Remarks" for details. `sscanf` is the same as `fscanf` except that it reads the data from a MATLAB string variable rather than reading it from a file.

`A = sscanf(s,`*`format`*`,size)` reads the amount of data specified by `size` and converts it according to the specified *`format`* string. `size` is an argument that determines how much data is read. Valid options are:

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

If the matrix `A` results from using character conversions only and `size` is not of the form `[M,N]`, a row vector is returned.

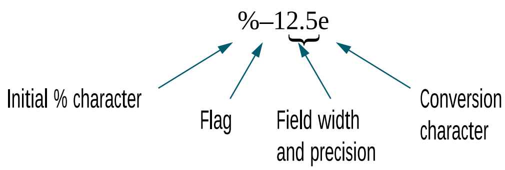
`sscanf` differs from its C language namesakes `scanf()` and `fscanf()` in an important respect — it is *vectorized* in order to return a matrix argument. The *`format`* string is cycled through the file until an end-of-file is reached or the amount of data specified by `size` is read in.

`[A,count,errmsg,nextindex] = sscanf(...)` reads data from MATLAB string variable `s`, converts it according to the specified *`format`* string, and returns it in matrix `A`. `count` is an optional output argument that returns the number of elements successfully read. `errmsg` is an optional output argument that returns an error message string if an error occurred or an empty matrix if an error did not occur. `nextindex` is an optional output argument specifying one more than the number of characters scanned in `s`.

Remarks

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

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



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

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

Valid conversion characters are:

`%c`	Sequence of characters; number specified by field width
`%d`	Decimal numbers
`%e`, `%f`, `%g`	Floating-point numbers
`%i`	Signed integer
`%o`	Signed octal integer
`%s`	A series of non-whitespace characters
`%u`	Signed decimal integer

`%x`	Signed hexadecimal integer
`[...]`	Sequence of characters (scanlist)

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

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

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

Examples

The statements

```
s = '2.7183  3.1416';
A = sscanf(s,'%f')
```

create a two-element vector containing poor approximations to e and `pi`.

See Also

`eval`	Interpret strings containing MATLAB expressions
`sprintf`	Write formatted data to a string

Purpose MATLAB startup M-file

Syntax

```
startup
```

Description At startup time, MATLAB automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`. On multiuser or networked systems, `matlabrc.m` is reserved for use by the system manager. The file `matlabrc.m` invokes the file `startup.m` if it exists on MATLAB's search path.

You can create a startup file in your own MATLAB directory. The file can include physical constants, handle graphics defaults, engineering conversion factors, or anything else you want predefined in your workspace.

Algorithm Only `matlabrc.m` is actually invoked by MATLAB at startup. However, `matlabrc.m` contains the statements

```
if exist('startup')==2
    startup
end
```

that invoke `startup.m`. You can extend this process to create additional startup M-files, if required.

See Also

`!`	Operating system command
`exist`	Check if a variable or file exists
`matlabrc`	MATLAB startup M-file
`path`	Control MATLAB's directory search path
`quit`	Terminate MATLAB

Purpose

Standard deviation

Syntax

```
s = std(X)
s = std(X,flag)
s = std(X,flag,dim)
```

Definition

There are two common textbook definitions for the standard deviation `s` of a data vector `X`:

$$(1)\quad s = \left(\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^{\frac{1}{2}} \qquad \text{and} \qquad (2)\quad s = \left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^{\frac{1}{2}}$$

where

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

and n is the number of elements in the sample. The two forms of the equation differ only in $n-1$ versus n in the divisor.

Description

`s = std(X)`, where `X` is a vector, returns the standard deviation using (1) above. If `X` is a random sample of data from a normal distribution, s^2 is the best *unbiased* estimate of its variance.

If `X` is a matrix, `std(X)` returns a row vector containing the standard deviation of the elements of each column of `X`. If `X` is a multidimensional array, `std(X)` is the standard deviation of th elements along the first nonsingleton dimension of `X`.

`s = std(X,flag)` for `flag = 0`, is the same as `std(X)`. For `flag = 1`, `std(X,1)` returns the standard deviation using (2) above, producing the second moment of the sample about its mean.

`s = std(X,flag,dim)` computes the standard deviations along the dimension of `X` specified by scalar `dim`.

Examples

For matrix X

```
X =
     1     5     9
     7    15    22

s = std(X,0,1)
s =
   4.2426   7.0711   9.1924

s = std(X,0,2)
s =
   4.000
   7.5056
```

See Also

corrcoef, cov, mean, median

str2num

Purpose

String to number conversion

Syntax

```
x = str2num('str')
```

Description

`x = str2num('str')` converts the string *str*, which is an ASCII character representation of a numeric value, to MATLAB's numeric representation. The string can contain:

- Digits
- A decimal point
- A leading + or – sign
- A letter `e` preceding a power of 10 scale factor
- A letter `i` indicating a complex or imaginary number.

The `str2num` function can also convert string matrices.

Examples

`str2num('3.14159e0')` is approximately π.

To convert a string matrix:

```
str2num(['1 2';'3 4'])

ans =

     1     2
     3     4
```

See Also

`[]` (special characters)	Build arrays
`;` (special characters)	End array rows; suppress printing; separate statements.
`hex2num`	Hexadecimal to double number conversion
`num2str`	Number to string conversion
`sparse`	Create sparse matrix
`sscanf`	Read string under format control

Purpose String concatenation

Syntax

```
t = strcat(s1,s2,s3,...)
```

Description `t = strcat(s1,s2,s3,...)` horizontally concatenates corresponding rows of the character arrays `s1`, `s2`, `s3`, etc. The trailing padding is ignored. All the inputs must have the same number of rows (or any can be a single string). When the inputs are all character arrays, the output is also a character array.

When any of the inputs is a cell array of strings, `strcat` returns a cell array of strings formed by concatenating corresponding elements of `s1`,`s2`, etc. The inputs must all have the same size (or any can be a scalar). Any of the inputs can also be a character array.

Examples Given two 1-by-2 cell arrays `a` and `b`,

```
a =                           b =
    'abcde'    'fghi'             'jkl'    'mn'
```

the command `t = strcat(a,b)` yields:

```
t =
    'abcdejkl'    'fghimn'
```

Given the 1-by-1 cell array `c = {‘Q’}`, the command `t = strcat(a,b,c)` yields:

```
t =
    'abcdejklQ'    'fghimnQ'
```

See Also

`cat`	Concatenate arrays
`cellstr`	Create cell array of strings
`strvcat`	Vertical concatenation of strings

strcmp

Purpose

Compare strings

Syntax

```
k = strcmp('str1','str2')
TF = strcmp(S,T)
```

Description

`k = strcmp('`*str1*`','`*str2*`')` compares the strings *str1* and `str2` and returns logical true (`1`) if the two are identical, and logical false (`0`) otherwise.

`TF = strcmp(S,T)` where either `S` or `T` is a cell array of strings, returns an array `TF` the same size as `S` and `T` containing `1` for those elements of `S` and `T` that match, and `0` otherwise. `S` and `T` must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

Remarks

Note that the value returned by `strcmp` is not the same as the C language convention. In addition, the `strcmp` function is case sensitive; any leading and trailing blanks in either of the strings are explicitly included in the comparison.

Examples

```
strcmp('Yes','No') =
   0
strcmp('Yes','Yes') =
   1

A =
    'MATLAB'              'SIMULINK'
    'Toolboxes'           'The MathWorks'

B =
    'Handle Graphics'     'Real Time Workshop'
    'Toolboxes'           'The MathWorks'

C =
    'Signal Processing'     'Image Processing'
    'MATLAB'                'SIMULINK'

strcmp(A,B)
ans =
     0     0
     1     1

strcmp(A,C)
ans =
     0     0
     0     0
```

See Also

`findstr`	Find one string within another
`strncmp`	Compare the first n characters of two strings
`strmatch`	Find possible matches for a string

strings

Purpose

MATLAB string handling

Syntax

```
S = 'Any Characters'
S = string(X)
X = numeric(S)
```

Description

`S = 'Any Characters'` is a vector whose components are the numeric codes for the characters (the first 127 codes are ASCII). The actual characters displayed depend on the character set encoding for a given font. The length of `S` is the number of characters. A quote within the string is indicated by two quotes.

`S = string(X)` can be used to convert an array that contains positive integers representing numeric codes into a MATLAB character array.

`X = double(S)` converts the string to its equivalent numeric codes.

`isstr(S)` tells if `S` is a string variable.

Example

```
s = ['It is 1 o''clock', 7]
```

See Also

`char`	Create character array (string)

Purpose

Justify a character array

Syntax

`strjust(S)`

Description

`strjust(S)` returns a right-justified version of the character array `S`.

See Also

`deblank`	Strip trailing blanks from the end of a string

strmatch

Purpose Find possible matches for a string

Syntax

```
i = strmatch('str',STRS)
i = strmatch('str',STRS,'exact')
```

Description `i = strmatch('`*`str`*`',STRS)` looks through the rows of the character array or cell array of strings `STRS` to find strings that begin with string *str*, returning the matching row indices. `strmatch` is fastest when `STRS` is a character array.

`i = strmatch('`*`str`*`',STRS,'exact')` returns only the indices of the strings in `STRS` matching *str* exactly.

Examples The statement

```
i = strmatch('max',strvcat('max','minimax','maximum'))
```

returns `i = [1; 3]` since rows 1 and 3 begin with `'max'`. The statement

```
i = strmatch('max',strvcat('max','minimax','maximum'),'exact')
```

returns `i = 1`, since only row 1 matches `'max'` exactly.

See Also

`findstr`	Find one string within another
`strcmp`	Compare strings
`strncmp`	Compare the first n characters of two strings
`strvcat`	Vertical concatenation of strings

Purpose Compare the first n characters of two strings

Syntax

```
k = strncmp('str1','str2',n)
TF = strncmp(S,T,n)
```

Description `k = strncmp('str1','str2',n)` returns logical true (1) if the first n characters of the strings str1 and str2 are the same, and returns logical false (0) otherwise. Arguments *str1* and *str2* may also be cell arrays of strings.

`TF = strncmp(S,T,N)` where either S or T is a cell array of strings, returns an array TF the same size as S and T containing 1 for those elements of S and T that match (up to n characters), and 0 otherwise. S and T must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

Remarks The command strncmp is case sensitive. Any leading and trailing blanks in either of the strings are explicitly included in the comparison.

See Also

findstr	Find one string within another
strcmp	Compare strings
strmatch	Find possible matches for a string

strrep

Purpose String search and replace

Syntax

`str = strrep(`*str1*`,`*str2*`,`*str3*`)`

Description

`str = strrep(`*str1*`,`*str2*`,`*str3*`)` replaces all occurrences of the string *str2* within string *str1* with the string `str3`.

`strrep(`*str1*`,`*str2*`,`*str3*`)`, when any of *str1*, *str2*, or *str3* is a cell array of strings, returns a cell array the same size as *str1*, *str2* and *str3* obtained by performing a `strrep` using corresponding elements of the inputs. The inputs must all be the same size (or any can be a scalar cell). Any one of the strings can also be a character array with the right number of rows.

Examples

```
s1 = 'This is a good example.';
str = strrep(s1,'good','great')
str =
This is a great example.

A =
    'MATLAB'                 'SIMULINK'
    'Toolboxes'              'The MathWorks'

B =
    'Handle Graphics'    'Real Time Workshop'
    'Toolboxes'          'The MathWorks'

C =
    'Signal Processing'    'Image Processing'
    'MATLAB'               'SIMULINK'

strrep(A,B,C)
ans =
    'MATLAB'    'SIMULINK'
    'MATLAB'    'SIMULINK'
```

See Also

`findstr` Find one string within another

Purpose

First token in string

Syntax

```
token = strtok('str',delimiter)
token = strtok('str')
[token,rem] = strtok(...)
```

Description

`token = strtok('str',delimiter)` returns the first token in the text string *str*, that is, the first set of characters before a delimiter is encountered. The vector `delimiter` contains valid delimiter characters.

`token = strtok('str')` uses the default delimiters, the white space characters. These include tabs (ASCII 9), carriage returns (ASCII 13), and spaces (ASCII 32).

`[token,rem] = strtok(...)` returns the remainder `rem` of the original string. The remainder consists of all characters from the first delimiter on.

Examples

```
s = 'This is a good example.';
[token,rem] = strtok(s)
token =
This
rem =
 is a good example.
```

See Also

`findstr`	Find one string within another
`strmatch`	Find possible matches for a string

Purpose Create structure array

Syntax

```
s = struct('field1',values1,'field2',values2,...)
```

Description `s = struct('field1',values1,'field2',values2,...)` creates a structure array with the specified fields and values. The value arrays `values1`, `values2`, etc. must be cell arrays of the same size or scalar cells. Corresponding elements of the value arrays are placed into corresponding structure array elements. The size of the resulting structure is the same size as the value arrays.

Examples The command

```
s = struct('type',{'big','little'},'color',{'red'},'x',{3 4})
```

produces a structure array `s`:

```
s =
1x2 struct array with fields:
    type
    color
    x
```

The value arrays have been distributed among the fields of `s`:

```
s(1)
ans =
     type: 'big'
    color: 'red'
        x: 3

s(2)
ans =
     type: 'little'
    color: 'red'
        x: 4
```

See Also

`fieldnames`	Field names of a structure
`getfield`	Get field of structure array
`rmfield`	Remove structure fields
`setfield`	Set field of structure array

Purpose Structure to cell array conversion

Syntax

```
c = struct2cell(s)
```

Description `c = struct2cell(s)` converts the `m`-by-`n` structure `s` (with `p` fields) into a `p`-by-`m`-by-`n` cell array `c`.

If structure `s` is multidimensional, cell array `c` has size `[p size(s)]`.

Examples The commands

```
clear s, s.category = 'tree';
s.height = 37.4; s.name = 'birch';
```

create the structure

```
s =
    category: 'tree'
      height: 37.4000
        name: 'birch'
```

Converting the structure to a cell array,

```
c = struct2cell(s)

c =
    'tree'
    [37.4000]
    'birch'
```

See Also `cell2struct`, `fields`

strvcat

Purpose Vertical concatenation of strings

Syntax

```
S = strvcat(t1,t2,t3,...)
```

Description `S = strvcat(t1,t2,t3,...)` forms the character array `S` containing the text strings (or string matrices) `t1,t2,t3,...` as rows. Spaces are appended to each string as necessary to form a valid matrix. Empty arguments are ignored.

Remarks If each text parameter, `ti`, is itself a character array, `strvcat` appends them vertically to create arbitrarily large string matrices.

Examples The command `strvcat('Hello','Yes')` is the same as `['Hello';'Yes  ']`, except that `strvcat` performs the padding automatically.

```
t1 = 'first';t2 = 'string';t3 = 'matrix';t4 = 'second';

S1 = strvcat(t1,t2,t3)          S2 = strvcat(t4,t2,t3)

S1 =                            S2 =

first                           second
string                          string
matrix                          matrix

S3 = strvcat(S1,S2)

S3 =
first
string
matrix
second
string
matrix
```

See Also

`cat`	Concatenate arrays
`int2str`	Integer to string conversion
`mat2str`	Convert a matrix into a string
`num2str`	Number to string conversion
`string`	Convert numeric values to string

Purpose

Single index from subscripts

Syntax

```
IND = sub2ind(siz,I,J)
IND = sub2ind(siz,I1,I2,...,In)
```

Description

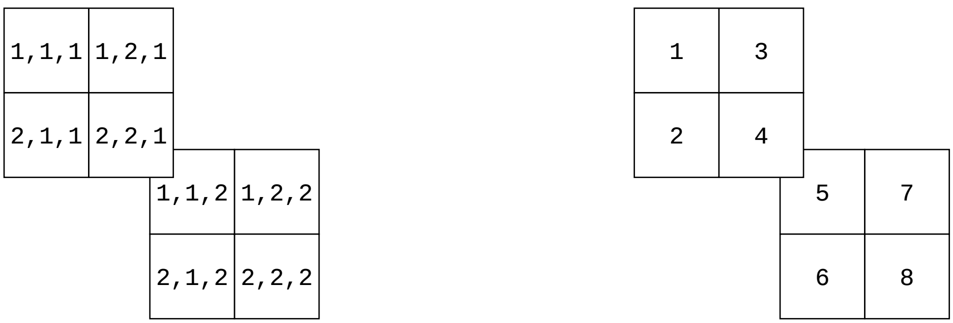
The `sub2ind` command determines the equivalent single index corresponding to a set of subscript values.

`IND = sub2ind(siz,I,J)` returns the linear index equivalent to the row and column subscripts in the arrays `I` and `J` for an matrix of size *siz*.

`IND = sub2ind(siz,I1,I2,...,In)` returns the linear index equivalent to the `n` subscripts in the arrays `I1,I2,...,In` for an array of size *siz*.

Examples

The mapping from subscripts to linear index equivalents for a 2-by-2-by-2 array is:



See Also

`ind2sub`	Subscripts from linear index
`find`	Find indices and values of nonzero elements

subsasgn

Purpose Overloaded method for `A(i)=B`, `A{i}=B`, and `A.field=B`

Syntax `A = subsasgn(A,S,B)`

Description `A = subsasgn(A,S,B)` is called for the syntax `A(i)=B`, `A{i}=B`, or `A.i=B` when `A` is an object. `S` is a structure array with the fields:

- `type`: A string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts; `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.
- `subs`: A cell array or string containing the actual subscripts.

Examples The syntax `A(1:2,:)=B` calls `A=subsasgn(A,S,B)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs = {1:2,':'}`. A colon used as a subscript is passed as the string `':'`.

The syntax `A{1:2}=B` calls `A=subsasgn(A,S,B)` where `S.type='{}'` .

The syntax `A.field=B` calls `subsasgn(A,S,B)` where `S.type='.'` and `S.subs='field'`.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)=B` calls `A=subsasgn(A,S,B)` where `S` is 3-by-1 structure array with the following values:

`S(1).type='()'`	`S(2).type='.'`	`S(3).type='()'`
`S(1).subs={1,2}`	`S(2).subs='name'`	`S(3).subs={3:5}`

See Also `subsref` Overloaded method for `A(i)`, `A{i}` and `A.field`

See *Using MATLAB* for more information about overloaded methods and `subsasgn`.

Purpose

Overloaded method for `X(A)`

Syntax

```
i = subsindex(A)
```

Description

`i = subsindex(A)` is called for the syntax `'X(A)'` when `A` is an object. `subsindex` must return the value of the object as a zero-based integer index (`i` must contain integer values in the range `0` to `prod(size(X))-1`). `subsindex` is called by the default `subsref` and `subsasgn` functions, and you can call it if you overload these functions.

See Also

`subsasgn`	Overloaded method for `A(i)=B`, `A{i}=B`, and `A.field=B`
`subsref`	Overloaded method for `A(i)`, `A{i}` and `A.field`

subsref

Purpose

Overloaded method for `A(I)`, `A{I}` and `A.field`

Syntax

```
B = subsref(A,S)
```

Description

`B = subsref(A,S)` is called for the syntax `A(i)`, `A{i}`, or `A.i` when `A` is an object. `S` is a structure array with the fields:

- `type`: A string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts; `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.
- `subs`: A cell array or string containing the actual subscripts.

Examples

The syntax `A(1:2,:)` calls `subsref(A,S)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs = {1:2,':'}`. A colon used as a subscript is passed as the string `':'`.

The syntax `A{1:2}` calls `subsref(A,S)` where `S.type='{}'` .

The syntax `A.field` calls `subsref(A,S)` where `S.type='.'` and `S.subs='field'`.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)` calls `subsref(A,S)` where `S` is 3-by-1 structure array with the following values:

`S(1).type='()'`	`S(2).type='.'`	`S(3).type='()'`
`S(1).subs={1,2}`	`S(2).subs='name'`	`S(3).subs={3:5}`

See Also

`subsasgn` Overloaded method for A(i)=B, A{i}=B, and A.field=B

See *Using MATLAB* for more information about overloaded methods and `subsref`.

Purpose

Angle between two subspaces

Syntax

```
theta = subspace(A,B)
```

Description

`theta = subspace(A,B)` finds the angle between two subspaces specified by the columns of `A` and `B`. If `A` and `B` are column vectors of unit length, this is the same as `acos(A'*B)`.

Remarks

If the angle between the two subspaces is small, the two spaces are nearly linearly dependent. In a physical experiment described by some observations `A`, and a second realization of the experiment described by `B`, `subspace(A,B)` gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.

Examples

Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.

```
H = hadamard(8);
A = H(:,2:4);
B = H(:,5:8);
```

Note that matrices `A` and `B` are different sizes— `A` has three columns and `B` four. It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

```
theta = subspace(A,B)
theta =
    1.5708
```

That `A` and `B` are orthogonal is shown by the fact that `theta` is equal to $\pi/2$.

```
theta - pi/2
ans =
     0
```

Purpose Sum of array elements

Syntax

```
B = sum(A)
B = sum(A,dim)
```

Description

`B = sum(A)` returns sums along different dimensions of an array.

If `A` is a vector, `sum(A)` returns the sum of the elements.

If `A` is a matrix, `sum(A)` treats the columns of `A` as vectors, returning a row vector of the sums of each column.

If `A` is a multidimensional array, `sum(A)` treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

`B = sum(A,dim)` sums along the dimension of `A` specified by scalar `dim`.

Remarks

`sum(diag(X))` is the `trace` of `X`.

Examples

The magic square of order 3 is

```
M = magic(3)
M =
     8    1    6
     3    5    7
     4    9    2
```

This is called a magic square because the sums of the elements in each column are the same.

```
sum(M) =
     15    15    15
```

as are the sums of the elements in each row, obtained by transposing:

```
sum(M') =
     15    15    15
```

See Also

`cumsum`	Cumulative sum
`diff`	Differences and approximate derivatives
`prod`	Product of array elements
`trace`	Sum of diagonal elements

Purpose Superior class relationship

Syntax

```
superiorto('class1','class2',...)
```

Description The `superiorto` function establishes a hierarchy that determines the order in which MATLAB calls object methods.

`superiorto('class1','class2',...)` invoked within a class constructor method (say `myclass.m`) indicates that `myclass`'s method should 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: `superiorto('class_a')`. Then `e = fun(a,c)` or `e = fun(c,a)` invokes `class_c/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 `inferiorto` Inferior class relationship

Purpose Singular value decomposition

Syntax

```
s = svd(X)
[U,S,V] = svd(X)
[U,S,V] = svd(X,0)
```

Description The svd command computes the matrix singular value decomposition.

s = svd(X) returns a vector of singular values.

[U,S,V] = svd(X) produces a diagonal matrix S of the same dimension as X, with nonnegative diagonal elements in decreasing order, and unitary matrices U and V so that X = U*S*V'.

[U,S,V] = svd(X,0) produces the "economy size" decomposition. If X is m-by-n with m > n, then svd computes only the first n columns of U and S is n-by-n.

Examples For the matrix

```
X =
     1    2
     3    4
     5    6
     7    8
```

the statement

```
[U,S,V] = svd(X)
```

produces

```
U =
    0.1525    0.8226   -0.3945   -0.3800
    0.3499    0.4214    0.2428    0.8007
    0.5474    0.0201    0.6979   -0.4614
    0.7448   -0.3812   -0.5462    0.0407
```

```
S =
    14.2691         0
          0    0.6268
          0         0
          0         0
V =
    0.6414   -0.7672
    0.7672    0.6414
```

The economy size decomposition generated by

```
[U,S,V] = svd(X,0)
```

produces

```
U =
    0.1525    0.8226
    0.3499    0.4214
    0.5474    0.0201
    0.7448   -0.3812
S =
   14.2691         0
         0    0.6268
V =
    0.6414   -0.7672
    0.7672    0.6414
```

Algorithm

The svd command uses the LINPACK routine ZSVDC.

Diagnostics

If the limit of 75 QR step iterations is exhausted while seeking a singular value, this message appears:

```
Solution will not converge.
```

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

svds

Purpose A few singular values

Syntax

```
[U,S,V] = svds(A,k)
```

Description

`[U,S,V] = svds(A,k)` computes the `k` largest singular values and singular vectors of the matrix `A`. `k = 5` is the default. If `A` is `m`-by-`n`, then `U` is `m`-by-`k` with orthonormal columns, `S` is `k`-by-`k` diagonal, `V` is `n`-by-`k` with orthonromal columns, and `U*S*V'` is the closest rank `k` approximation to `A`.

`[U,S,V] = svds(A,k,0)` computes the `k` smallest singular values and singular vectors.

`s = svds(A,k,...)` returns just a vector of singular values.

See Also

svd	Singular value decomposition
`eigs`	Find a few eigenvalues and eigenvectors

Purpose Switch among several cases based on expression

Syntax

```
switch switch_expr
  case case_expr
    statement,...,statement
  case {case_expr1,case_expr2,case_expr3,...}
    statement,...,statement
...
  otherwise
    statement,...,statement
end
```

Discussion The `switch` statement syntax is a means of conditionally executing code. In particular, `switch` executes one set of statements selected from an arbitrary number of alternatives. Each alternative is called a *case*, and consists of:

- The `case` statement
- One or more case expressions
- One or more statements

In its most basic syntax, `switch` executes only the statements associated with the first case where *switch_expr* == *case_expr*. When the case expression is a cell array (as in the second case above), the `case_expr` matches if any of the elements of the cell array match the switch expression.If none of the case expressions matches the switch expression, then control passes to the `otherwise` case (if it exists). Only one case is executed, and program execution resumes with the statement after the `end`.

The `switch_expr` can be a scalar or a string. A scalar `switch_expr` matches a `case_expr` if `switch_expr==case_expr`. A string `switch_expr` matches a `case_expr` if `strcmp(switch_expr,case_expr)` returns `1` (true).

Examples

Assume `method` exists as a string variable:

```
switch lower(method)
    case {'linear','bilinear'}, disp('Method is linear')
    case 'cubic', disp('Method is cubic')
    case 'nearest', disp('Method is nearest')
    otherwise, disp('Unknown method.')
end
```

See Also

`case`, `end`, `if`, `otherwise`, `while`

symmmd

Purpose

Sparse symmetric minimum degree ordering

Syntax

```
p = symmmd(S)
```

Description

`p = symmmd(S)` returns a symmetric minimum degree ordering of `S`. For a symmetric positive definite matrix `S`, this is a permutation `p` such that `S(p,p)` tends to have a sparser Cholesky factor than `S`. Sometimes `symmmd` works well for symmetric indefinite matrices too.

Remarks

The minimum degree ordering is automatically used by `\` and `/` for the solution of symmetric, positive definite, sparse linear systems.

Some options and parameters associated with heuristics in the algorithm can be changed with `spparms`.

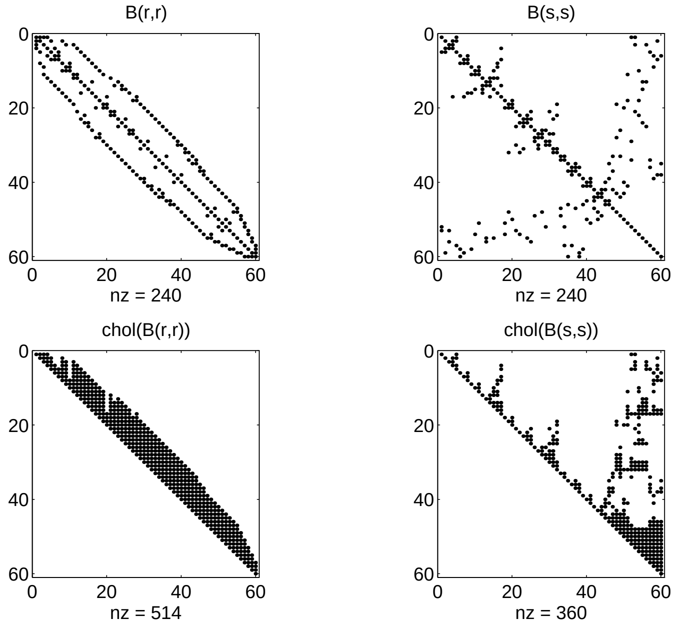
Algorithm

The symmetric minimum degree algorithm is based on the column minimum degree algorithm. In fact, `symmmd(A)` just creates a nonzero structure `K` such that `K'*K` has the same nonzero structure as `A` and then calls the column minimum degree code for `K`.

Examples

Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the symrcm reference page.

```
B = bucky+4*speye(60);
r = symrcm(B);
p = symmmd(B);
R = B(r,r);
S = B(p,p);
subplot(2,2,1), spy(R), title('B(r,r)')
subplot(2,2,2), spy(S), title('B(s,s)')
subplot(2,2,3), spy(chol(R)), title('chol(B(r,r))')
subplot(2,2,4), spy(chol(S)), title('chol(B(s,s))')
```



Even though this is a very small problem, the behavior of both orderings is typical. RCM produces a matrix with a narrow bandwidth which fills in almost completely during the Cholesky factorization. Minimum degree produces a structure with large blocks of contiguous zeros which do not fill in during the factorization. Consequently, the minimum degree ordering requires less time and storage for the factorization.

See Also

`colmmd`	Sparse column minimum degree permutation
`colperm`	Sparse column permutation based on nonzero count
`symrcm`	Sparse reverse Cuthill-McKee ordering

References

[1] Gilbert, John R., Cleve Moler, and Robert Schreiber, “Sparse Matrices in MATLAB: Design and Implementation,” *SIAM Journal on Matrix Analysis and Applications 13*, 1992, pp. 333-356.

Purpose

Sparse reverse Cuthill-McKee ordering

Syntax

```
r = symrcm(S)
```

Description

`r = symrcm(S)` returns the symmetric reverse Cuthill-McKee ordering of `S`. This is a permutation `r` such that `S(r,r)` tends to have its nonzero elements closer to the diagonal. This is a good preordering for LU or Cholesky factorization of matrices that come from long, skinny problems. The ordering works for both symmetric and nonsymmetric `S`.

For a real, symmetric sparse matrix, `S`, the eigenvalues of `S(r,r)` are the same as those of `S`, but `eig(S(r,r))` probably takes less time to compute than `eig(S)`.

Algorithm

The algorithm first finds a pseudoperipheral vertex of the graph of the matrix. It then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudoperipheral vertex. The implementation is based closely on the SPARSPAK implementation described by George and Liu.

Examples

The statement

```
B = bucky
```

uses an M-file in the `demos` toolbox to generate the adjacency graph of a truncated icosahedron. This is better known as a soccer ball, a Buckminster Fuller geodesic dome (hence the name `bucky`), or, more recently, as a 60-atom carbon molecule. There are 60 vertices. The vertices have been ordered by numbering half of them from one hemisphere, pentagon by pentagon; then reflecting into the other hemisphere and gluing the two halves together. With this numbering, the matrix does not have a particularly narrow bandwidth, as the first spy plot shows

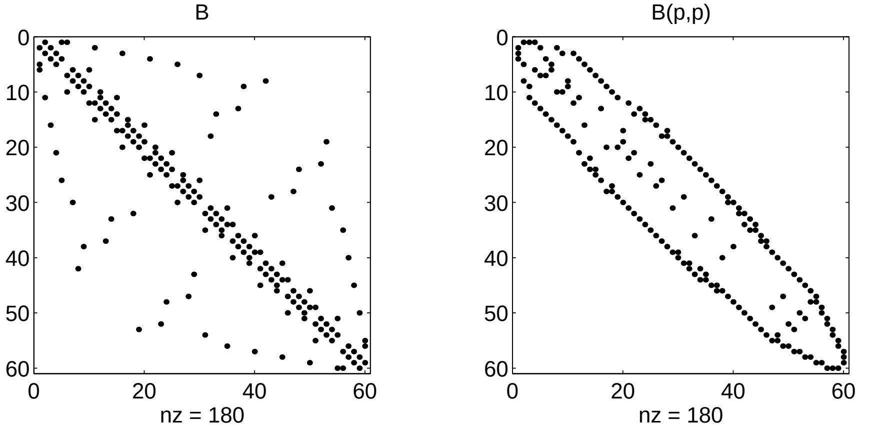
```
subplot(1,2,1), spy(B), title('B')
```

The reverse Cuthill-McKee ordering is obtained with

```
p = symrcm(B);
R = B(p,p);
```

The spy plot shows a much narrower bandwidth:

```
subplot(1,2,2), spy(R), title('B(p,p)')
```



This example is continued in the reference pages for `symmmd`.

The bandwidth can also be computed with

```
[i,j] = find(B);
bw = max(i-j) + 1
```

The bandwidths of `B` and `R` are 35 and 12, respectively.

See Also

`colmmd`	Sparse column minimum degree permutation
`colperm`	Sparse column permutation based on nonzero count
`symmmd`	Sparse symmetric minimum degree ordering

References

[1] George, Alan and Joseph Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[2] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," to appear in *SIAM Journal on Matrix Analysis*, 1992. A slightly expanded version is also available as a technical report from the Xerox Palo Alto Research Center.

tan, tanh

Purpose

Tangent and hyperbolic tangent

Syntax

```
Y = tan(X)
Y = tanh(X)
```

Description

The `tan` and `tanh` functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

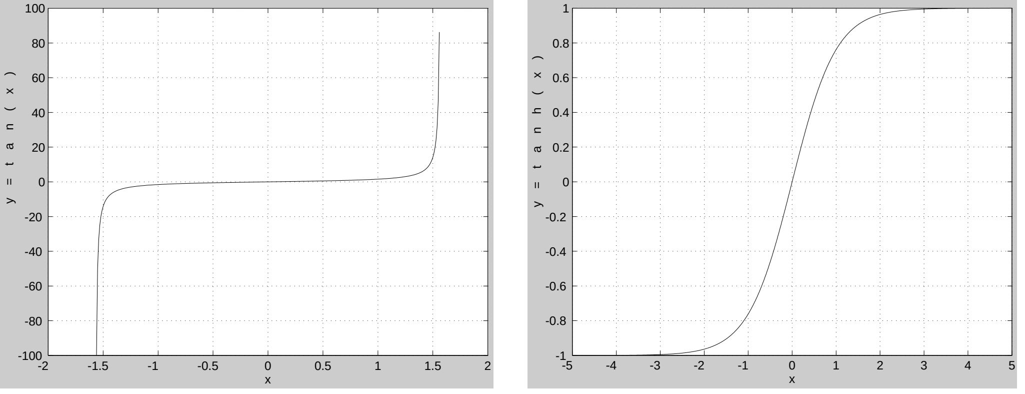
`Y = tan(X)` returns the circular tangent of each element of `X`.

`Y = tanh(X)` returns the hyperbolic tangent of each element of `X`.

Examples

Graph the tangent function over the domain $-\pi/2 < x < \pi/2$, and the hyperbolic tangent function over the domain $-5 \le x \le 5$.

```
x = (-pi/2)+0.01:0.01:(pi/2)-0.01; plot(x,tan(x))
x = -5:0.01:5; plot(x,tanh(x))
```



The expression `tan(pi/2)` does not evaluate as infinite but as the reciprocal of the floating point accuracy `eps` since `pi` is only a floating-point approximation to the exact value of π.

Algorithm

$$\tan(z) = \frac{\sin(z)}{\cos(z)}$$

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$$

See Also

`atan`, `atan2`

tempdir

Purpose Return the name of the system's temporary directory

Syntax

```
tmp_dir = tempdir
```

Description `tmp_dir = tempdir` returns the name of the system's temporary directory, if one exists. This function does not create a new directory.

See Also

`tempname`	Unique name for temporary file

Purpose Unique name for temporary file

Syntax

```
tempname
```

Description `tempname` returns a unique string beginning with the characters `tp`. This string is useful as a name for a temporary file.

See Also

`tempdir`	Return the name of the system's temporary directory

tic, toc

Purpose Stopwatch timer

Syntax

```
tic
   any statements
toc
t = toc
```

Description `tic` starts a stopwatch timer.

`toc` prints the elapsed time since `tic` was used.

`t = toc` returns the elapsed time in `t`.

Examples This example measures how the time required to solve a linear system varies with the order of a matrix.

```
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic
    x = A\b;
    t(n) = toc;
end
plot(t)
```

See Also

`clock`	Current time as a date vector
`cputime`	Elapsed CPU time
`etime`	Elapsed time

Purpose

Toeplitz matrix

Syntax

```
T = toeplitz(c,r)
T = toeplitz(r)
```

Description

A *Toeplitz* matrix is defined by one row and one column. A *symmetric Toeplitz* matrix is defined by just one row. `toeplitz` generates Toeplitz matrices given just the row or row and column description.

`T = toeplitz(c,r)` returns a nonsymmetric Toeplitz matrix `T` having `c` as its first column and `r` as its first row. If the first elements of `c` and `r` are different, a message is printed and the column element is used.

`T = toeplitz(r)` returns the symmetric or Hermitian Toeplitz matrix formed from vector `r`, where `r` defines the first row of the matrix.

Examples

A Toeplitz matrix with diagonal disagreement is

```
c = [1  2  3  4  5];
r = [1.5  2.5  3.5  4.5  5.5];
toeplitz(c,r)
Column wins diagonal conflict:
ans =
    1.000    2.500    3.500    4.500    5.500
    2.000    1.000    2.500    3.500    4.500
    3.000    2.000    1.000    2.500    3.500
    4.000    3.000    2.000    1.000    2.500
    5.000    4.000    3.000    2.000    1.000
```

See Also

`hankel`	Hankel matrix

trace

Purpose Sum of diagonal elements

Syntax

```
b = trace(A)
```

Description `b = trace(A)` is the sum of the diagonal elements of the matrix `A`.

Algorithm `trace` is a single-statement M-file.

```
t = sum(diag(A));
```

See Also

`det`	Matrix determinant
`eig`	Eigenvalues and eigenvectors

Purpose Trapezoidal numerical integration

Syntax

```
Z = trapz(Y)
Z = trapz(X,Y)
Z = trapz(...,dim)
```

Description `Z = trapz(Y)` computes an approximation of the integral of `Y` via the trapezoidal method (with unit spacing). To compute the integral for spacing other than one, multiply `Z` by the spacing increment.

If `Y` is a vector, `trapz(Y)` is the integral of `Y`.

If `Y` is a matrix,`trapz(Y)` is a row vector with the integral over each column.

If `Y` is a multidimensional array, `trapz(Y)` works across the first nonsingleton dimension.

`Z = trapz(X,Y)` computes the integral of `Y` with respect to `X` using trapezoidal integration.

If `X` is a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`, `trapz(X,Y)` operates across this dimension.

`Z = trapz(...,dim)` integrates across the dimension of `Y` specified by scalar `dim`. The length of `X`, if given, must be the same as `size(Y,dim)`.

Examples The exact value of $\int_0^{\pi} \sin(x)\,dx$ is 2.

To approximate this numerically on a uniformly spaced grid, use

```
X = 0:pi/100:pi;
Y = sin(x);
```

Then both

```
Z = trapz(X,Y)
```

and

```
Z = pi/100*trapz(Y)
```

produce

```
Z =
    1.9998
```

A nonuniformly spaced example is generated by

```
X = sort(rand(1,101)*pi);
Y = sin(X);
Z = trapz(X,Y);
```

The result is not as accurate as the uniformly spaced grid. One random sample produced

```
Z =
    1.9984
```

See Also

`cumsum`	Cumulative sum
`cumtrapz`	Cumulative trapezoidal numerical integration

Purpose Lower triangular part of a matrix

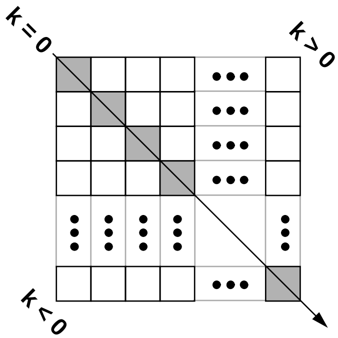
Syntax

```
L = tril(X)
L = tril(X,k)
```

Description

`L = tril(X)` returns the lower triangular part of `X`.

`L = tril(X,k)` returns the elements on and below the kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

`tril(ones(4,4),-1)` is

```
     0     0     0     0
     1     0     0     0
     1     1     0     0
     1     1     1     0
```

See Also

`diag`	Diagonal matrices and diagonals of a matrix
`triu`	Upper triangular part of a matrix

triu

Purpose

Upper triangular part of a matrix

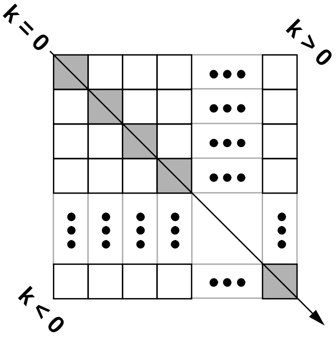
Syntax

```
U = triu(X)
U = triu(X,k)
```

Description

`U = triu(X)` returns the upper triangular part of `X`.

`U = triu(X,k)` returns the element on and above the kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

`triu(ones(4,4),-1)` is

```
    1    1    1    1
    1    1    1    1
    0    1    1    1
    0    0    1    1
```

See Also

`diag`	Diagonal matrices and diagonals of a matrix
`tril`	Lower triangular part of a matrix

Purpose Search for enclosing Delaunay triangle

Syntax

```
T = tsearch(x,y,TRI,xi,yi)
```

Description

`T = tsearch(x,y,TRI,xi,yi)` returns an index into the rows of `TRI` for each point in `xi`,`yi`. The `tsearch` command returns `NaN` for all points outside the convex hull. Requires a triangulation `TRI` of the points `x`,`y` obtained from `delaunay`.

See Also

`delaunay`	Delaunay triangulation
`dsearch`	Search for nearest point

type

Purpose

List file

Syntax

```
type filename
```

Description

`type` *filename* displays the contents of the specified file in the MATLAB command window given a full pathname or a `MATLABPATH` relative partial pathname. Use pathnames and drive designators in the usual way for your computer's operating system.

If you do not specify a filename extension, the `type` command adds the `.m` extension by default. The `type` command checks the directories specified in MATLAB's search path, which makes it convenient for listing the contents of M-files on the screen.

Examples

`type foo.bar` lists the file `foo.bar`.

`type foo` lists the file `foo.m`.

See Also

`!`	Operating system command
`cd`	Change working directory
`dbtype`	List M-file with line numbers
`delete`	Delete files and graphics objects
`dir`	Directory listing
`path`	Control MATLAB's directory search path
`what`	Directory listing of M-files, MAT-files, and MEX-files
`who`	List directory of variables in memory

See also `partialpath`.

Purpose

Convert to unsigned 8-bit integer

Syntax

```
i = uint8(x)
```

Description

`i = uint8(x)` converts the vector `x` into an unsigned 8-bit integer. `x` can be any numeric object (such as a `double`). The elements of an `uint8` range from 0 to 255. The result for any elements of `x` outside this range is not defined (and may vary from platform to platform). If `x` is already an unsigned 8-bit integer, `uint8` has no effect.

The `uint8` class is primarily meant to store integer values. Most operations that manipulate arrays without changing their elements are defined (examples are `reshape`, `size`, subscripted assignment and subscripted reference). No math operations are defined for `uint8` since such operations are ambiguous on the boundary of the set (for example they could wrap or truncate there). You can define your own methods for `uint8` (as you can for any object) by placing the appropriately named method in an `@uint8` directory within a directory on your path. The Image Processing Toolbox does just that to define additional methods for the `uint8` (such as the logical operators <,>,&, etc.).

Type `help oopfun` for the names of the methods you can overload.

See Also

`double`	Convert to double precision

union

Purpose

Set union of two vectors

Syntax

```
c = union(a,b)
c = union(a,b,'rows')
[c,ia,ib] = union(...)
```

Description

`c = union(a,b)` returns the combined values from `a` and `b` but with no repetitions. The resulting vector is sorted in ascending order. In set theoretic terms, `c = a`$\cup$ `b`.

`c = union(a,b,'rows')` when `a` and `b` are matrices with the same number of columns returns the combined rows from `a` and `b` with no repetitions.

`[c,ia,ib] = union(...)` also returns index vectors `ia` and `ib` such that `c = a(ia)` and `c = b(ib)` or, for row combinations, `c = a(ia,:)` and `c = b(ib,:)`.

A nonvector input array `A` is regarded as a column vector `A(:)`.

Examples

```
a = [-1 0 2 4 6];
b = [-1 0 1 3];
[c,ia,ib] = union(a,b);
c =

    -1     0     1     2     3     4     6

ia =

     3     4     5

ib =

     1     2     3     4
```

See Also

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

Purpose Unique elements of a vector

Syntax

```
b = unique(a)
b = unique(a,'rows')
[b,index] = unique(...)
```

Description `b = unique(A)` returns the same values as in `A` but with no repetitions. The resulting vector is sorted in ascending order. A nonvector input array is regarded as the column vector `a = A(:)`.

`b = unique(a,'rows')` returns the unique rows of `A`.

`[b,i,j] = unique(...)` also returns index vectors `i` and `j` such that `b = a(i)` and `a = b(j)` (or `b = a(i,:)` and `a = b(j,:)`).

Examples

```
a = [1 1 5 6 2 3 3 9 8 6 2 4]
a =
1     1     5     6     2     3     3     9     8     6     2     4
[b,i,j] = unique(a)
b =
     1     2     3     4     5     6     8     9
i =
     2    11     7    12     3    10     9     8
j =
1     1     5     6     2     3     3     8     7     6     2     4
a(i)
ans =
     1     2     3     4     5     6     8     9
b(j)
ans =
1     1     5     6     2     3     3     9     8     6     2     4
```

See Also

`intersect`	Set intersection of two vectors
`ismember`	True for a set member
`setdiff`	Return the set difference of two vectors
`setxor`	Set exclusive-or of two vectors
`union`	Set union of two vectors

unwrap

Purpose

Correct phase angles

Syntax

```
Q = unwrap(P)
Q = unwrap(P,tol)
Q = unwrap(P,[],dim)
Q = unwrap(P,tol,dim)
```

Description

`Q = unwrap(P)` corrects the radian phase angles in array `P` by adding multiples of $\pm 2\pi$ when absolute jumps between consecutive array elements are greater than π radians. If `P` is a matrix, `unwrap` operates columnwise. If `P` is a multidimensional array, `unwrap` operates on the first nonsingleton dimension.

`Q = unwrap(P,tol)` uses a jump tolerance `tol` instead of the default value, π.

`Q = unwrap(P,[],dim)` unwraps along `dim` using the default tolerance.

`Q = unwrap(P,tol,dim)` uses a jump tolerance of `tol`.

Examples

Array `P` features smoothly increasing phase angles except for discontinuities at elements `(3,1)` and `(1,2)`.

```
P =
         0    7.0686    1.5708    2.3562
    0.1963    0.9817    1.7671    2.5525
    6.6759    1.1781    1.9635    2.7489
    0.5890    1.3744    2.1598    2.9452
```

The function `Q = unwrap(P)` eliminates these discontinuities.

```
Q =
         0    0.7854    1.5708    2.3562
    0.1963    0.9817    1.7671    2.5525
    0.3927    1.1781    1.9635    2.7489
    0.5890    1.3744    2.1598    2.9452
```

Limitations

The `unwrap` function detects branch cut crossings, but it can be fooled by sparse, rapidly changing phase values.

See Also

`abs`	Absolute value and complex magnitude
`angle`	Phase angle

Purpose Convert string to upper case

Syntax

```
t = upper('str')
```

Description

`t = upper('str')` converts any lower-case characters in the string *str* to the corresponding upper-case characters and leaves all other characters unchanged.

Examples

`upper('attention!')` is `ATTENTION!`.

Remarks

Character sets supported:

- Mac: Standard Roman
- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

See Also

`lower`	Convert string to lower case

varargin, varargout

Purpose Pass or return variable numbers of arguments

Syntax

```
function varargout = foo(n)
y = function bar(varargin)
```

Description

`function varargout = foo(n)` returns a variable number of arguments from function `foo.m`.

`y = function bar(varargin)` accepts a variable number of arguments into function `bar.m`.

The `varargin` and `varargout` statements are used only inside a function M-file to contain the optional arguments to the function. Each must be declared as the last argument to a function, collecting all the inputs or outputs from that point onwards. In the declaration, `varargin` and `varargout` must be lowercase.

Examples

The function

```
function myplot(x,varargin)
plot(x,varargin{:})
```

collects all the inputs starting with the second input into the variable `varargin`. `myplot` uses the comma-separated list syntax `varargin{:}` to pass the optional parameters to `plot`. The call

```
myplot(sin(0:.1:1),'color',[.5 .7 .3],'linestyle',':')
```

results in `varargin` being a 1-by-4 cell array containing the values `'color'`, [.5 .7 .3], `'linestyle'`, and `':'`.

The function

```
function [s,varargout] = mysize(x)
nout = max(nargout,1)-1;
s = size(x);
for i=1:nout, varargout(i) = {s(i)}; end
```

returns the size vector and, optionally, individual sizes. So

```
[s,rows,cols] = mysize(rand(4,5));
```

returns `s = [4 5], rows = 4, cols = 5`.

See Also

`nargin`	Number of function arguments
`nargout`	Number of function arguments
`nargchk`	Check number of input arguments

version

Purpose MATLAB version number

Syntax

```
v = version
[v,d] = version
```

Description

`v = version` returns a string `v` containing the MATLAB version number.

`[v,d] = version` also returns a string `d` containing the date of the version.

See Also

`help`	Online help for MATLAB functions and M-files
`whatsnew`	Display README files for MATLAB and toolboxes
`version`	MATLAB version number

Purpose

Voronoi diagram

Syntax

```
voronoi(x,y)
voronoi(x,y,TRI)
h = voronoi(...,'LineSpec')
[vx,vy] = voronoi(...)
```

Definition

Consider a set of coplanar points P. For each point P_x in the set P, you can draw a boundary enclosing all the intermediate points lying closer to P_x than to other points in the set P. Such a boundary is called a *Voronoi polygon*, and the set of all Voronoi polygons for a given point set is called a *Voronoi diagram*.

Description

`voronoi(x,y)` plots the Voronoi diagram for the points `x,y`.

`voronoi(x,y,TRI)` uses the triangulation `TRI` instead of computing it via `delaunay`.

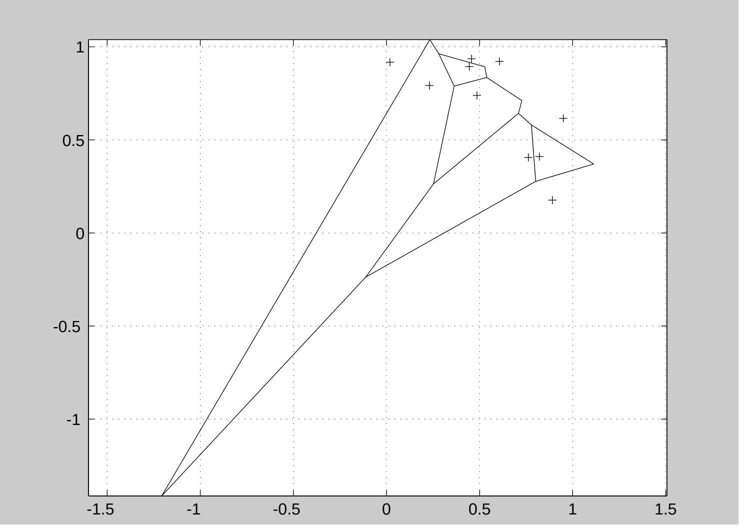
`h = voronoi(...,'LineSpec')` plots the diagram with color and line style specified and returns handles to the line objects created in `h`.

`[vx,vy] = voronoi(...)` returns the vertices of the Voronoi edges in `vx` and `vy` so that `plot(vx,vy,'-',x,y,'.')` creates the Voronoi diagram.

Examples

This code plots the Voronoi diagram for 10 randomly generated points.

```
rand('state',0);
x = rand(1,10); y = rand(1,10);
[vx, vy] = voronoi(x,y);
plot(x,y,'r+',vx,vy,'b-'); axis equal
```



See Also

The `LineSpec` entry in *Using MATLAB Graphics*, and

`convhull`	Convex hull
`delaunay`	Delaunay triangulation
`dsearch`	Search for nearest point

Purpose Display warning message

Syntax

```
warning('message')
warning on
warning off
warning backtrace
warning debug
warning once
warning always
[s,f] = warning
```

Description

`warning('message')` displays the text `'message'` as does the `disp` function, except that with `warning`, message display can be suppressed.

`warning off` suppresses all subsequent warning messages.

`warning on` re-enables them.

`warning backtrace` is the same as `warning on` except that the file and line number that produced the warning are displayed.

`warning debug` is the same as `dbstop if warning` and triggers the debugger when a warning is encountered.

`warning once` displays Handle Graphics backwards compatibility warnings only once per session.

`warning always` displays Handle Graphics backwards compatibility warnings as they are encountered (subject to current warning state).

`[s,f] = warning` returns the current warning state `s` and the current warning frequency `f` as strings.

Remarks

Use `dbstop on warning` to trigger the debugger when a warning is encountered.

See Also

`dbstop`	Set breakpoints in an M-file function
`disp`	Display text or array
`error`	Display error messages

wavread

Purpose Read Microsoft WAVE (`.wav`) sound file

Syntax

```
y = wavread('filename')
[y,Fs,bits] = wavread('filename')
[...] = wavread('filename',N)
[...] = wavread('filename',[N1 N2])
[...] = wavread('filename','size')
```

Description

`wavread` supports multichannel data, with up to 16 bits per sample.

`y = wavread('`*`filename`*`')` loads a WAVE file specified by the string *filename*, returning the sampled data in `y`. The `.wav` extension is appended if no extension is given. Amplitude values are in the range `[-1,+1]`.

`[y,Fs,bits] = wavread('`*`filename`*`')` returns the sample rate (`Fs`) in Hertz and the number of bits per sample (`bits`) used to encode the data in the file.

`[...] = wavread('`*`filename`*`',N)` returns only the first `N` samples from each channel in the file.

`[...] = wavread('`*`filename`*`',[N1 N2])` returns only samples `N1` through `N2` from each channel in the file.

`siz = wavread('`*`filename`*`','size')` returns the size of the audio data contained in the file in place of the actual audio data, returning the vector `siz = [samples channels]`.

See Also

`auread`	Read NeXT/SUN (`.au`) sound file
`wavwrite`	Write Microsoft WAVE (`.wav`) sound file

Purpose Write Microsoft WAVE (`.wav`) sound file

Syntax

```
wavwrite(y,'filename')
wavwrite(y,Fs,'filename')
wavwrite(y,Fs,N,'filename')
```

Description `wavwrite` supports multi-channel 8- or 16-bit WAVE data.

`wavwrite(y,'`*`filename`*`')` writes a WAVE file specified by the string *filename*. The data should be arranged with one channel per column. Amplitude values outside the range `[-1,+1]` are clipped prior to writing.

`wavwrite(y,Fs,'`*`filename`*`')` specifies the sample rate `Fs`, in Hertz, of the data.

`wavwrite(y,Fs,N,'`*`filename`*`')` forces an `N`-bit file format to be written, where `N <= 16`.

See Also

`auwrite`	Write NeXT/SUN (`.au`) sound file
`wavread`	Read Microsoft WAVE (`.wav`) sound file

web

Purpose Point Web browser at file or Web site

Syntax

```
web url
```

Description `web` *url* opens a Web browser and loads the file or Web site specified in the URL (Uniform Resource Locator). The URL can be in any form your browser supports. Generally, the URL specifies a local file or a Web site on the Internet.

Examples `web file:/disk/dir1/dir2/foo.html` points the browser to the file `foo.html`. If the file is on the MATLAB path, `web(['file:' which('foo.html')])` also works.

`web http://www.mathworks.com` loads The MathWorks Web page into your browser. Use `web mailto:email_address` to send e-mail to another site.

The Web browser used is specified in the `docopt` M-file.

See Also

`doc`	Load hypertext documentation
`docopt`	Configure local `doc` access defaults (in online help)

Purpose Day of the week

Syntax

```
[N,S] = weekday(D)
```

Description `[N,S] = weekday(D)` returns the day of the week in numeric (`N`) and string (`S`) form for each element of a serial date number array or date string. The days of the week are assigned these numbers and abbreviations:

N	S	N	S
1	Sun	5	Thu
2	Mon	6	Fri
3	Tue	7	Sat
4	Wed		

Examples Either

```
[n,s] = weekday(728647)
```

or

```
[n,s] = weekday('19-Dec-1994')
```

returns `n = 2` and `s = Mon`.

See Also

`datenum`	Serial date number
`datevec`	Date components
`eomday`	End of month

what

Purpose

Directory listing of M-files, MAT-files, and MEX-files

Syntax

```
what
what dirname
```

Description

`what` by itself, lists the M-files, MAT-files, and MEX-files in the current directory.

`what` *dirname* lists the files in directory *dirname* on MATLAB's search path. It is not necessary to enter the full pathname of the directory. The last component, or last couple of components, is sufficient. Use `what` *class* or `what` *dirname*/`private` to list the files in a method directory or a private directory (for the class named *class*).

Examples

The statements

```
what general
```

and

```
what matlab/general
```

both list the M-files in the `general` directory. The syntax of the path depends on your operating system:

UNIX:	`matlab/general`
VMS:	`MATLAB.GENERAL`
MS-DOS:	`MATLAB\GENERAL`
Macintosh:	`MATLAB:General`

See Also

`dir`	Directory listing
`lookfor`	Keyword search through all help entries
`path`	Control MATLAB's directory search path
`which`	Locate functions and files
`who`	List directory of variables in memory

Purpose Display README files for MATLAB and toolboxes

Syntax

```
whatsnew
whatsnew matlab
whatsnew toolboxpath
```

Description `whatsnew`, by itself, displays the `README` file for the MATLAB product or a specified toolbox. If present, the `README` file summarizes new functionality that is not described in the documentation.

`whatsnew matlab` displays the `README` file for MATLAB.

`whatsnew` *toolboxpath* displays the `README` file for the toolbox specified by the string *toolboxpath*.

Examples

```
whatsnew matlab % MATLAB README file
whatsnew signal % Signal Processing Toolbox README file
```

See Also

`help`	Online help for MATLAB functions and M-files
`lookfor`	Keyword search through all help entries
`path`	Control MATLAB's directory search path
`version`	MATLAB version number
`which`	Locate functions and files

Purpose Locate functions and files

Syntax

```
which fun
which fun -all
which file.ext
which fun1 in fun2
which fun(a,b,c,...)
s = which(...)
```

Description

`which` *fun* displays the full pathname of the specified function. The function can be an M-file, MEX-file, workspace variable, built-in function, or SIMULINK model. The latter three display a message indicating that they are variable, built in to MATLAB, or are part of SIMULINK. Use `which private/`*fun* or `which` *class/fun* or `which` *class*/`private`/*fun* to further qualify the function name for private functions, methods, and private methods (for the class named *class*).

`which` *fun* `-all` displays the paths to all functions with the name *fun*. The first one in the list is the one normally returned by `which`. The others are either shadowed or can be executed in special circumstances. The `-all` flag can be used with all forms of `which`.

`which` *file.ext* displays the full pathname of the specified file.

`which` *fun1* `in` *fun2* displays the pathname to function *fun1* in the context of the M-file *fun2*. While debugging *fun2*, `which` *fun1* does the same thing. You can use this to determine if a local or private version of a function is being called instead of a function on the path.

`which` *fun(a,b,c,...)* displays the path to the specified function with the given input arguments. For example, `which feval(g)`, when `g=inline('sin(x)')`, indicates that `inline/feval.m` is invoked.

`s = which(...)` returns the results of `which` in the string `s` instead of printing it to the screen. `s` will be the string `built-in` or `variable` for built-in functions or variables in the workspace. You must use the functional form of `which` when there is an output argument.

Examples

For example,

```
which inv
```

reveals that `inv` is a built-in function, and

```
which pinv
```

indicates that `pinv` is in the `matfun` directory of the MATLAB Toolbox.

The statement

```
which jacobian
```

probably says

```
jacobian not found
```

because there is no file `jacobian.m` on MATLAB's search path. Contrast this with `lookfor jacobian`, which takes longer to run, but finds several matches to the keyword `jacobian` in its search through all the help entries. (If `jacobian.m` does exist in the current directory, or in some private directory that has been added to MATLAB's search path, `which jacobian` finds it.)

See Also

`dir`, `exist`, `help`, `lookfor`, `path`, `what`, `who`

while

Purpose

Repeat statements an indefinite number of times

Syntax

```
while expression
    statements
end
```

Description

`while` repeats `statements` an indefinite number of times. The `statements` are executed while the real part of *expression* has all nonzero elements. *expression* is usually of the form

```
expression rop expression
```

where *rop* is ==, <, >, <=, >=, or ~=.

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

Examples

The variable `eps` is a tolerance used to determine such things as near singularity and rank. Its initial value is the *machine epsilon*, the distance from `1.0` to the next largest floating-point number on your machine. Its calculation demonstrates `while` loops:

```
eps = 1;
while (1+eps) > 1
    eps = eps/2;
end
eps = eps*2
```

See Also

`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzeros
`break`	Break out of flow control structures
`end`	Terminate for, while, switch, and if statements or indicate last index
`for`	Repeat statements a specific number of times
`if`	Conditionally execute statements
`return`	Return to the invoking function
`switch`	Switch among several cases based on expression

Purpose List directory of variables in memory

Syntax

```
who
whos
who global
whos global
who -file filename
whos -file filename
who ... var1 var2
whos ... var1 var2
s = who(...)
s = whos(...)
```

Description

`who` by itself, lists the variables currently in memory.

`whos` by itself, lists the current variables, their sizes, and whether they have nonzero imaginary parts.

`who global` and `whos global` list the variables in the global workspace.

`who -file` *filename* and `whos -file` *filename* list the variables in the specified MAT-file.

`who ...` *var1 var2* and `whos ...` *var1 var2* restrict the display to the variables specified. The wildcard character `*` can be used to display variables that match a pattern. For instance, `who A*` finds all variables in the current workspace that start with `A`. Use the functional form, such as `whos('-file',` *filename*, *v1*, *v2*), when the filename or variable names are stored in strings.

`s = who(...)` returns a cell array containing the names of the variables in the workspace or file. Use the functional form of `who` when there is an output argument.

`s = whos(...)` returns a structure with the fields:

`name`	variable name
`bytes`	number of bytes allocated for the array
`class`	class of variable

Use the functional form of `whos` when there is an output argument.

See Also `dir`, `exist`, `help`, `what`

Purpose

Wilkinson's eigenvalue test matrix

Syntax

```
W = wilkinson(n)
```

Description

`W = wilkinson(n)` returns one of J. H. Wilkinson's eigenvalue test matrices. It is a symmetric, tridiagonal matrix with pairs of nearly, but not exactly, equal eigenvalues.

Examples

`wilkinson(7)` is

```
3    1    0    0    0    0    0
1    2    1    0    0    0    0
0    1    1    1    0    0    0
0    0    1    0    1    0    0
0    0    0    1    1    1    0
0    0    0    0    1    2    1
0    0    0    0    0    1    3
```

The most frequently used case is `wilkinson(21)`. Its two largest eigenvalues are both about 10.746; they agree to 14, but not to 15, decimal places.

See Also

`eig`	Eigenvalues and eigenvectors
`gallery`	Test matrices
`pascal`	Pascal matrix

wk1read

Purpose

Read a Lotus123 WK1 spreadsheet file into a matrix

Syntax

```
M = wk1read(filename)
M = wk1read(filename,r,c)
M = wk1read(filename,r,c,range)
```

Description

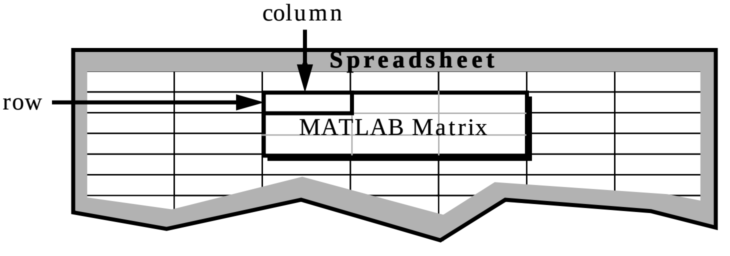
`M = wk1read(`*`filename`*`)` reads a Lotus123 WK1 spreadsheet file into the matrix `M`.

`M = wk1read(`*`filename`*`,r,c)` starts reading at the row-column cell offset specified by `(r,c)`. `r` and `c` are zero based so that `r=0`, `c=0` specifies the first value in the file.

`M = wk1read(`*`filename`*`,r,c,range)` reads the range of values specified by the parameter `range`, where `range` can be:

- A four-element vector specifying the cell range in the format

  `[upper_left_row upper_left_col lower_right_row lower_right_col]`



- A cell range specified as a string; for example, `'A1...C5'`.
- A named range specified as a string; for example, `'Sales'`.

See Also

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

Purpose

Write a matrix to a Lotus123 WK1 spreadsheet file

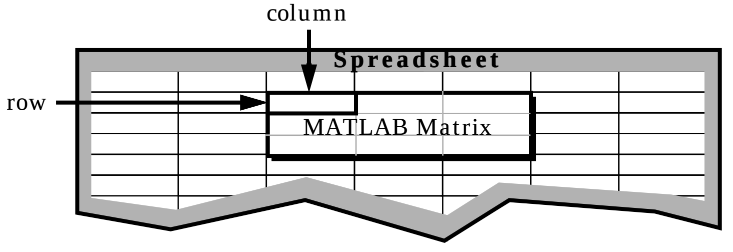
Syntax

```
wk1write(filename,M)
wk1write(filename,M,r,c)
```

Description

`wk1write(`*`filename`*`,M)` writes the matrix `M` into a Lotus123 WK1 spreadsheet file named *filename*.

`wk1write(`*`filename`*`,M,r,c)` writes the matrix starting at the spreadsheet location `(r,c)`. `r` and `c` are zero based so that `r=0`, `c=0` specifies the first cell in the spreadsheet.



See Also

wk1read	Read a Lotus123 WK1 spreadsheet file into a matrix

xor

Purpose

Exclusive or

Syntax

```
C = xor(A,B)
```

Description

`C = xor(A,B)` performs an exclusive OR operation on the corresponding elements of arrays `A` and `B`. The resulting element `C(i,j,...)` is logical true (`1`) if `A(i,j,...)` or `B(i,j,...)`, but not both, is nonzero.

A	B	C
zero	zero	0
zero	nonzero	1
nonzero	zero	1
nonzero	nonzero	0

Examples

Given `A = [0 0 pi eps]` and `B = [0 -2.4 0 1]`, then

```
C = xor(A,B)
C =
     0    1    1    0
```

To see where either `A` or `B` has a nonzero element and the other matrix does not,

```
spy(xor(A,B))
```

See Also

`&`	Logical AND operator
`|`	Logical OR operator
`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzeros
`find`	Find indices and values of nonzero elements

Purpose

Create an array of all zeros

Syntax

```
B = zeros(n)
B = zeros(m,n)
B = zeros([m n])
B = zeros(d1,d2,d3...)
B = zeros([d1 d2 d3...])
B = zeros(size(A))
```

Description

`B = zeros(n)` returns an `n`-by-`n` matrix of zeros. An error message appears if `n` is not a scalar.

`B = zeros(m,n)` or `B = zeros([m n])` returns an `m`-by-`n` matrix of zeros.

`B = zeros(d1,d2,d3...)` or `B = zeros([d1 d2 d3...])` returns an array of zeros with dimensions `d1`-by-`d2`-by-`d3`-by-`...` .

`B = zeros(size(A))` returns an array the same size as `A` consisting of all zeros.

Remarks

The MATLAB language does not have a dimension statement—MATLAB automatically allocates storage for matrices. Nevertheless, most MATLAB programs execute faster if the `zeros` function is used to set aside storage for a matrix whose elements are to be generated one at a time, or a row or column at a time.

Examples

With `n = 1000`, the `for` loop

```
for i = 1:n, x(i) = i; end
```

takes about 1.2 seconds to execute on a Sun SPARC-1. If the loop is preceded by the statement `x = zeros(1,n);` the computations require less than 0.2 seconds.

See Also

`eye`	Identity matrix
`ones`	Create an array of all ones
`rand`	Uniformly distributed random numbers and arrays
`randn`	Normally distributed random numbers and arrays

A

List of Commands

This appendix lists MATLAB commands and functions alphabetically. For a list of commands grouped by functional category, see the Command Summary.

Function Names

Index

Symbols

Numerics

A

B

G

H

M

Q

R

T

U

V