



Computation

Visualization

Programming

# MATLAB Function Reference (Volume 1: Language)

*Version 5*

## How to Contact The MathWorks:

	(508) 647-7000	Phone
	(508) 647-7001	Fax
	The MathWorks, Inc. 24 Prime Park Way Natick, MA 01760-1500	Mail
	<a href="http://www.mathworks.com">http://www.mathworks.com</a> <a href="ftp.mathworks.com">ftp.mathworks.com</a> <a href="comp.soft-sys.matlab">comp.soft-sys.matlab</a>	Web Anonymous FTP server Newsgroup
@	<a href="mailto:support@mathworks.com">support@mathworks.com</a> <a href="mailto:suggest@mathworks.com">suggest@mathworks.com</a> <a href="mailto:bugs@mathworks.com">bugs@mathworks.com</a> <a href="mailto:doc@mathworks.com">doc@mathworks.com</a> <a href="mailto:subscribe@mathworks.com">subscribe@mathworks.com</a> <a href="mailto:service@mathworks.com">service@mathworks.com</a> <a href="mailto:info@mathworks.com">info@mathworks.com</a>	Technical support Product enhancement suggestions Bug reports Documentation error reports Subscribing user registration Order status, license renewals, passcodes Sales, pricing, and general information

*MATLAB Function Reference* (online version, January 1998: Revised for MATLAB 5.2)

© COPYRIGHT 1984 - 1998 by The MathWorks, Inc. All Rights Reserved.

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

U.S. GOVERNMENT: If Licensee is acquiring the software on behalf of any unit or agency of the U. S. Government, the following shall apply:

(a) for units of the Department of Defense:

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013.

(b) for any other unit or agency:

NOTICE - Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, the computer software and accompanying documentation, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Clause 52.227-19(c)(2) of the FAR.

Contractor/manufacturer is The MathWorks Inc., 24 Prime Park Way, Natick, MA 01760-1500.

MATLAB, Simulink, Handle Graphics, and Real-Time Workshop are registered trademarks and Stateflow and Target Language Compiler are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

## Command Summary

1

<b>General Purpose Commands</b> .....	<b>1-2</b>
<b>Operators and Special Characters</b> .....	<b>1-3</b>
<b>Logical Functions</b> .....	<b>1-4</b>
<b>Language Constructs and Debugging</b> .....	<b>1-4</b>
<b>Elementary Matrices and Matrix Manipulation</b> .....	<b>1-6</b>
<b>Specialized Matrices</b> .....	<b>1-7</b>
<b>Elementary Math Functions</b> .....	<b>1-7</b>
<b>Specialized Math Functions</b> .....	<b>1-8</b>
<b>Coordinate System Conversion</b> .....	<b>1-9</b>
<b>Matrix Functions - Numerical Linear Algebra</b> .....	<b>1-9</b>
<b>Data Analysis and Fourier Transform Functions</b> .....	<b>1-10</b>
<b>Polynomial and Interpolation Functions</b> .....	<b>1-12</b>
<b>Function Functions – Nonlinear Numerical Methods</b> .....	<b>1-13</b>
<b>Sparse Matrix Functions</b> .....	<b>1-13</b>
<b>Sound Processing Functions</b> .....	<b>1-15</b>
<b>Character String Functions</b> .....	<b>1-15</b>
<b>Low-Level File I/O Functions</b> .....	<b>1-16</b>

<b>Bitwise Functions</b>	.....	<b>1-18</b>
<b>Structure Functions</b>	.....	<b>1-18</b>
<b>Object Functions</b>	.....	<b>1-18</b>
<b>Cell Array Functions</b>	.....	<b>1-18</b>
<b>Multidimensional Array Functions</b>	.....	<b>1-18</b>

## Reference

2

### List of Commands

<b>Function Names</b>	.....	<b>A-2</b>
-----------------------	-------	------------

# 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 2-25
doc	Display HTML documentation in Web browser.....	page 2-212
help	Online help for MATLAB functions and M-files.....	page 2-369
lasterr	Last error message.....	page 2-433
lastwarn	Last warning message .....	page 2-435
lookfor	Keyword search through all help entries.....	page 2-451
path	Control MATLAB's directory search path .....	page 2-533
profile	Measure and display M-file execution profiles.....	page 2-565
rmpath	Remove directories from MATLAB's search path .....	page 2-606
type	List file .....	page 2-725
version	MATLAB version number .....	page 2-734
what	Directory listing of M-files, MAT-files, and MEX-files .....	page 2-742
whatsnew	Display README files for MATLAB and toolboxes .....	page 2-744
which	Locate functions and files.....	page 2-745

### Managing Variables and the Workspace

clear	Remove items from memory .....	page 2-120
disp	Display text or array .....	page 2-207
length	Length of vector.....	page 2-439
load	Retrieve variables from disk.....	page 2-442
unlock	Prevent M-file clearing.....	page 2-481
ununlock	Allow M-file clearing .....	page 2-485
pack	Consolidate workspace memory .....	page 2-529
save	Save workspace variables on disk.....	page 2-616
size	Array dimensions .....	page 2-632
who, whos	List directory of variables in memory .....	page 2-748

### Controlling the Command Window

echo	Echo M-files during execution.....	page 2-215
format	Control the output display format .....	page 2-291
more	Control paged output for the command window .....	page 2-483

### Working with Files and the Operating Environment

acopy	Copy Macintosh file from one folder to another.....	page 2-20
amove	Move Macintosh file from one folder to another.....	page 2-30
appl escript	Load a compiled AppleScript from a file and execute it .....	page 2-35

arename	Rename Macintosh File.....	page 2-36
areveal	Reveal filename on Macintosh desktop.....	page 2-37
cd	Change working directory .....	page 2-92
copyfile	Copy file .....	page 2-139
delete	Delete files and graphics objects.....	page 2-200
diary	Save session in a disk file.....	page 2-203
dir	Directory listing.....	page 2-206
edit	Edit an M-file.....	page 2-216
fileparts	Filename parts .....	page 2-267
fullfile	Build full filename from parts .....	page 2-308
gestalt	Macintosh gestalt function .....	page 2-343
inmem	Functions in memory .....	page 2-396
matlabroot	Root directory of MATLAB installation .....	page 2-468
tempdir	Return the name of the system's temporary directory.....	page 2-714
mkdir	Make directory.....	page 2-480
tempname	Unique name for temporary file .....	page 2-715
!	Execute operating system command.....	page 2-13

## Starting and Quitting MATLAB

matlabrc	MATLAB startup M-file .....	page 2-467
quit	Terminate MATLAB .....	page 2-582
startup	MATLAB startup M-file .....	page 2-674

## Operators and Special Characters

+	Plus .....	page 2-2
-	Minus .....	page 2-2
*	Matrix multiplication .....	page 2-2
.*	Array multiplication .....	page 2-2
^	Matrix power .....	page 2-2
.^	Array power .....	page 2-2
kron	Kronecker tensor product.....	page 2-432
\	Backslash or left division.....	page 2-2
/	Slash or right division .....	page 2-2
. / and . \	Array division, right and left.....	page 2-2
:	Colon .....	page 2-16
( )	Parentheses.....	page 2-13
[ ]	Brackets.....	page 2-13
{ }	Curly braces .....	page 2-13
.	Decimal point .....	page 2-13
...	Continuation .....	page 2-13

,	Comma.....	page 2-13
;	Semicolon .....	page 2-13
%	Comment.....	page 2-13
!	Exclamation point.....	page 2-13
'	Transpose and quote.....	page 2-13
.	Nonconjugated transpose.....	page 2-13
=	Assignment .....	page 2-13
==	Equality .....	page 2-9
< >	Relational operators .....	page 2-9
&	Logical AND .....	page 2-11
	Logical OR.....	page 2-11
~	Logical NOT .....	page 2-11
xor	Logical EXCLUSIVE OR .....	page 2-756

## Logical Functions

all	Test to determine if all elements are nonzero.....	page 2-28
any	Test for any nonzeros .....	page 2-33
exist	Check if a variable or file exists .....	page 2-244
find	Find indices and values of nonzero elements.....	page 2-271
is*	Detect state.....	page 2-419
*isa	Detect an object of a given class.....	page 2-423
logical	Convert numeric values to logical.....	page 2-447
mislocked	True if M-file cannot be cleared .....	page 2-479

## Language Constructs and Debugging

### MATLAB as a Programming Language

builtin	Execute builtin function from overloaded method .....	page 2-83
eval	Interpret strings containing MATLAB expressions .....	page 2-241
evalin	Evaluate expression in workspace .....	page 2-243
feval	Function evaluation .....	page 2-257
function	Function M-files .....	page 2-309
global	Define global variables .....	page 2-345
nargchk	Check number of input arguments.....	page 2-487
persistent	Define persistent variable.....	page 2-543
script	Script M-files .....	page 2-621

## Control Flow

break	Terminate execution of for or while loop .....	page 2-82
case	Case switch.....	page 2-89
catch	Begin catch block.....	page 2-91
else	Conditionally execute statements .....	page 2-230
elseif	Conditionally execute statements .....	page 2-231
end	Terminate for, while, switch, try, and if statements or indicate last index .....	page 2-233
error	Display error messages .....	page 2-238
for	Repeat statements a specific number of times .....	page 2-289
if	Conditionally execute statements .....	page 2-373
otherwise	Default part of switch statement .....	page 2-528
return	Return to the invoking function .....	page 2-604
switch	Switch among several cases based on expression .....	page 2-704
try	Begin try block.....	page 2-723
warning	Display warning message .....	page 2-737
while	Repeat statements an indefinite number of times .....	page 2-747

## Interactive Input

input	Request user input.....	page 2-398
keyboard	Invoke the keyboard in an M-file.....	page 2-431
menu	Generate a menu of choices for user input .....	page 2-472
pause	Halt execution temporarily.....	page 2-535

## Object-Oriented Programming

class	Create object or return class of object .....	page 2-119
double	Convert to double precision .....	page 2-213
inferior	Inferior class relationship .....	page 2-392
inline	Construct an inline object.....	page 2-393
isa	Detect an object of a given class.....	page 2-423
superior	Superior class relationship .....	page 2-699
uint8	Convert to unsigned 8-bit integer .....	page 2-726

## Debugging

dbclear	Clear breakpoints.....	page 2-160
dbcont	Resume execution .....	page 2-162
dbdown	Change local workspace context .....	page 2-163
dbmex	Enable MEX-file debugging .....	page 2-166
dbquit	Quit debug mode.....	page 2-167
dbstack	Display function call stack .....	page 2-168

<code>dbstatus</code>	List all breakpoints .....	page 2-169
<code>dbstep</code>	Execute one or more lines from a breakpoint .....	page 2-170
<code>dbstop</code>	Set breakpoints in an M-file function .....	page 2-171
<code>dbtype</code>	List M-file with line numbers. ....	page 2-174
<code>dbup</code>	Change local workspace context .....	page 2-175

## Elementary Matrices and Matrix Manipulation

### Elementary Matrices and Arrays

<code>eye</code>	Identity matrix.....	page 2-251
<code>linspace</code>	Generate linearly spaced vectors .....	page 2-441
<code>logspace</code>	Generate logarithmically spaced vectors.....	page 2-450
<code>ones</code>	Create an array of all ones .....	page 2-526
<code>rand</code>	Uniformly distributed random numbers and arrays.....	page 2-584
<code>randn</code>	Normally distributed random numbers and arrays.....	page 2-586
<code>zeros</code>	Create an array of all zeros.....	page 2-757
<code>:</code> (colon)	Regularly spaced vector .....	page 2-16

### Special Variables and Constants

<code>ans</code>	The most recent answer .....	page 2-32
<code>computer</code>	Identify the computer on which MATLAB is running .....	page 2-128
<code>eps</code>	Floating-point relative accuracy .....	page 2-235
<code>flops</code>	Count floating-point operations .....	page 2-279
<code>i</code>	Imaginary unit.....	page 2-372
<code>Inf</code>	Infinity .....	page 2-391
<code>inputname</code>	Input argument name.....	page 2-399
<code>j</code>	Imaginary unit.....	page 2-430
<code>NaN</code>	Not-a-Number .....	page 2-486
<code>nargin, nargout</code>	Number of function arguments.....	page 2-488
<code>pi</code>	Ratio of a circle's circumference to its diameter, $\pi$ .....	page 2-544
<code>realmax</code>	Largest positive floating-point number .....	page 2-596
<code>realmin</code>	Smallest positive floating-point number.....	page 2-597
<code>varargin, varargout</code>	Pass or return variable numbers of arguments.....	page 2-731

### Time and Dates

<code>calendar</code>	Calendar.....	page 2-85
<code>clock</code>	Current time as a date vector.....	page 2-122

cputime	Elapsed CPU time .....	page 2-145
date	Current date string .....	page 2-155
datenum	Serial date number .....	page 2-156
datestr	Date string format .....	page 2-157
datevec	Date components .....	page 2-159
eomday	End of month .....	page 2-234
etime	Elapsed time .....	page 2-240
now	Current date and time .....	page 2-500
tic, toc	Stopwatch timer .....	page 2-716
weekday	Day of the week .....	page 2-741

## Matrix Manipulation

cat	Concatenate arrays .....	page 2-90
diag	Diagonal matrices and diagonals of a matrix .....	page 2-202
fliplr	Flip matrices left-right .....	page 2-276
fliptud	Flip matrices up-down .....	page 2-277
repmat	Replicate and tile an array .....	page 2-600
reshape	Reshape array .....	page 2-601
rot90	Rotate matrix 90 degrees .....	page 2-609
tril	Lower triangular part of a matrix .....	page 2-721
triu	Upper triangular part of a matrix .....	page 2-722
:	(colon) Index into array, rearrange array .....	page 2-16

## Specialized Matrices

compan	Companion matrix .....	page 2-127
gallery	Test matrices .....	page 2-319
hadamard	Hadamard matrix .....	page 2-362
hankel	Hankel matrix .....	page 2-363
hilb	Hilbert matrix .....	page 2-371
invhilb	Inverse of the Hilbert matrix .....	page 2-417
magic	Magic square .....	page 2-464
pascal	Pascal matrix .....	page 2-532
toeplitz	Toeplitz matrix .....	page 2-717
wilkinson	Wilkinson's eigenvalue test matrix .....	page 2-750

## Elementary Math Functions

abs	Absolute value and complex magnitude .....	page 2-19
acos, acosh	Inverse cosine and inverse hyperbolic cosine .....	page 2-21

acot, acoth	Inverse cotangent and inverse hyperbolic cotangent.....	page 2-22
acsc,acsch	Inverse cosecant and inverse hyperbolic cosecant.....	page 2-23
angl e	angl e Phase angle.....	page 2-31
asec, asech	Inverse secant and inverse hyperbolic secant.....	page 2-38
asi n, asi nh	Inverse sine and inverse hyperbolic sine.....	page 2-39
atan, atanh	Inverse tangent and inverse hyperbolic tangent.....	page 2-41
atan2	Four-quadrant inverse tangent .....	page 2-43
ceil	Round toward infinity .....	page 2-95
conj	Complex conjugate .....	page 2-133
cos, cosh	Cosine and hyperbolic cosine .....	page 2-141
cot, coth	Cotangent and hyperbolic cotangent .....	page 2-142
csc, csch	Cosecant and hyperbolic cosecant.....	page 2-147
exp	Exponential .....	page 2-246
fix	Round towards zero .....	page 2-274
floor	Round towards minus infinity.....	page 2-278
gcd	Greatest common divisor.....	page 2-341
imag	Imaginary part of a complex number .....	page 2-379
lcm	Least common multiple .....	page 2-436
log	Natural logarithm .....	page 2-444
log2	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa .....	page 2-445
log10	Common (base 10) logarithm .....	page 2-446
mod	Modulus (signed remainder after division) .....	page 2-482
nchoosek	Binomial coefficient or all combinations .....	page 2-490
real	Real part of complex number .....	page 2-595
rem	Remainder after division .....	page 2-599
round	Round to nearest integer .....	page 2-610
sec, sech	Secant and hyperbolic secant .....	page 2-622
sgn	Signum function .....	page 2-629
sin, si nh	Sine and hyperbolic sine.....	page 2-630
sqr t	Square root .....	page 2-666
tan, tanh	Tangent and hyperbolic tangent .....	page 2-712

## Specialized Math Functions

airy	Airy functions .....	page 2-26
bessel h	Bessel functions of the third kind (Hankel functions).....	page 2-51
bessel i, bessel k	Modified Bessel functions .....	page 2-53
bessel j, bessel y	Bessel functions.....	page 2-56

beta, betai nc, betal n	Beta functions . . . . .	page 2-59
ellipj	Jacobi elliptic functions. . . . .	page 2-226
ellipke	Complete elliptic integrals of the first and second kind. . . . .	page 2-228
erf, erfc, erfcx, erfinv	Error functions. . . . .	page 2-236
expint	Exponential integral . . . . .	page 2-247
gamma, gammairc, gammaln	Gamma functions . . . . .	page 2-339
legendre	Associated Legendre functions. . . . .	page 2-437
pow2	Base 2 power and scale floating-point numbers. . . . .	page 2-562
rat, rats	Rational fraction approximation . . . . .	page 2-590

## Coordinate System Conversion

cart2pol	Transform Cartesian coordinates to polar or cylindrical. . . . .	page 2-86
cart2sph	Transform Cartesian coordinates to spherical . . . . .	page 2-88
pol2cart	Transform polar or cylindrical coordinates to Cartesian. . . . .	page 2-548
sph2cart	Transform spherical coordinates to Cartesian . . . . .	page 2-650

## Matrix Functions - Numerical Linear Algebra

### Matrix Analysis

cond	Condition number with respect to inversion . . . . .	page 2-130
condeig	Condition number with respect to eigenvalues . . . . .	page 2-131
det	Matrix determinant. . . . .	page 2-201
norm	Vector and matrix norms . . . . .	page 2-498
null	Null space of a matrix . . . . .	page 2-501
orth	Range space of a matrix. . . . .	page 2-527
rank	Rank of a matrix . . . . .	page 2-589
rcond	Matrix reciprocal condition number estimate . . . . .	page 2-593
rref, rrefmove	Reduced row echelon form. . . . .	page 2-611
subspace	Angle between two subspaces. . . . .	page 2-697
trace	Sum of diagonal elements. . . . .	page 2-718

### Linear Equations

\ /	Linear equation solution. . . . .	page 2-2
chol	Cholesky factorization. . . . .	page 2-107

i nv	Matrix inverse .....	page 2-414
l scov	Least squares solution in the presence of known covariance	page 2-453
l u	LU matrix factorization .....	page 2-454
n nl s	Nonnegative least squares .....	page 2-494
pi nv	Moore-Penrose pseudoinverse of a matrix .....	page 2-545
qr	Orthogonal-triangular decomposition .....	page 2-571

## Eigenvalues and Singular Values

bal ance	Improve accuracy of computed eigenvalues .....	page 2-47
cdf2rdf	Convert complex diagonal form to real block diagonal form	page 2-93
ei g	Eigenvalues and eigenvectors .....	page 2-217
gsvd	Generalized singular value decomposition .....	page 2-357
hess	Hessenberg form of a matrix .....	page 2-367
pol y	Polynomial with specified roots .....	page 2-550
qz	QZ factorization for generalized eigenvalues .....	page 2-583
rsf2csf	Convert real Schur form to complex Schur form .....	page 2-613
schur	Schur decomposition .....	page 2-619
svd	Singular value decomposition .....	page 2-700

## Matrix Functions

expm	Matrix exponential .....	page 2-249
funm	Evaluate functions of a matrix .....	page 2-311
logm	Matrix logarithm .....	page 2-448
sqrtm	Matrix square root .....	page 2-667

## Low Level Functions

qrdel ete	Delete column from QR factorization .....	page 2-574
qrinsert	Insert column in QR factorization .....	page 2-575

# Data Analysis and Fourier Transform Functions

## Basic Operations

convhull	Convex hull .....	page 2-137
cumprod	Cumulative product .....	page 2-148
cumsum	Cumulative sum .....	page 2-149
cumtrapz	Cumulative trapezoidal numerical integration .....	page 2-150
delaunay	Delaunay triangulation .....	page 2-197
dsearch	Search for nearest point .....	page 2-214

<b>factor</b>	Prime factors . . . . .	page 2-253
<b>i n pol ygon</b>	Detect points inside a polygonal region . . . . .	page 2-397
<b>max</b>	Maximum elements of an array . . . . .	page 2-469
<b>mean</b>	Average or mean value of arrays . . . . .	page 2-470
<b>medi an</b>	Median value of arrays . . . . .	page 2-471
<b>mi n</b>	Minimum elements of an array . . . . .	page 2-478
<b>perms</b>	All possible permutations . . . . .	page 2-541
<b>pol yarea</b>	Area of polygon . . . . .	page 2-553
<b>pri mes</b>	Generate list of prime numbers . . . . .	page 2-563
<b>prod</b>	Product of array elements . . . . .	page 2-564
<b>sort</b>	Sort elements in ascending order . . . . .	page 2-634
<b>sortrows</b>	Sort rows in ascending order . . . . .	page 2-635
<b>std</b>	Standard deviation . . . . .	page 2-675
<b>sum</b>	Sum of array elements . . . . .	page 2-698
<b>trapz</b>	Trapezoidal numerical integration . . . . .	page 2-719
<b>tsearch</b>	Search for enclosing Delaunay triangle . . . . .	page 2-724
<b>voronoi</b>	Voronoi diagram . . . . .	page 2-735

## Finite Differences

<b>del 2</b>	Discrete Laplacian . . . . .	page 2-194
<b>di ff</b>	Differences and approximate derivatives . . . . .	page 2-204
<b>gradi ent</b>	Numerical gradient . . . . .	page 2-351

## Correlation

<b>corrcoef</b>	Correlation coefficients . . . . .	page 2-140
<b>cov</b>	Covariance matrix . . . . .	page 2-143

## Filtering and Convolution

<b>conv</b>	Convolution and polynomial multiplication . . . . .	page 2-134
<b>conv2</b>	Two-dimensional convolution . . . . .	page 2-135
<b>deconv</b>	Deconvolution and polynomial division . . . . .	page 2-193
<b>fi lter</b>	Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter . . . . .	page 2-268
<b>fi lter2</b>	Two-dimensional digital filtering . . . . .	page 2-270

## Fourier Transforms

<b>abs</b>	Absolute value and complex magnitude . . . . .	page 2-19
<b>angl e</b>	Phase angle . . . . .	page 2-31
<b>cpl xpai r</b>	Sort complex numbers into complex conjugate pairs . . . . .	page 2-144

fft	One-dimensional fast Fourier transform .....	page 2-258
fft2	Two-dimensional fast Fourier transform .....	page 2-261
fftshift	Shift DC component of fast Fourier transform to center of spectrum .....	page 2-263
ifft	Inverse one-dimensional fast Fourier transform .....	page 2-375
ifft2	Inverse two-dimensional fast Fourier transform .....	page 2-376
ifftshift	Inverse FFT shift .....	page 2-378
nextpow2	Next power of two .....	page 2-493
unwrap	Correct phase angles .....	page 2-729

## Vector Functions

cross	Vector cross product .....	page 2-146
intersect	Set intersection of two vectors .....	page 2-413
ismember	Detect members of a set .....	page 2-424
setdiff	Return the set difference of two vectors .....	page 2-624
setxor	Set exclusive-or of two vectors .....	page 2-627
union	Set union of two vectors .....	page 2-727
unique	Unique elements of a vector .....	page 2-728

# Polynomial and Interpolation Functions

## Polynomials

conv	Convolution and polynomial multiplication .....	page 2-134
deconv	Deconvolution and polynomial division .....	page 2-193
poly	Polynomial with specified roots .....	page 2-550
polyder	Polynomial derivative .....	page 2-554
polyeig	Polynomial eigenvalue problem .....	page 2-555
polyfit	Polynomial curve fitting .....	page 2-556
polyval	Polynomial evaluation .....	page 2-559
polyvalm	Matrix polynomial evaluation .....	page 2-560
residue	Convert between partial fraction expansion and polynomial coefficients .....	page 2-602
roots	Polynomial roots .....	page 2-607

## Data Interpolation

griddata	Data gridding .....	page 2-354
interp1	One-dimensional data interpolation (table lookup) .....	page 2-401
interp2	Two-dimensional data interpolation (table lookup) .....	page 2-404
interp3	Three-dimensional data interpolation (table lookup) .....	page 2-408

<code>interpft</code>	One-dimensional interpolation using the FFT method . . . . .	page 2-410
<code>interpn</code>	Multidimensional data interpolation (table lookup) . . . . .	page 2-411
<code>meshgrid</code>	Generate X and Y matrices for three-dimensional plots . . . . .	page 2-473
<code>ndgrid</code>	Generate arrays for multidimensional functions and interpolation . . . . .	page 2-491
<code>spline</code>	Cubic spline interpolation . . . . .	page 2-651

## Function Functions – Nonlinear Numerical Methods

<code>dblquad</code>	Numerical double integration . . . . .	page 2-164
<code>fmin</code>	Minimize a function of one variable . . . . .	page 2-280
<code>fmins</code>	Minimize a function of several variables . . . . .	page 2-282
<code>fzero</code>	Zero of a function of one variable . . . . .	page 2-316
<code>ode45</code> , <code>ode23</code> , <code>ode113</code> , <code>ode15s</code> , <code>ode23s</code> , <code>ode23t</code> , <code>ode23tb</code>	Solve differential equations . . . . .	page 2-505
<code>odeset</code>	Define a differential equation problem for ODE solvers . . . . .	page 2-513
<code>odeget</code>	Extract properties from options structure created with <code>odeset</code> . . . . .	page 2-519
<code>quad</code> , <code>quad8</code>	Create or alter options structure for input to ODE solvers . . . . .	page 2-520
<code>vectorize</code>	Numerical evaluation of integrals . . . . .	page 2-580
	Vectorize expression . . . . .	page 2-733

## Sparse Matrix Functions

### Elementary Sparse Matrices

<code>spdiags</code>	Extract and create sparse band and diagonal matrices . . . . .	page 2-644
<code>speye</code>	Sparse identity matrix . . . . .	page 2-648
<code>sprand</code>	Sparse uniformly distributed random matrix . . . . .	page 2-658
<code>sprandn</code>	Sparse normally distributed random matrix . . . . .	page 2-659
<code>sprandsym</code>	Sparse symmetric random matrix . . . . .	page 2-660

### Full to Sparse Conversion

<code>find</code>	Find indices and values of nonzero elements . . . . .	page 2-271
<code>full</code>	Convert sparse matrix to full matrix . . . . .	page 2-307
<code>sparse</code>	Create sparse matrix . . . . .	page 2-640
<code>spconvert</code>	Import matrix from sparse matrix external format . . . . .	page 2-642

## Working with Nonzero Entries of Sparse Matrices

nnz	Number of nonzero matrix elements .....	page 2-496
nonzeros	Nonzero matrix elements.....	page 2-497
nzmax	Amount of storage allocated for nonzero matrix elements..	page 2-504
spalloc	Allocate space for sparse matrix.....	page 2-639
spfun	Apply function to nonzero sparse matrix elements.....	page 2-649
spones	Replace nonzero sparse matrix elements with ones.....	page 2-654

## Visualizing Sparse Matrices

spy	Visualize sparsity pattern .....	page 2-665
-----	----------------------------------	------------

## Reordering Algorithms

colmmd	Sparse column minimum degree permutation .....	page 2-123
colperm	Sparse column permutation based on nonzero count.....	page 2-126
dmp perm	Dulmage-Mendelsohn decomposition .....	page 2-211
randperm	Random permutation .....	page 2-588
symmmd	Sparse symmetric minimum degree ordering .....	page 2-706
symrcm	Sparse reverse Cuthill-McKee ordering .....	page 2-708

## Norm, Condition Number, and Rank

condest	1-norm matrix condition number estimate .....	page 2-132
normest	2-norm estimate .....	page 2-499

## Sparse Systems of Linear Equations

bicg	BiConjugate Gradients method .....	page 2-61
bicgstab	BiConjugate Gradients Stabilized method .....	page 2-68
cgs	Conjugate Gradients Squared method .....	page 2-101
cholinc	Sparse Incomplete Cholesky and Cholesky-Infinity factorizations .....	page 2-109
cholupdate	Rank 1 update to Cholesky factorization .....	page 2-116
gmres	Generalized Minimum Residual method (with restarts)....	page 2-347
lui nc	Incomplete LU matrix factorizations .....	page 2-457
pcg	Preconditioned Conjugate Gradients method .....	page 2-536
qmr	Quasi-Minimal Residual method .....	page 2-567
qr	Orthogonal-triangular decomposition .....	page 2-571
qrdelete	Delete column from QR factorization.....	page 2-574
qrinsert	Insert column in QR factorization.....	page 2-575
qrupdate	Rank 1 update to QR factorization .....	page 2-576

---

## Sparse Eigenvalues and Singular Values

eig	Find a few eigenvalues and eigenvectors .....	page 2-220
svds	A few singular values.....	page 2-702

## Miscellaneous

spparms	Set parameters for sparse matrix routines .....	page 2-655
---------	---	------------

# Sound Processing Functions

## General Sound Functions

sound	Convert vector into sound .....	page 2-636
-------	---------------------------------	------------

## SPARCstation-specific Sound Functions

auread	Read NeXT/SUN (.au) sound file .....	page 2-44
auwrite	Write NeXT/SUN (.au) sound file.....	page 2-45

## .WAV Sound Functions

wavread	Read Microsoft WAVE (.wav) sound file .....	page 2-738
wavwrite	Write Microsoft WAVE (.wav) sound file.....	page 2-739

## Macintosh Sound Functions

readsnd	Read snd resources and files.....	page 2-594
recordsound	Record sound.....	page 2-598
soundcap	Sound capabilities .....	page 2-637
speak	Speak text string.....	page 2-647
writesnd	Write snd resources and files .....	page 2-753

# Character String Functions

## General

abs	Absolute value and complex magnitude .....	page 2-19
eval	Interpret strings containing MATLAB expressions .....	page 2-241
real	Real part of complex number.....	page 2-595
strings	MATLAB string handling .....	page 2-683

## String Manipulation

deblank	Strip trailing blanks from the end of a string .....	page 2-189
findstr	Find one string within another .....	page 2-273
lower	Convert string to lower case.....	page 2-452
strcat	String concatenation.....	page 2-678
strcmp	Compare strings .....	page 2-680
strcmpi	Compare strings ignoring case.....	page 2-682
strjustify	Justify a character array .....	page 2-684
strmatch	Find possible matches for a string.....	page 2-685
strncmp	Compare the first n characters of two strings.....	page 2-686
strrep	String search and replace .....	page 2-688
strtok	First token in string.....	page 2-689
strvcat	Vertical concatenation of strings.....	page 2-692
upper	Convert string to upper case .....	page 2-730

## String to Number Conversion

char	Create character array (string).....	page 2-105
int2str	Integer to string conversion .....	page 2-400
mat2str	Convert a matrix into a string .....	page 2-466
num2str	Number to string conversion.....	page 2-503
sprintf	Write formatted data to a string .....	page 2-661
sscanf	Read string under format control .....	page 2-671
str2num	String to number conversion .....	page 2-677

## Radix Conversion

b2n2dec	Binary to decimal number conversion .....	page 2-72
dec2bin	Decimal to binary number conversion .....	page 2-191
dec2hex	Decimal to hexadecimal number conversion.....	page 2-192
hex2dec	IEEE hexadecimal to decimal number conversion .....	page 2-369
hex2num	Hexadecimal to double number conversion .....	page 2-370

## Low-Level File I/O Functions

### File Opening and Closing

fclose	Close one or more open files .....	page 2-254
fopen	Open a file or obtain information about open files.....	page 2-286

## Unformatted I/O

fread	Read binary data from file .....	page 2-297
fwrite	Write binary data to a file .....	page 2-313

## Formatted I/O

fgetl	Return the next line of a file as a string without line terminator(s) .....	page 2-264
fgets	Return the next line of a file as a string with line terminator(s) .....	page 2-265
fprintf	Write formatted data to file .....	page 2-292
fscanf	Read formatted data from file .....	page 2-302

## File Positioning

feof	Test for end-of-file .....	page 2-255
ferror	Query MATLAB about errors in file input or output .....	page 2-256
frewind	Rewind an open file .....	page 2-301
fseek	Set file position indicator .....	page 2-305
ftell	Get file position indicator .....	page 2-306

## String Conversion

sprintf	Write formatted data to a string .....	page 2-661
sscanf	Read string under format control .....	page 2-671

## Specialized File I/O

qfwrite	Write QuickTime movie file to disk .....	page 2-579
dlmread	Read an ASCII delimited file into a matrix .....	page 2-208
dlmwrite	Write a matrix to an ASCII delimited file .....	page 2-210
hdf	HDF interface .....	page 2-364
imfinfo	Return information about a graphics file .....	page 2-380
imread	Read image from graphics file .....	page 2-383
imwrite	Write an image to a graphics file .....	page 2-386
wk1read	Read a Lotus123 WK1 spreadsheet file into a matrix .....	page 2-751
wk1write	Write a matrix to a Lotus123 WK1 spreadsheet file .....	page 2-752
xlgetrange	Get range of cells from Microsoft Excel worksheet .....	page 2-754
xlssetrange	Set range of cells in Microsoft Excel worksheet .....	page 2-755

## Bitwise Functions

<code>bitand</code>	Bit-wise AND.....	page 2-73
<code>bitecmp</code>	Complement bits.....	page 2-74
<code>bitor</code>	Bit-wise OR .....	page 2-77
<code>bitemax</code>	Maximum floating-point integer.....	page 2-76
<code>bittset</code>	Set bit .....	page 2-78
<code>bithshift</code>	Bit-wise shift .....	page 2-79
<code>bittget</code>	Get bit.....	page 2-75
<code>bittxor</code>	Bit-wise XOR.....	page 2-80

## Structure Functions

<code>fieldnames</code>	Field names of a structure .....	page 2-266
<code>getfield</code>	Get field of structure array.....	page 2-344
<code>rmfield</code>	Remove structure fields .....	page 2-605
<code>setfield</code>	Set field of structure array .....	page 2-625
<code>struct</code>	Create structure array.....	page 2-690
<code>struct2cell</code>	Structure to cell array conversion .....	page 2-691

## Object Functions

<code>class</code>	Create object or return class of object.....	page 2-119
<code>isa</code>	Detect an object of a given class.....	page 2-423

## Cell Array Functions

<code>cell</code>	Create cell array.....	page 2-96
<code>cellstr</code>	Create cell array of strings from character array .....	page 2-100
<code>cell2struct</code>	Cell array to structure array conversion .....	page 2-97
<code>celldisp</code>	Display cell array contents.....	page 2-98
<code>cellplot</code>	Graphically display the structure of cell arrays.....	page 2-99
<code>num2cell</code>	Convert a numeric array into a cell array .....	page 2-502

## Multidimensional Array Functions

<code>cat</code>	Concatenate arrays .....	page 2-90
<code>flipdim</code>	Flip array along a specified dimension .....	page 2-275
<code>ind2sub</code>	Subscripts from linear index .....	page 2-390

---

i permute	Inverse permute the dimensions of a multidimensional array	page 2-418
ndgrid	Generate arrays for multidimensional functions and interpolation	page 2-491
ndims	Number of array dimensions	page 2-492
permute	Rearrange the dimensions of a multidimensional array	page 2-542
reshape	Reshape array	page 2-601
shiftdim	Shift dimensions	page 2-628
squeeze	Remove singleton dimensions	page 2-670
sub2ind	Single index from subscripts	page 2-693

# **1** Command Summary

---

# Reference

---

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

# Arithmetic Operators + - \* / \ ^ '

<b>Purpose</b>	Matrix and array arithmetic
<b>Syntax</b>	A+B A-B A*B      A.*B A/B      A./B A\B      A.\B A^B      A.^B A'        A.'
<b>Description</b>	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 \.

# Arithmetic Operators + - \* / \ ^ '

- . / 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,  $A \backslash B$  is roughly the same as  $\text{inv}(A) * B$ , 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  $X = A \backslash B$  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  $m \approx n$  and B is a column vector with m components, or a matrix with several such columns, then  $X = A \backslash B$  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  $\text{pinv}(A) * B$ , which is the least squares solution with the smallest norm,  $| |X| |$ .
- . \ Array left division.  $A . \backslash B$  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  $[V, D] = \text{eig}(X)$ , then  $X^p = V * D.^p * V^{-1}$ .

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  $A(i, j)^B$  to the power B(i, j). 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	plus(A)
Binary subtraction	A-B	minus(A, B)
Unary minus	-A	minus(A)
Matrix multiplication	A*B	times(A, B)
Array-wise multiplication	A.*B	times(A, B)
Matrix right division	A/B	rdivide(A, B)
Array-wise right division	A./B	rdivide(A, B)
Matrix left division	A\B	ldivide(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)

# Arithmetic Operators + - \* / \ ^ '

## 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  $X = A \setminus B$  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,

$$X = R \setminus (R' \setminus B)$$

- 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 `l u`). If  $A$  is sparse, a non-symmetric minimum degree preordering is applied (see `col mmd` 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.

$$X = U \setminus (L \setminus B)$$

- 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

$$X = P * (R \setminus (Q' * B))$$

- 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  $c = \max(\max(\text{abs}(A))) / 1000$  (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 ZGEC0, ZGEFA and ZGESL for

# Arithmetic Operators + - \* / \ ^ '

---

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.

# Relational Operators < > <= >= == ~=

<b>Purpose</b>	Relational operations									
<b>Syntax</b>	A < B A > B A <= B A >= B A == B A ~= B									
<b>Description</b>	The relational operators are $<$ , $\leq$ , $>$ , $\geq$ , $==$ , 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 $<$ , $\leq$ , $>$ , and $\geq$ 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 <code>strcmp</code> , which allows vectors of dissimilar length to be compared.									
<b>Examples</b>	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:  <code>X = 5; X &gt;= [1 2 3; 4 5 6; 7 8 10]</code> <code>X = 5*ones(3, 3); X &gt;= [1 2 3; 4 5 6; 7 8 10]</code>  produce the same result:  <code>ans =</code>  <table><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr></table>	1	1	1	1	1	0	0	0	0
1	1	1								
1	1	0								
0	0	0								

# Relational Operators < > <= >= == ~=

---

## 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**

$A \& B$   
 $A \mid B$   
 $\sim A$

**Description**

The symbols  $\&$ ,  $\mid$ , and  $\sim$  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  $\mid$  operator does a logical OR, and  $\sim A$  complements the elements of A. The function  $xor(A, B)$  implements the exclusive OR operation. Truth tables for these operators and functions follow.

Inputs		and	or	xor	NOT
A	B	$A \& B$	$A \mid B$	$xor(A, B)$	$\sim A$
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	$A \mid B$	$or(A, B)$
not	$\sim A$	$not(A)$

# Logical Operators & | ~

---

## 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

# Special Characters [ ] ( ) { } = ' . . . , ; % !

<b>Purpose</b>	Special characters
<b>Syntax</b>	[ ] ( ) { } = ' . . . , ; % !
<b>Description</b>	<p>[ ] 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.</p> <p>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.</p> <p>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.</p> <p>[A1, A2, A3. . . ] = function assigns function output to multiple variables.</p> <p>For the use of [ and ] on the left of an “=” in multiple assignment statements, see l u, ei g, svd, and so on.</p> <p>{ } 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 hel p paren for more information about { }.</p>

## Special Characters [ ] ( ) {} = ' . . . , ; % !

---

( ) 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  $[X(V(1)), X(V(2)), \dots, X(V(n))]$ . 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.

## Special Characters [ ] ( ) { } = ' . . . , ; % !

- , 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	<code>[A, B, C. . . ]</code>	<code>horzcat(A, B, C. . . )</code>
Vertical concatenation	<code>[A; B; C. . . ]</code>	<code>vertcat(A, B, C. . . )</code>
Subscript reference	<code>A(i, j, k. . . )</code>	<code>subsref(A, S)</code> . See help <code>subsref</code> .
Subscript assignment	<code>A(i, j, k. . . ) = B</code>	<code>subsasgn(A, S, B)</code> . See help <code>subsasgn</code> .

### See Also

## Arithmetic, relational, and logical operators.

## Colon :

---

<b>Purpose</b>	Create vectors, array subscripting, and <i>for</i> iterations
<b>Description</b>	The colon is one of the most useful operators in MATLAB. It can create vectors, subscript arrays, and specify <i>for</i> iterations.
The colon operator uses the following rules to create regularly spaced vectors:	
$j : k$	is the same as $[j, j+1, \dots, k]$
$j : k$	is empty if $j > k$
$j : i : k$	is the same as $[j, j+i, j+2i, \dots, k]$
$j : i : k$	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 $A(j), A(j+1), \dots, A(k)$
$A(:, :, j : k)$	is $A(:, :, j), A(:, :, j+1), \dots, A(:, :, k)$
$A(:, :, :, k)$	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`  
`linspace`  
`logspace`  
`reshape`

Repeat statements a specific number of times  
Generate linearly spaced vectors  
Generate logarithmically spaced vectors  
Reshape array

## **Colon :**

---

---

<b>Purpose</b>	Absolute value and complex magnitude		
<b>Syntax</b>	$Y = \text{abs}(X)$		
<b>Description</b>	$\text{abs}(X)$ returns the absolute value, $ X $ , for each element of $X$ . If $X$ is complex, $\text{abs}(X)$ returns the complex modulus (magnitude): $\text{abs}(X) = \sqrt{\text{real}(X)^2 + \text{imag}(X)^2}$		
<b>Examples</b>	$\text{abs}(-5) = 5$ $\text{abs}(3+4i) = 5$		
<b>See Also</b>	<code>angle</code>	<code>Phase angle</code>	
	<code>sign</code>	<code>Signum function</code>	
	<code>unwrap</code>	<code>Correct phase angles</code>	

# acopy

---

<b>Purpose</b>	Copy Macintosh file from one folder to another								
<b>Syntax</b>	<code>acopy(<i>filename</i>, <i>foldername</i>)</code>								
<b>Description</b>	<code>acopy(filename, foldername)</code> copies the file <i>filename</i> to the folder <i>foldername</i> . Both <i>filename</i> and <i>foldername</i> can be full or partial path names.								
<b>See Also</b>	<table><tr><td><code>amove</code></td><td>Move Macintosh file from one folder to another</td></tr><tr><td><code>appl escript</code></td><td>Load a compiled AppleScript from a file and execute it</td></tr><tr><td><code>arename</code></td><td>Rename Macintosh File</td></tr><tr><td><code>areveal</code></td><td>Reveal filename on Macintosh desktop</td></tr></table>	<code>amove</code>	Move Macintosh file from one folder to another	<code>appl escript</code>	Load a compiled AppleScript from a file and execute it	<code>arename</code>	Rename Macintosh File	<code>areveal</code>	Reveal filename on Macintosh desktop
<code>amove</code>	Move Macintosh file from one folder to another								
<code>appl escript</code>	Load a compiled AppleScript from a file and execute it								
<code>arename</code>	Rename Macintosh File								
<code>areveal</code>	Reveal filename on Macintosh desktop								

**Purpose** Inverse cosine and inverse hyperbolic cosine

**Syntax**

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

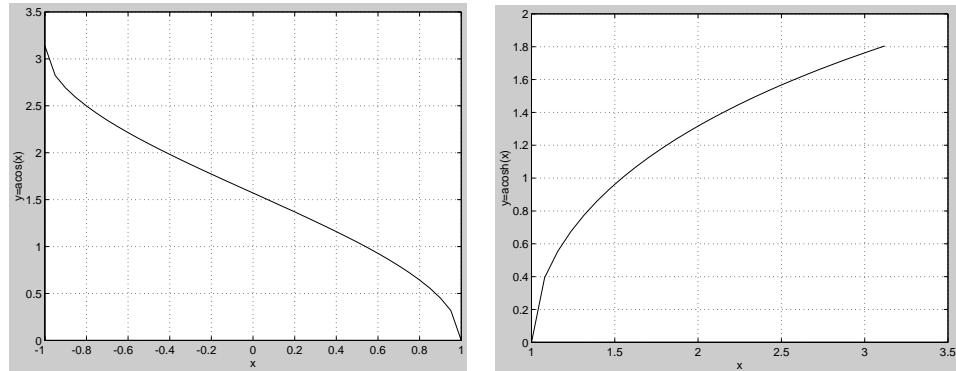
**Description** The `acos` and `acosh` functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

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

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



**Algorithm**

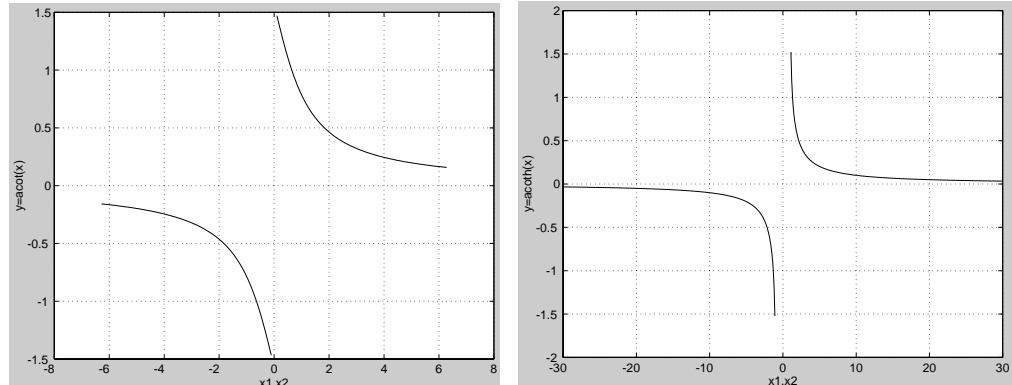
$$\cos^{-1}(z) = -i \log\left[z + i(1 - z^2)^{\frac{1}{2}}\right]$$

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

**See Also** `cos, cosh` Cosine and hyperbolic cosine

# acot, acoth

<b>Purpose</b>	Inverse cotangent and inverse hyperbolic cotangent
<b>Syntax</b>	$Y = \text{acot}(X)$ $Y = \text{acoth}(X)$
<b>Description</b>	The <code>acot</code> and <code>acoth</code> functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.
	$Y = \text{acot}(X)$ returns the inverse cotangent (arccotangent) for each element of $X$ .
	$Y = \text{acoth}(X)$ returns the inverse hyperbolic cotangent for each element of $X$ .
<b>Examples</b>	Graph the inverse cotangent over the domains $-2\pi \leq x < 0$ and $0 < x \leq 2\pi$ , and the inverse hyperbolic cotangent over the domains $-30 \leq x < -1$ and $1 < x \leq 30$ .  <pre>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))</pre>
<b>Algorithm</b>	$\cot^{-1}(z) = \tan^{-1}\left(\frac{1}{z}\right)$ $\coth^{-1}(z) = \tanh^{-1}\left(\frac{1}{z}\right)$
<b>See Also</b>	<a href="#">cot, coth</a> <a href="#">Cotangent and hyperbolic cotangent</a>



**Purpose**

Inverse cosecant and inverse hyperbolic cosecant

**Syntax**

$Y = \text{acsc}(X)$

$Y = \text{acsch}(X)$

**Description**

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

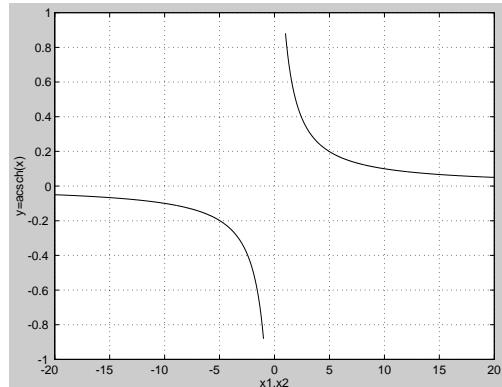
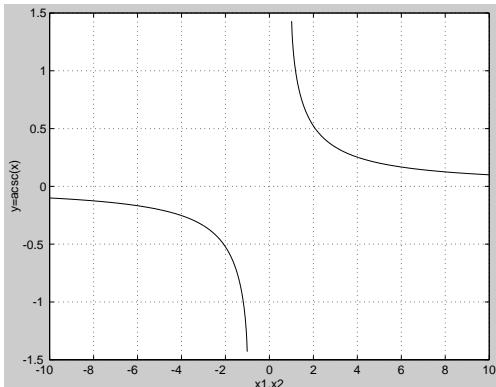
$Y = \text{acsc}(X)$  returns the inverse cosecant (arccosecant) for each element of  $X$ .

$Y = \text{acsch}(X)$  returns the inverse hyperbolic cosecant for each element of  $X$ .

**Examples**

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

```
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**

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

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

## **acsc,acsch**

---

**See Also**

csc, csch

Cosecant and hyperbolic cosecant

<b>Purpose</b>	Add directories to MATLAB's search path				
<b>Syntax</b>	<pre>addpath('directory') addpath('dir1', 'dir2', 'dir3', ...) addpath(..., '-flag')</pre>				
<b>Description</b>	<p><code>addpath ('directory')</code> prepends the specified directory to MATLAB's current search path.</p> <p><code>addpath ('dir1', 'dir2', 'dir3', ...)</code> prepends all the specified directories to the path.</p> <p><code>addpath (... , '-flag')</code> either prepends or appends the specified directories to the path depending the value of <i>flag</i>:</p> <table> <tr> <td>0 or begin</td> <td>Prepend specified directories</td> </tr> <tr> <td>1 or end</td> <td>Append specified directories</td> </tr> </table>	0 or begin	Prepend specified directories	1 or end	Append specified directories
0 or begin	Prepend specified directories				
1 or end	Append specified directories				
<b>Examples</b>	<pre>path     MATLABPATH     c:\matlab\tool box\general     c:\matlab\tool box\ops     c:\matlab\tool box\strfun  addpath('c:\matlab\myfiles')  path     MATLABPATH     c:\matlab\myfiles     c:\matlab\tool box\general     c:\matlab\tool box\ops     c:\matlab\tool box\strfun</pre>				
<b>See Also</b>	<table> <tr> <td><code>path</code></td> <td>Control MATLAB's directory search path</td> </tr> <tr> <td><code>rmpath</code></td> <td>Remove directories from MATLAB's search path</td> </tr> </table>	<code>path</code>	Control MATLAB's directory search path	<code>rmpath</code>	Remove directories from MATLAB's search path
<code>path</code>	Control MATLAB's directory search path				
<code>rmpath</code>	Remove directories from MATLAB's search path				

# airy

**Purpose** Airy functions

**Syntax**  $W = \text{airy}(Z)$

$W = \text{airy}(k, Z)$

$[W, \text{err}] = \text{airy}(k, Z)$

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

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

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**  $W = \text{airy}(Z)$  returns the Airy function,  $Ai(Z)$ , for each element of the complex array  $Z$ .

$W = \text{airy}(k, Z)$  returns different results depending on the value of  $k$ :

<b>k</b>	<b>Returns</b>
0	The same result as $\text{airy}(Z)$ .
1	The derivative, $Ai'(Z)$ .
2	The Airy function of the second kind, $Bi(Z)$ .
3	The derivative, $Bi'(Z)$ .

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

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

**See Also**

bessel i                    Modified Bessel functions of the first kind  
bessel j                    Bessel functions of the first kind  
bessel k                    Modified Bessel functions of the third kind  
bessel y                    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

<b>Purpose</b>	Test to determine if all elements are nonzero													
<b>Syntax</b>	$B = \text{all}(A)$ $B = \text{all}(A, dim)$													
<b>Description</b>	<p><math>B = \text{all}(A)</math> tests whether <i>all</i> the elements along various dimensions of an array are nonzero or logical true (1).</p> <p>If A is a vector, <math>\text{all}(A)</math> returns logical true (1) if all of the elements are nonzero, and returns logical false (0) if one or more elements are zero.</p> <p>If A is a matrix, <math>\text{all}(A)</math> treats the columns of A as vectors, returning a row vector of 1s and 0s.</p> <p>If A is a multidimensional array, <math>\text{all}(A)</math> treats the values along the first non-singleton dimension as vectors, returning a logical condition for each vector.</p>													
	$B = \text{all}(A, dim)$ tests along the dimension of A specified by scalar <i>dim</i> .													
	<table border="1"><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	1	1	1	1	1	0	<table border="1"><tr><td>1</td><td>1</td><td>0</td></tr></table>	1	1	0	<table border="1"><tr><td>1</td></tr><tr><td>0</td></tr></table>	1	0
1	1	1												
1	1	0												
1	1	0												
1														
0														
	A	$\text{all}(A,1)$	$\text{all}(A,2)$											
<b>Examples</b>	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 $\text{all}$ function reduces such a vector of logical conditions to a single condition. In this case, $\text{all}(B)$ yields 0. This makes $\text{all}$ particularly useful in if statements,													
	<pre>if all(A &lt; 0.5)     do something end</pre>													

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`

# amove

---

<b>Purpose</b>	Move Macintosh file from one folder to another								
<b>Syntax</b>	<code>amove(<i>filename</i>, <i>foldername</i>)</code>								
<b>Description</b>	<code>amove(filename, foldername)</code> moves the file <i>filename</i> to the folder <i>foldername</i> . Both <i>filename</i> and <i>foldername</i> can be full or partial path names.								
<b>See Also</b>	<table><tr><td><code>acopy</code></td><td>Copy Macintosh file from one folder to another</td></tr><tr><td><code>appl escri pt</code></td><td>Load a compiled AppleScript from a file and execute it</td></tr><tr><td><code>arename</code></td><td>Rename Macintosh File</td></tr><tr><td><code>areveal</code></td><td>Reveal filename on Macintosh desktop</td></tr></table>	<code>acopy</code>	Copy Macintosh file from one folder to another	<code>appl escri pt</code>	Load a compiled AppleScript from a file and execute it	<code>arename</code>	Rename Macintosh File	<code>areveal</code>	Reveal filename on Macintosh desktop
<code>acopy</code>	Copy Macintosh file from one folder to another								
<code>appl escri pt</code>	Load a compiled AppleScript from a file and execute it								
<code>arename</code>	Rename Macintosh File								
<code>areveal</code>	Reveal filename on Macintosh desktop								

<b>Purpose</b>	Phase angle																																
<b>Syntax</b>	$P = \text{angle}(Z)$																																
<b>Description</b>	$P = \text{angle}(Z)$ 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																																
	$R = \text{abs}(Z) \quad \% \text{ magnitude}$ $\theta = \text{angle}(Z) \quad \% \text{ phase angle}$																																
	and the statement																																
	$Z = R * \exp(i * \theta)$																																
	converts back to the original complex $Z$ .																																
<b>Examples</b>	$Z =$ <table border="0"> <tr> <td>1.0000 - 1.0000i</td> <td>2.0000 + 1.0000i</td> <td>3.0000 - 1.0000i</td> <td>4.0000 + 1.0000i</td> </tr> <tr> <td>1.0000 + 2.0000i</td> <td>2.0000 - 2.0000i</td> <td>3.0000 + 2.0000i</td> <td>4.0000 - 2.0000i</td> </tr> <tr> <td>1.0000 - 3.0000i</td> <td>2.0000 + 3.0000i</td> <td>3.0000 - 3.0000i</td> <td>4.0000 + 3.0000i</td> </tr> <tr> <td>1.0000 + 4.0000i</td> <td>2.0000 - 4.0000i</td> <td>3.0000 + 4.0000i</td> <td>4.0000 - 4.0000i</td> </tr> </table> $P = \text{angle}(Z)$ $P =$ <table border="0"> <tr> <td>-0.7854</td> <td>0.4636</td> <td>-0.3218</td> <td>0.2450</td> </tr> <tr> <td>1.1071</td> <td>-0.7854</td> <td>0.5880</td> <td>-0.4636</td> </tr> <tr> <td>-1.2490</td> <td>0.9828</td> <td>-0.7854</td> <td>0.6435</td> </tr> <tr> <td>1.3258</td> <td>-1.1071</td> <td>0.9273</td> <td>-0.7854</td> </tr> </table>	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	-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
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																														
-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																														
<b>Algorithm</b>	angle can be expressed as:																																
	$\text{angle}(z) = \text{imag}(\log(z)) = \text{atan2}(\text{imag}(z), \text{real}(z))$																																
<b>See Also</b>	<a href="#">abs</a> <a href="#">Absolute value and complex magnitude</a> <a href="#">unwrap</a> <a href="#">Correct phase angles</a>																																

## ans

---

<b>Purpose</b>	The most recent answer
<b>Syntax</b>	ans
<b>Description</b>	The ans variable is created automatically when no output argument is specified.
<b>Examples</b>	<p>The statement <math>2+2</math> is the same as <code>ans = 2+2</code></p>

<b>Purpose</b>	Test for any nonzeros						
<b>Syntax</b>	$B = \text{any}(A)$ $B = \text{any}(A, dim)$						
<b>Description</b>	<p><math>B = \text{any}(A)</math> tests whether <i>any</i> of the elements along various dimensions of an array are nonzero or logical true (1).</p> <p>If A is a vector, <math>\text{any}(A)</math> returns logical true (1) if any of the elements of A are nonzero, and returns logical false (0) if all the elements are zero.</p> <p>If A is a matrix, <math>\text{any}(A)</math> treats the columns of A as vectors, returning a row vector of 1s and 0s.</p> <p>If A is a multidimensional array, <math>\text{any}(A)</math> treats the values along the first non-singleton dimension as vectors, returning a logical condition for each vector.</p> <p><math>B = \text{any}(A, dim)</math> tests along the dimension of A specified by scalar <i>dim</i>.</p>						
	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> </table> <span style="margin-left: 20px;"><math>A</math></span> <span style="margin-left: 20px;"><math>\text{any}(A, 1)</math></span> <span style="margin-left: 20px;"><math>\text{any}(A, 2)</math></span>	1	0	1	0	0	0
1	0	1					
0	0	0					

<b>Examples</b>	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,  $\text{any}(B)$  yields 1.

This makes any particularly useful in if statements,

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

any

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

<b>Purpose</b>	Load a compiled AppleScript from a file and execute it
<b>Syntax</b>	<pre>appl escri pt( <i>filename</i>) appl escri pt( <i>filename</i> ' -useEngl i sh') resul t = appl escri pt( <i>filename</i>) appl escri pt( <i>filename</i>, ' VarName1', ' VarVal ue1', ...)</pre>
<b>Description</b>	<p><code>appl escri pt( <i>filename</i>)</code> loads a compiled AppleScript from the file <i>filename</i> and executes it. If <i>filename</i> is not a full path name, then <code>appl escri pt</code> searches for <i>filename</i> along the MATLAB path.</p> <p><code>appl escri pt( <i>filename</i> ' -useEngl i sh')</code> forces <code>appl escri pt</code> to use the English AppleScript dialect when compiling both the script in <i>filename</i> and any AppleScript variables passed to the script. By default, <code>appl escri pt</code> uses the current system AppleScript dialect, which can be set with (for example) the Script Editor application.</p> <p><code>resul t = appl escri pt( <i>filename</i>)</code> returns in <code>resul t</code> the value that the AppleScript returns, converted to a string.</p> <p><code>appl escri pt( <i>filename</i>, ' VarName1', ' VarVal ue1', ...)</code> sets the value of the AppleScript's property or variable whose name is specified in <code>VarName</code> to the value specified in <code>VarVal ue</code>.</p>
<b>Remarks</b>	<code>appl escri pt</code> is available on the Macintosh only.
<b>Examples</b>	<p>Compile an AppleScript and save it to the file <code>rename</code>:</p> <pre>tell application "Fi nger"     set name of item <i>itemName</i> to <i>newName</i> end tell</pre> <p>The <code>appl escri pt</code> command renames file <code>hell o</code> on volume <code>MyDi sk</code> to the new name <code>wor l d</code>.</p> <pre>appl escri pt(' rename', ' <i>itemName</i>', '"MyDi sk: hell o"', ...               ' <i>newName</i>', '"wor l d"');</pre>

# arename

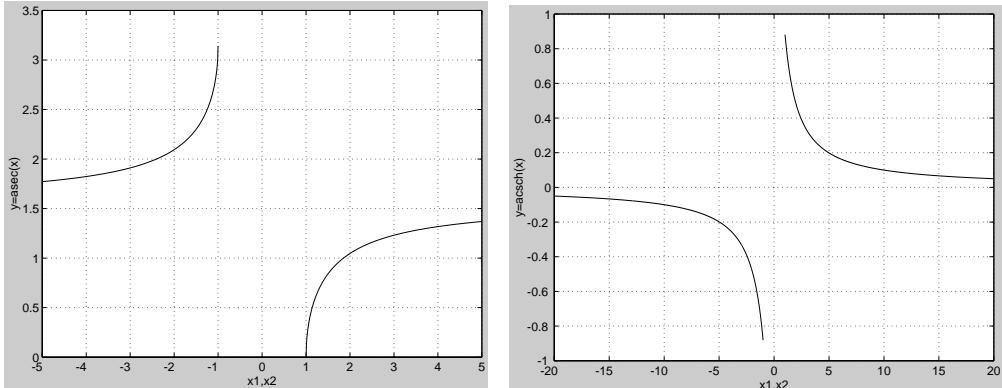
---

<b>Purpose</b>	Rename Macintosh File		
<b>Syntax</b>	<code>arename(<i>oldfilename, newname</i>)</code>		
<b>Description</b>	<code>arename(<i>oldfilename, newname</i>)</code> renames the file <i>oldfilename</i> to have the name <i>newname</i> . <i>oldfilename</i> can be a full or partial path name.		
<b>See Also</b>	<code>acopy</code>	Copy Macintosh file from one folder to another	
	<code>amove</code>	Move Macintosh file from one folder to another	
	<code>areveal</code>	Reveal filename on Macintosh desktop	
	<code>appl scri pt</code>	Load a compiled AppleScript from a file and execute it	

<b>Purpose</b>	Reveal filename on Macintosh desktop								
<b>Syntax</b>	<code>areveal (<i>filename</i>)</code>								
<b>Description</b>	<code>areveal (<i>filename</i>)</code> opens the window of the folder containing <i>filename</i> on the Macintosh desktop. <i>filename</i> can be a full or partial path name.								
<b>See Also</b>	<table><tr><td><code>acopy</code></td><td>Copy Macintosh file from one folder to another</td></tr><tr><td><code>amove</code></td><td>Move Macintosh file from one folder to another</td></tr><tr><td><code>appl escript</code></td><td>Load a compiled AppleScript from a file and execute it</td></tr><tr><td><code>arename</code></td><td>Rename Macintosh File</td></tr></table>	<code>acopy</code>	Copy Macintosh file from one folder to another	<code>amove</code>	Move Macintosh file from one folder to another	<code>appl escript</code>	Load a compiled AppleScript from a file and execute it	<code>arename</code>	Rename Macintosh File
<code>acopy</code>	Copy Macintosh file from one folder to another								
<code>amove</code>	Move Macintosh file from one folder to another								
<code>appl escript</code>	Load a compiled AppleScript from a file and execute it								
<code>arename</code>	Rename Macintosh File								

## asec, asech

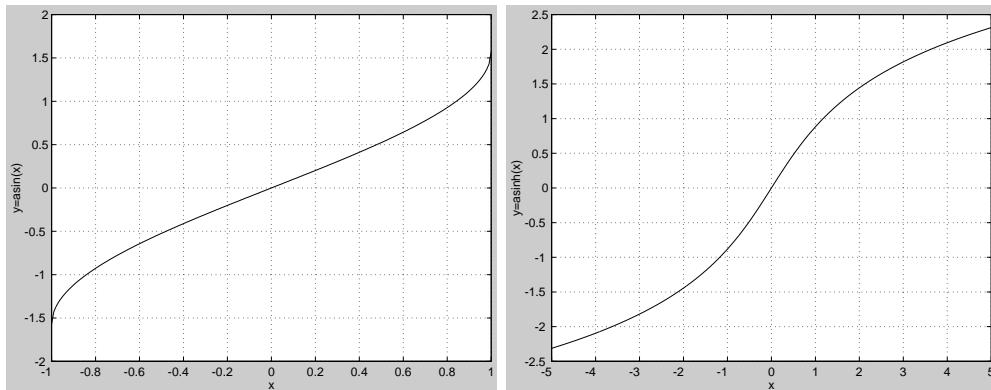
<b>Purpose</b>	Inverse secant and inverse hyperbolic secant
<b>Syntax</b>	$Y = \text{asec}(X)$ $Y = \text{asech}(X)$
<b>Description</b>	The asec and asech functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.
	$Y = \text{asec}(X)$ returns the inverse secant (arcsecant) for each element of $X$ .
	$Y = \text{asech}(X)$ returns the inverse hyperbolic secant for each element of $X$ .
<b>Examples</b>	Graph the inverse secant over the domains $1 \leq x \leq 5$ and $-5 \leq x \leq -1$ , and the inverse hyperbolic secant over the domain $0 < x \leq 1$ .  <pre>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))</pre>



<b>Algorithm</b>	$\sec^{-1}(z) = \cos^{-1}\left(\frac{1}{z}\right)$
	$\operatorname{sech}^{-1}(z) = \cosh^{-1}\left(\frac{1}{z}\right)$

**See Also** sec, sech Secant and hyperbolic secant

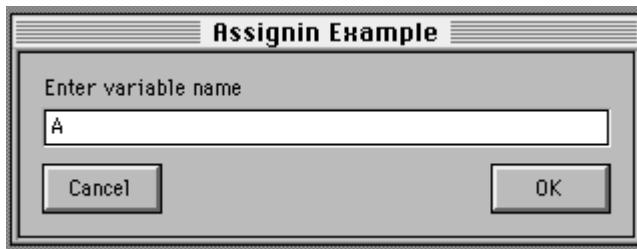
<b>Purpose</b>	Inverse sine and inverse hyperbolic sine
<b>Syntax</b>	$Y = \text{asin}(X)$ $Y = \text{asinh}(X)$
<b>Description</b>	The <code>asin</code> and <code>asinh</code> functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.
	$Y = \text{asin}(X)$ returns the inverse sine (arcsine) for each element of $X$ . For real elements of $X$ in the domain $[-1, 1]$ , $\text{asin}(X)$ is in the range $[-\pi/2, \pi/2]$ . For real elements of $x$ outside the range $[-1, 1]$ , $\text{asin}(X)$ is complex.
	$Y = \text{asinh}(X)$ returns the inverse hyperbolic sine for each element of $X$ .
<b>Examples</b>	Graph the inverse sine function over the domain $-1 \leq x \leq 1$ , and the inverse hyperbolic sine function over the domain $-5 \leq x \leq 5$ .
	<pre>x = -1: .01: 1; plot(x, asin(x)) x = -5: .01: 5; plot(x, asinh(x))</pre>
<b>Algorithm</b>	$\sin^{-1}(z) = -i \log\left[iz + (1 - z^2)^{\frac{1}{2}}\right]$ $\sinh^{-1}(z) = \log\left[z + (z^2 + 1)^{\frac{1}{2}}\right]$
<b>See Also</b>	<code>sin</code> , <code>sinh</code> Sine and hyperbolic sine



# assignin

<b>Purpose</b>	Assign value to variable in workspace
<b>Syntax</b>	<code>assignin(ws, 'name', v)</code>
<b>Description</b>	<code>assignin(ws, 'name', v)</code> assigns the variable ' <i>name</i> ' in the workspace <i>ws</i> the value <i>v</i> . ' <i>name</i> ' is created if it doesn't exist. <i>ws</i> can be either 'caller' or 'base'.
<b>Examples</b>	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))
```



<b>See Also</b>	<code>evalin</code>	Evaluate expression in workspace.
-----------------	---------------------	-----------------------------------

**Purpose**

Inverse tangent and inverse hyperbolic tangent

**Syntax**

$Y = \text{atan}(X)$

$Y = \text{atanh}(X)$

**Description**

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

$Y = \text{atan}(X)$  returns the inverse tangent (arctangent) for each element of  $X$ .

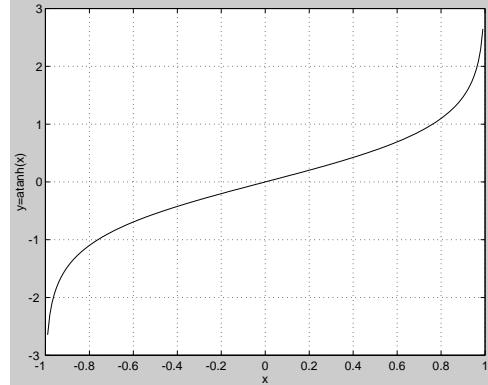
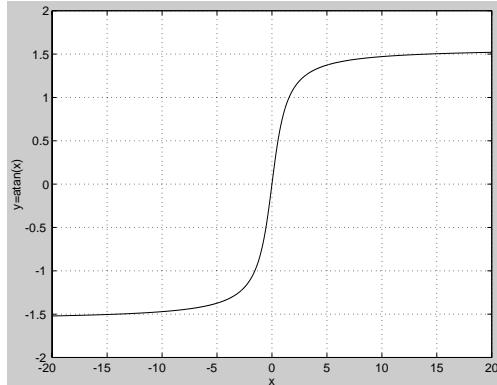
For real elements of  $X$ ,  $\text{atan}(X)$  is in the range  $[-\pi/2, \pi/2]$ .

$Y = \text{atanh}(X)$  returns the inverse hyperbolic tangent for each element of  $X$ .

**Examples**

Graph the inverse tangent function over the domain  $-20 \leq x \leq 20$ , 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

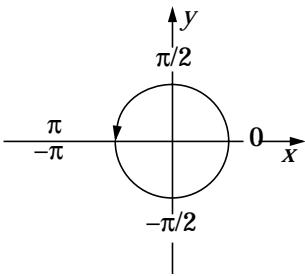
Four-quadrant inverse tangent

## **atan, atanh**

---

`tan, tanh`

Tangent and hyperbolic tangent

<b>Purpose</b>	Four-quadrant inverse tangent	
<b>Syntax</b>	$P = \text{atan2}(Y, X)$	
<b>Description</b>	$P = \text{atan2}(Y, X)$ 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 closed interval $[-\pi, \pi]$ , where $\pi$ is MATLAB's floating-point representation of $\pi$ . The specific quadrant is determined by $\text{sign}(Y)$ and $\text{sign}(X)$ :	
		
	This contrasts with the result of $\text{atan}(Y/X)$ , which is limited to the interval $[-\pi/2, \pi/2]$ , or the right side of this diagram.	
<b>Examples</b>	Any complex number $z = x+iy$ is converted to polar coordinates with	
	<pre>r = abs(z) theta = atan2(imag(z), real(z))</pre>	
	To convert back to the original complex number:	
	<pre>z = r *exp(i *theta)</pre>	
	This is a common operation, so MATLAB provides a function, $\text{angle}(z)$ , that simply computes $\text{atan2}(\text{imag}(z), \text{real}(z))$ .	
<b>See Also</b>	<a href="#">atan</a> , <a href="#">atanh</a> <a href="#">tan</a> , <a href="#">tanh</a>	Inverse tangent and inverse hyperbolic tangent Tangent and hyperbolic tangent

# auread

<b>Purpose</b>	Read NeXT/SUN (. au) sound file
<b>Syntax</b>	<pre>y = auread(aufile) [y, Fs, bits] = auread(aufile) [...] = auread(aufile, N) [...] = auread(aufile, [N1, N2]) size = auread(aufile, 'size')</pre>
<b>Description</b>	<p>Supports multi-channel data in the following formats:</p> <ul style="list-style-type: none"><li>• 8-bit mu-law</li><li>• 8-, 16-, and 32-bit linear</li><li>• floating-point</li></ul>
	<p><code>y = auread(aufile)</code> loads a sound file specified by the string <code>aufile</code>, returning the sampled data in <code>y</code>. The . au extension is appended if no extension is given. Amplitude values are in the range <math>[-1, +1]</math>.</p>
	<p><code>[y, Fs, bits] = auread(aufile)</code> returns the sample rate (<code>Fs</code>) in Hertz and the number of bits per sample (<code>bits</code>) used to encode the data in the file.</p>
	<p><code>[...] = auread(aufile, N)</code> returns only the first <code>N</code> samples from each channel in the file.</p>
	<p><code>[...] = auread(aufile, [N1 N2])</code> returns only samples <code>N1</code> through <code>N2</code> from each channel in the file.</p>
	<p><code>size = auread(aufile, 'size')</code> returns the size of the audio data contained in the file in place of the actual audio data, returning the vector <code>size = [samples channels]</code>.</p>
<b>See Also</b>	<code>afwrite</code> Write NeXT/SUN (. au) sound file <code>wavread</code> Read Microsoft WAVE (. wav) sound file

<b>Purpose</b>	Write NeXT/SUN (. au) sound file
<b>Syntax</b>	<pre>auwrite(y, <i>aufile</i>) auwrite(y, Fs, <i>aufile</i>) auwrite(y, Fs, N, <i>aufile</i>) auwrite(y, Fs, N, <i>method</i>, <i>aufile</i>)</pre>
<b>Description</b>	auwrite supports multi-channel data for 8-bit mu-law, and 8- and 16-bit linear formats.
	auwrite(y, <i>aufile</i> ) writes a sound file specified by the string <i>aufile</i> . 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, <i>aufile</i> ) specifies the sample rate of the data in Hertz.
	auwrite(y, Fs, N, <i>aufile</i> ) selects the number of bits in the encoder. Allowable settings are N = 8 and N = 16.
	auwrite(y, Fs, N, <i>method</i> , <i>aufile</i> ) allows selection of the encoding method, which can be either 'mu' or 'linear'. Note that mu-law files must be 8-bit. By default, <i>method</i> ='mu'.
<b>See Also</b>	<a href="#">auread</a> <a href="#">Read NeXT/SUN (. au) sound file</a> <a href="#">wavwrite</a> <a href="#">Write Microsoft WAVE (. wav) sound file</a>

# auwrite

---

<b>Purpose</b>	Improve accuracy of computed eigenvalues
<b>Syntax</b>	$[D, B] = \text{balance}(A)$ $B = \text{balance}(A)$
<b>Description</b>	<p><math>[D, B] = \text{balance}(A)</math> returns a diagonal matrix <math>D</math> whose elements are integer powers of two, and a balanced matrix <math>B</math> so that <math>B = D \setminus A * D</math>. If <math>A</math> is symmetric, then <math>B == A</math> and <math>D</math> is the identity matrix.</p> <p><math>B = \text{balance}(A)</math> returns just the balanced matrix <math>B</math>.</p>
<b>Remarks</b>	<p>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,</p> $\text{cond}(V) = \text{norm}(V) * \text{norm}(\text{inv}(V))$ <p>where</p> $[V, D] = \text{eig}(A)$ <p>(The condition number of <math>A</math> itself is irrelevant to the eigenvalue problem.)</p> <p>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.</p> <p>MATLAB's eigenvalue function, <math>\text{eig}(A)</math>, automatically balances <math>A</math> before computing its eigenvalues. Turn off the balancing with <math>\text{eig}(A, 'nobalance')</math>.</p>

# balance

## 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  $\text{cond}(V)$  is 31.9814. The ill conditioning is concentrated in the scaling matrix;  $\text{cond}(D)$  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

---

<b>Purpose</b>	Base to decimal number conversion
<b>Syntax</b>	<code>d = base2dec('strn', base)</code>
<b>Description</b>	<code>d = base2dec('strn', base)</code> converts the string number <i>strn</i> of the specified base into its decimal (base 10) equivalent. <i>base</i> must be an integer between 2 and 36. If ' <i>strn</i> ' is a character array, each row is interpreted as a string in the specified base.
<b>Examples</b>	The expression <code>base2dec('212', 3)</code> converts $212_3$ to decimal, returning 23.
<b>See Also</b>	<code>dec2base</code>

**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, i err] = besselh( . . . )
```

**Definitions** The differential equation

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

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

$$Y_v(z) = \frac{J_v(z)\cos(v\pi) - J_{-v}(z)}{\sin(v\pi)}$$

The relationship between the Hankel and Bessel functions is:

**Description**  $H = \text{besselh}(nu, K, Z)$  for  $K = 1$  or  $2$  computes the Hankel functions

$H_v^{(1)}(z)$  or  $H_v^{(2)}(z)$  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.

$H = \text{besselh}(nu, Z)$  uses  $K = 1$ .

$H = \text{besselh}(nu, 1, Z, 1)$  scales  $H_v^{(1)}(z)$  by  $\exp(-i * z)$ .

$H = \text{besselh}(nu, 2, Z, 1)$  scales  $H_v^{(2)}(z)$  by  $\exp(+i * z)$ .

## besselh

---

[H, i err] = bessel h( . . . ) also returns an array of error flags:

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

**Purpose**

Modified Bessel functions

**Syntax**

<code>I = bessel i (nu, Z)</code>	Modified Bessel function of the 1st kind
<code>K = bessel k(nu, Z)</code>	Modified Bessel function of the 3rd kind
<code>E = bessel i (nu, Z, 1)</code>	
<code>K = bessel k(nu, Z, 1)</code>	
<code>[I, i err] = bessel i ( . . . )</code>	
<code>[K, i err] = bessel k( . . . )</code>	

**Definitions**

The differential equation

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

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

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

$I_v(z)$  and  $K_v(z)$  are defined by:

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

$$K_v(z) = \left(\frac{\pi}{2}\right) \frac{I_{-v}(z) - I_v(z)}{\sin(v\pi)}$$

**Description**

`I = bessel i (nu, Z)` computes modified Bessel functions of the first kind,  $I_v(z)$ , 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.

# besseli, besselk

---

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

`E = bessel i (nu, Z, 1)` computes  $bessel i (nu, Z) \cdot \exp(-Z)$ .

`K = bessel k(nu, Z, 1)` computes  $bessel k(nu, Z) \cdot \exp(-Z)$ .

`[I, i err] = bessel i(...)` and `[K, i err] = bessel k(...)` also return an array of error flags.

`i err = 1`                   Illegal arguments.

`i err = 2`                   Overflow. Return Inf.

`i err = 3`                   Some loss of accuracy in argument reduction.

`i err = 4`                   Unacceptable loss of accuracy, `Z` or `nu` too large.

`i err = 5`                   No convergence. Return NaN.

## Examples

`bessel i (3: 9, (0: . 2: 10)', 1)` generates the entire table on page 423 of Abramowitz and Stegun, *Handbook of Mathematical Functions*.

`bessel k(3: 9, (0: . 2: 10)', 1)` generates part of the table on page 424 of Abramowitz and Stegun, *Handbook of Mathematical Functions*.

## Algorithm

The `bessel i` and `bessel k` 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.

# besselj, bessely

<b>Purpose</b>	Bessel functions
<b>Syntax</b>	$J = \text{besselj}(\text{nu}, Z)$ Bessel function of the 1st kind $Y = \text{bessely}(\text{nu}, Z)$ Bessel function of the 2nd kind $[J, \text{ierr}] = \text{besselj}(\text{nu}, Z)$ $[Y, \text{ierr}] = \text{bessely}(\text{nu}, Z)$
<b>Definition</b>	The differential equation
	$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} + (z^2 - v^2)y = 0$
	where $v$ is a nonnegative constant, is called <i>Bessel's equation</i> , and its solutions are known as <i>Bessel functions</i> .
	$J_v(z)$ and $J_{-v}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger $v$ . $J_v(z)$ is defined by:
	$J_v(z) = \left(\frac{z}{2}\right)^v \sum_{k=0}^{\infty} \frac{\left(-\frac{z^2}{4}\right)^k}{k! \Gamma(v+k+1)}, \quad \text{where } \Gamma(a) = \int_0^{\infty} e^{-t} t^{a-1} dt$
	$Y_v(z)$ is a second solution of Bessel's equation—linearly independent of $J_v(z)$ —defined by:
	$Y_v(z) = \frac{J_v(z) \cos(v\pi) - J_{-v}(z)}{\sin(v\pi)}$
<b>Description</b>	$J = \text{besselj}(\text{nu}, Z)$ computes Bessel functions of the first kind, $J_v(z)$ , for each element of the complex array $Z$ . The order $\text{nu}$ need not be an integer, but must be real. The argument $Z$ can be complex. The result is real where $Z$ is positive.
	If $\text{nu}$ 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 = bessel y(nu, Z)` computes Bessel functions of the second kind,  $Y_v(z)$ , for real, nonnegative order nu and argument Z.

`[J, i err] = besselj (nu, Z)` and `[Y, i err] = bessel y(nu, Z)` also return an array of error flags.

- |                        |   |
|------------------------|---|
| <code>i err = 1</code> | Illegal arguments.                                |
| <code>i err = 2</code> | Overflow. Return Inf.                             |
| <code>i err = 3</code> | Some loss of accuracy in argument reduction.      |
| <code>i err = 4</code> | Unacceptable loss of accuracy, Z or nu too large. |
| <code>i err = 5</code> | No convergence. Return NaN.                       |

## Remarks

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

$$H_v^{(1)}(z) = J_v(z) + i Y_v(z)$$

$$H_v^{(2)}(z) = J_v(z) - i Y_v(z)$$

where  $J_v(z)$  is `besselj`, and  $Y_v(z)$  is `bessel y`. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see `bessel h`).

## 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 `bessel y` functions use a Fortran MEX-file to call a library developed by D. E. Amos [3] [4].

## See Also

`airy` Airy functions  
`bessel i, bessel k` 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.

## **besselj, bessely**

---

- [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** 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**  $B = \text{beta}(Z, W)$  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).

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

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

## **beta, betainc, betaln**

---

### **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,

$$\begin{aligned}\text{beta}(n, 3) &= (n-1)! * 2! / (n+2)! \\ &= 2 / (n * (n+1) * (n+2))\end{aligned}$$

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

For  $x = 510$ ,  $\text{betaln}(x, x) = -708.8616$ , which, on a computer with IEEE arithmetic, is slightly less than  $\log(\text{realmin})$ . Here  $\text{beta}(x, x)$  would underflow (or be denormal).

### **Algorithm**

$$\begin{aligned}\text{beta}(z, w) &= \exp(\text{gammaln}(z) + \text{gammaln}(w) - \text{gammaln}(z+w)) \\ \text{betaln}(z, w) &= \text{gammaln}(z) + \text{gammaln}(w) - \text{gammaln}(z+w)\end{aligned}$$

<b>Purpose</b>	BiConjugate Gradients method
<b>Syntax</b>	$x = \text{bi cg}(A, b)$ $\text{bi cg}(A, b, tol)$ $\text{bi cg}(A, b, tol, maxi t)$ $\text{bi cg}(A, b, tol, maxi t, M)$ $\text{bi cg}(A, b, tol, maxi t, M1, M2)$ $\text{bi cg}(A, b, tol, maxi t, M1, M2, x0)$ $x = \text{bi cg}(A, b, tol, maxi t, M1, M2, x0)$ $[x, flag] = \text{bi cg}(A, b, tol, maxi t, M1, M2, x0)$ $[x, flag, rel res] = \text{bi cg}(A, b, tol, maxi t, M1, M2, x0)$ $[x, flag, rel res, iter] = \text{bi cg}(A, b, tol, maxi t, M1, M2, x0)$ $[x, flag, rel res, iter, resvec] = \text{bi cg}(A, b, tol, maxi t, M1, M2, x0)$
<b>Description</b>	<p><math>x = \text{bi cg}(A, b)</math> attempts to solve the system of linear equations <math>A^*x = b</math> for <math>x</math>. The coefficient matrix <math>A</math> must be square and the right hand side (column) vector <math>b</math> must have length <math>n</math>, where <math>A</math> is <math>n</math>-by-<math>n</math>. <math>\text{bi cg}</math> will start iterating from an initial estimate that by default is an all zero vector of length <math>n</math>. Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate <math>x</math> has relative residual <math>\text{norm}(b - A^*x) / \text{norm}(b)</math> less than or equal to the tolerance of the method. The default tolerance is <math>1e-6</math>. The default maximum number of iterations is the minimum of <math>n</math> and 20. No preconditioning is used.</p> <p><math>\text{bi cg}(A, b, tol)</math> specifies the tolerance of the method, <math>tol</math>.</p> <p><math>\text{bi cg}(A, b, tol, maxi t)</math> additionally specifies the maximum number of iterations, <math>maxi t</math>.</p> <p><math>\text{bi cg}(A, b, tol, maxi t, M)</math> and <math>\text{bi cg}(A, b, tol, maxi t, M1, M2)</math> use left preconditioner <math>M</math> or <math>M = M1 * M2</math> and effectively solve the system <math>\text{inv}(M)^* A^* x = \text{inv}(M)^* b</math> for <math>x</math>. If <math>M1</math> or <math>M2</math> is given as the empty matrix (<math>[]</math>), it is considered to be the identity matrix, equivalent to no preconditioning at all. Since systems of equations of the form <math>M^*y = r</math> are solved using backslash within <math>\text{bi cg}</math>, it is wise to factor preconditioners into their LU factors first. For example, replace <math>\text{bi cg}(A, b, tol, maxi t, M)</math> with:</p> $[M1, M2] = \text{lu}(M);$ $\text{bi cg}(A, b, tol, maxi t, M1, M2).$

## bicg

`bi cg(A, b, tol, maxi t, M1, M2, x0)` specifies the initial estimate  $x_0$ . If  $x_0$  is given as the empty matrix ([ ]), the default all zero vector is used.

`x = bi cg(A, b, tol, maxi t, M1, M2, x0)` returns a solution  $x$ . If `bi cg` converged, a message to that effect is displayed. If `bi cg` failed to converge after the maximum number of iterations or halted for any reason, a warning message is printed displaying the relative residual  $\text{norm}(b - A^*x) / \text{norm}(b)$  and the iteration number at which the method stopped or failed.

`[x, flag] = bi cg(A, b, tol, maxi t, M1, M2, x0)` returns a solution  $x$  and a flag that describes the convergence of `bi cg`:

Flag	Convergence
0	<code>bi cg</code> converged to the desired tolerance <code>tol</code> within <code>maxi t</code> iterations without failing for any reason.
1	<code>bi cg</code> iterated <code>maxi t</code> 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 <code>bi cg</code> 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, rel res] = bi cg(A, b, tol, maxi t, M1, M2, x0)` also returns the relative residual  $\text{norm}(b - A^*x) / \text{norm}(b)$ . If `flag` is 0, then  $\text{rel res} \leq \text{tol}$ .

`[x, flag, rel res, iter] = bi cg(A, b, tol, maxi t, M1, M2, x0)` also returns the iteration number at which  $x$  was computed. This always satisfies  $0 \leq \text{iter} \leq \text{maxi t}$ .

`[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.

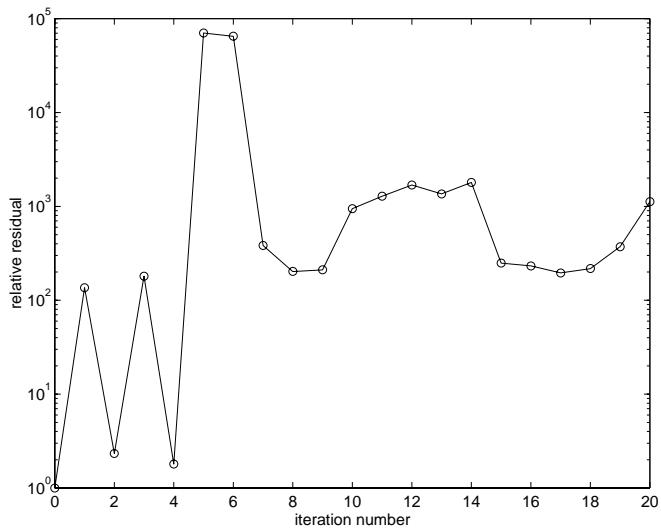
```
x = A \ b
norm(b-A*x) / norm(b) =
6.8476e-18
```

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 `semi logy(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:

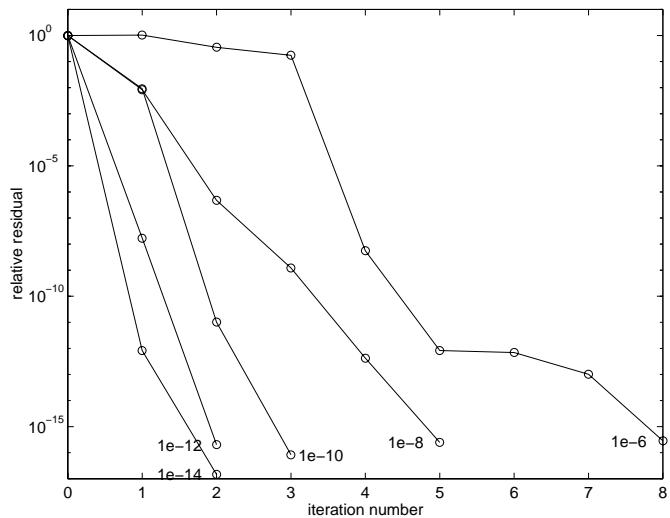
```
[L2, U2] = lunc(A, 1e-6)
nnz(L2) =
6231
nnz(U2) =
4559
```

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
```

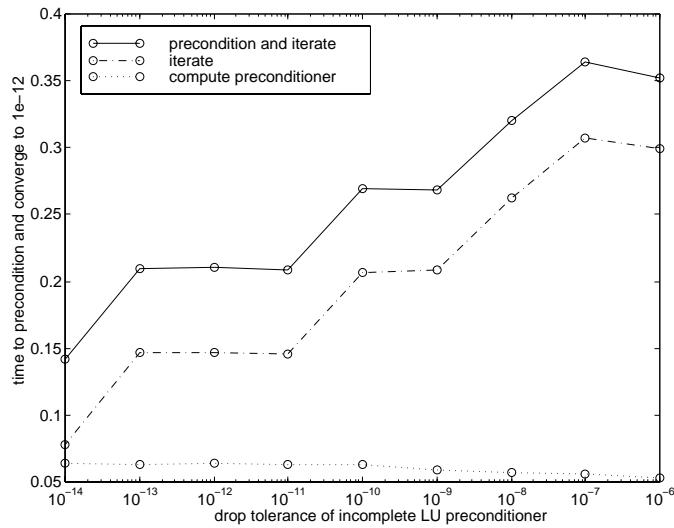
and bicg converges to within the desired tolerance at iteration number 8. Decreasing the value of the drop tolerance increases the fill-in of the incomplete factors but also increases the accuracy of the approximation to the original matrix. Thus, the preconditioned system becomes closer to  $i\text{nv}(U)^{-1}i\text{nv}(L)^{-1}L^{-1}U^{-1}x = i\text{nv}(U)^{-1}i\text{nv}(L)^{-1}b$ , 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 bi cg using six different incomplete LU factors as preconditioners. Each line in the graph is labelled with the drop tolerance of the preconditioner used in bi cg.



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 bi cg still takes longer than backslash.



## See Also

[bi\\_cgstab](#)      BiConjugate Gradients Stabilized method  
[cgs](#)      Conjugate Gradients Squared method  
[gmres](#)      Generalized Minimum Residual method (with restarts)  
[lui\\_nc](#)      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.

# bicgstab

---

Purpose	BiConjugate Gradients Stabilized method
Syntax	<pre>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)</pre>
Description	<p><code>x = bicgstab(A, b)</code> attempts to solve the system of linear equations <math>A^*x = b</math> for <math>x</math>. The coefficient matrix <math>A</math> must be square and the right hand side (column) vector <math>b</math> must have length <math>n</math>, where <math>A</math> is <math>n</math>-by-<math>n</math>. <code>bicgstab</code> will start iterating from an initial estimate that by default is an all zero vector of length <math>n</math>. Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate <math>x</math> has relative residual <math>\text{norm}(b - A^*x) / \text{norm}(b)</math> less than or equal to the tolerance of the method. The default tolerance is <math>1e-6</math>. The default maximum number of iterations is the minimum of <math>n</math> and <math>20</math>. No preconditioning is used.</p> <p><code>bicgstab(A, b, tol)</code> specifies the tolerance of the method, <code>tol</code>.</p> <p><code>bicgstab(A, b, tol, maxit)</code> additionally specifies the maximum number of iterations, <code>maxit</code>.</p> <p><code>bicgstab(A, b, tol, maxit, M)</code> and <code>bicgstab(A, b, tol, maxit, M1, M2)</code> use left preconditioner <math>M</math> or <math>M = M1 * M2</math> and effectively solve the system <math>\text{inv}(M)^* A^* x = \text{inv}(M)^* b</math> for <math>x</math>. If <math>M1</math> or <math>M2</math> 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 <math>M^* y = r</math> are solved using backslash within <code>bicgstab</code>, it</p>

is wise to factor preconditioners into their LU factors first. For example, replace `bi cgstab(A, b, tol, maxi t, M)` with:

```
[M1, M2] = lu(M);
bi cgstab(A, b, tol, maxi t, M1, M2).
```

`bi cgstab(A, b, tol, maxi t, M1, M2, x0)` specifies the initial estimate  $x_0$ . If  $x_0$  is given as the empty matrix ([ ]), the default all zero vector is used.

`x = bi cgstab(A, b, tol, maxi t, M1, M2, x0)` returns a solution  $x$ . If `bi cgstab` converged, a message to that effect is displayed. If `bi cgstab` failed to converge after the maximum number of iterations or halted for any reason, a warning message is printed displaying the relative residual  $\text{norm}(b - A^*x) / \text{norm}(b)$  and the iteration number at which the method stopped or failed.

`[x, flag] = bi cgstab(A, b, tol, maxi t, M1, M2, x0)` returns a solution  $x$  and a flag that describes the convergence of `bi cgstab`:

Flag	Convergence
0	<code>bi cgstab</code> converged to the desired tolerance <code>tol</code> within <code>maxi t</code> iterations without failing for any reason.
1	<code>bi cgstab</code> iterated <code>maxi t</code> 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 <code>bi cgstab</code> 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.

# bicgstab

---

[x, flag, relres] = bicgstab(A, b, tol, maxit, M1, M2, x0) also returns the relative residual  $\text{norm}(b - A^*x) / \text{norm}(b)$ . If flag is 0, then  $\text{relres} \leq \text{tol}$ .

[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  $0 \leq \text{iter} \leq \text{maxit}$ . 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.

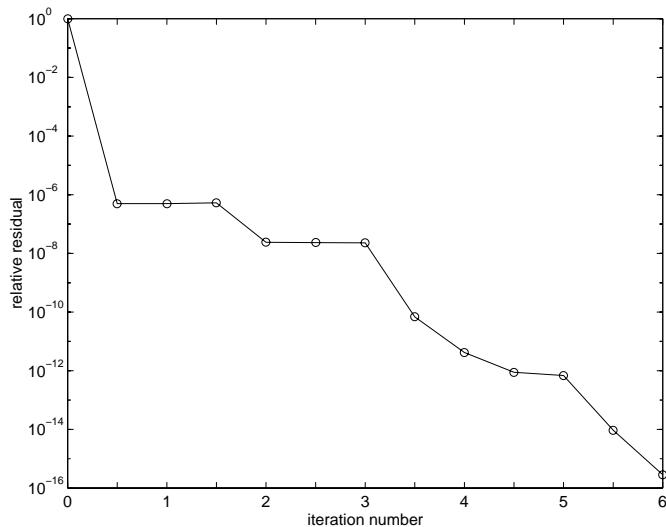
```
[L1, U1] = luinc(A, 1e-5)
[x1, flag1] = bicgstab(A, b, 1e-6, 20, L1, U1)
```

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  
`semi logy(0: 0.5: iter2, resvec2/norm(b), '-o')`

**See Also**

[bi cg](#) BiConjugate Gradients method  
[cgs](#) Conjugate Gradients Squared method  
[gmres](#) Generalized Minimum Residual method (with restarts)  
[l ui nc](#) 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.

## bin2dec

---

<b>Purpose</b>	Binary to decimal number conversion
<b>Syntax</b>	<code>bin2dec(<i>binarystr</i>)</code>
<b>Description</b>	<code>bin2dec(<i>binarystr</i>)</code> interprets the binary string <i>binarystr</i> and returns the equivalent decimal number.
<b>Examples</b>	<code>bin2dec('010111')</code> returns 23.
<b>See Also</b>	<code>dec2bin</code>

---

<b>Purpose</b>	Bit-wise AND														
<b>Syntax</b>	<code>C = bitand(A, B)</code>														
<b>Description</b>	<code>C = bitand(A, B)</code> returns the bit-wise AND of two nonnegative integer arguments <code>A</code> and <code>B</code> . To ensure the operands are integers, use the <code>ceil</code> , <code>fix</code> , <code>floor</code> , and <code>round</code> functions.														
<b>Examples</b>	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.  <code>C = bitand(13, 27)</code>  <code>C =</code>  9														
<b>See Also</b>	<table><tr><td><code>bitcmp</code></td><td>Complement bits</td></tr><tr><td><code>bitget</code></td><td>Get bit</td></tr><tr><td><code>bitmакс</code></td><td>Maximum floating-point integer</td></tr><tr><td><code>bitor</code></td><td>Bit-wise OR</td></tr><tr><td><code>bitsel</code></td><td>Set bit</td></tr><tr><td><code>bitshift</code></td><td>Bit-wise shift</td></tr><tr><td><code>bitxor</code></td><td>Bit-wise XOR</td></tr></table>	<code>bitcmp</code>	Complement bits	<code>bitget</code>	Get bit	<code>bitmакс</code>	Maximum floating-point integer	<code>bitor</code>	Bit-wise OR	<code>bitsel</code>	Set bit	<code>bitshift</code>	Bit-wise shift	<code>bitxor</code>	Bit-wise XOR
<code>bitcmp</code>	Complement bits														
<code>bitget</code>	Get bit														
<code>bitmакс</code>	Maximum floating-point integer														
<code>bitor</code>	Bit-wise OR														
<code>bitsel</code>	Set bit														
<code>bitshift</code>	Bit-wise shift														
<code>bitxor</code>	Bit-wise XOR														

# bitcmp

---

<b>Purpose</b>	Complement bits														
<b>Syntax</b>	<code>C = bitcmp(A, n)</code>														
<b>Description</b>	<code>C = bitcmp(A, n)</code> returns the bit-wise complement of <code>A</code> as an <code>n</code> -bit floating-point integer (flint).														
<b>Example</b>	With eight-bit arithmetic, the ones' complement of 01100011 (99, decimal) is 10011100 (156, decimal).  <code>C = bitcmp(99, 8)</code>  C =  156														
<b>See Also</b>	<table><tr><td><code>bitand</code></td><td>Bit-wise AND</td></tr><tr><td><code>bitget</code></td><td>Get bit</td></tr><tr><td><code>bitmax</code></td><td>Maximum floating-point integer</td></tr><tr><td><code>bitor</code></td><td>Bit-wise OR</td></tr><tr><td><code>bitsset</code></td><td>Set bit</td></tr><tr><td><code>bitshift</code></td><td>Bit-wise shift</td></tr><tr><td><code>bitxor</code></td><td>Bit-wise XOR</td></tr></table>	<code>bitand</code>	Bit-wise AND	<code>bitget</code>	Get bit	<code>bitmax</code>	Maximum floating-point integer	<code>bitor</code>	Bit-wise OR	<code>bitsset</code>	Set bit	<code>bitshift</code>	Bit-wise shift	<code>bitxor</code>	Bit-wise XOR
<code>bitand</code>	Bit-wise AND														
<code>bitget</code>	Get bit														
<code>bitmax</code>	Maximum floating-point integer														
<code>bitor</code>	Bit-wise OR														
<code>bitsset</code>	Set bit														
<code>bitshift</code>	Bit-wise shift														
<code>bitxor</code>	Bit-wise XOR														

<b>Purpose</b>	Get bit
<b>Syntax</b>	<code>C = bitget(A, bit)</code>
<b>Description</b>	<code>C = bitget(A, bit)</code> returns the value of the bit at position <code>bit</code> in <code>A</code> . Operand <code>A</code> must be a nonnegative integer, and <code>bit</code> must be a number between 1 and the number of bits in the floating-point integer (flint) representation of <code>A</code> (52 for IEEE flints). To ensure the operand is an integer, use the <code>ceil</code> , <code>fix</code> , <code>floor</code> , and <code>round</code> functions.
<b>Example</b>	The <code>dec2bin</code> function converts decimal numbers to binary. However, you can also use the <code>bitget</code> function to show the binary representation of a decimal number. Just test successive bits from most to least significant:  <code>disp(dec2bin(13))</code> 1101 <code>C = bitget(13, 4:-1:1)</code>  <code>C =</code> 1        1        0        1

<b>See Also</b>	<code>bitand</code>	Bit-wise AND
	<code>biteq</code>	Complement bits
	<code>bitemax</code>	Maximum floating-point integer
	<code>bitor</code>	Bit-wise OR
	<code>bitset</code>	Set bit
	<code>bitshift</code>	Bit-wise shift
	<code>bithxor</code>	Bit-wise XOR

# bitmax

---

<b>Purpose</b>	Maximum floating-point integer															
<b>Syntax</b>	<code>bitmax</code>															
<b>Description</b>	<code>bitmax</code> 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 $2^{53} - 1$ .															
<b>See Also</b>	<table><tr><td><code>bitand</code></td><td>Bit-wise AND</td></tr><tr><td><code>bitemp</code></td><td>Complement bits</td></tr><tr><td><code>bitget</code></td><td>Get bit</td></tr><tr><td><code>bitor</code></td><td>Bit-wise OR</td></tr><tr><td><code>bitsset</code></td><td>Set bit</td></tr><tr><td><code>bitsshift</code></td><td>Bit-wise shift</td></tr><tr><td><code>bitxor</code></td><td>Bit-wise XOR</td></tr></table>		<code>bitand</code>	Bit-wise AND	<code>bitemp</code>	Complement bits	<code>bitget</code>	Get bit	<code>bitor</code>	Bit-wise OR	<code>bitsset</code>	Set bit	<code>bitsshift</code>	Bit-wise shift	<code>bitxor</code>	Bit-wise XOR
<code>bitand</code>	Bit-wise AND															
<code>bitemp</code>	Complement bits															
<code>bitget</code>	Get bit															
<code>bitor</code>	Bit-wise OR															
<code>bitsset</code>	Set bit															
<code>bitsshift</code>	Bit-wise shift															
<code>bitxor</code>	Bit-wise XOR															

<b>Purpose</b>	Bit-wise OR														
<b>Syntax</b>	<code>C = bitor(A, B)</code>														
<b>Description</b>	<code>C = bitor(A, B)</code> returns the bit-wise OR of two nonnegative integer arguments <code>A</code> and <code>B</code> . To ensure the operands are integers, use the <code>ceil</code> , <code>fix</code> , <code>floor</code> , and <code>round</code> functions.														
<b>Examples</b>	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.  <code>C = bitor(13, 27)</code>  <code>C =</code>  31														
<b>See Also</b>	<table><tr><td><code>bitand</code></td><td>Bit-wise AND</td></tr><tr><td><code>biteq</code></td><td>Complement bits</td></tr><tr><td><code>bitget</code></td><td>Get bit</td></tr><tr><td><code>bitemax</code></td><td>Maximum floating-point integer</td></tr><tr><td><code>bitset</code></td><td>Set bit</td></tr><tr><td><code>bithshift</code></td><td>Bit-wise shift</td></tr><tr><td><code>bitor</code></td><td>Bit-wise XOR</td></tr></table>	<code>bitand</code>	Bit-wise AND	<code>biteq</code>	Complement bits	<code>bitget</code>	Get bit	<code>bitemax</code>	Maximum floating-point integer	<code>bitset</code>	Set bit	<code>bithshift</code>	Bit-wise shift	<code>bitor</code>	Bit-wise XOR
<code>bitand</code>	Bit-wise AND														
<code>biteq</code>	Complement bits														
<code>bitget</code>	Get bit														
<code>bitemax</code>	Maximum floating-point integer														
<code>bitset</code>	Set bit														
<code>bithshift</code>	Bit-wise shift														
<code>bitor</code>	Bit-wise XOR														

# bitset

---

## 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

<code>bitand</code>	Bit-wise AND
<code>bittcmp</code>	Complement bits
<code>bittget</code>	Get bit
<code>bittmax</code>	Maximum floating-point integer
<code>bitor</code>	Bit-wise OR
<code>bittshift</code>	Bit-wise shift
<code>bittxor</code>	Bit-wise XOR

<b>Purpose</b>	Bit-wise shift																
<b>Syntax</b>	<pre>C = bitshift(A, k, n) C = bitshift(A, k)</pre>																
<b>Description</b>	<p><code>C = bitshift(A, k, n)</code> returns the value of <code>A</code> shifted by <code>k</code> bits. If <code>k &gt; 0</code>, this is same as a multiplication by <math>2^k</math> (left shift). If <code>k &lt; 0</code>, this is the same as a division by <math>2^k</math> (right shift). An equivalent computation for this function is <code>C = fix(A * 2^k)</code>.</p> <p>If the shift causes <code>C</code> to overflow <code>n</code> bits, the overflowing bits are dropped. <code>A</code> must contain nonnegative integers between 0 and <code>BITMAX</code>, which you can ensure by using the <code>ceil</code>, <code>fix</code>, <code>floor</code>, and <code>round</code> functions.</p> <p><code>C = bitshift(A, k)</code> uses the default value of <code>n = 53</code>.</p>																
<b>Examples</b>	<p>Shifting 1100 (12, decimal) to the left two bits yields 110000 (48, decimal).</p> <pre>C = bitshift(12, 2)</pre> <p>C =</p> <pre>48</pre>																
<b>See Also</b>	<table border="0"> <tr> <td><code>bitand</code></td> <td>Bit-wise AND</td> </tr> <tr> <td><code>biteq</code></td> <td>Complement bits</td> </tr> <tr> <td><code>bitget</code></td> <td>Get bit</td> </tr> <tr> <td><code>bitmax</code></td> <td>Maximum floating-point integer</td> </tr> <tr> <td><code>bitor</code></td> <td>Bit-wise OR</td> </tr> <tr> <td><code>bitsel</code></td> <td>Set bit</td> </tr> <tr> <td><code>bitxor</code></td> <td>Bit-wise XOR</td> </tr> <tr> <td><code>fix</code></td> <td>Round towards zero</td> </tr> </table>	<code>bitand</code>	Bit-wise AND	<code>biteq</code>	Complement bits	<code>bitget</code>	Get bit	<code>bitmax</code>	Maximum floating-point integer	<code>bitor</code>	Bit-wise OR	<code>bitsel</code>	Set bit	<code>bitxor</code>	Bit-wise XOR	<code>fix</code>	Round towards zero
<code>bitand</code>	Bit-wise AND																
<code>biteq</code>	Complement bits																
<code>bitget</code>	Get bit																
<code>bitmax</code>	Maximum floating-point integer																
<code>bitor</code>	Bit-wise OR																
<code>bitsel</code>	Set bit																
<code>bitxor</code>	Bit-wise XOR																
<code>fix</code>	Round towards zero																

# bitxor

---

<b>Purpose</b>	Bit-wise XOR														
<b>Syntax</b>	<code>C = bitxor(A, B)</code>														
<b>Description</b>	<code>C = bitxor(A, B)</code> 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 <code>ceil</code> , <code>fix</code> , <code>floor</code> , and <code>round</code> functions.														
<b>Examples</b>	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.  <code>C = bitxor(13, 27)</code>  <code>C =</code> 22														
<b>See Also</b>	<table><tr><td><code>bitand</code></td><td>Bit-wise AND</td></tr><tr><td><code>bitecmp</code></td><td>Complement bits</td></tr><tr><td><code>bitget</code></td><td>Get bit</td></tr><tr><td><code>bitmax</code></td><td>Maximum floating-point integer</td></tr><tr><td><code>bitor</code></td><td>Bit-wise OR</td></tr><tr><td><code>bitsset</code></td><td>Set bit</td></tr><tr><td><code>bitshift</code></td><td>Bit-wise shift</td></tr></table>	<code>bitand</code>	Bit-wise AND	<code>bitecmp</code>	Complement bits	<code>bitget</code>	Get bit	<code>bitmax</code>	Maximum floating-point integer	<code>bitor</code>	Bit-wise OR	<code>bitsset</code>	Set bit	<code>bitshift</code>	Bit-wise shift
<code>bitand</code>	Bit-wise AND														
<code>bitecmp</code>	Complement bits														
<code>bitget</code>	Get bit														
<code>bitmax</code>	Maximum floating-point integer														
<code>bitor</code>	Bit-wise OR														
<code>bitsset</code>	Set bit														
<code>bitshift</code>	Bit-wise shift														

---

<b>Purpose</b>	A string of blanks						
<b>Syntax</b>	<code>blanks(n)</code>						
<b>Description</b>	<code>blanks(n)</code> is a string of n blanks.						
<b>Examples</b>	<code>blanks</code> is useful with the <code>display</code> function. For example,  <code>disp(['xxx' blanks(20) 'yyy'])</code> displays twenty blanks between the strings 'xxx' and 'yyy'. <code>disp(blanks(n))</code> moves the cursor down n lines.						
<b>See Also</b>	<table><tr><td><code>clc</code></td><td>Clear command window</td></tr><tr><td><code>home</code></td><td>Send the cursor home</td></tr><tr><td><code>format</code></td><td>See compact option for suppression of blank lines</td></tr></table>	<code>clc</code>	Clear command window	<code>home</code>	Send the cursor home	<code>format</code>	See compact option for suppression of blank lines
<code>clc</code>	Clear command window						
<code>home</code>	Send the cursor home						
<code>format</code>	See compact option for suppression of blank lines						

# break

---

<b>Purpose</b>	Terminate execution of <code>for</code> or <code>while</code> loop														
<b>Syntax</b>	<code>break</code>														
<b>Description</b>	<code>break</code> terminates the execution of <code>for</code> and <code>while</code> loops. In nested loops, <code>break</code> exits from the innermost loop only.														
<b>Examples</b>	The indented statements are repeatedly executed until nonpositive <code>n</code> is entered.														
	<pre>while 1     n = input('Enter n. n &lt;= 0 quits. n = ')     if n &lt;= 0, break, end     r = rank(magic(n)) end disp('That''s all.')</pre>														
<b>See Also</b>	<table><tr><td><code>end</code></td><td>Terminate <code>for</code>, <code>while</code>, and <code>if</code> statements and indicate the last index</td></tr><tr><td><code>error</code></td><td>Display error messages</td></tr><tr><td><code>for</code></td><td>Repeat statements a specific number of times</td></tr><tr><td><code>if</code></td><td>Conditionally execute statements</td></tr><tr><td><code>return</code></td><td>Return to the invoking function</td></tr><tr><td><code>switch</code></td><td>Switch among several cases based on expression</td></tr><tr><td><code>while</code></td><td>Repeat statements an indefinite number of times</td></tr></table>	<code>end</code>	Terminate <code>for</code> , <code>while</code> , and <code>if</code> statements and indicate the last index	<code>error</code>	Display error messages	<code>for</code>	Repeat statements a specific number of times	<code>if</code>	Conditionally execute statements	<code>return</code>	Return to the invoking function	<code>switch</code>	Switch among several cases based on expression	<code>while</code>	Repeat statements an indefinite number of times
<code>end</code>	Terminate <code>for</code> , <code>while</code> , and <code>if</code> statements and indicate the last index														
<code>error</code>	Display error messages														
<code>for</code>	Repeat statements a specific number of times														
<code>if</code>	Conditionally execute statements														
<code>return</code>	Return to the invoking function														
<code>switch</code>	Switch among several cases based on expression														
<code>while</code>	Repeat statements an indefinite number of times														

---

<b>Purpose</b>	Execute builtin function from overloaded method
<b>Syntax</b>	<code>builtin(function, x1, . . . , xn)</code> <code>[y1, . . . , yn] = builtin(function, x1, . . . , xn)</code>
<b>Description</b>	<code>builtin</code> is used in methods that overload builtin functions to execute the original builtin function. If <i>function</i> is a string containing the name of a builtin function, then:  <code>builtin(function, x1, . . . , xn)</code> evaluates that function at the given arguments.  <code>[y1, . . . , yn] = builtin(function, x1, . . . , xn)</code> returns multiple output arguments.
<b>Remarks</b>	<code>builtin(...)</code> is the same as <code>feval (...)</code> 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 <code>builtin</code> .)
<b>See Also</b>	<code>feval</code> Function evaluation

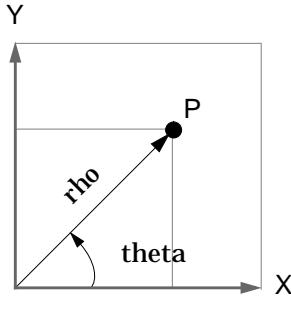
## **builtin**

---

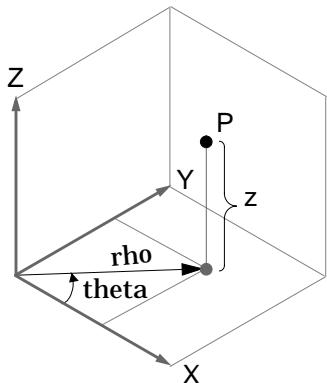
<b>Purpose</b>	Calendar																																																								
<b>Syntax</b>	<pre>c = calendar c = calendar(d) c = calendar(y, m)  calendar(...)</pre>																																																								
<b>Description</b>	<p><code>c = calendar</code> returns a 6-by-7 matrix containing a calendar for the current month. The calendar runs Sunday (first column) to Saturday.</p> <p><code>c = calendar(d)</code>, where <code>d</code> is a serial date number or a date string, returns a calendar for the specified month.</p> <p><code>c = calendar(y, m)</code>, where <code>y</code> and <code>m</code> are integers, returns a calendar for the specified month of the specified year.</p> <p><code>calendar(...)</code> displays the calendar on the screen.</p>																																																								
<b>Examples</b>	<p>The command:</p> <pre>calendar(1957, 10)</pre> <p>reveals that the Space Age began on a Friday (on October 4, 1957, when Sputnik 1 was launched).</p> <table border="0" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th colspan="7" style="text-align: center;">Oct 1957</th> </tr> <tr> <th>S</th><th>M</th><th>Tu</th><th>W</th><th>Th</th><th>F</th><th>S</th> </tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>1</td><td>2</td><td>3</td><td><u>4</u></td><td>5</td> </tr> <tr> <td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td> </tr> <tr> <td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td> </tr> <tr> <td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td> </tr> <tr> <td>27</td><td>28</td><td>29</td><td>30</td><td>31</td><td>0</td><td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td> </tr> </tbody> </table>	Oct 1957							S	M	Tu	W	Th	F	S	0	0	1	2	3	<u>4</u>	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
Oct 1957																																																									
S	M	Tu	W	Th	F	S																																																			
0	0	1	2	3	<u>4</u>	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																																																			
<b>See Also</b>	<a href="#">datenum</a> <a href="#">Serial date number</a>																																																								

# cart2pol

<b>Purpose</b>	Transform Cartesian coordinates to polar or cylindrical
<b>Syntax</b>	$[THETA, RHO, Z] = \text{cart2pol}(X, Y, Z)$ $[THETA, RHO] = \text{cart2pol}(X, Y)$
<b>Description</b>	$[THETA, RHO, Z] = \text{cart2pol}(X, Y, Z)$ 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).  $[THETA, RHO] = \text{cart2pol}(X, Y)$ transforms two-dimensional Cartesian coordinates stored in corresponding elements of arrays X and Y into polar coordinates.
<b>Algorithm</b>	The mapping from two-dimensional Cartesian coordinates to polar coordinates, and from three-dimensional Cartesian coordinates to cylindrical coordinates is:



Two-Dimensional Mapping  
 $\text{theta} = \text{atan2}(y, x)$   
 $\text{rho} = \sqrt{x.^2 + y.^2}$



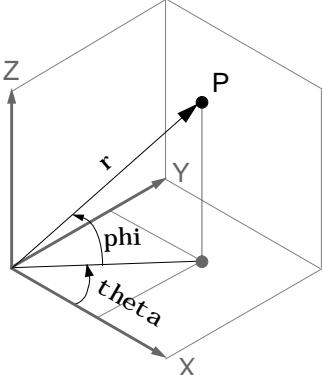
Three-Dimensional Mapping  
 $\text{theta} = \text{atan2}(y, x)$   
 $\text{rho} = \sqrt{x.^2 + y.^2}$   
 $z = z$

**See Also**

[cart2sph](#)  
[pol2cart](#)  
[sph2cart](#)

Transform Cartesian coordinates to spherical  
Transform polar or cylindrical coordinates to Cartesian  
Transform spherical coordinates to Cartesian

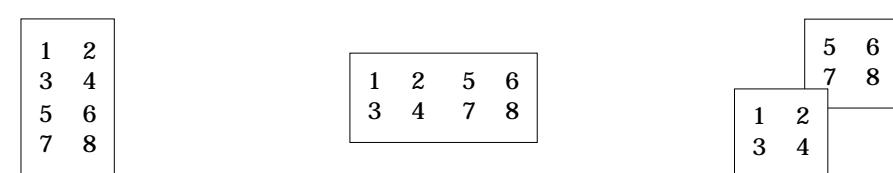
# cart2sph

<b>Purpose</b>	Transform Cartesian coordinates to spherical
<b>Syntax</b>	$[ \text{THETA}, \text{PHI}, \text{R} ] = \text{cart2sph}(X, Y, Z)$
<b>Description</b>	$[ \text{THETA}, \text{PHI}, \text{R} ] = \text{cart2sph}(X, Y, Z)$ 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.
<b>Algorithm</b>	The mapping from three-dimensional Cartesian coordinates to spherical coordinates is:  $\begin{aligned}\text{theta} &= \text{atan2}(y, x) \\ \text{phi} &= \text{atan2}(z, \sqrt{x.^2 + y.^2}) \\ \text{r} &= \sqrt{x.^2+y.^2+z.^2}\end{aligned}$
<b>See Also</b>	<a href="#">cart2pol</a> <a href="#">pol 2cart</a> <a href="#">sph2cart</a> Transform Cartesian coordinates to polar or cylindrical Transform polar or cylindrical coordinates to Cartesian Transform spherical coordinates to Cartesian

---

<b>Purpose</b>	Case switch
<b>Description</b>	<p>case is part of the switch statement syntax, which allows for conditional execution.</p> <p>A particular case consists of the case statement itself, followed by a case expression, and one or more statements.</p> <p>A case is executed only if its associated case expression (case_expr) is the first to match the switch expression (switch_expr).</p>
<b>Examples</b>	The general form of the switch statement is:
	<pre>switch switch_expr     case case_expr         statement, . . . , statement     case {case_expr1, case_expr2, case_expr3, . . . }         statement, . . . , statement     . . .     otherwise         statement, . . . , statement end</pre>
	See switch for more details.
<b>See Also</b>	switch      Switch among several cases based on expression

# cat

<b>Purpose</b>	Concatenate arrays
<b>Syntax</b>	$C = \text{cat}(\text{dim}, A, B)$ $C = \text{cat}(\text{dim}, A_1, A_2, A_3, A_4, \dots)$
<b>Description</b>	$C = \text{cat}(\text{dim}, A, B)$ concatenates the arrays $A$ and $B$ along $\text{dim}$ . $C = \text{cat}(\text{dim}, A_1, A_2, A_3, A_4, \dots)$ concatenates all the input arrays ( $A_1, A_2, A_3, A_4$ , and so on) along $\text{dim}$ . $\text{cat}(2, A, B)$ is the same as $[A, B]$ and $\text{cat}(1, A, B)$ is the same as $[A; B]$ .
<b>Remarks</b>	When used with comma separated list syntax, $\text{cat}(\text{dim}, C{:})$ or $\text{cat}(\text{dim}, C, \text{field})$ is a convenient way to concatenate a cell or structure array containing numeric matrices into a single matrix.
<b>Examples</b>	Given, $A = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix} \quad B = \begin{matrix} 5 & 6 \\ 7 & 8 \end{matrix}$ concatenating along different dimensions produces:  $C = \text{cat}(1, A, B)$ $C = \text{cat}(2, A, B)$ $C = \text{cat}(3, A, B)$
<b>The commands</b>	$A = \text{magic}(3); B = \text{pascal}(3);$ $C = \text{cat}(4, A, B);$ produce a 3-by-3-by-1-by-2 array.
<b>See Also</b>	<a href="#">[]</a> <a href="#">(Special characters) Build arrays</a> <a href="#">num2cell</a> <a href="#">Convert a numeric array into a cell array</a>

<b>Purpose</b>	Begin catch block
<b>Description</b>	<p>The general form of a try statement is:</p> <pre>try statement, ..., statement, catch statement, ..., statement end</pre> <p>Normally, only the statements between the try and catch are executed. However, if an error occurs while executing any of the statements, the error is captured into lasterr, and the statements between the catch and end are executed. If an error occurs within the catch statements, execution stops unless caught by another try...catch block. The error string produced by a failed try block can be obtained with lasterr.</p>
<b>See Also</b>	end, eval , eval in, try

---

<b>Purpose</b>	Change working directory
<b>Syntax</b>	<code>cd</code> <code>cd <i>directory</i></code> <code>cd ..</code>
<b>Description</b>	<code>cd</code> , by itself, prints out the current directory.  <code>cd <i>directory</i></code> sets the current directory to the one specified. On UNIX platforms, the character <code>~</code> is interpreted as the user's root directory.  <code>cd ..</code> changes to the directory above the current one.
<b>Examples</b>	UNIX: <code>cd /usr/local/matlab/toolbox/demos</code>  DOS: <code>cd C:\MATLAB\DEMONS</code>  VMS: <code>cd DISK1:[MATLAB.DEMOS]</code>  Macintosh: <code>cd Tool box: Demos</code>  To specify a Macintosh directory name that includes spaces, enclose the name in single quotation marks, as in ' <code>Tool box: New M-Files</code> '.
<b>See Also</b>	<code>dir</code> Directory listing <code>path</code> Control MATLAB's directory search path <code>what</code> Directory listing of M-files, MAT-files, and MEX-files

**Purpose** Convert complex diagonal form to real block diagonal form

**Syntax**  $[V, D] = \text{cdf2rdf}(V, D)$

**Description** If the eigensystem  $[V, D] = \text{eig}(X)$  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

$$\begin{matrix} X = \\ \begin{array}{ccc} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & -5 & 4 \end{array} \end{matrix}$$

has a pair of complex eigenvalues.

$$[V, D] = \text{eig}(X)$$

$$\begin{matrix} V = \\ \begin{array}{ccc} 1.0000 & 0.4002 - 0.0191i & 0.4002 + 0.0191i \\ 0 & 0.6479 & 0.6479 \\ 0 & 0 + 0.6479i & 0 - 0.6479i \end{array} \end{matrix}$$

$$\begin{matrix} D = \\ \begin{array}{ccc} 1.0000 & 0 & 0 \\ 0 & 4.0000 + 5.0000i & 0 \\ 0 & 0 & 4.0000 - 5.0000i \end{array} \end{matrix}$$

## cdf2rdf

---

Converting this to real block diagonal form produces

$[V, D] = \text{cdf2rdf}(V, D)$

$V =$

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

---

<b>Purpose</b>	Round toward infinity						
<b>Syntax</b>	<code>B = ceil(A)</code>						
<b>Description</b>	<code>B = ceil(A)</code> 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.						
<b>Examples</b>	<pre>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</pre>						
<b>See Also</b>	<table> <tr> <td><code>fix</code></td> <td>Round toward zero</td> </tr> <tr> <td><code>floor</code></td> <td>Round toward minus infinity</td> </tr> <tr> <td><code>round</code></td> <td>Round to nearest integer</td> </tr> </table>	<code>fix</code>	Round toward zero	<code>floor</code>	Round toward minus infinity	<code>round</code>	Round to nearest integer
<code>fix</code>	Round toward zero						
<code>floor</code>	Round toward minus infinity						
<code>round</code>	Round to nearest integer						

# cell

---

<b>Purpose</b>	Create cell array								
<b>Syntax</b>	<pre>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))</pre>								
<b>Description</b>	<p><code>c = cell(n)</code> creates an <math>n</math>-by-<math>n</math> cell array of empty matrices. An error message appears if <math>n</math> is not a scalar.</p> <p><code>c = cell(m, n)</code> or <code>c = cell([m, n])</code> creates an <math>m</math>-by-<math>n</math> cell array of empty matrices. Arguments <math>m</math> and <math>n</math> must be scalars.</p> <p><code>c = cell(m, n, p, . . . )</code> or <code>c = cell([m n p . . . ])</code> creates an <math>m</math>-by-<math>n</math>-by-<math>p</math>-... cell array of empty matrices. Arguments <math>m</math>, <math>n</math>, <math>p</math>,... must be scalars.</p> <p><code>c = cell(size(A))</code> creates a cell array the same size as <math>A</math> containing all empty matrices.</p>								
<b>Examples</b>	<pre>A = ones(2, 2)  A =     1     1     1     1  c = cell(size(A))  c =     []     []     []     []</pre>								
<b>See Also</b>	<table><tr><td><code>ones</code></td><td>Create an array of all ones</td></tr><tr><td><code>rand</code></td><td>Uniformly distributed random numbers and arrays</td></tr><tr><td><code>randn</code></td><td>Normally distributed random numbers and arrays</td></tr><tr><td><code>zeros</code></td><td>Create an array of all zeros</td></tr></table>	<code>ones</code>	Create an array of all ones	<code>rand</code>	Uniformly distributed random numbers and arrays	<code>randn</code>	Normally distributed random numbers and arrays	<code>zeros</code>	Create an array of all zeros
<code>ones</code>	Create an array of all ones								
<code>rand</code>	Uniformly distributed random numbers and arrays								
<code>randn</code>	Normally distributed random numbers and arrays								
<code>zeros</code>	Create an array of all zeros								

**Purpose** Cell array to structure array conversion

**Syntax** `s = cell2struct(c, fieldnames, dim)`

**Description** `s = cell2struct(c, fieldnames, 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 `fieldnames`. Argument `fieldnames` 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**

<code>fieldnames</code>	Field names of a structure
<code>struct2cell</code>	Structure to cell array conversion

# celldisp

---

<b>Purpose</b>	Display cell array contents.
<b>Syntax</b>	<code>celldisp(C)</code>
<b>Description</b>	<code>celldisp(c)</code> recursively displays the contents of a cell array.
<b>Example</b>	Use <code>celldisp</code> to display the contents of a 2-by-3 cell array:  <code>C = {[1 2] 'Tony' 3+4i; [1 2; 3 4] -5 'abc'};</code> <code>celldisp(C)</code>  <code>C{1, 1} =</code>  1        2  <code>C{2, 1} =</code>  1        2 3        4  <code>C{1, 2} =</code>  Tony  <code>C{2, 2} =</code>  -5  <code>C{1, 3} =</code>  3. 0000+ 4. 0000i  <code>C{2, 3} =</code>  abc
<b>See Also</b>	<code>cellplot</code> 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**

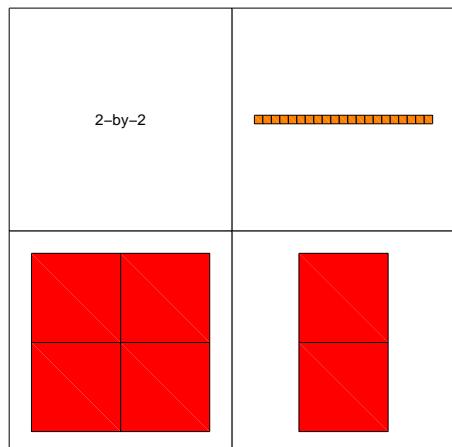
The `cellplot` function can display only two-dimensional cell arrays.

**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:



# cellstr

---

<b>Purpose</b>	Create cell array of strings from character array
<b>Syntax</b>	<code>c = cellstr(S)</code>
<b>Description</b>	<code>c = cellstr(S)</code> places each row of the character array <code>S</code> into separate cells of <code>c</code> . Use the <code>string</code> function to convert back to a string matrix.
<b>Examples</b>	Given the string matrix  <code>S =</code> <code>abc</code> <code>defg</code> <code>hi</code>  The command <code>c = cellstr(S)</code> returns the 3-by-1 cell array:  <code>c =</code> <code>'abc'</code> <code>'defg'</code> <code>'hi'</code>
<b>See Also</b>	<code>iscellstr</code> True for cell array of strings <code>strings</code> MATLAB string handling

<b>Purpose</b>	Conjugate Gradients Squared method
<b>Syntax</b>	$x = \text{cgs}(A, b)$ $\text{cgs}(A, b, tol)$ $\text{cgs}(A, b, tol, \text{maxit})$ $\text{cgs}(A, b, tol, \text{maxit}, M)$ $\text{cgs}(A, b, tol, \text{maxit}, M1, M2)$ $\text{cgs}(A, b, tol, \text{maxit}, M1, M2, x0)$ $x = \text{cgs}(A, b, tol, \text{maxit}, M1, M2, x0)$ $[x, flag] = \text{cgs}(A, b, tol, \text{maxit}, M1, M2, x0)$ $[x, flag, relres] = \text{cgs}(A, b, tol, \text{maxit}, M1, M2, x0)$ $[x, flag, relres, iter] = \text{cgs}(A, b, tol, \text{maxit}, M1, M2, x0)$ $[x, flag, relres, iter, resvec] = \text{cgs}(A, b, tol, \text{maxit}, M1, M2, x0)$
<b>Description</b>	<p><math>x = \text{cgs}(A, b)</math> attempts to solve the system of linear equations <math>A^*x = b</math> for <math>x</math>. The coefficient matrix <math>A</math> must be square and the right hand side (column) vector <math>b</math> must have length <math>n</math>, where <math>A</math> is <math>n</math>-by-<math>n</math>. <math>\text{cgs}</math> will start iterating from an initial estimate that by default is an all zero vector of length <math>n</math>. Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate <math>x</math> has relative residual <math>\text{norm}(b - A^*x) / \text{norm}(b)</math> less than or equal to the tolerance of the method. The default tolerance is <math>1e-6</math>. The default maximum number of iterations is the minimum of <math>n</math> and 20. No preconditioning is used.</p> <p><math>\text{cgs}(A, b, tol)</math> specifies the tolerance of the method, <math>tol</math>.</p> <p><math>\text{cgs}(A, b, tol, \text{maxit})</math> additionally specifies the maximum number of iterations, <math>\text{maxit}</math>.</p> <p><math>\text{cgs}(A, b, tol, \text{maxit}, M)</math> and <math>\text{cgs}(A, b, tol, \text{maxit}, M1, M2)</math> use left preconditioner <math>M</math> or <math>M = M1 * M2</math> and effectively solve the system <math>\text{inv}(M)^* A^* x = \text{inv}(M)^* b</math> for <math>x</math>. If <math>M1</math> or <math>M2</math> is given as the empty matrix (<math>[]</math>), it is considered to be the identity matrix, equivalent to no preconditioning at all. Since systems of equations of the form <math>M^*y = r</math> are solved using backslash within <math>\text{cgs}</math>, it is wise to factor preconditioners into their LU factors first. For example, replace <math>\text{cgs}(A, b, tol, \text{maxit}, M)</math> with:</p> $[M1, M2] = \text{lu}(M);$ $\text{cgs}(A, b, tol, \text{maxit}, M1, M2).$

`cgs(A, b, tol, maxit, M1, M2, x0)` specifies the initial estimate  $x_0$ . If  $x_0$  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  $\text{norm}(b - A^*x) / \text{norm}(b)$  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  $\text{norm}(b - A^*x) / \text{norm}(b)$ . 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  $0 \leq \text{iter} \leq \text{maxit}$ .

`[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.

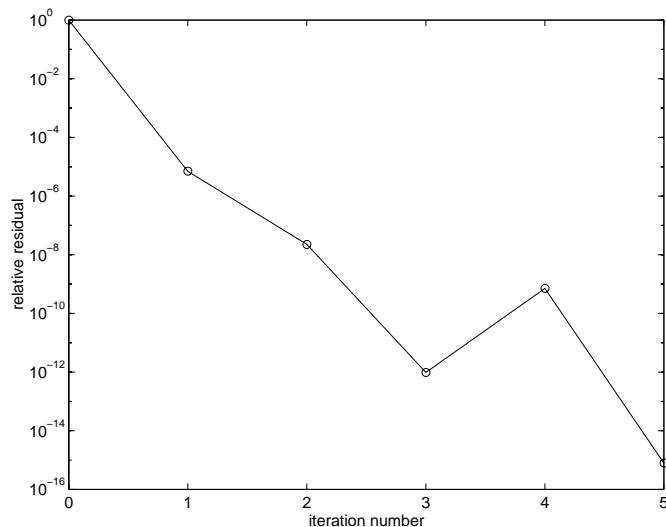
```
[L1, U1] = luinc(A, 1e-5)
[x1, flag1] = cgs(A, b, 1e-6, 20, L1, U1)
```

`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)
- `lui nc` 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.

<b>Purpose</b>	Create character array (string)
<b>Syntax</b>	<pre>S = char(X) S = char(C) S = char(t1, t2, t3, ...)</pre>
<b>Description</b>	<p>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.</p> <p>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.</p> <p>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, Ti, can itself be a character array. This allows the creation of arbitrarily large character arrays. Empty strings are significant.</p>
<b>Remarks</b>	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)).
<b>Examples</b>	To print a 3-by-32 display of the printable ASCII characters:
	<pre>asci i = char(reshape(32:127, 32, 3)') asci i = ! " # \$ % &amp; ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; &lt; = &gt; ? @ 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 {   } ~</pre>

# char

---

## 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
```

# chol

---

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.

## See Also

cholinc

Sparse Incomplete Cholesky and Cholesky-Infinity factorizations

cholupdate

Rank 1 update to Cholesky factorization

**Purpose** Sparse Incomplete Cholesky and Cholesky-Infinity factorizations

**Syntax**

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

**Description**

cholinc produces two different kinds of incomplete Cholesky factorizations: the drop tolerance and the 0 level of fill-in factorizations. These factors may be useful as preconditioners for a symmetric positive definite system of linear equations being solved by an iterative method such as pcg (Preconditioned Conjugate Gradients). cholinc works only for sparse matrices.

`R = cholinc(X, droptol)` performs the incomplete Cholesky factorization of X, with drop tolerance droptol .

`R = cholinc(X, options)` allows additional options to the incomplete Cholesky factorization. options is a structure with up to three fields:

droptol	drop tolerance of the incomplete factorization
mi chol	modified incomplete Cholesky
rdi ag	replace zeros on the diagonal of R

Only the fields of interest need to be set.

droptol is a non-negative scalar used as the drop tolerance for the incomplete Cholesky factorization. This factorization is computed by performing the incomplete LU factorization with the pivot threshold option set to 0 (which forces diagonal pivoting) and then scaling the rows of the incomplete upper triangular factor, U, by the square root of the diagonal entries in that column. Since the nonzero entries  $U(i, j)$  are bounded below by  $\text{droptol} * \text{norm}(X(:, j))$  (see luinc), the nonzero entries  $R(i, j)$  are bounded below by the local drop tolerance  $\text{droptol} * \text{norm}(X(:, j)) / R(i, i)$ .

Setting `droptol = 0` produces the complete Cholesky factorization, which is the default.

`mi chol` stands for modified incomplete Cholesky factorization. Its value is either 0 (unmodified, the default) or 1 (modified). This performs the modified incomplete LU factorization of `X` and scales the returned upper triangular factor as described above.

`rdi ag` is either 0 or 1. If it is 1, any zero diagonal entries of the upper triangular factor `R` are replaced by the square root of the local drop tolerance in an attempt to avoid a singular factor. The default is 0.

`R = cholinc(X, '0')` produces the incomplete Cholesky factor of a real symmetric positive definite sparse matrix with 0 level of fill-in (i.e. no fill-in). The upper triangular `R` has the same sparsity pattern as `triu(X)`, although `R` may be zero in some positions where `X` is nonzero due to cancellation. The lower triangle of `X` is assumed to be the transpose of the upper. Note that the positive definiteness of `X` does not guarantee the existence of a factor with the required sparsity. An error message results if the factorization is not possible. If the factorization is successful, `R' *R` agrees with `X` over its sparsity pattern.

`[R, p] = cholinc(X, '0')` with two output arguments, never produces an error message. If `R` exists, `p` is 0. But, if the incomplete factor does not exist, then `p` is a positive integer and `R` is an upper triangular matrix of size `q`-by-`n` where `q = p-1` so that the sparsity pattern of `R` is that of the `q`-by-`n` upper triangle of `X`. `R' *R` agrees with `X` over the sparsity pattern of its first `q` rows and first `q` columns.

`R = cholinc(X, 'inf')` produces the Cholesky-Infinity factorization. This factorization is based on the Cholesky factorization, and additionally handles real positive semi-definite matrices. It may be useful for finding a solution to systems which arise in interior-point methods. When a zero pivot is encountered in the ordinary Cholesky factorization, the diagonal of the Cholesky-Infinity factor is set to `Inf` and the rest of that row is set to 0. This is designed to force a 0 in the corresponding entry of the solution vector in the associated system of linear equations. A negative pivot still results in an error.

`[R, p] = cholinc(X, 'inf')` with two output arguments, never produces an error message. If `X` is positive semi-definite, then `p` is 0. Otherwise, `p` is a positive integer and `R` is an upper triangular matrix of size `(p-1)`-by-`n`.

**Remarks**

1. The 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 rdi ag 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.
2. The Cholesky-Infinity factorization is meant to be used within interior-point methods. Its use otherwise cannot be recommended.

**Examples**

Example 1.

Start with a symmetric positive definite matrix, S.

```
S = del sq(numgrid('C', 15));
```

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

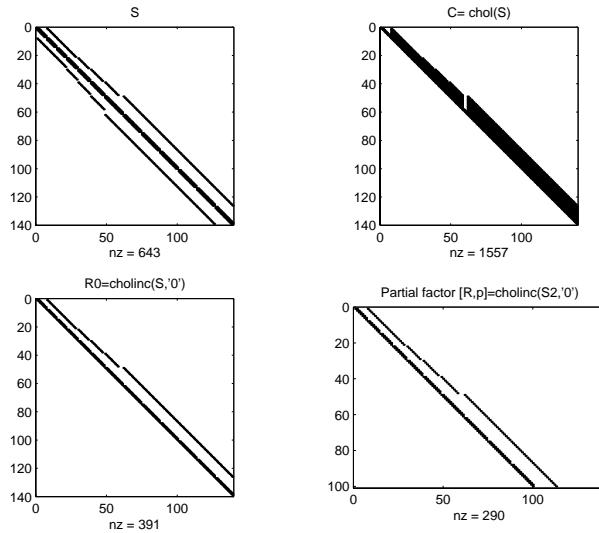
Compute the Cholesky factorization and the incomplete Cholesky factorization of level 0 to compare the fill-in. Make S singular by zeroing out a diagonal entry and compute the (partial) incomplete Cholesky factorization of level 0.

```
C = chol(S);
R0 = cholinc(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;
```

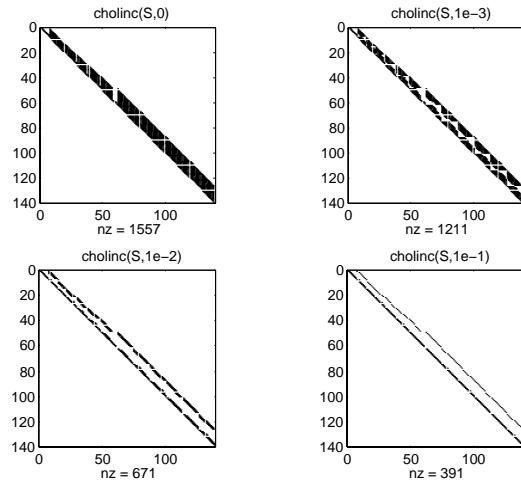
D1 has elements of the order of eps, showing that  $R0' * R0$  agrees with S over its sparsity pattern. D2 has elements of the order of eps over its first 100 rows and first 100 columns,  $D2(1: 100, :)$  and  $D2(:, 1: 100)$ .



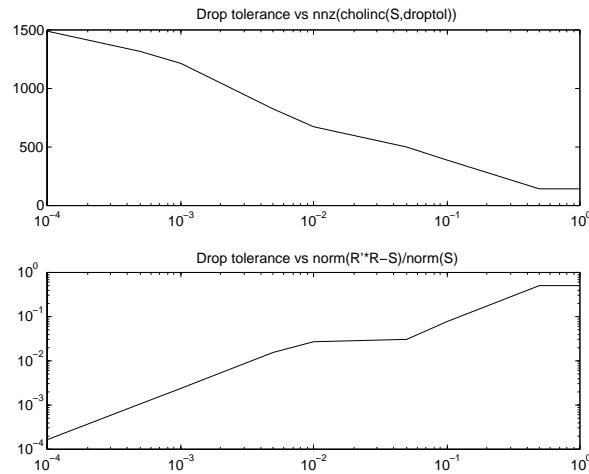
## Example 2.

The first subplot below shows that  $\text{cholinc}(S, 0)$ , 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  $\text{norm}(R^T R - S, 1) / \text{norm}(S, 1)$  in the next figure.



### Example 3.

```
S = sparse([1 0 3 0; 0 25 0 30; 3 0 9 0; 0 30 0 661])
```

This symmetric sparse matrix is singular. The Cholesky factorization fails at the zero pivot in the third row, but cholinc succeeds in computing all rows of the Cholesky-Infinity factorization.

```
[R, p] = chol(S);
full(R)
ans =
    1     0     3     0
    0     5     0     6

Rinf = cholinc(S, 'inf');
full(Rinf)
ans =
    1     0     3     0
    0     5     0     6
    0     0     Inf    0
    0     0     0     25
```

## Limitations

cholinc works on square sparse matrices only. For cholinc(X, '0') and cholinc(X, 'inf'), 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 uppertriangular 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 m chol option.

R = 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.

R = cholinc(X, 'inf') is based on the algorithm in Zhang.

**See Also**

[chol](#) Cholesky factorization  
[lui nc](#) 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.
- Zhang, Yin, *Solving Large-Scale Linear Programs by Interior-Point Methods Under the MATLAB Environment*, Department of Mathematics and Statistics, University of Maryland Baltimore County, Technical Report TR96-01

# cholupdate

---

<b>Purpose</b>	Rank 1 update to Cholesky factorization
<b>Syntax</b>	<pre>R1 = cholupdate(R, x) R1 = cholupdate(R, x, '+') R1 = cholupdate(R, x, '-') [R1, p] = cholupdate(R, x, '-')</pre>
<b>Description</b>	<p><code>R1 = cholupdate(R, x)</code> where <math>R = \text{chol}(A)</math> is the original Cholesky factorization of <math>A</math>, returns the upper triangular Cholesky factor of <math>A + x^*x'</math>, where <math>x</math> is a column vector of appropriate length. <code>cholupdate</code> uses only the diagonal and upper triangle of <math>R</math>. The lower triangle of <math>R</math> is ignored.</p> <p><code>R1 = cholupdate(R, x, '+')</code> is the same as <code>R1 = cholupdate(R, x)</code>.</p> <p><code>R1 = cholupdate(R, x, '-')</code> returns the Cholesky factor of <math>A - x^*x'</math>. An error message reports when <math>R</math> is not a valid Cholesky factor or when the downdated matrix is not positive definite and so does not have a Cholesky factorization.</p> <p><code>[R1, p] = cholupdate(R, x, '-')</code> will not return an error message. If <math>p</math> is 0, <math>R1</math> is the Cholesky factor of <math>A - x^*x'</math>. If <math>p</math> is greater than 0, <math>R1</math> is the Cholesky factor of the original <math>A</math>. If <math>p</math> is 1, <code>cholupdate</code> failed because the downdated matrix is not positive definite. If <math>p</math> is 2, <code>cholupdate</code> failed because the upper triangle of <math>R</math> was not a valid Cholesky factor.</p>
<b>Remarks</b>	<code>cholupdate</code> works only for full matrices.
<b>Example</b>	<pre>A = pascal(4) A =</pre> $\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 6 & 10 \\ 1 & 4 & 10 & 20 \end{matrix}$

```
R = chol (A)
R =
```

```
1     1     1     1
0     1     2     3
0     0     1     3
0     0     0     1
```

```
x = [0 0 0 1]';
```

This is called a rank one update to A since  $\text{rank}(x^*x')$  is 1:

```
A + x*x'
ans =
```

```
1     1     1     1
1     2     3     4
1     3     6    10
1     4    10    21
```

Instead of computing the Cholesky factor with  $R1 = \text{chol}(A + x^*x')$ , we can use `cholupdate`:

```
R1 = cholupdate(R, x)
R1 =
```

```
1.0000    1.0000    1.0000    1.0000
0         1.0000    2.0000    3.0000
0         0         1.0000    3.0000
0         0         0         1.4142
```

Next destroy the positive definiteness (and actually make the matrix singular) by subtracting 1 from the last element of A. The downdated matrix is:

# cholupdate

```
A = x*x'  
ans =  
  
1 1 1 1  
1 2 3 4  
1 3 6 10  
1 4 10 19
```

Compare chol with chol update:

```
R1 = chol(A-x*x')  
??? Error using ==> chol  
Matrix must be positive definite.  
  
R1 = cholupdate(R, x, '-')
```

??? Error using ==> cholupdate  
Downdated matrix must be positive definite.

However, subtracting 0.5 from the last element of A produces a positive definite matrix, and we can use chol update to compute its Cholesky factor:

```
x = [0 0 0 1/sqrt(2)]';  
R1 = cholupdate(R, x, '-')  
R1 =  
1.0000 1.0000 1.0000 1.0000  
0 1.0000 2.0000 3.0000  
0 0 1.0000 3.0000  
0 0 0 0.7071
```

## Algorithm

chol update uses the algorithms from the LINPACK subroutines ZCHUD and ZCHDD. chol update is useful since computing the new Cholesky factor from scratch is an  $O(N^3)$  algorithm, while simply updating the existing factor in this way is an  $O(N^2)$  algorithm.

## References

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

## See Also

[chol](#)      Cholesky factorization  
[qrupdate](#)      Rank 1 update to QR factorization

**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`.

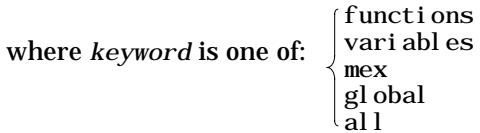
---

`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**

inferioro	Inferior class relationship
isa	Detect an object of a given class
superioro	Superior class relationship

# clear

<b>Purpose</b>	Remove items from memory
<b>Syntax</b>	<code>clear</code> <code>clear name</code> <code>clear name1 name2 name3...</code> <code>clear global name</code>  <code>clear keyword</code>
	where <i>keyword</i> is one of:  functions variables mex global all
<b>Description</b>	<code>clear</code> , by itself, clears all variables from the workspace.  <code>clear name</code> removes just the M-file or MEX-file function or variable <i>name</i> from the workspace. A MATLABPATH relative partial pathname is permitted. If <i>name</i> is global, it is removed from the current workspace, but left accessible to any functions declaring it global. If <i>name</i> has been locked by <code>lock</code> , it will remain in memory.  <code>clear name1 name2 name3</code> removes <i>name1</i> , <i>name2</i> , and <i>name3</i> from the workspace.  <code>clear global name</code> removes the global variable <i>name</i> .  <code>clear keyword</code> clears the indicated items:
	<code>clear functions</code> Clears all the currently compiled M-functions from memory. <code>clear variables</code> Clears all variables from the workspace. <code>clear mex</code> Clears all MEX-files from memory. <code>clear global</code> Clears all global variables. <code>clear all</code> Removes all variables, functions, and MEX-files from memory, leaving the workspace empty.
<b>Remarks</b>	You can use wildcards (*) to remove items selectively. For instance, <code>clear my*</code> removes any variables whose names begin with the string "my." The function form of the syntax, <code>clear (' name')</code> , is also permitted.

**Limitations**

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

**See Also**

`mllock`      Prevent M-file clearing  
`munlock`      Prevent M-file clearing  
`pack`              Consolidate workspace memory

# clock

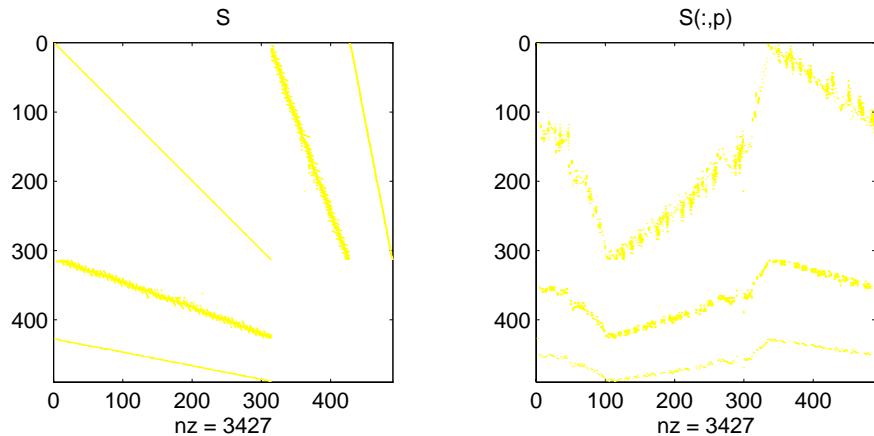
---

<b>Purpose</b>	Current time as a date vector												
<b>Syntax</b>	<code>c = clock</code>												
<b>Description</b>	<code>c = clock</code> returns a 6-element date vector containing the current date and time in decimal form:  <code>c = [year month day hour minute seconds]</code>												
<b>See Also</b>	<table><tr><td><code>cputime</code></td><td>CPU time in seconds</td></tr><tr><td><code>datenum</code></td><td>Serial date number</td></tr><tr><td><code>datevec</code></td><td>Date components</td></tr><tr><td><code>etime</code></td><td>Elapsed time</td></tr><tr><td><code>tic</code></td><td>Start a stopwatch timer</td></tr><tr><td><code>toc</code></td><td>Read the stopwatch timer</td></tr></table>	<code>cputime</code>	CPU time in seconds	<code>datenum</code>	Serial date number	<code>datevec</code>	Date components	<code>etime</code>	Elapsed time	<code>tic</code>	Start a stopwatch timer	<code>toc</code>	Read the stopwatch timer
<code>cputime</code>	CPU time in seconds												
<code>datenum</code>	Serial date number												
<code>datevec</code>	Date components												
<code>etime</code>	Elapsed time												
<code>tic</code>	Start a stopwatch timer												
<code>toc</code>	Read the stopwatch timer												

<b>Purpose</b>	Sparse column minimum degree permutation
<b>Syntax</b>	<code>p = colmmd(S)</code>
<b>Description</b>	<p><code>p = colmmd(S)</code> returns the column minimum degree permutation vector for the sparse matrix <code>S</code>. For a nonsymmetric matrix <code>S</code>, this is a column permutation <code>p</code> such that <code>S(:, p)</code> tends to have sparser LU factors than <code>S</code>.</p> <p>The <code>colmmd</code> permutation is automatically used by \ and / for the solution of nonsymmetric and symmetric indefinite sparse linear systems.</p> <p>Use <code>spparms</code> to change some options and parameters associated with heuristics in the algorithm.</p>
<b>Algorithm</b>	<p>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 <math>A' * A</math>, but does not actually form <math>A' * A</math>.</p> <p>Each stage of the algorithm chooses a vertex in the graph of <math>A' * A</math> of lowest degree (that is, a column of <math>A</math> 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 <code>S</code> is of size <math>m</math>-by-<math>n</math>, the columns are all eliminated and the permutation is complete after <math>n</math> stages. To speed up the process, several heuristics are used to carry out multiple stages simultaneously.</p>
<b>Examples</b>	<p>The Harwell-Boeing collection of sparse matrices includes a test matrix <code>ABB313</code>. 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 <code>spaugment</code>), 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 <code>colmmd</code> ordering scrambles this</p>

structure. (Note that this example requires the Harwell-Boeing collection of software.)

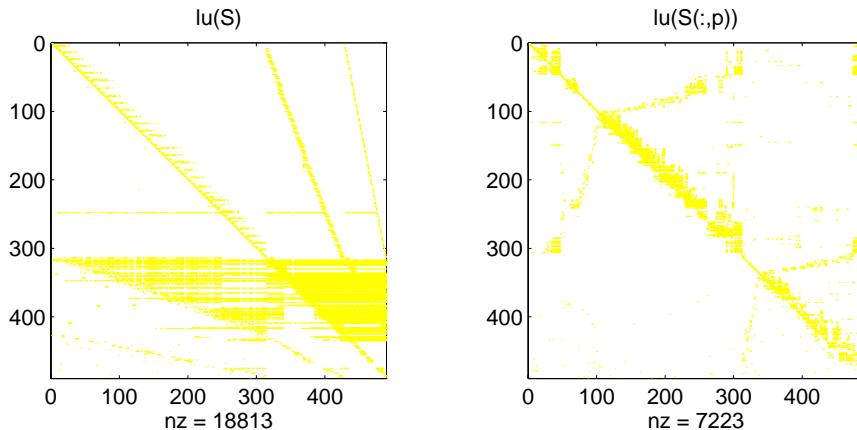
```
load('abb313.mat')
S = spaugment(A);
p = colmmd(S);
spy(S)
spy(S(:, p))
```



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

<b>Purpose</b>	Sparse column permutation based on nonzero count								
<b>Syntax</b>	$j = \text{colperm}(S)$								
<b>Description</b>	$j = \text{colperm}(S)$ 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 $\text{lu}(S(:, j))$ .  If $S$ is symmetric, then $j = \text{colperm}(S)$ 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 $\text{chol}(S(j, j))$ .								
<b>Algorithm</b>	The algorithm involves a sort on the counts of nonzeros in each column.								
<b>Examples</b>	<p>The n-by-n <i>arrowhead</i> matrix</p> <pre>A = [ ones(1, n); ones(n-1, 1) speye(n-1, n-1) ]</pre> <p>has a full first row and column. Its LU factorization, <math>\text{lu}(A)</math>, is almost completely full. The statement</p> <pre>j = colperm(A)</pre> <p>returns <math>j = [2:n 1]</math>. So <math>A(j, j)</math> sends the full row and column to the bottom and the rear, and <math>\text{lu}(A(j, j))</math> has the same nonzero structure as <math>A</math> itself.</p> <p>On the other hand, the Bucky ball example, <math>B = \text{bucky}</math>,</p> <p>has exactly three nonzero elements in each row and column, so <math>j = \text{colperm}(B)</math> is the identity permutation and is no help at all for reducing fill-in with subsequent factorizations.</p>								
<b>See Also</b>	<table><tr><td><code>chol</code></td><td>Cholesky factorization</td></tr><tr><td><code>colmmd</code></td><td>Sparse minimum degree ordering</td></tr><tr><td><code>lu</code></td><td>LU matrix factorization</td></tr><tr><td><code>symrcm</code></td><td>Sparse reverse Cuthill-McKee ordering</td></tr></table>	<code>chol</code>	Cholesky factorization	<code>colmmd</code>	Sparse minimum degree ordering	<code>lu</code>	LU matrix factorization	<code>symrcm</code>	Sparse reverse Cuthill-McKee ordering
<code>chol</code>	Cholesky factorization								
<code>colmmd</code>	Sparse minimum degree ordering								
<code>lu</code>	LU matrix factorization								
<code>symrcm</code>	Sparse reverse Cuthill-McKee ordering								

<b>Purpose</b>	Companion matrix
<b>Syntax</b>	<code>A = compan(u)</code>
<b>Description</b>	<code>A = compan(u)</code> returns the corresponding companion matrix whose first row is $-u(2:n)/u(1)$ , where <code>u</code> is a vector of polynomial coefficients. The eigenvalues of <code>compan(u)</code> are the roots of the polynomial.
<b>Examples</b>	The polynomial $(x-1)(x-2)(x+3) = x^3 - 7x + 6$ has a companion matrix given by <pre>u = [1 0 -7 6] A = compan(u) A =     0    7   -6     1    0    0     0    1    0</pre> <p>The eigenvalues are the polynomial roots:</p> <pre>eig(compan(u)) ans = -3.0000 2.0000 1.0000</pre> <p>This is also <code>roots(u)</code>.</p>

<b>See Also</b>	<code>eig</code>	Eigenvalues and eigenvectors
	<code>poly</code>	Polynomial with specified roots
	<code>polyval</code>	Polynomial evaluation
	<code>roots</code>	Polynomial roots

# computer

<b>Purpose</b>	Identify the computer on which MATLAB is running
<b>Syntax</b>	<code>str = computer</code> <code>[str, maxsize] = computer</code>
<b>Description</b>	<code>str = computer</code> returns a string with the computer type on which MATLAB is running.  <code>[str, maxsize] = computer</code> returns the integer <code>maxsize</code> , 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)
SGI 64	Silicon Graphics (R8000)
IBM_RS	IBM RS6000 workstation

**See Also**

[i si eee](#), [i suni x](#), [i svms](#)

# cond

<b>Purpose</b>	Condition number with respect to inversion
<b>Syntax</b>	$c = \text{cond}(X)$ $c = \text{cond}(X, p)$
<b>Description</b>	The <i>condition number</i> 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 $\text{cond}(X)$ and $\text{cond}(X, p)$ near 1 indicate a well-conditioned matrix.
	$c = \text{cond}(X)$ returns the 2-norm condition number, the ratio of the largest singular value of $X$ to the smallest.
	$c = \text{cond}(X, p)$ returns the matrix condition number in $p$ -norm: $\text{norm}(X, p) * \text{norm}(\text{inv}(X), p)$
If $p$ is...	Then $\text{cond}(X, p)$ returns the...
1	1-norm condition number
2	2-norm condition number
'fro'	Frobenius norm condition number
inf	Infinity norm condition number
<b>Algorithm</b>	The algorithm for $\text{cond}$ (when $p = 2$ ) uses the singular value decomposition, $\text{svd}$ .
<b>See Also</b>	<a href="#">condeig</a> Condition number with respect to eigenvalues <a href="#">condest</a> 1-norm matrix condition number estimate <a href="#">norm</a> Vector and matrix norms <a href="#">rank</a> Rank of a matrix <a href="#">svd</a> Singular value decomposition
<b>References</b>	[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, <i>LINPACK Users' Guide</i> , SIAM, Philadelphia, 1979.

**Purpose** Condition number with respect to eigenvalues

**Syntax**

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

**Description**  $c = \text{condeig}(A)$  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.

$[V, D, s] = \text{condeig}(A)$  is equivalent to:  $[V, D] = \text{eig}(A); s = \text{condeig}(A); .$

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

**See Also**

<code>balance</code>	Improve accuracy of computed eigenvalues
<code>cond</code>	Condition number with respect to inversion
<code>eig</code>	Eigenvalues and eigenvectors

## condest

---

<b>Purpose</b>	1-norm matrix condition number estimate
<b>Syntax</b>	$c = \text{condest}(A)$ $[c, v] = \text{condest}(A)$
<b>Description</b>	$c = \text{condest}(A)$ 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.  $[c, v] = \text{condest}(A)$ 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.
<b>See Also</b>	<a href="#">cond</a> Condition number with respect to inversion <a href="#">normest</a> 2-norm estimate
<b>Reference</b>	[1] Higham, N.J. "Fortran Codes for Estimating the One-Norm of a Real or Complex Matrix, with Applications to Condition Estimation." <i>ACM Trans. Math. Soft.</i> , 14, 1988, pp. 381-396.

---

<b>Purpose</b>	Complex conjugate						
<b>Syntax</b>	$ZC = conj(Z)$						
<b>Description</b>	$ZC = conj(Z)$ returns the complex conjugate of the elements of $Z$ .						
<b>Algorithm</b>	If $Z$ is a complex array: $conj(Z) = real(Z) - i * imag(Z)$						
<b>See Also</b>	<table><tr><td><math>i, j</math></td><td>Imaginary unit (<math>\sqrt{-1}</math>)</td></tr><tr><td><math>imag</math></td><td>Imaginary part of a complex number</td></tr><tr><td><math>real</math></td><td>Real part of a complex number</td></tr></table>	$i, j$	Imaginary unit ( $\sqrt{-1}$ )	$imag$	Imaginary part of a complex number	$real$	Real part of a complex number
$i, j$	Imaginary unit ( $\sqrt{-1}$ )						
$imag$	Imaginary part of a complex number						
$real$	Real part of a complex number						

## conv

<b>Purpose</b>	Convolution and polynomial multiplication				
<b>Syntax</b>	$w = \text{conv}(u, v)$				
<b>Description</b>	$w = \text{conv}(u, v)$ convolves vectors $u$ and $v$ . Algebraically, convolution is the same operation as multiplying the polynomials whose coefficients are the elements of $u$ and $v$ .				
<b>Definition</b>	Let $m = \text{length}(u)$ and $n = \text{length}(v)$ . Then $w$ is the vector of length $m+n-1$ whose $k$ th 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				
	$\begin{aligned} w(1) &= u(1)*v(1) \\ w(2) &= u(1)*v(2)+u(2)*v(1) \\ w(3) &= u(1)^*v(3)+u(2)*v(2)+u(3)*v(1) \\ &\dots \\ w(n) &= u(1)*v(n)+u(2)*v(n-1)+\dots+u(n)*v(1) \\ &\dots \\ w(2*n-1) &= u(n)*v(n) \end{aligned}$				
<b>Algorithm</b>	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				
	$X = \text{fft}([x \text{ zeros}(1, \text{length}(y)-1)])$ and $Y = \text{fft}([y \text{ zeros}(1, \text{length}(x)-1)])$ then $\text{conv}(x, y) = \text{ifft}(X.*Y)$				
<b>See Also</b>	convmtx, xconv2, xcorr, in the Signal Processing Toolbox, and:  <table><tr><td><code>deconv</code></td><td>Deconvolution and polynomial division</td></tr><tr><td><code>filter</code></td><td>Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter</td></tr></table>	<code>deconv</code>	Deconvolution and polynomial division	<code>filter</code>	Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter
<code>deconv</code>	Deconvolution and polynomial division				
<code>filter</code>	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)
```

## conv2

---

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)
```

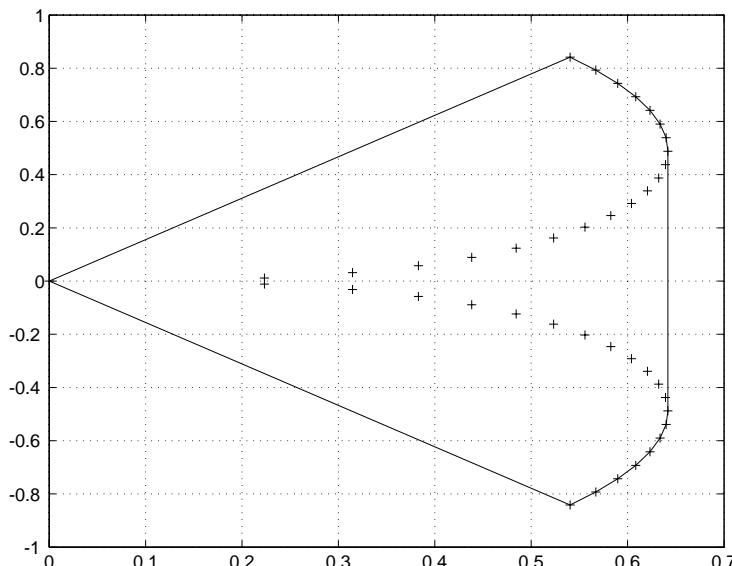
**Description**

`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`  
`polyarea`  
`voronoi`

Delauney triangulation  
Area of polygon  
Voronoi diagram

## convn

---

<b>Purpose</b>	N-dimensional convolution
<b>Syntax</b>	$C = \text{convn}(A, B)$ $C = \text{convn}(A, B, 'shape')$
<b>Description</b>	$C = \text{convn}(A, B)$ computes the N-dimensional convolution of the arrays A and B. The size of the result is $\text{size}(A) + \text{size}(B) - 1$ .  $C = \text{convn}(A, B, 'shape')$ returns a subsection of the N-dimensional convolution, as specified by the <i>shape</i> parameter: <ul style="list-style-type: none"><li>• 'full' returns the full N-dimensional convolution (default).</li><li>• 'same' returns the central part of the result that is the same size as A.</li><li>• '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 <math>\max(\text{size}(A) - \text{size}(B) + 1, 0)</math>.</li></ul>
<b>See Also</b>	<a href="#">conv</a> Convolution and polynomial multiplication <a href="#">conv2</a> Two-dimensional convolution

---

<b>Purpose</b>	Copy file
<b>Syntax</b>	<pre>copyfile(source, dest) copyfile(source, dest, 'writable') status = copyfile(...) [status, msg] = copyfile(...)</pre>
<b>Description</b>	<p><code>copyfile(source, dest)</code> copies the file <i>source</i> to the new file <i>dest</i>. <i>source</i> and <i>dest</i> may be absolute pathnames or pathnames relative to the current directory.</p> <p><code>copyfile(source, dest, 'writable')</code> checks that <i>dest</i> is writable.</p> <p><code>status = copyfile(...)</code> returns 1 if the file is copied successfully and 0 otherwise.</p> <p><code>[status, msg] = copyfile(...)</code> returns a non-empty error message string when an error occurs.</p>
<b>See Also</b>	<p><code>delete</code>                      Delete files and graphics objects <code>mkdir</code>                      Make directory</p>

## corrcoef

---

<b>Purpose</b>	Correlation coefficients						
<b>Syntax</b>	$S = \text{corrcoef}(X)$ $S = \text{corrcoef}(x, y)$						
<b>Description</b>	$S = \text{corrcoef}(X)$ returns a matrix of correlation coefficients calculated from an input matrix whose rows are observations and whose columns are variables. The matrix $S = \text{corrcoef}(X)$ is related to the covariance matrix $C = \text{cov}(X)$ by						
	$S(i, j) = \frac{C(i, j)}{\sqrt{C(i, i)C(j, j)}}$						
	$\text{corrcoef}(X)$ is the zeroth lag of the covariance function, that is, the zeroth lag of $\text{xcov}(x, 'coeff')$ packed into a square array.						
	$S = \text{corrcoef}(x, y)$ where $x$ and $y$ are column vectors is the same as $\text{corrcoef}([x \ y])$ .						
<b>See Also</b>	<a href="#">xcorr</a> , <a href="#">xcov</a> in the Signal Processing Toolbox, and:  <table><tr><td><a href="#">cov</a></td><td>Covariance matrix</td></tr><tr><td><a href="#">mean</a></td><td>Average or mean value of arrays</td></tr><tr><td><a href="#">std</a></td><td>Standard deviation</td></tr></table>	<a href="#">cov</a>	Covariance matrix	<a href="#">mean</a>	Average or mean value of arrays	<a href="#">std</a>	Standard deviation
<a href="#">cov</a>	Covariance matrix						
<a href="#">mean</a>	Average or mean value of arrays						
<a href="#">std</a>	Standard deviation						

**Purpose** Cosine and hyperbolic cosine

**Syntax**

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

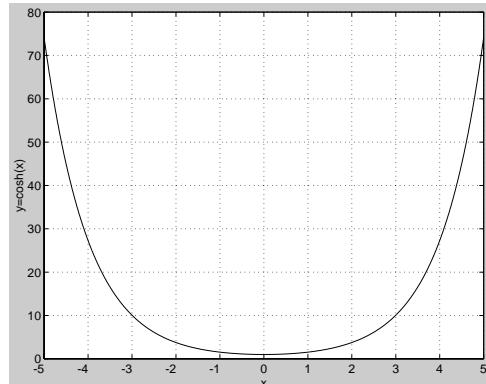
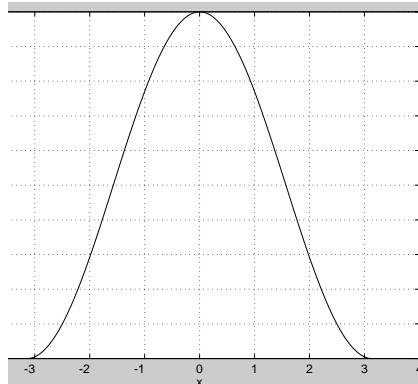
**Description** The cos and cosh functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

`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  $-\pi \leq x \leq \pi$ , and the hyperbolic cosine function over the domain  $-5 \leq x \leq 5$ .

```
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,  $\text{eps}$ , because  $\pi$  is only a floating-point approximation to the exact value of  $\pi$ .

**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

## **cot, coth**

<b>Purpose</b>	Cotangent and hyperbolic cotangent
<b>Syntax</b>	$Y = \cot(X)$ $Y = \coth(X)$
<b>Description</b>	The <code>cot</code> and <code>coth</code> 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 $X$ .
<b>Examples</b>	Graph the cotangent and hyperbolic cotangent over the domains $-\pi < x < 0$ and $0 < x < \pi$ .
	<pre>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))</pre>
<b>Algorithm</b>	$\cot(z) = \frac{1}{\tan(z)}$ $\coth(z) = \frac{1}{\tanh(z)}$
<b>See Also</b>	<a href="#">acot</a> , <a href="#">acoth</a> Inverse cotangent and inverse hyperbolic cotangent

<b>Purpose</b>	Covariance matrix
<b>Syntax</b>	$C = \text{cov}(X)$ $C = \text{cov}(x, y)$
<b>Description</b>	$C = \text{cov}(x)$ 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, $\text{cov}(x)$ is the covariance matrix. $\text{diag}(\text{cov}(x))$ is a vector of variances for each column, and $\text{sqrt}(\text{diag}(\text{cov}(x)))$ is a vector of standard deviations.  $C = \text{cov}(x, y)$ , where $x$ and $y$ are column vectors of equal length, is equivalent to $\text{cov}([x \ y])$ .
<b>Remarks</b>	$\text{cov}$ removes the mean from each column before calculating the result. The <i>covariance</i> function is defined as
	where $E$ is the mathematical expectation and $\mu_i = E x_i$ .
<b>Examples</b>	Consider $A = [-1 \ 1 \ 2 ; -2 \ 3 \ 1 ; 4 \ 0 \ 3]$ . To obtain a vector of variances for each column of $A$ :
	$v = \text{diag}(\text{cov}(A))'$ $v =$ 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 $j$ .
<b>See Also</b>	$\text{xcorr}$ , $\text{xcov}$ in the Signal Processing Toolbox, and: $\text{corrcoef}$ Correlation coefficients $\text{mean}$ Average or mean value of arrays $\text{std}$ Standard deviation

# cplxpair

<b>Purpose</b>	Sort complex numbers into complex conjugate pairs
<b>Syntax</b>	$B = \text{cplxpai r}(A)$ $B = \text{cplxpai r}(A, tol)$ $B = \text{cplxpai r}(A, [], dim)$ $B = \text{cplxpai r}(A, tol, dim)$
<b>Description</b>	$B = \text{cplxpai r}(A)$ 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 $100*\text{eps}$ relative to $\text{abs}(A(i))$ determines which numbers are real and which elements are paired complex conjugates.  If $A$ is a vector, $\text{cplxpai r}(A)$ returns $A$ with complex conjugate pairs grouped together.  If $A$ is a matrix, $\text{cplxpai r}(A)$ returns $A$ with its columns sorted and complex conjugates paired.  If $A$ is a multidimensional array, $\text{cplxpai r}(A)$ treats the values along the first non-singleton dimension as vectors, returning an array of sorted elements.  $B = \text{cplxpai r}(A, tol)$ overrides the default tolerance.  $B = \text{cplxpai r}(A, [], dim)$ sorts $A$ along the dimension specified by scalar $dim$ .  $B = \text{cplxpai r}(A, tol, dim)$ sorts $A$ along the specified dimension and overrides the default tolerance.
<b>Diagnostics</b>	If there are an odd number of complex numbers, or if the complex numbers cannot be grouped into complex conjugate pairs within the tolerance, $\text{cplxpai r}$ generates the error message:  <code>Complex numbers can't be paired.</code>

**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
```

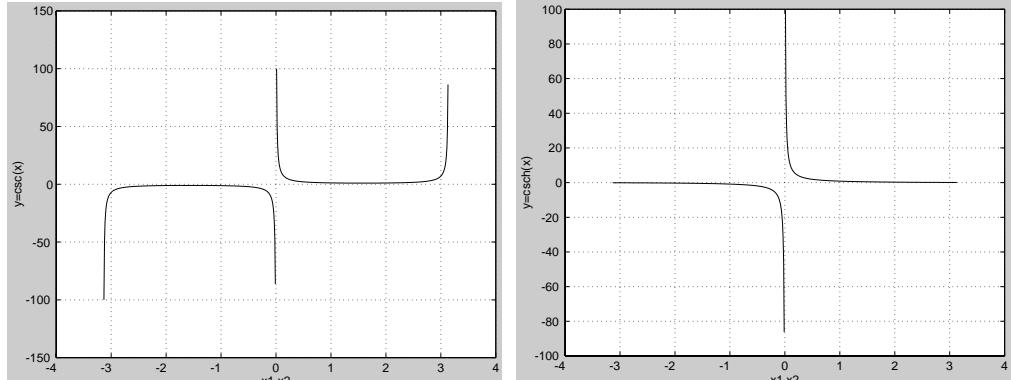
returns the CPU time used to run `surf(peaks(40))`.

**See Also** `clock` Current time as a date vector  
`etime` Elapsed time  
`tic, toc` Stopwatch timer

## CROSS

---

<b>Purpose</b>	Vector cross product
<b>Syntax</b>	$W = \text{cross}(U, V)$ $W = \text{cross}(U, V, \text{dim})$
<b>Description</b>	$W = \text{cross}(U, V)$ returns the cross product of the vectors U and V. That is, $W = U \times V$ . 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, $\text{cross}(U, V)$ treats the first size 3 dimension of U and V as vectors, returning pages whose columns are cross products.  $W = \text{cross}(U, V, \text{dim})$ where U and V are multidimensional arrays, returns the cross product of U and V in dimension $\text{dim}$ . U and V must have the same size, and both $\text{size}(U, \text{dim})$ and $\text{size}(V, \text{dim})$ must be 3.
<b>Remarks</b>	To perform a dot (scalar) product of two vectors of the same size, use: $c = \text{sum}(a.*b)$ or, if a and b are row vectors, $c = a.'*b$ .
<b>Examples</b>	The cross and dot products of two vectors are calculated as shown:  $a = [1 2 3]; b = [4 5 6];$ $c = \text{cross}(a, b)$  $c =$  $\begin{array}{ccc} -3 & 6 & -3 \end{array}$  $d = \text{sum}(a.*b)$  $d =$  32

<b>Purpose</b>	Cosecant and hyperbolic cosecant
<b>Syntax</b>	$Y = \text{csc}(x)$ $Y = \text{csch}(x)$
<b>Description</b>	The csc and csch functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.
	$Y = \text{csc}(x)$ returns the cosecant for each element of $x$ .
	$Y = \text{csch}(x)$ returns the hyperbolic cosecant for each element of $x$ .
<b>Examples</b>	Graph the cosecant and hyperbolic cosecant over the domains $-\pi < x < 0$ and $0 < x < \pi$ . <pre>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))</pre> 
<b>Algorithm</b>	$\text{csc}(z) = \frac{1}{\sin(z)}$ $\text{csch}(z) = \frac{1}{\sinh(z)}$
<b>See Also</b>	acsc,acsch      Inverse cosecant and inverse hyperbolic cosecant

# cumprod

---

**Purpose** Cumulative product

**Syntax**

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

**Description**  $B = \text{cumprod}(A)$  returns the cumulative product along different dimensions of an array.

If A is a vector,  $\text{cumprod}(A)$  returns a vector containing the cumulative product of the elements of A.

If A is a matrix,  $\text{cumprod}(A)$  returns a matrix the same size as A containing the cumulative products for each column of A.

If A is a multidimensional array,  $\text{cumprod}(A)$  works on the first nonsingleton dimension.

$B = \text{cumprod}(A, \text{dim})$  returns the cumulative product of the elements along the dimension of A specified by scalar dim. For example,  $\text{cumprod}(A, 1)$  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**

<code>cumsum</code>	Cumulative sum
<code>prod</code>	Product of array elements
<code>sum</code>	Sum of array elements

<b>Purpose</b>	Cumulative sum												
<b>Syntax</b>	$B = \text{cumsum}(A)$ $B = \text{cumsum}(A, \text{dim})$												
<b>Description</b>	<p><math>B = \text{cumsum}(A)</math> returns the cumulative sum along different dimensions of an array.</p> <p>If <math>A</math> is a vector, <math>\text{cumsum}(A)</math> returns a vector containing the cumulative sum of the elements of <math>A</math>.</p> <p>If <math>A</math> is a matrix, <math>\text{cumsum}(A)</math> returns a matrix the same size as <math>A</math> containing the cumulative sums for each column of <math>A</math>.</p> <p>If <math>A</math> is a multidimensional array, <math>\text{cumsum}(A)</math> works on the first nonsingleton dimension.</p> <p><math>B = \text{cumsum}(A, \text{dim})</math> returns the cumulative sum of the elements along the dimension of <math>A</math> specified by scalar <math>\text{dim}</math>. For example, <math>\text{cumsum}(A, 1)</math> works across the first dimension (the rows).</p>												
<b>Examples</b>	<pre>cumsum(1:5)    = [1 3 6 10 15]</pre> <pre>A = [1 2 3; 4 5 6];</pre> <pre>disp(cumsum(A))</pre> <table style="margin-left: 200px;"> <tr><td>1</td><td>2</td><td>3</td></tr> <tr><td>5</td><td>7</td><td>9</td></tr> </table> <pre>disp(cumsum(A, 2))</pre> <table style="margin-left: 200px;"> <tr><td>1</td><td>3</td><td>6</td></tr> <tr><td>4</td><td>9</td><td>15</td></tr> </table>	1	2	3	5	7	9	1	3	6	4	9	15
1	2	3											
5	7	9											
1	3	6											
4	9	15											
<b>See Also</b>	<a href="#">sum</a> Sum of array elements <a href="#">prod</a> Product of array elements <a href="#">cumprod</a> Cumulative product of elements												

# cumtrapz

---

<b>Purpose</b>	Cumulative trapezoidal numerical integration
<b>Syntax</b>	$Z = \text{cumtrapz}(Y)$ $Z = \text{cumtrapz}(X, Y)$ $Z = \text{cumtrapz}(\dots, \text{dim})$
<b>Description</b>	<p><math>Z = \text{cumtrapz}(Y)</math> computes an approximation of the cumulative integral of <math>Y</math> via the trapezoidal method with unit spacing. (This is similar to <math>\text{cumsum}(Y)</math>, except that trapezoidal approximation is used.) To compute the integral with other than unit spacing, multiply <math>Z</math> by the spacing increment.</p> <p>For vectors, <math>\text{cumtrapz}(Y)</math> is the cumulative integral of <math>Y</math>.</p> <p>For matrices, <math>\text{cumtrapz}(Y)</math> is a row vector with the cumulative integral over each column.</p> <p>For multidimensional arrays, <math>\text{cumtrapz}(Y)</math> works across the first nonsingleton dimension.</p> <p><math>Z = \text{cumtrapz}(X, Y)</math> computes the cumulative integral of <math>Y</math> with respect to <math>X</math> using trapezoidal integration. <math>X</math> and <math>Y</math> must be vectors of the same length, or <math>X</math> must be a column vector and <math>Y</math> an array.</p> <p>If <math>X</math> is a column vector and <math>Y</math> an array whose first nonsingleton dimension is <math>\text{length}(X)</math>, <math>\text{cumtrapz}(X, Y)</math> operates across this dimension.</p> <p><math>Z = \text{cumtrapz}(\dots, \text{dim})</math> integrates across the dimension of <math>Y</math> specified by scalar <math>\text{dim}</math>. The length of <math>X</math> must be the same as <math>\text{size}(Y, \text{dim})</math>.</p>
<b>Example</b>	<p>Example: If <math>Y = [0 \ 1 \ 2; \ 3 \ 4 \ 5]</math></p> <pre>cumtrapz(Y, 1) ans =     0    1.0000    2.0000     1.5000   2.5000    3.5000</pre> <p>and</p> <pre>cumtrapz(Y, 2) ans =     0    0.5000    2.0000     3.0000   3.5000    8.0000</pre>

**See Also**

[cumsum](#)  
[trapz](#)

Cumulative sum  
Trapezoidal numerical integration

## cumtrapz

---



## cumtrapz

---

---

<b>Purpose</b>	Current date string	
<b>Syntax</b>	<code>str = date</code>	
<b>Description</b>	<code>str = date</code> returns a string containing the date in dd-mmm-yyyy format.	
<b>See Also</b>	<code>clock</code>	Current time as a date vector
	<code>datenum</code>	Serial date number
	<code>now</code>	Current date and time

# datenum

---

<b>Purpose</b>	Serial date number
<b>Syntax</b>	$N = \text{datenum}(\text{str})$ $N = \text{datenum}(Y, M, D)$ $N = \text{datenum}(Y, M, D, H, MI, S)$
<b>Description</b>	The <code>datenum</code> 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.  <code>N = datenum(str)</code> converts the date string <code>str</code> into a serial date number.
<hr/>	
	<b>NOTE</b> The string <code>str</code> must be in one of the date formats 0, 1, 2, 6, 13, 14, 15, or 16 as defined by <code>datestr</code> .
<hr/>	
	$N = \text{datenum}(Y, M, D)$ returns the serial date number for corresponding elements of the <code>Y</code> , <code>M</code> , and <code>D</code> (year, month, day) arrays. <code>Y</code> , <code>M</code> , and <code>D</code> 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.
	$N = \text{datenum}(Y, M, D, H, MI, S)$ returns the serial date number for corresponding elements of the <code>Y</code> , <code>M</code> , <code>D</code> , <code>H</code> , <code>MI</code> , and <code>S</code> (year, month, hour, minute, and second) array values. <code>Y</code> , <code>M</code> , <code>D</code> , <code>H</code> , <code>MI</code> , and <code>S</code> must be arrays of the same size (or any can be a scalar).
<b>Examples</b>	<code>n = datenum('19-May-1995')</code> returns <code>n = 728798</code> . <code>n = datenum(1994, 12, 19)</code> returns <code>n = 728647</code> . <code>n = datenum(1994, 12, 19, 18, 0, 0)</code> returns <code>n = 7.2865e+05</code> .
<b>See Also</b>	<code>datestr</code> Date string format <code>datevec</code> Date components <code>now</code> Current date and time

**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- mmmm-yyyy HH:MM:SS'	01- Mar- 1995 03: 45
1	' dd- mmmm-yyyy'	01- Mar- 1995
2	' mm/dd/yy'	03/01/95
3	' mmmm'	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	' mmmmyy'	Mar95
13	' HH: MM: SS'	15: 45: 17

# datestr

<i>dateform</i> (number)	<i>dateform</i> (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`  
`datenum`  
`datevec`

Current date string  
Serial date number  
Date components

<b>Purpose</b>	Date components
	$C = \text{datevec}(A)$ $[Y, M, D, H, MI, S] = \text{datevec}(A)$
<b>Description</b>	$C = \text{datevec}(A)$ 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 <code>datestr</code> function, or scalars of the sort produced by the <code>datenum</code> and now functions.  $[Y, M, D, H, MI, S] = \text{datevec}(A)$ 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.
<b>Examples</b>	Let  <code>d = '12/24/1984'</code> <code>t = '725000.00',</code>  Then <code>datevec(d)</code> and <code>datevec(t)</code> generate [1984 12 24 0 0 0].
<b>See Also</b>	<code>clock</code> Current time as date vector <code>datenum</code> Serial date number <code>datestr</code> Date string format

# dbclear

<b>Purpose</b>	Clear breakpoints				
<b>Syntax</b>	<pre>dbclear dbclear at <i>lineno</i> in <i>function</i> dbclear all in <i>function</i> dbclear all dbclear in <i>mfilename</i>  dbclear if <i>keyword</i></pre>				
	where <i>keyword</i> is one of: <div style="text-align: right;"><math display="block">\begin{cases} \text{error} \\ \text{nani nf} \\ \text{inf nan} \\ \text{warning} \end{cases}</math></div>				
<b>Description</b>	<p>The at, in, and if keywords, familiar to users of the UNIX debugger dbx, are optional.</p> <p>dbclear, by itself, clears the breakpoint(s) set by a corresponding dbstop command.</p> <p>dbclear at <i>lineno</i> in <i>function</i> clears the breakpoint set at the specified line in the specified M-file. <i>function</i> must be the name of an M-file function or a MATLABPATH relative partial pathname.</p> <p>dbclear all in <i>function</i> clears all breakpoints in the specified M-file.</p> <p>dbclear all clears all breakpoints in all M-file functions, except for errors and warning breakpoints.</p> <p>dbclear in <i>function</i> clears the breakpoint set at the first executable line in the specified M-file.</p> <p>dbclear if <i>keyword</i> clears the indicated statement or breakpoint:</p> <table><tr><td>dbclear if error</td><td>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.</td></tr><tr><td>dbclear if nani nf</td><td>Clears the dbstop nani nf statement, if set.</td></tr></table>	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 nani nf	Clears the dbstop nani nf statement, if set.
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 nani nf	Clears the dbstop nani nf statement, if set.				

---

<b>dbclear if infnan</b>	Clears the dbstop infnan statement, if set.
<b>dbclear if warning</b>	Clears warning breakpoints.

**See Also**

<b>dbcont</b>	Resume execution
<b>dbdown</b>	Change local workspace context (down)
<b>dbquit</b>	Quit debug mode
<b>dbstack</b>	Display function call stack
<b>dbstatus</b>	List all breakpoints
<b>dbstep</b>	Execute one or more lines from a breakpoint
<b>dbstop</b>	Set breakpoints in an M-file function
<b>dbtype</b>	List M-file with line numbers
<b>dbup</b>	Change local workspace context (up)
See also <a href="#">partialpath</a> .	

# dbcont

---

<b>Purpose</b>	Resume execution																		
<b>Syntax</b>	dbcont																		
<b>Description</b>	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.																		
<b>See Also</b>	<table><tr><td>dbclear</td><td>Clear breakpoints</td></tr><tr><td>dbdown</td><td>Change local workspace context (down)</td></tr><tr><td>dbquit</td><td>Quit debug mode</td></tr><tr><td>dbstack</td><td>Display function call stack</td></tr><tr><td>dbstatus</td><td>List all breakpoints</td></tr><tr><td>dbstep</td><td>Execute one or more lines from a breakpoint</td></tr><tr><td>dbstop</td><td>Set breakpoints in an M-file function</td></tr><tr><td>dbtype</td><td>List M-file with line numbers</td></tr><tr><td>dbup</td><td>Change local workspace context (up)</td></tr></table>	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)
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)																		

<b>Purpose</b>	Change local workspace context																		
<b>Syntax</b>	dbdown																		
<b>Description</b>	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.																		
<b>See Also</b>	<table><tr><td>dbclear</td><td>Clear breakpoints</td></tr><tr><td>dbcont</td><td>Resume execution</td></tr><tr><td>dbquit</td><td>Quit debug mode</td></tr><tr><td>dbstack</td><td>Display function call stack</td></tr><tr><td>dbstatus</td><td>List all breakpoints</td></tr><tr><td>dbstep</td><td>Execute one or more lines from a breakpoint</td></tr><tr><td>dbstop</td><td>Set breakpoints in an M-file function</td></tr><tr><td>dbtype</td><td>List M-file with line numbers</td></tr><tr><td>dbup</td><td>Change local workspace context (up)</td></tr></table>	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)
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)																		

# dblquad

---

<b>Purpose</b>	Numerical double integration
<b>Syntax</b>	<pre>resul t = dbl quad(' fun' , i nmi n, i nmax, out mi n, out max) resul t = dbl quad(' fun' , i nmi n, i nmax, out mi n, out max, tol , trace) resul t = dbl quad(' fun' , i nmi n, i nmax, out mi n, out max, tol , trace, order)</pre>
<b>Description</b>	<p><code>resul t = dbl quad(' fun' , i nmi n, i nmax, out mi n, out max)</code> evaluates the double integral <math>\int_{\text{out\_inner}}^{\text{out\_outer}} \int_{\text{in\_inner}}^{\text{in\_outer}} \text{fun}(\text{in\_inner}, \text{out\_inner}) \, d\text{in\_inner} \, d\text{out\_inner}</math> using the quad quadrature function. <code>i nner</code> is the inner variable, ranging from <code>i nmi n</code> to <code>i nmax</code>, and <code>outer</code> is the outer variable, ranging from <code>out mi n</code> to <code>out max</code>. The first argument '<code>fun</code>' is a string representing the integrand function. This function must be a function of two variables of the form <code>fout = fun(i nner, outer)</code>. The function must take a vector <code>i nner</code> and a scalar <code>outer</code> and return a vector <code>fout</code> that is the function evaluated at <code>outer</code> and each value of <code>i nner</code>.</p> <p><code>resul t = dbl quad(' fun' , i nmi n, i nmax, out mi n, out max, tol , trace)</code> passes <code>tol</code> and <code>trace</code> to the <code>quad</code> function. See the help entry for <code>quad</code> for a description of the <code>tol</code> and <code>trace</code> parameters.</p> <p><code>resul t = dbl quad(' fun' , i nmi n, i nmax, out mi n, out max, tol , trace, order)</code> passes <code>tol</code> and <code>trace</code> to the <code>quad</code> or <code>quad8</code> function depending on the value of the string <code>order</code>. Valid values for <code>order</code> are '<code>quad</code>' and '<code>quad8</code>' or the name of any user-defined quadrature method with the same calling and return arguments as <code>quad</code> and <code>quad8</code>.</p>
<b>Example</b>	<p><code>resul t = dbl quad(' integrnd' , pi , 2*pi , 0, pi)</code> integrates the function <math>y \sin(x) + x \cos(y)</math>, where <math>x</math> ranges from <math>\pi</math> to <math>2\pi</math>, and <math>y</math> ranges from 0 to <math>\pi</math>, assuming:</p> <ul style="list-style-type: none"><li>• <math>x</math> is the inner variable in the integration.</li><li>• <math>y</math> is the outer variable.</li><li>• the M-file <code>integrnd.m</code> is defined as:</li></ul> <pre>function out = integrnd(x, y) out = y*sin(x)+x*cos(y);</pre> <p>Note that <code>integrnd.m</code> is valid when <math>x</math> is a vector and <math>y</math> is a scalar. Also, <math>x</math> must be the first argument to <code>integrnd.m</code> since it is the inner variable.</p>

**See Also**

quad, quad8

Numerical evaluation of integrals

# dbmex

---

<b>Purpose</b>	Enable MEX-file debugging
<b>Syntax</b>	<code>dbmex on</code> <code>dbmex off</code> <code>dbmex stop</code> <code>dbmex print</code>
<b>Description</b>	<code>dbmex on</code> enables MEX-file debugging. To use this option, first start MATLAB from within a debugger by typing: <code>matlab -Ddebugger</code> , where <code>debugger</code> is the name of the debugger.  <code>dbmex off</code> disables MEX-file debugging.  <code>dbmex stop</code> returns to the debugger prompt.  <code>dbmex print</code> displays MEX-file debugging information.  <code>dbmex</code> is not available on the Macintosh or the PC.
<b>See Also</b>	<code>dbstop</code> Set breakpoints in an M-file function <code>dbclear</code> Clear breakpoints <code>dbcont</code> Resume execution <code>dbdown</code> Change local workspace context (down) <code>dbquit</code> Quit debug mode <code>dbstack</code> Display function call stack <code>dbstatus</code> List all breakpoints <code>dbstep</code> Execute one or more lines from a breakpoint <code>dbtype</code> List M-file with line numbers <code>dbup</code> Change local workspace context (up)

<b>Purpose</b>	Quit debug mode
<b>Syntax</b>	dbquit
<b>Description</b>	dbquit immediately terminates the debugger and returns control to the base workspace prompt. The M-file being processed is <i>not</i> completed and no results are returned.  All breakpoints remain in effect.
<b>See Also</b>	<a href="#">dbclear</a> Clear breakpoints <a href="#">dbcont</a> Resume execution <a href="#">dbdown</a> Change local workspace context (down) <a href="#">dbstack</a> Display function call stack <a href="#">dbstatus</a> List all breakpoints <a href="#">dbstep</a> Execute one or more lines from a breakpoint <a href="#">dbstop</a> Set breakpoints in an M-file function <a href="#">dbtype</a> List M-file with line numbers <a href="#">dbup</a> Change local workspace context (up)

# dbstack

<b>Purpose</b>	Display function call stack																		
<b>Syntax</b>	<code>dbstack</code> <code>[ ST, I ] = dbstack</code>																		
<b>Description</b>	<code>dbstack</code> 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.																		
	<code>[ ST, I ] = dbstack</code> returns the stack trace information in an <code>m</code> -by-1 structure <code>ST</code> with the fields:																		
	<code>name</code> function name <code>line</code> function line number																		
	The current workspace index is returned in <code>I</code> .																		
<b>Examples</b>	<pre>&gt;&gt; dbstack &gt; In /usr/local/matlab/toolbox/matlab/cond.m at line 13     In test1.m at line 2     In test.m at line 3</pre>																		
<b>See Also</b>	<table><tr><td><code>dbclear</code></td><td>Clear breakpoints</td></tr><tr><td><code>dbcont</code></td><td>Resume execution</td></tr><tr><td><code>dbdown</code></td><td>Change local workspace context (down)</td></tr><tr><td><code>dbquit</code></td><td>Quit debug mode</td></tr><tr><td><code>dbstatus</code></td><td>List all breakpoints</td></tr><tr><td><code>dbstep</code></td><td>Execute one or more lines from a breakpoint</td></tr><tr><td><code>dbstop</code></td><td>Set breakpoints in an M-file function</td></tr><tr><td><code>dbtype</code></td><td>List M-file with line numbers</td></tr><tr><td><code>dbup</code></td><td>Change local workspace context (up)</td></tr></table>	<code>dbclear</code>	Clear breakpoints	<code>dbcont</code>	Resume execution	<code>dbdown</code>	Change local workspace context (down)	<code>dbquit</code>	Quit debug mode	<code>dbstatus</code>	List all breakpoints	<code>dbstep</code>	Execute one or more lines from a breakpoint	<code>dbstop</code>	Set breakpoints in an M-file function	<code>dbtype</code>	List M-file with line numbers	<code>dbup</code>	Change local workspace context (up)
<code>dbclear</code>	Clear breakpoints																		
<code>dbcont</code>	Resume execution																		
<code>dbdown</code>	Change local workspace context (down)																		
<code>dbquit</code>	Quit debug mode																		
<code>dbstatus</code>	List all breakpoints																		
<code>dbstep</code>	Execute one or more lines from a breakpoint																		
<code>dbstop</code>	Set breakpoints in an M-file function																		
<code>dbtype</code>	List M-file with line numbers																		
<code>dbup</code>	Change local workspace context (up)																		

<b>Purpose</b>	List all breakpoints
<b>Syntax</b>	<code>dbstatus</code> <code>dbstatus <i>function</i></code> <code>s = dbstatus(...)</code>
<b>Description</b>	<code>dbstatus</code> , by itself, lists all breakpoints in effect including error, warning, and nani nf.  <code>dbstatus <i>function</i></code> displays a list of the line numbers for which breakpoints are set in the specified M-file.  <code>s = dbstatus(...)</code> 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 nani nf)  Use <code>dbstatus <i>class/function</i></code> or <code>dbstatus private/function</code> or <code>dbstatus <i>class/private/function</i></code> to determine the status for methods, private functions, or private methods (for a class named <i>class</i> ). In all of these forms you can further qualify the function name with a subfunction name as in <code>dbstatus <i>function/subfunction</i></code> .
<b>See Also</b>	<code>dbclear</code> Clear breakpoints <code>dbcont</code> Resume execution <code>dbdown</code> Change local workspace context (down) <code>dbquit</code> Quit debug mode <code>dbstack</code> Display function call stack <code>dbstep</code> Execute one or more lines from a breakpoint <code>dbstop</code> Set breakpoints in an M-file function <code>dbtype</code> List M-file with line numbers <code>dbup</code> Change local workspace context (up)

# dbstep

<b>Purpose</b>	Execute one or more lines from a breakpoint																		
<b>Syntax</b>	<code>dbstep</code> <code>dbstep nlines</code> <code>dbstep in</code>																		
<b>Description</b>	This command allows you to debug an M-file by following its execution from the current breakpoint. At a breakpoint, the <code>dbstep</code> command steps through execution of the current M-file one line at a time or at the rate specified by <code>nlines</code> .  <code>dbstep</code> , by itself, executes the next executable line of the current M-file. <code>dbstep</code> steps over the current line, skipping any breakpoints set in functions called by that line.  <code>dbstep nlines</code> executes the specified number of executable lines.  <code>dbstep in</code> 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, <code>dbstep in</code> is the same as <code>dbstep</code> .																		
<b>See Also</b>	<table><tr><td><code>dbclear</code></td><td>Clear breakpoints</td></tr><tr><td><code>dbcont</code></td><td>Resume execution</td></tr><tr><td><code>dbdown</code></td><td>Change local workspace context (down)</td></tr><tr><td><code>dbquit</code></td><td>Quit debug mode</td></tr><tr><td><code>dbstack</code></td><td>Display function call stack</td></tr><tr><td><code>dbstatus</code></td><td>List all breakpoints</td></tr><tr><td><code>dbstop</code></td><td>Set breakpoints in an M-file function</td></tr><tr><td><code>dbtype</code></td><td>List M-file with line numbers</td></tr><tr><td><code>dbup</code></td><td>Change local workspace context (up)</td></tr></table>	<code>dbclear</code>	Clear breakpoints	<code>dbcont</code>	Resume execution	<code>dbdown</code>	Change local workspace context (down)	<code>dbquit</code>	Quit debug mode	<code>dbstack</code>	Display function call stack	<code>dbstatus</code>	List all breakpoints	<code>dbstop</code>	Set breakpoints in an M-file function	<code>dbtype</code>	List M-file with line numbers	<code>dbup</code>	Change local workspace context (up)
<code>dbclear</code>	Clear breakpoints																		
<code>dbcont</code>	Resume execution																		
<code>dbdown</code>	Change local workspace context (down)																		
<code>dbquit</code>	Quit debug mode																		
<code>dbstack</code>	Display function call stack																		
<code>dbstatus</code>	List all breakpoints																		
<code>dbstop</code>	Set breakpoints in an M-file function																		
<code>dbtype</code>	List M-file with line numbers																		
<code>dbup</code>	Change local workspace context (up)																		

<b>Purpose</b>	Set breakpoints in an M-file function								
<b>Syntax</b>	<pre>dbstop at <i>lineno</i> in <i>function</i> dbstop in <i>function</i></pre>								
<code>dbstop if <i>keyword</i></code>	where <i>keyword</i> is one of: <div style="display: flex; align-items: center;"> <span style="margin-right: 10px;">error</span> <span style="font-size: 2em; font-weight: bold; margin-right: 10px;">{</span> <span style="font-size: 0.8em; font-family: monospace; margin-right: 10px;">nani nf</span> <span style="font-size: 0.8em; font-family: monospace; margin-right: 10px;">infnan</span> <span style="font-size: 0.8em; font-family: monospace; margin-right: 10px;">warning</span> </div>								
<b>Description</b>	<p>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.</p> <p><code>dbstop at <i>lineno</i> in <i>function</i></code> stops execution just prior to execution of that line of the specified M-file function. <i>function</i> must be the name of an M-file function or a MATLABPATH relative partial pathname.</p> <p><code>dbstop in <i>function</i></code> stops execution before the first executable line in the M-file function when it is called.</p> <p><code>dbstop if <i>keyword</i></code> stops execution under the specified conditions:</p> <table border="0"> <tr> <td><code>dbstop if error</code></td> <td>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.</td> </tr> <tr> <td><code>dbstop if nani nf</code></td> <td>Stops execution when it detects Not-a-Number (NaN) or Infinity (Inf).</td> </tr> <tr> <td><code>dbstop if infnan</code></td> <td>Stops execution when it detects Not-a-Number (NaN) or Infinity (Inf).</td> </tr> <tr> <td><code>dbstop if warning</code></td> <td>Stops execution if a runtime warning occurs in any M-file function.</td> </tr> </table>	<code>dbstop if error</code>	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.	<code>dbstop if nani nf</code>	Stops execution when it detects Not-a-Number (NaN) or Infinity (Inf).	<code>dbstop if infnan</code>	Stops execution when it detects Not-a-Number (NaN) or Infinity (Inf).	<code>dbstop if warning</code>	Stops execution if a runtime warning occurs in any M-file function.
<code>dbstop if error</code>	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.								
<code>dbstop if nani nf</code>	Stops execution when it detects Not-a-Number (NaN) or Infinity (Inf).								
<code>dbstop if infnan</code>	Stops execution when it detects Not-a-Number (NaN) or Infinity (Inf).								
<code>dbstop if warning</code>	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

# dbstop

---

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 nani nf
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 <a href="#">partialpath</a> .	

# dbtype

---

<b>Purpose</b>	List M-file with line numbers																		
<b>Syntax</b>	<code>dbtype <i>function</i></code> <code>dbtype <i>function start:end</i></code>																		
<b>Description</b>	<code>dbtype <i>function</i></code> displays the contents of the specified M-file function with line numbers preceding each line. <i>function</i> must be the name of an M-file function or a MATLABPATH relative partial pathname.  <code>dbtype <i>function start:end</i></code> displays the portion of the file specified by a range of line numbers.																		
<b>See Also</b>	<table><tr><td><code>dbclear</code></td><td>Clear breakpoints</td></tr><tr><td><code>dbcont</code></td><td>Resume execution</td></tr><tr><td><code>dbdown</code></td><td>Change local workspace context (down)</td></tr><tr><td><code>dbquit</code></td><td>Quit debug mode</td></tr><tr><td><code>dbstack</code></td><td>Display function call stack</td></tr><tr><td><code>dbstatus</code></td><td>List all breakpoints</td></tr><tr><td><code>dbstep</code></td><td>Execute one or more lines from a breakpoint</td></tr><tr><td><code>dbstop</code></td><td>Set breakpoints in an M-file function</td></tr><tr><td><code>dbup</code></td><td>Change local workspace context (up)</td></tr></table> <p>See also <code>partialpath</code>.</p>	<code>dbclear</code>	Clear breakpoints	<code>dbcont</code>	Resume execution	<code>dbdown</code>	Change local workspace context (down)	<code>dbquit</code>	Quit debug mode	<code>dbstack</code>	Display function call stack	<code>dbstatus</code>	List all breakpoints	<code>dbstep</code>	Execute one or more lines from a breakpoint	<code>dbstop</code>	Set breakpoints in an M-file function	<code>dbup</code>	Change local workspace context (up)
<code>dbclear</code>	Clear breakpoints																		
<code>dbcont</code>	Resume execution																		
<code>dbdown</code>	Change local workspace context (down)																		
<code>dbquit</code>	Quit debug mode																		
<code>dbstack</code>	Display function call stack																		
<code>dbstatus</code>	List all breakpoints																		
<code>dbstep</code>	Execute one or more lines from a breakpoint																		
<code>dbstop</code>	Set breakpoints in an M-file function																		
<code>dbup</code>	Change local workspace context (up)																		

<b>Purpose</b>	Change local workspace context																		
<b>Syntax</b>	dbup																		
<b>Description</b>	<p>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.</p> <p>dbup changes the current workspace context (at a breakpoint) to the workspace of the calling M-file.</p> <p>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.)</p>																		
<b>See Also</b>	<table><tr><td>dbcl ear</td><td>Clear breakpoints</td></tr><tr><td>dbcont</td><td>Resume execution</td></tr><tr><td>dbdown</td><td>Change local workspace context (down)</td></tr><tr><td>dbquit</td><td>Quit debug mode</td></tr><tr><td>dbstack</td><td>Display function call stack</td></tr><tr><td>dbstatus</td><td>List all breakpoints</td></tr><tr><td>dbstep</td><td>Execute one or more lines from a breakpoint</td></tr><tr><td>dbstop</td><td>Set breakpoints in an M-file function</td></tr><tr><td>dbtype</td><td>List M-file with line numbers</td></tr></table>	dbcl ear	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
dbcl ear	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																		

# ddeadv

<b>Purpose</b>	Set up advisory link										
<b>Syntax</b>	<pre>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)</pre>										
<b>Description</b>	<p>ddeadv sets up an advisory link between MATLAB and a server application. When the data identified by the <i>item</i> argument changes, the string specified by the <i>callback</i> argument is passed to the eval function and evaluated. If the advisory link is a hot link, DDE modifies <i>upmtx</i>, the update matrix, to reflect the data in <i>item</i>.</p> <p>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).</p>										
<b>Arguments</b>	<table><tr><td><i>rc</i></td><td>Return code: 0 indicates failure, 1 indicates success.</td></tr><tr><td><i>channel</i></td><td>Conversation channel from ddeinit.</td></tr><tr><td><i>item</i></td><td>String specifying the DDE item name for the advisory link. Changing the data identified by <i>item</i> at the server triggers the advisory link.</td></tr><tr><td><i>callback</i></td><td>String specifying the callback that is evaluated on update notification. Changing the data identified by <i>item</i> at the server causes <i>callback</i> to get passed to the eval function to be evaluated.</td></tr><tr><td><i>upmtx</i> (optional)</td><td>String specifying the name of a matrix that holds data sent with an update notification. If <i>upmtx</i> is included, changing <i>item</i> at the server causes <i>upmtx</i> to be updated with the revised data. Specifying <i>upmtx</i> creates a hot link. Omitting <i>upmtx</i> or specifying it as an empty string creates a warm link. If <i>upmtx</i> exists in the workspace, its contents are overwritten. If <i>upmtx</i> does not exist, it is created.</td></tr></table>	<i>rc</i>	Return code: 0 indicates failure, 1 indicates success.	<i>channel</i>	Conversation channel from ddeinit.	<i>item</i>	String specifying the DDE item name for the advisory link. Changing the data identified by <i>item</i> at the server triggers the advisory link.	<i>callback</i>	String specifying the callback that is evaluated on update notification. Changing the data identified by <i>item</i> at the server causes <i>callback</i> to get passed to the eval function to be evaluated.	<i>upmtx</i> (optional)	String specifying the name of a matrix that holds data sent with an update notification. If <i>upmtx</i> is included, changing <i>item</i> at the server causes <i>upmtx</i> to be updated with the revised data. Specifying <i>upmtx</i> creates a hot link. Omitting <i>upmtx</i> or specifying it as an empty string creates a warm link. If <i>upmtx</i> exists in the workspace, its contents are overwritten. If <i>upmtx</i> does not exist, it is created.
<i>rc</i>	Return code: 0 indicates failure, 1 indicates success.										
<i>channel</i>	Conversation channel from ddeinit.										
<i>item</i>	String specifying the DDE item name for the advisory link. Changing the data identified by <i>item</i> at the server triggers the advisory link.										
<i>callback</i>	String specifying the callback that is evaluated on update notification. Changing the data identified by <i>item</i> at the server causes <i>callback</i> to get passed to the eval function to be evaluated.										
<i>upmtx</i> (optional)	String specifying the name of a matrix that holds data sent with an update notification. If <i>upmtx</i> is included, changing <i>item</i> at the server causes <i>upmtx</i> to be updated with the revised data. Specifying <i>upmtx</i> creates a hot link. Omitting <i>upmtx</i> or specifying it as an empty string creates a warm link. If <i>upmtx</i> exists in the workspace, its contents are overwritten. If <i>upmtx</i> does not exist, it is created.										

<b>format</b> <i>(optional)</i>	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].
<b>timeout</b> <i>(optional)</i>	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
dderek	Request data from application
ddeterm	Terminate DDE conversation
ddeunadv	Release advisory link

# ddeexec

<b>Purpose</b>	Send string for execution												
<b>Syntax</b>	<pre>rc = ddeexec(channel, 'command') rc = ddeexec(channel, 'command', 'item') rc = ddeexec(channel, 'command', 'item', timeout)</pre>												
<b>Description</b>	<p>ddeexec sends a string for execution to another application via an established DDE conversation. Specify the string as the <i>command</i> argument.</p> <p>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).</p>												
<b>Arguments</b>	<table><tr><td><i>rc</i></td><td>Return code: 0 indicates failure, 1 indicates success.</td></tr><tr><td><i>channel</i></td><td>Conversation channel from ddeinit.</td></tr><tr><td><i>command</i></td><td>String specifying the command to be executed.</td></tr><tr><td><i>item</i> <i>(optional)</i></td><td>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 <i>command</i>. Consult your server documentation for more information.</td></tr><tr><td><i>timeout</i> <i>(optional)</i></td><td>Scalar specifying the time-out limit for this operation. <i>timeout</i> is specified in milliseconds. (1000 milliseconds = 1 second). The default value of <i>timeout</i> is three seconds.</td></tr></table>	<i>rc</i>	Return code: 0 indicates failure, 1 indicates success.	<i>channel</i>	Conversation channel from ddeinit.	<i>command</i>	String specifying the command to be executed.	<i>item</i> <i>(optional)</i>	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 <i>command</i> . Consult your server documentation for more information.	<i>timeout</i> <i>(optional)</i>	Scalar specifying the time-out limit for this operation. <i>timeout</i> is specified in milliseconds. (1000 milliseconds = 1 second). The default value of <i>timeout</i> is three seconds.		
<i>rc</i>	Return code: 0 indicates failure, 1 indicates success.												
<i>channel</i>	Conversation channel from ddeinit.												
<i>command</i>	String specifying the command to be executed.												
<i>item</i> <i>(optional)</i>	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 <i>command</i> . Consult your server documentation for more information.												
<i>timeout</i> <i>(optional)</i>	Scalar specifying the time-out limit for this operation. <i>timeout</i> is specified in milliseconds. (1000 milliseconds = 1 second). The default value of <i>timeout</i> is three seconds.												
<b>Examples</b>	<p>Given the channel assigned to a conversation, send a command to Excel:</p> <pre>rc = ddeexec(channel, '[formula.goto("r1c1")]')</pre> <p>Communication with Excel must have been established previously with a ddeinit command.</p>												
<b>See Also</b>	<table><tr><td>ddeadv</td><td>Set up advisory link</td></tr><tr><td>ddeinit</td><td>Initiate DDE conversation</td></tr><tr><td>ddepoke</td><td>Send data to application</td></tr><tr><td>ddereq</td><td>Request data from application</td></tr><tr><td>ddeterm</td><td>Terminate DDE conversation</td></tr><tr><td>ddeunadv</td><td>Release advisory link</td></tr></table>	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
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												

<b>Purpose</b>	Initiate DDE conversation												
<b>Syntax</b>	<code>channel = ddeinit('service', 'topic')</code>												
<b>Description</b>	<code>channel = ddeinit('service', 'topic')</code> returns a channel handle assigned to the conversation, which is used with other MATLAB DDE functions. ' <i>service</i> ' is a string specifying the service or application name for the conversation. ' <i>topic</i> ' is a string specifying the topic for the conversation.												
<b>Examples</b>	To initiate a conversation with Excel for the spreadsheet 'stocks.xls':  <code>channel = ddeinit('excel', 'stocks.xls')</code>  <code>channel =</code> 0.00												
<b>See Also</b>	<table><tr><td><code>ddeadv</code></td><td>Set up advisory link</td></tr><tr><td><code>ddeexec</code></td><td>Send string for execution</td></tr><tr><td><code>ddepoke</code></td><td>Send data to application</td></tr><tr><td><code>ddereq</code></td><td>Request data from application</td></tr><tr><td><code>ddeterm</code></td><td>Terminate DDE conversation</td></tr><tr><td><code>ddeunadv</code></td><td>Release advisory link</td></tr></table>	<code>ddeadv</code>	Set up advisory link	<code>ddeexec</code>	Send string for execution	<code>ddepoke</code>	Send data to application	<code>ddereq</code>	Request data from application	<code>ddeterm</code>	Terminate DDE conversation	<code>ddeunadv</code>	Release advisory link
<code>ddeadv</code>	Set up advisory link												
<code>ddeexec</code>	Send string for execution												
<code>ddepoke</code>	Send data to application												
<code>ddereq</code>	Request data from application												
<code>ddeterm</code>	Terminate DDE conversation												
<code>ddeunadv</code>	Release advisory link												

# ddepoke

<b>Purpose</b>	Send data to application												
<b>Syntax</b>	<pre>rc = ddepoke(channel, 'item', data) rc = ddepoke(channel, 'item', data, format) rc = ddepoke(channel, 'item', data, format, timeout)</pre>												
<b>Description</b>	<p>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:</p> <ul style="list-style-type: none"><li>• String matrices are converted, element by element, to characters and the resulting character buffer is sent.</li><li>• 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.</li></ul> <p>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).</p>												
<b>Arguments</b>	<table><tr><td>rc</td><td>Return code: 0 indicates failure, 1 indicates success.</td></tr><tr><td>channel</td><td>Conversation channel from ddeinit.</td></tr><tr><td><i>item</i></td><td>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.</td></tr><tr><td>data</td><td>Matrix containing the data to send.</td></tr><tr><td>format (optional)</td><td>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.</td></tr><tr><td>timeout (optional)</td><td>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.</td></tr></table>	rc	Return code: 0 indicates failure, 1 indicates success.	channel	Conversation channel from ddeinit.	<i>item</i>	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.
rc	Return code: 0 indicates failure, 1 indicates success.												
channel	Conversation channel from ddeinit.												
<i>item</i>	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 ddeini t. 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
ddeini t	Initiate DDE conversation
ddereq	Request data from application
ddeterm	Terminate DDE conversation
ddeunadv	Release advisory link

# ddreq

---

<b>Purpose</b>	Request data from application										
<b>Syntax</b>	<pre>data = ddreq(channel, 'item') data = ddreq(channel, 'item', format) data = ddreq(channel, 'item', format, timeout)</pre>										
<b>Description</b>	<p>ddreq requests data from a server application via an established DDE conversation. ddreq returns a matrix containing the requested data or an empty matrix if the function is unsuccessful.</p> <p>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).</p>										
<b>Arguments</b>	<table><tr><td><code>data</code></td><td>Matrix containing requested data, empty if function fails.</td></tr><tr><td><code>channel</code></td><td>Conversation channel from ddeinit.</td></tr><tr><td><code>item</code></td><td>String specifying the server application's DDE item name for the data requested.</td></tr><tr><td><code>format</code> <i>(optional)</i></td><td>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].</td></tr><tr><td><code>timeout</code> <i>(optional)</i></td><td>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.</td></tr></table>	<code>data</code>	Matrix containing requested data, empty if function fails.	<code>channel</code>	Conversation channel from ddeinit.	<code>item</code>	String specifying the server application's DDE item name for the data requested.	<code>format</code> <i>(optional)</i>	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].	<code>timeout</code> <i>(optional)</i>	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.
<code>data</code>	Matrix containing requested data, empty if function fails.										
<code>channel</code>	Conversation channel from ddeinit.										
<code>item</code>	String specifying the server application's DDE item name for the data requested.										
<code>format</code> <i>(optional)</i>	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].										
<code>timeout</code> <i>(optional)</i>	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.										
<b>Examples</b>	<p>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:</p> <pre>channel = ddeinit('excel', 'stocks.xls')</pre> <p>DDE functions require the rxcy reference style for Excel worksheets. In Excel terminology the prices are in r3c1:r3c3 and the shares in r6c2:r8c2.</p>										

To request the prices from Excel:

```
prices = ddreq(channel, 'r3c1:r3c3')
```

```
prices =  
        42.50      15.00      78.88
```

To request the number of shares of each stock:

```
shares = ddreq(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

# ddeterm

---

<b>Purpose</b>	Terminate DDE conversation												
<b>Syntax</b>	<code>rc = ddeterm(channel)</code>												
<b>Description</b>	<code>rc = ddeterm(channel)</code> accepts a channel handle returned by a previous call to <code>ddeinit</code> that established the DDE conversation. <code>ddeterm</code> terminates this conversation. <code>rc</code> is a return code where 0 indicates failure and 1 indicates success.												
<b>Examples</b>	To close a conversation channel previously opened with <code>ddeinit</code> :												
	<pre>rc = ddeterm(channel)  rc = 1. 00</pre>												
<b>See Also</b>	<table><tr><td><code>ddeadv</code></td><td>Set up advisory link</td></tr><tr><td><code>ddeexec</code></td><td>Send string for execution</td></tr><tr><td><code>ddeinit</code></td><td>Initiate DDE conversation</td></tr><tr><td><code>ddepoke</code></td><td>Send data to application</td></tr><tr><td><code>dderek</code></td><td>Request data from application</td></tr><tr><td><code>ddeunadv</code></td><td>Release advisory link</td></tr></table>	<code>ddeadv</code>	Set up advisory link	<code>ddeexec</code>	Send string for execution	<code>ddeinit</code>	Initiate DDE conversation	<code>ddepoke</code>	Send data to application	<code>dderek</code>	Request data from application	<code>ddeunadv</code>	Release advisory link
<code>ddeadv</code>	Set up advisory link												
<code>ddeexec</code>	Send string for execution												
<code>ddeinit</code>	Initiate DDE conversation												
<code>ddepoke</code>	Send data to application												
<code>dderek</code>	Request data from application												
<code>ddeunadv</code>	Release advisory link												

<b>Purpose</b>	Release advisory link
<b>Syntax</b>	<pre>rc = ddeunadv(channel, 'item') rc = ddeunadv(channel, 'item', format) rc = ddeunadv(channel, 'item', format, timeout)</pre>
<b>Description</b>	ddeunadv releases the advisory link between MATLAB and the server application established by an earlier ddeadv call. The channel, <i>item</i> , 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.
<b>Arguments</b>	<p><b>rc</b>           Return code: 0 indicates failure, 1 indicates success.</p> <p><b>channel</b>      Conversation channel from ddeinit.</p> <p><b>item</b>          String specifying the DDE item name for the advisory link. Changing the data identified by item at the server triggers the advisory link.</p> <p><b>format</b>        Two-element array. This must be the same as the format argument for the corresponding ddeadv call.</p> <p><b>timeout</b>       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.</p>
<b>Example</b>	To release an advisory link established previously with ddeadv:
	<pre>rc = ddeadv(channel, 'r1c1:r5c5') rc = 1.00</pre>
<b>See Also</b>	<p><b>ddeadv</b>       Set up advisory link</p> <p><b>ddeexec</b>      Send string for execution</p> <p><b>ddeinit</b>      Initiate DDE conversation</p> <p><b>ddepoke</b>       Send data to application</p> <p><b>ddereq</b>        Request data from application</p> <p><b>ddeterm</b>       Release advisory link</p>

# deal

---

<b>Purpose</b>	Deal inputs to outputs
<b>Syntax</b>	$[Y_1, Y_2, Y_3, \dots] = \text{deal}(X)$ $[Y_1, Y_2, Y_3, \dots] = \text{deal}(X_1, X_2, X_3, \dots)$
<b>Description</b>	$[Y_1, Y_2, Y_3, \dots] = \text{deal}(X)$ copies the single input to all the requested outputs. It is the same as $Y_1 = X, Y_2 = X, Y_3 = X, \dots$ $[Y_1, Y_2, Y_3, \dots] = \text{deal}(X_1, X_2, X_3, \dots)$ is the same as $Y_1 = X_1; Y_2 = X_2; Y_3 = X_3; \dots$
<b>Remarks</b>	deal is most useful when used with cell arrays and structures via comma separated list expansion. Here are some useful constructions:  $[S.\text{field}] = \text{deal}(X)$ sets all the fields with the name field in the structure array S to the value X. If S doesn't exist, use $[S(1:m).\text{field}] = \text{deal}(X)$ .  $[X{:}] = \text{deal}(A.\text{field})$ copies the values of the field with name field to the cell array X. If X doesn't exist, use $[X\{1:m\}] = \text{deal}(A.\text{field})$ .  $[Y_1, Y_2, Y_3, \dots] = \text{deal}(X\{:})$ copies the contents of the cell array X to the separate variables Y1, Y2, Y3, ...  $[Y_1, Y_2, Y_3, \dots] = \text{deal}(S.\text{field})$ 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
```

## deal

---

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
```

**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} = ' SI MULI NK      ';
A{2, 1} = ' Tool boxes      ';
A{2, 2} = ' The MathWorks      ';
```

`A =`

```
' MATLAB'          ' SI MULI NK'
' Tool boxes'       ' The MathWorks'
```

`deblank(A)`

`ans =`

```
' MATLAB'          ' SI MULI NK'
' Tool boxes'       ' The MathWorks'
```

## dec2base

---

<b>Purpose</b>	Decimal number to base conversion
<b>Syntax</b>	<code>str = dec2base(d, base)</code> <code>str = dec2base(d, base, n)</code>
<b>Description</b>	<code>str = dec2base(d, base)</code> 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 <code>str</code> is a string.  <code>str = dec2base(d, base, n)</code> produces a representation with at least $n$ digits.
<b>Examples</b>	The expression <code>dec2base(23, 2)</code> converts $23_{10}$ to base 2, returning the string ' <code>10111</code> '.
<b>See Also</b>	<code>base2dec</code>

---

<b>Purpose</b>	Decimal to binary number conversion
<b>Syntax</b>	<code>str = dec2bin(d)</code> <code>str = dec2bin(d, n)</code>
<b>Description</b>	<code>str = dec2bin(d)</code> returns the binary representation of d as a string. d must be a nonnegative integer smaller than $2^{52}$ .  <code>str = dec2bin(d, n)</code> produces a binary representation with at least n bits.
<b>Examples</b>	<code>dec2bin(23)</code> returns '10111'.
<b>See Also</b>	<code>bin2dec</code> Binary to decimal number conversion <code>dec2hex</code> Decimal to hexadecimal number conversion

# dec2hex

---

<b>Purpose</b>	Decimal to hexadecimal number conversion
<b>Syntax</b>	<code>str = dec2hex(d)</code> <code>str = dec2hex(d, n)</code>
<b>Description</b>	<code>str = dec2hex(d)</code> converts the decimal integer $d$ to its hexadecimal representation stored in a MATLAB string. $d$ must be a nonnegative integer smaller than $2^{52}$ .  <code>str = dec2hex(d, n)</code> produces a hexadecimal representation with at least $n$ digits.
<b>Examples</b>	<code>dec2hex(1023)</code> is the string ' <code>3ff</code> '.
<b>See Also</b>	<code>dec2bin</code> Decimal to binary number conversion <code>format</code> Control the output display format <code>hex2dec</code> IEEE hexadecimal to decimal number conversion <code>hex2num</code> Hexadecimal to double number conversion

<b>Purpose</b>	Deconvolution and polynomial division															
<b>Syntax</b>	$[q, r] = \text{deconv}(v, u)$															
<b>Description</b>	$[q, r] = \text{deconv}(v, u)$ 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 $v = \text{conv}(u, q) + r$ .  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 $r$ .															
<b>Examples</b>	If  $u = [1 \quad 2 \quad 3 \quad 4]$ $v = [10 \quad 20 \quad 30]$ the convolution is  $c = \text{conv}(u, v)$ $c =$ <table style="margin-left: 100px;"> <tr><td>10</td><td>40</td><td>100</td><td>160</td><td>170</td><td>120</td></tr> </table> Use deconvolution to recover $u$ :  $[q, r] = \text{deconv}(c, u)$ $q =$ <table style="margin-left: 100px;"> <tr><td>10</td><td>20</td><td>30</td></tr> </table> $r =$ <table style="margin-left: 100px;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table> This gives a quotient equal to $v$ and a zero remainder.	10	40	100	160	170	120	10	20	30	0	0	0	0	0	0
10	40	100	160	170	120											
10	20	30														
0	0	0	0	0	0											
<b>Algorithm</b>	deconv uses the filter primitive.															
<b>See Also</b>	convmtx, conv2, and filter in the Signal Processing Toolbox, and:  <table style="width: 100%;"><tr> <td style="width: 30%; vertical-align: top;"> <a href="#">conv</a>  <a href="#">residue</a> </td> <td style="width: 70%; vertical-align: top;"> <a href="#">Convolution and polynomial multiplication</a>  <a href="#">Convert between partial fraction expansion and polynomial coefficients</a> </td> </tr></table>	<a href="#">conv</a> <a href="#">residue</a>	<a href="#">Convolution and polynomial multiplication</a> <a href="#">Convert between partial fraction expansion and polynomial coefficients</a>													
<a href="#">conv</a> <a href="#">residue</a>	<a href="#">Convolution and polynomial multiplication</a> <a href="#">Convert between partial fraction expansion and polynomial coefficients</a>															

## del2

<b>Purpose</b>	Discrete Laplacian
<b>Syntax</b>	$L = \text{del2}(U)$ $L = \text{del2}(U, h)$ $L = \text{del2}(U, hx, hy)$ $L = \text{del2}(U, hx, hy, hz, \dots)$
<b>Definition</b>	If the matrix $U$ is regarded as a function $u(x,y)$ evaluated at the point on a square grid, then $4*\text{del2}(U)$ is a finite difference approximation of Laplace's differential operator applied to $u$ , that is:
	$I = \frac{\nabla^2 u}{4} = \frac{1}{4} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} \right)$
	where:
	$I_{ij} = \frac{1}{4} (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) - u_{i,j}$
	in the interior. On the edges, the same formula is applied to a cubic extrapolation.
	For functions of more variables $u(x,y,z,\dots)$ , $\text{del2}(U)$ is an approximation,
	$I = \frac{\nabla^2 u}{2N} = \frac{1}{2N} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} + \frac{d^2 u}{dz^2} + \dots \right)$
	where $N$ is the number of variables in $u$ .
<b>Description</b>	$L = \text{del2}(U)$ where $U$ is a rectangular array is a discrete approximation of
	$I = \frac{\nabla^2 u}{4} = \frac{1}{4} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} \right)$
	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.

$L = \text{del2}(U)$  when  $U$  is an multidimensional array, returns an approximation of

$$\frac{\nabla^2 u}{2N}$$

where  $N$  is  $\text{ndims}(u)$ .

$L = \text{del2}(U, h)$  where  $H$  is a scalar uses  $H$  as the spacing between points in each direction ( $h=1$  by default).

$L = \text{del2}(U, hx, hy)$  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  $\text{size}(u, 2)$  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  $\text{size}(u, 1)$  and specifies the  $y$ -coordinates of the points.

$L = \text{del2}(U, hx, hy, hz, \dots)$  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,  $4 * \text{del2}(U)$  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
```

## del2

---

$V = 4 * \text{del2}(U)$

$V =$

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	4	4	4	4	4	4	4
4	4	4	4	4	4	4	4	4	4

### See Also

diff  
gradient

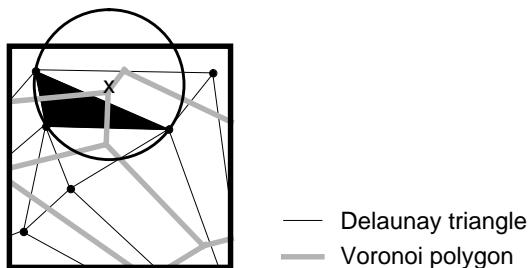
Differences and approximate derivatives  
Numerical gradient

**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.

# delaunay

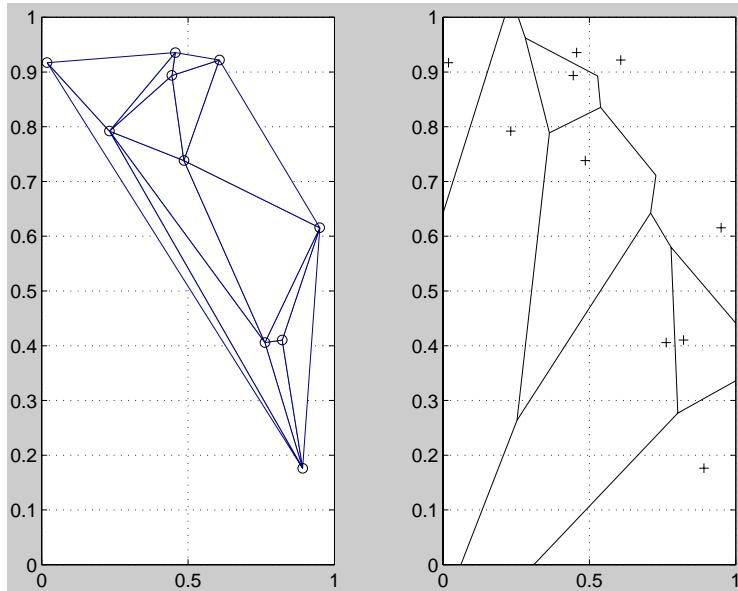
## Examples

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**  
**griddata**  
**tsearch**  
**voronoi**

Search for nearest point  
Data gridding  
Search for enclosing Delaunay triangle  
Voronoi diagram

# delete

---

<b>Purpose</b>	Delete files and graphics objects
<b>Syntax</b>	<code>delete filename</code> <code>delete(h)</code>
<b>Description</b>	<code>delete filename</code> deletes the named file. Wildcards may be used.  <code>delete(h)</code> 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 <code>delete('filename')</code> , when the filename is stored in a string.
<b>See Also</b>	<code>!</code> Operating system command <code>dir</code> Directory listing <code>type</code> List file

<b>Purpose</b>	Matrix determinant									
<b>Syntax</b>	<code>d = det(X)</code>									
<b>Description</b>	<code>d = det(X)</code> returns the determinant of the square matrix <code>X</code> . If <code>X</code> contains only integer entries, the result <code>d</code> is also an integer.									
<b>Remarks</b>	Using <code>det(X) == 0</code> as a test for matrix singularity is appropriate only for matrices of modest order with small integer entries. Testing singularity using <code>abs(det(X)) &lt;= tolerance</code> is not recommended as it is difficult to choose the correct tolerance. The function <code>cond(X)</code> can check for singular and nearly singular matrices.									
<b>Algorithm</b>	The determinant is computed from the triangular factors obtained by Gaussian elimination									
	<code>[L, U] = lu(A)</code> <code>s = det(L)</code> % This is always +1 or -1 <code>det(A) = s*prod(diag(U))</code>									
<b>Examples</b>	The statement <code>A = [1 2 3; 4 5 6; 7 8 9]</code> produces									
	<code>A =</code> <table style="margin-left: 20px;"> <tr><td>1</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>5</td><td>6</td></tr> <tr><td>7</td><td>8</td><td>9</td></tr> </table>	1	2	3	4	5	6	7	8	9
1	2	3								
4	5	6								
7	8	9								
	This happens to be a singular matrix, so <code>d = det(A)</code> produces <code>d = 0</code> . Changing <code>A(3, 3)</code> with <code>A(3, 3) = 0</code> turns <code>A</code> into a nonsingular matrix. Now <code>d = det(A)</code> produces <code>d = 27</code> .									
<b>See Also</b>	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 20%; vertical-align: top;"> <code>\</code>  <code>/</code>  <code>cond</code>  <code>condest</code>  <code>inv</code>  <code>lu</code>  <code>rref</code> </td> <td style="width: 80%; vertical-align: top;"> <p>Matrix left division (backslash)          Matrix right division (slash)          Condition number with respect to inversion          1-norm matrix condition number estimate          Matrix inverse          LU matrix factorization          Reduced row echelon form</p> </td> </tr> </table>	<code>\</code> <code>/</code> <code>cond</code> <code>condest</code> <code>inv</code> <code>lu</code> <code>rref</code>	<p>Matrix left division (backslash)          Matrix right division (slash)          Condition number with respect to inversion          1-norm matrix condition number estimate          Matrix inverse          LU matrix factorization          Reduced row echelon form</p>							
<code>\</code> <code>/</code> <code>cond</code> <code>condest</code> <code>inv</code> <code>lu</code> <code>rref</code>	<p>Matrix left division (backslash)          Matrix right division (slash)          Condition number with respect to inversion          1-norm matrix condition number estimate          Matrix inverse          LU matrix factorization          Reduced row echelon form</p>									

# diag

<b>Purpose</b>	Diagonal matrices and diagonals of a matrix
<b>Syntax</b>	$X = \text{diag}(v, k)$ $X = \text{diag}(v)$ $v = \text{diag}(X, k)$ $v = \text{diag}(X)$
<b>Description</b>	$X = \text{diag}(v, k)$ when $v$ is a vector of $n$ components, returns a square matrix $X$ of order $n + \text{abs}(k)$ , 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.
	$X = \text{diag}(v)$ puts $v$ on the main diagonal, same as above with $k = 0$ .
	$v = \text{diag}(X, k)$ for matrix $X$ , returns a column vector $v$ formed from the elements of the $k$ th diagonal of $X$ .
	$v = \text{diag}(X)$ returns the main diagonal of $X$ , same as above with $k = 0$ .
<b>Examples</b>	$\text{diag}(\text{diag}(X))$ is a diagonal matrix. $\text{sum}(\text{diag}(X))$ is the trace of $X$ . The statement $\text{diag}(-m:m) + \text{diag}(\text{ones}(2*m, 1), 1) + \text{diag}(\text{ones}(2*m, 1), -1)$ produces a tridiagonal matrix of order $2*m+1$ .
<b>See Also</b>	<code>spdiags</code> , <code>tril</code> , <code>triu</code>

---

<b>Purpose</b>	Save session in a disk file
<b>Syntax</b>	<code>di ary</code> <code>di ary <i>filename</i></code> <code>di ary off</code> <code>di ary on</code>
<b>Description</b>	The <code>di ary</code> command creates a log of keyboard input and system responses. The output of <code>di ary</code> is an ASCII file, suitable for printing or for inclusion in reports and other documents.  <code>di ary</code> , by itself, toggles <code>di ary</code> mode on and off.  <code>di ary <i>filename</i></code> 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.  <code>di ary off</code> suspends the diary.  <code>di ary on</code> resumes diary mode using the current filename, or the default filename <code>di ary</code> if none has yet been specified.
<b>Remarks</b>	The function form of the syntax, <code>di ary('filename')</code> , is also permitted.
<b>Limitations</b>	You cannot put a diary into the files named <code>off</code> and <code>on</code> .

# diff

---

<b>Purpose</b>	Differences and approximate derivatives
<b>Syntax</b>	$Y = \text{diff}(X)$ $Y = \text{diff}(X, n)$ $Y = \text{diff}(X, n, \text{dim})$
<b>Description</b>	$Y = \text{diff}(X)$ calculates differences between adjacent elements of $X$ . If $X$ is a vector, then $\text{diff}(X)$ returns a vector, one element shorter than $X$ , of differences between adjacent elements: $[X(2) - X(1) \ X(3) - X(2) \ \dots \ X(n) - X(n-1)]$ If $X$ is a matrix, then $\text{diff}(X)$ returns a matrix of column differences: $[X(2:m, :) - X(1:m-1, :)]$ In general, $\text{diff}(X)$ returns the differences calculated along the first non-singleton ( $\text{size}(X, \text{dim}) > 1$ ) dimension of $X$ . $Y = \text{diff}(X, n)$ applies $\text{diff}$ recursively $n$ times, resulting in the $n$ th difference. Thus, $\text{diff}(X, 2)$ is the same as $\text{diff}(\text{diff}(X))$ . $Y = \text{diff}(X, n, \text{dim})$ is the $n$ th difference function calculated along the dimension specified by scalar $\text{dim}$ . If order $n$ equals or exceeds the length of dimension $\text{dim}$ , $\text{diff}$ returns an empty array.
<b>Remarks</b>	Since each iteration of $\text{diff}$ reduces the length of $X$ along dimension $\text{dim}$ , it is possible to specify an order $n$ sufficiently high to reduce $\text{dim}$ to a singleton ( $\text{size}(X, \text{dim}) = 1$ ) dimension. When this happens, $\text{diff}$ continues calculating along the next nonsingleton dimension.

**Examples**

The quantity  $\text{diff}(y) ./ \text{diff}(x)$  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**

<code>gradient</code>	Approximate gradient.
<code>int</code>	Integrate (see Symbolic Toolbox).
<code>prod</code>	Product of array elements
<code>sum</code>	Sum of array elements

# dir

---

<b>Purpose</b>	Directory listing								
<b>Syntax</b>	<code>dir</code> <code>dir <i>dirname</i></code> <code>names = dir</code> <code>names = dir('dirname')</code>								
<b>Description</b>	<code>dir</code> , by itself, lists the files in the current directory.  <code>dir <i>dirname</i></code> lists the files in the specified directory. Use pathnames, wildcards, and any options available in your operating system.  <code>names = dir('dirname')</code> or <code>names = dir</code> returns the results in an m-by-1 structure with the fields:								
	<table><tr><td><code>name</code></td><td>Filename</td></tr><tr><td><code>date</code></td><td>Modification date</td></tr><tr><td><code>bytes</code></td><td>Number of bytes allocated to the file</td></tr><tr><td><code>isdir</code></td><td>1 if name is a directory; 0 if not</td></tr></table>	<code>name</code>	Filename	<code>date</code>	Modification date	<code>bytes</code>	Number of bytes allocated to the file	<code>isdir</code>	1 if name is a directory; 0 if not
<code>name</code>	Filename								
<code>date</code>	Modification date								
<code>bytes</code>	Number of bytes allocated to the file								
<code>isdir</code>	1 if name is a directory; 0 if not								
<b>Examples</b>	<pre>cd /Matlab/Tool box/Local; dir</pre> Contents.m matlabrc.m siteid.m userpath.m  <code>names = dir</code>  <code>names =</code>  4x1 struct array with fields: <table><tr><td><code>name</code></td></tr><tr><td><code>date</code></td></tr><tr><td><code>bytes</code></td></tr><tr><td><code>isdir</code></td></tr></table>	<code>name</code>	<code>date</code>	<code>bytes</code>	<code>isdir</code>				
<code>name</code>									
<code>date</code>									
<code>bytes</code>									
<code>isdir</code>									
<b>See Also</b>	<code>!</code> , <code>cd</code> , <code>delete</code> , <code>type</code> , <code>what</code>								

---

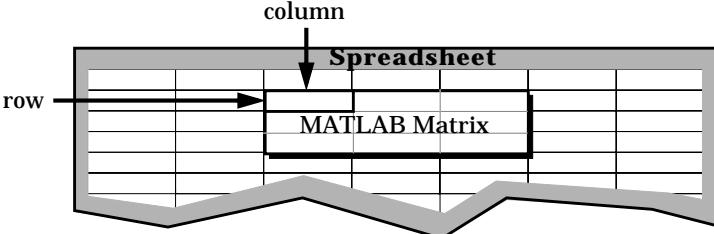
<b>Purpose</b>	Display text or array																		
<b>Syntax</b>	<code>disp(X)</code>																		
<b>Description</b>	<code>disp(X)</code> displays an array, without printing the array name. If <code>X</code> contains a text string, the string is displayed. Another way to display an array on the screen is to type its name, but this prints a leading “ <code>X =</code> ,” which is not always desirable.																		
<b>Examples</b>	One use of <code>disp</code> in an M-file is to display a matrix with column labels:  <code>disp(['Corn' 'Oats' 'Hay'])</code> <code>disp(rand(5, 3))</code>  which results in  <table><thead><tr><th>Corn</th><th>Oats</th><th>Hay</th></tr></thead><tbody><tr><td>0.2113</td><td>0.8474</td><td>0.2749</td></tr><tr><td>0.0820</td><td>0.4524</td><td>0.8807</td></tr><tr><td>0.7599</td><td>0.8075</td><td>0.6538</td></tr><tr><td>0.0087</td><td>0.4832</td><td>0.4899</td></tr><tr><td>0.8096</td><td>0.6135</td><td>0.7741</td></tr></tbody></table>	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
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																	
<b>See Also</b>	<code>format</code> Control the output display format <code>int2str</code> Integer to string conversion <code>num2str</code> Number to string conversion <code>rats</code> Rational fraction approximation <code>sprintf</code> Write formatted data to a string																		

# dlmread

<b>Purpose</b>	Read an ASCII delimited file into a matrix						
<b>Syntax</b>	<pre>M = dlmread(filename, delimiter) M = dlmread(filename, delimiter, r, c) M = dlmread(filename, delimiter, r, c, range)</pre>						
<b>Description</b>	<p><code>M = dlmread(filename, delimiter)</code> reads data from the ASCII delimited format <code>filename</code>, using the delimiter <code>delimiter</code>. Use '\t' to specify a tab.</p> <p><code>M = dlmread(filename, delimiter, r, c)</code> reads data from the ASCII delimited format <code>filename</code>, using the delimiter <code>delimiter</code>, starting at file offset <code>r</code> and <code>c</code>. <code>r</code> and <code>c</code> are zero based so that <code>r=0, c=0</code> specifies the first value in the file.</p> <p><code>M = dlmread(filename, delimiter, r, c, range)</code> imports an indexed or named range of ASCII-delimited data. To use the cell range, specify range by:</p> <pre>range = [UpperLeftRow UpperLeftColumn LowerRightRow LowerRightColumn]</pre>						
	<p>The diagram shows a 'Spreadsheet' represented as a grid of cells. A specific cell in the middle-left column is highlighted with a thick black border and labeled 'MATLAB Matrix'. An arrow points from the word 'row' to the top-left cell of the grid, indicating the starting point for row indexing. Another arrow points from the word 'column' to the top-left cell, indicating the starting point for column indexing. The grid has several rows and columns of empty cells below and to the right of the highlighted cell.</p>						
<b>Arguments</b>	<table><tr><td><code>delimiter</code></td><td>The character separating individual matrix elements in the ASCII- format spreadsheet file. A comma (,) is the default delimiter.</td></tr><tr><td><code>r, c</code></td><td>The spreadsheet cell from which the upper-left-most matrix element is taken.</td></tr><tr><td><code>range</code></td><td>A vector specifying a range of spreadsheet cells.</td></tr></table>	<code>delimiter</code>	The character separating individual matrix elements in the ASCII- format spreadsheet file. A comma (,) is the default delimiter.	<code>r, c</code>	The spreadsheet cell from which the upper-left-most matrix element is taken.	<code>range</code>	A vector specifying a range of spreadsheet cells.
<code>delimiter</code>	The character separating individual matrix elements in the ASCII- format spreadsheet file. A comma (,) is the default delimiter.						
<code>r, c</code>	The spreadsheet cell from which the upper-left-most matrix element is taken.						
<code>range</code>	A vector specifying a range of spreadsheet cells.						
<b>See Also</b>	<table><tr><td><code>dlmwrit</code></td><td>Write a matrix to an ASCII delimited file</td></tr><tr><td><code>wk1read</code></td><td>Read a Lotus123 WK1 spreadsheet file into a matrix</td></tr><tr><td><code>wk1write</code></td><td>Write a matrix to a Lotus123 WK1 spreadsheet file</td></tr></table>	<code>dlmwrit</code>	Write a matrix to an ASCII delimited file	<code>wk1read</code>	Read a Lotus123 WK1 spreadsheet file into a matrix	<code>wk1write</code>	Write a matrix to a Lotus123 WK1 spreadsheet file
<code>dlmwrit</code>	Write a matrix to an ASCII delimited file						
<code>wk1read</code>	Read a Lotus123 WK1 spreadsheet file into a matrix						
<code>wk1write</code>	Write a matrix to a Lotus123 WK1 spreadsheet file						



# dlmwrite

<b>Purpose</b>	Write a matrix to an ASCII delimited file
<b>Syntax</b>	<code>dlmwrite(filename, A, delimiter)</code> <code>dlmwrite(filename, A, delimiter, r, c)</code>
<b>Description</b>	The <code>dlmwrite</code> command converts a MATLAB matrix into an ASCII-format file readable by spreadsheet programs.
	<code>dlmwrite(filename, A, delimiter)</code> writes matrix A into the upper left-most cell of the ASCII-format spreadsheet file <code>filename</code> , and uses the delimiter to separate matrix elements. Specify ' <code>\t</code> ' 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 ' <code>1, , 2</code> ' when the delimiter is a comma.
	<code>dlmwrite(filename, A, delimiter, r, c)</code> writes A into <code>filename</code> , starting at spreadsheet cell r and c, with <code>delimiter</code> used to separate matrix elements.
<b>Arguments</b>	 <p><code>delimiter</code> The character separating individual matrix elements in the ASCII-format spreadsheet file. A comma (,) is the default delimiter.</p> <p><code>r, c</code> The spreadsheet cell into which the upper-left-most matrix element is written.</p>
<b>See Also</b>	<code>dlmread</code> Read an ASCII delimited file into a matrix <code>wk1read</code> Read a Lotus123 WK1 spreadsheet file into a matrix <code>wk1write</code> 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.

$p = \text{dmperm}(A)$  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*.

$[p, q, r] = \text{dmperm}(A)$  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  $k$ th block of  $A(p, q)$  has indices  $r(k) : r(k+1) - 1$ .

$[p, q, r, s] = \text{dmperm}(A)$ , 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

<b>Purpose</b>	Display HTML documentation in Web browser
<b>Syntax</b>	<code>doc</code> <code>doc <i>function</i></code> <code>doc <i>toolbox/funciton</i></code>
<b>Description</b>	<p><code>doc</code>, by itself, launches the Help Desk.</p> <p><code>doc <i>function</i></code> displays the HTML documentation for the MATLAB function <i>function</i>. If <i>function</i> is overloaded, <code>doc</code> lists the overloaded functions in the MATLAB command window.</p> <p><code>doc <i>toolbox/funciton</i></code> displays the HTML documentation for the specified toolbox function.</p>
<b>See Also</b>	<code>help</code> Online help for MATLAB functions and M-files <code>type</code> List file

---

<b>Purpose</b>	Convert to double precision
<b>Syntax</b>	<code>double(X)</code>
<b>Description</b>	<code>double(x)</code> returns the double precision value for <code>X</code> . If <code>X</code> is already a double precision array, <code>double</code> has no effect.
<b>Remarks</b>	<code>double</code> is called for the expressions in <code>for</code> , <code>if</code> , and <code>while</code> loops if the expression isn't already double precision. <code>double</code> should be overloaded for any object when it makes sense to convert it to a double precision value.

# dsearch

---

<b>Purpose</b>	Search for nearest point
<b>Syntax</b>	$K = \text{dsearch}(x, y, \text{TRI}, xi, yi)$ $K = \text{dsearch}(x, y, \text{TRI}, xi, yi, S)$
<b>Description</b>	$K = \text{dsearch}(x, y, \text{TRI}, xi, yi)$ 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.
	$K = \text{dsearch}(x, y, \text{TRI}, xi, yi, S)$ uses the sparse matrix S instead of computing it each time:  $S = \text{sparse}(\text{TRI}(:, [1 1 2 2 3 3]), \text{TRI}(:, [2 3 1 3 1 2]), 1, \text{nxy}, \text{nxy})$ where $\text{nxy} = \text{prod}(\text{size}(x))$ .

## See Also

[delaunay](#) Delaunay triangulation  
[tsearch](#) Search for enclosing Delaunay triangle  
[voronoi](#) Voronoi diagram

<b>Purpose</b>	Echo M-files during execution
<b>Syntax</b>	<code>echo on</code> <code>echo off</code> <code>echo</code> <code>echo <i>fcnname</i> on</code> <code>echo <i>fcnname</i> off</code> <code>echo <i>fcnname</i></code> <code>echo on all</code> <code>echo off all</code>
<b>Description</b>	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:  <code>echo on</code> Turns on the echoing of commands in all script files. <code>echo off</code> Turns off the echoing of commands in all script files. <code>echo</code> 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.  <code>echo <i>fcnname</i> on</code> Turns on echoing of the named function file. <code>echo <i>fcnname</i> off</code> Turns off echoing of the named function file. <code>echo <i>fcnname</i></code> Toggles the echo state of the named function file. <code>echo on all</code> Set echoing on for all function files. <code>echo off all</code> Set echoing off for all function files.
<b>See Also</b>	<code>function</code>

# edit

---

<b>Purpose</b>	Edit an M-file
<b>Syntax</b>	<code>edit</code> <code>edit <i>fun</i></code> <code>edit <i>file.ext</i></code> <code>edit <i>class/fun</i></code> <code>edit <i>private/fun</i></code> <code>edit <i>class/private/fun</i></code>
<b>Description</b>	<code>edit</code> opens a new editor window.  <code>edit <i>fun</i></code> opens the M-file <code>fun.m</code> in a text editor.  <code>edit <i>file.ext</i></code> opens the specified text file.  <code>edit <i>class/fun</i></code> , <code>edit <i>private/fun</i></code> , or <code>edit <i>class/private/fun</i></code> can be used to edit a method, private function, or private method (for the class named <code>class</code> .)

<b>Purpose</b>	Eigenvalues and eigenvectors
<b>Syntax</b>	$d = \text{eig}(A)$ $[V, D] = \text{eig}(A)$ $[V, D] = \text{eig}(A, 'nobalance')$ $d = \text{eig}(A, B)$ $[V, D] = \text{eig}(A, B)$
<b>Description</b>	<p><math>d = \text{eig}(A)</math> returns a vector of the eigenvalues of matrix A.</p> <p><math>[V, D] = \text{eig}(A)</math> produces matrices of eigenvalues (D) and eigenvectors (V) of matrix A, so that <math>A*V = V*D</math>. Matrix D is the <i>canonical form</i> of A—a diagonal matrix with A's eigenvalues on the main diagonal. Matrix V is the <i>modal matrix</i>—its columns are the eigenvectors of A.</p> <p>The eigenvectors are scaled so that the norm of each is 1.0. Use <math>[W, D] = \text{eig}(A')</math>; <math>W = W'</math> to compute the <i>left eigenvectors</i>, which satisfy <math>W*A = D*W</math>.</p> <p><math>[V, D] = \text{eig}(A, 'nobalance')</math> 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.</p> <p><math>d = \text{eig}(A, B)</math> returns a vector containing the generalized eigenvalues, if A and B are square matrices.</p> <p><math>[V, D] = \text{eig}(A, B)</math> produces a diagonal matrix D of generalized eigenvalues and a full matrix V whose columns are the corresponding eigenvectors so that <math>A*V = B*V*D</math>. The eigenvectors are scaled so that the norm of each is 1.0.</p>
<b>Remarks</b>	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  $\lambda$  is a scalar. The  $n$  values of  $\lambda$  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  $\lambda$ , 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  $\lambda$  is a scalar. The values of  $\lambda$  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

$B = [3 -2 -.9 2*\text{eps}; -2 4 -1 -\text{eps}; -\text{eps}/4 \text{eps}/2 -1 0; -.5 -.5 .1 1];$   
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**

<code>balance</code>	Improve accuracy of computed eigenvalues
<code>condeig</code>	Condition number with respect to eigenvalues
<code>hess</code>	Hessenberg form of a matrix
<code>qz</code>	QZ factorization for generalized eigenvalues
<code>schur</code>	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.

# eigs

---

**Purpose** Find a few eigenvalues and eigenvectors

## Syntax

```
d = eig(A)
d = eig('Afun', n)
d = eig(A, B, k, sigma, options)
d = eig('Afun', n, B, k, sigma, options)
[V, D] = eig(A, ...)
[V, D] = eig('Afun', n, ...)
[V, D, flag] = eig(A, ...)
[V, D, flag] = eig('Afun', n, ...)
```

## Description

`eig` solves the eigenvalue problem  $A \cdot v = \lambda \cdot v$  or the generalized eigenvalue problem  $A \cdot v = \lambda \cdot B \cdot v$ . Only a few selected eigenvalues, or eigenvalues and eigenvectors, are computed, in contrast to `eig`, which computes all eigenvalues and eigenvectors.

`eig(A)` or `eig('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, `eig('fft', ...)` is much faster than `eig(F, ...)`, where `F` is the explicit FFT matrix.

With one output argument, `d` is a vector containing `k` eigenvalues. With two output arguments, `V` is a matrix with `k` columns and `D` is a `k`-by-`k` diagonal matrix so that  $A \cdot V = V \cdot D$  or  $A \cdot V = B \cdot V \cdot D$ . With three output arguments, `flag` indicates whether or not the eigenvalues were computed to the desired tolerance. `flag = 0` indicates convergence; `flag = 1` indicates no convergence.

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 = min(n, 6) eigenvalues are computed.
sigma	A scalar shift or a two letter string. If sigma is not specified, the k eigenvalues largest in magnitude are computed. If sigma is 0, the k eigenvalues smallest in magnitude are computed. If sigma is a real or complex scalar, the <i>shift</i> , the k 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.)

**Note 1.** If sigma is a scalar with no fractional part, k must be specified first. For example, eigs(A, 2, 0) finds the two largest magnitude eigenvalues, not the six eigenvalues closest to 2.0, as you may have wanted.

**Note 2.** If sigma is exactly an eigenvalue of A, eigs will encounter problems when it performs divisions of the form  $1/(lambda - sigma)$ , where lambda is an approximation of an eigenvalue of A. Restart with eigs(A, sigma2), where sigma2 is close to, but not equal to, sigma.

The options structure specifies certain parameters in the algorithm.

# eigs

Parameter	Description	Default Value
options.tol	Convergence tolerance $\text{norm}(A*V - V*D) \leq \text{tol} * \text{norm}(A)$	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	<code>rand(n, 1) - .5</code>

## Remarks

`d = eigs(A, k)` is not a substitute for

```
d = eig(full(A))
d = sort(d)
d = d(end-k+1: end)
```

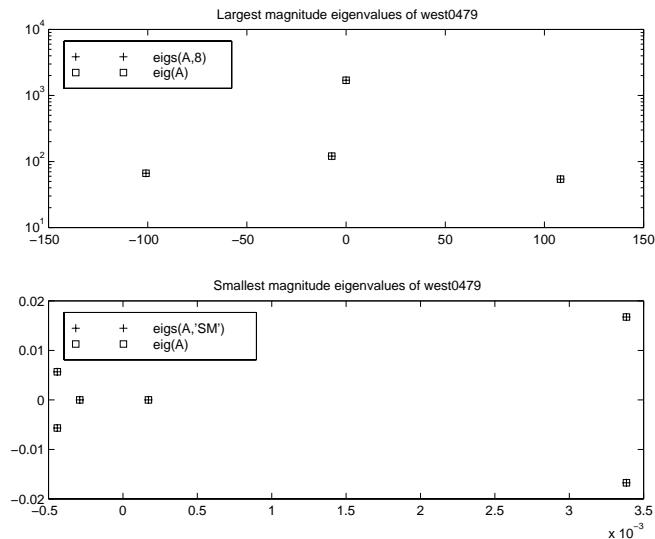
but is most appropriate for large sparse matrices. If the problem fits into memory, it may be quicker to use `eig(full(A))`.

**Examples****Example 1:**

`west0479` is a real 479-by-479 sparse matrix with both real and pairs of complex conjugate eigenvalues. `eig` computes all 479 eigenvalues. `eigs` easily picks out the smallest and largest magnitude eigenvalues.

```
load west0479
d = eig(full(west0479))
dlm = eigs(west0479, 8)
dsm = eigs(west0479, 'sm')
```

These plots show the eigenvalues of `west0479` as computed by `eig` and `eigs`. The first plot shows the four largest magnitude eigenvalues in the top half of the complex plane (but not their complex conjugates in the bottom half). The second subplot shows the six smallest magnitude eigenvalues.

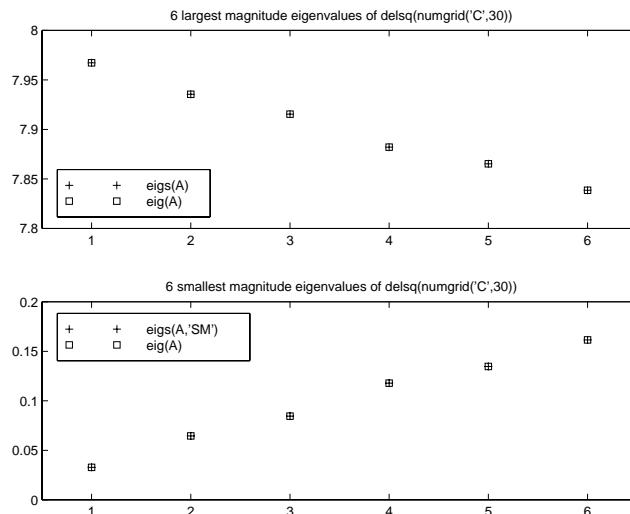


# eigs

## Example 2:

`A = delsq(numgrid('C', 30))` is a symmetric positive definite matrix of size 632 with eigenvalues reasonably well-distributed in the interval (0 8), but with 18 eigenvalues repeated at 4. `eig` computes all 632 eigenvalues. `eigs` computes the six largest and smallest magnitude eigenvalues of `A` successfully with:

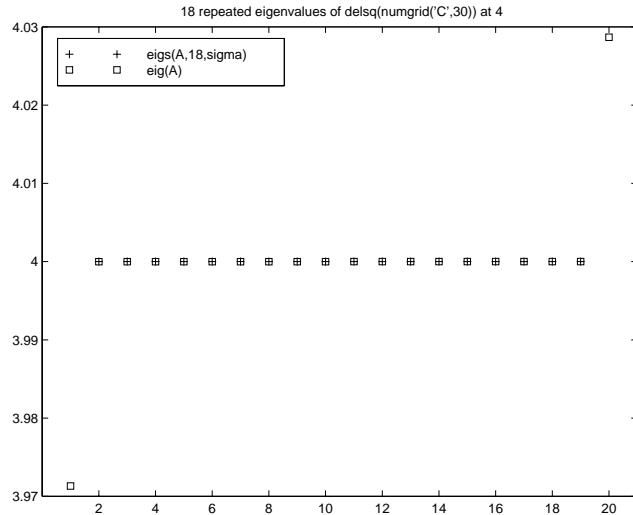
```
d = eig(full(A))
dlm = eigs(A)
dsm = eigs(A, 'sm')
```



However, the repeated eigenvalue at 4 must be handled more carefully. The call `eigs(A, 18, 4, 0)` to compute 18 eigenvalues near 4.0 tries to find eigenvalues of  $A - 4.0 \cdot I$ . This involves divisions of the form  $1 / (\lambda - 4.0)$ , where  $\lambda$  is an estimate of an eigenvalue of `A`. As  $\lambda$  gets closer to 4.0, `eigs` fails. We must use `sigma` near but not equal to 4 to find those 18 eigenvalues.

```
sigma = 4 - 1e-6
[V, D] = eigs(A, 18, sigma)
```

The plot shows the 20 eigenvalues closest to 4 that were computed by `eig`.



## See Also

`eig`  
`svds`

Eigenvalues and eigenvectors  
A few singular values

## References

- [1] R. Radke, "A MATLAB Implementation of the Implicitly Restarted Arnoldi Method for Solving Large-Scale Eigenvalue Problems," Dept. of Computational and Applied Math, Rice University, Houston, Texas.
- [2] D. C. Sorensen, "Implicit Application of Polynomial Filters in a k-step Arnoldi Method," *SIAM Journal on Matrix Analysis and Applications*, volume 13, number 1, 1992, pp 357-385.
- [3] R. B. Lehoucq and D. C. Sorensen, "Deflation Techniques within an Implicitly Restarted Iteration," *SIAM Journal on Matrix Analysis and Applications*, volume 17, 1996, pp 789-821.

# ellipj

<b>Purpose</b>	Jacobi elliptic functions
<b>Syntax</b>	$[SN, CN, DN] = \text{ellipj}(U, M)$ $[SN, CN, DN] = \text{ellipj}(U, M, tol)$
<b>Definition</b>	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
	$sn(u) = \sin \phi, \quad cn(u) = \cos \phi, \quad dn(u) = (1 - \sin^2 \phi)^{\frac{1}{2}}, \quad am(u) = \phi$
	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].
<b>Description</b>	$[SN, CN, DN] = \text{ellipj}(U, M)$ 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).
	$[SN, CN, DN] = \text{ellipj}(U, M, tol)$ computes the Jacobi elliptic functions to accuracy tol. The default is eps; increase this for a less accurate but more quickly computed answer.
<b>Algorithm</b>	<code>ellipj</code> 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:

$$\begin{aligned}a_i &= \frac{1}{2}(a_{i-1} + b_{i-1}) \\b_i &= (a_{i-1} b_{i-1})^{\frac{1}{2}} \\c_i &= \frac{1}{2}(a_{i-1} - b_{i-1})\end{aligned}$$

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:

$$\begin{aligned}\text{sn}(u) &= \sin\phi_0 \\ \text{cn}(u) &= \cos\phi_0 \\ \text{dn}(u) &= (1 - m \cdot \text{sn}(u)^2)^{\frac{1}{2}}\end{aligned}$$

## Limitations

The `ellipj` function is limited to the input domain  $0 \leq m \leq 1$ . 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.

# ellipke

**Purpose** Complete elliptic integrals of the first and second kind

**Syntax**  $K = \text{ellipke}(M)$

$[K, E] = \text{ellipke}(M)$

$[K, E] = \text{ellipke}(M, tol)$

**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,

$$E(m) = E(K(m)) = E(\pi/2|m),$$

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**  $K = \text{ellipke}(M)$  returns the complete elliptic integral of the first kind for the elements of  $M$ .

$[K, E] = \text{ellipke}(M)$  returns the complete elliptic integral of the first and second kinds.

$[K, E] = \text{ellipke}(M, tol)$  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:

$$a_0 = 1, \quad b_0 = (1 - m)^{\frac{1}{2}}, \quad c_0 = (m)^{\frac{1}{2}}$$

`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  $0 \leq m \leq 1$ .

**See Also**

`ellipj`      Jacobi elliptic functions

**References**

[1] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, 17.6.

# else

---

<b>Purpose</b>	Conditionally execute statements																
<b>Syntax</b>	<pre>if expression     statements else     statements end</pre>																
<b>Description</b>	The else command is used to delineate an alternate block of statements.																
	<pre>if expression     statements else     statements end</pre>																
	The second set of <i>statements</i> is executed if the <i>expression</i> has any zero elements. The expression is usually the result of <i>expression rop expression</i> where <i>rop</i> is ==, <, >, <=, >=, or ~=.																
<b>See Also</b>	<table><tr><td>break</td><td>Terminate execution of for or while loop</td></tr><tr><td>elseif</td><td>Conditionally execute statements</td></tr><tr><td>end</td><td>Terminate for, while, and if statements and indicate the last index</td></tr><tr><td>for</td><td>Repeat statements a specific number of times</td></tr><tr><td>if</td><td>Conditionally execute statements</td></tr><tr><td>return</td><td>Return to the invoking function</td></tr><tr><td>switch</td><td>Switch among several cases based on expression</td></tr><tr><td>while</td><td>Repeat statements an indefinite number of times</td></tr></table>	break	Terminate execution of for or while loop	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
break	Terminate execution of for or while loop																
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																

**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 i f, with a space between the el se and the i f, differs from el sei f, with no space. The former introduces a new, nested, i f, which must have a matching end. The latter is used in a linear sequence of conditional statements with only one terminating end.

# elseif

---

The two segments

```
if A
    x = a
else
    if B
        x = b
    else
        if C
            x = c
        else
            x = d
    end
end
end
```

```
if A
    x = a
elseif B
    x = b
elseif C
    x = c
else
    x = d
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	Terminate execution of for or while loop
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

<b>Purpose</b>	Terminate for, while, switch, try, and if statements or indicate last index														
<b>Syntax</b>	<pre>while expression% (or if, for, or try)     statements end</pre> <p>B = A(index: end, index)</p>														
<b>Description</b>	<p>end is used to terminate for, while, switch, try, and if statements. Without an end statement, for, while, switch, try, and if wait for further input. Each end is paired with the closest previous unpaired for, while, switch, try, or if and serves to delimit its scope.</p> <p>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 of this use are X(3: end) and X(1, 1: 2: end- 1). When using end to grow an array, as in X(end+1)=5, make sure X exists first.</p>														
<b>Examples</b>	<p>This example shows end used with for and if. Indentation provides easier readability.</p> <pre>for i = 1: n     if a(i) == 0         a(i) = a(i) + 2;     end end</pre> <p>Here, end is used in an indexing expression:</p> <pre>A = rand(5, 4) B = A(end, 2: end)</pre> <p>In this example, B is a 1-by-3 vector equal to [A(5, 2) A(5, 3) A(5, 4)].</p>														
<b>See Also</b>	<table> <tr> <td>break</td> <td>Terminate execution of for or while loop</td> </tr> <tr> <td>for</td> <td>Repeat statements a specific number of times</td> </tr> <tr> <td>if</td> <td>Conditionally execute statements</td> </tr> <tr> <td>return</td> <td>Return to the invoking function</td> </tr> <tr> <td>switch</td> <td>Switch among several cases based on expression</td> </tr> <tr> <td>try</td> <td>Begin try block</td> </tr> <tr> <td>while</td> <td>Repeat statements an indefinite number of times</td> </tr> </table>	break	Terminate execution of for or while loop	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	try	Begin try block	while	Repeat statements an indefinite number of times
break	Terminate execution of for or while loop														
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														
try	Begin try block														
while	Repeat statements an indefinite number of times														

# eomday

<b>Purpose</b>	End of month																														
<b>Syntax</b>	<code>E = eomday(Y, M)</code>																														
<b>Description</b>	<code>E = eomday(Y, M)</code> returns the last day of the year and month given by corresponding elements of arrays Y and M																														
<b>Examples</b>	<p>Because 1996 is a leap year, the statement <code>eomday(1996, 2)</code> returns 29.</p> <p>To show all the leap years in this century, try:</p> <pre>y = 1900:1999; E = eomday(y, 2*ones(length(y), 1)'); y(find(E==29))'</pre> <p><code>ans =</code></p> <table><thead><tr><th colspan="6">Columns 1 through 6</th></tr><tr><th>1904</th><th>1908</th><th>1912</th><th>1916</th><th>1920</th><th>1924</th></tr></thead><tbody><tr><td>1928</td><td>1932</td><td>1936</td><td>1940</td><td>1944</td><td>1948</td></tr><tr><td>1952</td><td>1956</td><td>1960</td><td>1964</td><td>1968</td><td>1972</td></tr><tr><td>1976</td><td>1980</td><td>1984</td><td>1988</td><td>1992</td><td>1996</td></tr></tbody></table>	Columns 1 through 6						1904	1908	1912	1916	1920	1924	1928	1932	1936	1940	1944	1948	1952	1956	1960	1964	1968	1972	1976	1980	1984	1988	1992	1996
Columns 1 through 6																															
1904	1908	1912	1916	1920	1924																										
1928	1932	1936	1940	1944	1948																										
1952	1956	1960	1964	1968	1972																										
1976	1980	1984	1988	1992	1996																										
<b>See Also</b>	<table><tr><td><code>datenum</code></td><td>Serial date number</td></tr><tr><td><code>datevec</code></td><td>Date components</td></tr><tr><td><code>weekday</code></td><td>Day of the week</td></tr></table>	<code>datenum</code>	Serial date number	<code>datevec</code>	Date components	<code>weekday</code>	Day of the week																								
<code>datenum</code>	Serial date number																														
<code>datevec</code>	Date components																														
<code>weekday</code>	Day of the week																														

<b>Purpose</b>	Floating-point relative accuracy
<b>Syntax</b>	eps
<b>Description</b>	eps returns the distance from 1.0 to the next largest floating-point number. The value eps is a default tolerance for <code>pi</code> , <code>inv</code> and <code>rank</code> , as well as several other MATLAB functions. On machines with IEEE floating-point arithmetic, <code>eps = 2^(-52)</code> , which is roughly <code>2.22e-16</code> .
<b>See Also</b>	<code>real max</code> Largest positive floating-point number <code>real min</code> Smallest positive floating-point number

## **erf, erfc, erfcx, erfinv**

<b>Purpose</b>	Error functions
<b>Syntax</b>	$Y = \text{erf}(X)$ $Y = \text{erfc}(X)$ $Y = \text{erfcx}(X)$ $X = \text{erfinv}(Y)$
	Error function Complementary error function Scaled complementary error function Inverse of the error function
<b>Definition</b>	The error function $\text{erf}(X)$ is twice the integral of the Gaussian distribution with 0 mean and variance of $1/2$ :
	$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
	The complementary error function $\text{erfc}(X)$ is defined as:
	$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt = 1 - \text{erf}(x)$
	The scaled complementary error function $\text{erfcx}(X)$ is defined as:
	$\text{erfcx}(x) = e^{x^2} \text{erfc}(x)$
	For large X, $\text{erfcx}(X)$ is approximately $\left(\frac{1}{\sqrt{\pi}}\right)_X^1$ .
<b>Description</b>	$Y = \text{erf}(X)$ returns the value of the error function for each element of real array X.  $Y = \text{erfc}(X)$ computes the value of the complementary error function.  $Y = \text{erfcx}(X)$ computes the value of the scaled complementary error function.  $X = \text{erfinv}(Y)$ returns the value of the inverse error function for each element of Y. The elements of Y must fall within the domain $-1 < Y < 1$ .
<b>Examples</b>	$\text{erfinv}(1)$ is Inf $\text{erfinv}(-1)$ is -Inf.  For $\text{abs}(Y) > 1$ , $\text{erfinv}(Y)$ 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

# error

---

<b>Purpose</b>	Display error messages								
<b>Syntax</b>	<code>error('error_message')</code>								
<b>Description</b>	<code>error('error_message')</code> displays an error message and returns control to the keyboard. The error message contains the input string <code>error_message</code> . The error command has no effect if <code>error_message</code> is a null string.								
<b>Examples</b>	The error command provides an error return from M-files.  <pre>function foo(x,y) if nargin ~= 2     error('Wrong number of input arguments') end</pre> The returned error message looks like:  <pre>» foo(pi) ??? Error using ==&gt; foo Wrong number of input arguments</pre>								
<b>See Also</b>	<table><tr><td><code>dbstop</code></td><td>Set breakpoints in an M-file function</td></tr><tr><td><code>disp</code></td><td>Display text or array</td></tr><tr><td><code>lasterr</code></td><td>Last error message</td></tr><tr><td><code>warning</code></td><td>Display warning message</td></tr></table>	<code>dbstop</code>	Set breakpoints in an M-file function	<code>disp</code>	Display text or array	<code>lasterr</code>	Last error message	<code>warning</code>	Display warning message
<code>dbstop</code>	Set breakpoints in an M-file function								
<code>disp</code>	Display text or array								
<code>lasterr</code>	Last error message								
<code>warning</code>	Display warning message								

<b>Purpose</b>	Continue execution after errors during testing
<b>Syntax</b>	<code>errortrap on</code> <code>errortrap off</code>
<b>Description</b>	<code>errortrap on</code> continues execution after errors when they occur. Execution continues with the next statement in a top level script.  <code>errortrap off</code> (the default) stops execution when an error occurs.

# etime

---

<b>Purpose</b>	Elapsed time						
<b>Syntax</b>	<code>e = etime(t2, t1)</code>						
<b>Description</b>	<code>e = etime(t2, t1)</code> returns the time in seconds between vectors <code>t1</code> and <code>t2</code> . The two vectors must be six elements long, in the format returned by <code>clock</code> :						
	<code>T = [Year Month Day Hour Minute Second]</code>						
<b>Examples</b>	Calculate how long a 2048-point real FFT takes.						
	<pre>x = rand(2048, 1); t = clock; fft(x); etime(clock, t) ans =     0.4167</pre>						
<b>Limitations</b>	As currently implemented, the <code>etime</code> function fails across month and year boundaries. Since <code>etime</code> is an M-file, you can modify the code to work across these boundaries if needed.						
<b>See Also</b>	<table><tr><td><code>clock</code></td><td>Current time as a date vector</td></tr><tr><td><code>cputime</code></td><td>Elapsed CPU time</td></tr><tr><td><code>tic, toc</code></td><td>Stopwatch timer</td></tr></table>	<code>clock</code>	Current time as a date vector	<code>cputime</code>	Elapsed CPU time	<code>tic, toc</code>	Stopwatch timer
<code>clock</code>	Current time as a date vector						
<code>cputime</code>	Elapsed CPU time						
<code>tic, toc</code>	Stopwatch timer						

<b>Purpose</b>	Interpret strings containing MATLAB expressions
<b>Syntax</b>	<pre>a = eval (' expression ') [a1, a2, a3. . . ] = eval (' expression ') eval (string, catchstring)</pre>
<b>Description</b>	<p><code>a = eval (' expression ')</code> returns the value of <i>expression</i>, a MATLAB expression, enclosed in single quotation marks. Create '<i>expression</i>' by concatenating substrings and variables inside square brackets.</p> <p><code>[a1, a2, a3. . . ] = eval (' expression ')</code> evaluates and returns the results in separate variables. Use of this syntax is recommended over:</p> <pre>eval (' [a1, a2, a3. . . ] = expression ')</pre> <p>which hides information from the MATLAB parser and can produce unexpected behavior.</p> <p><code>eval (string, catchstring)</code> provides the ability to catch errors. It executes <i>string</i> and returns if the operation was successful. If the operation generates an error, <i>catchstring</i> is evaluated before returning. Use <code>lasterr</code> to obtain the error string produced by <i>string</i>.</p>
<b>Examples</b>	<pre>A = '1+4'; eval (A) ans = 5</pre> <pre>P = 'pwd'; eval (P) ans = /home/myname</pre> <p>The loop</p> <pre>for n = 1:12     eval(['M', int2str(n), ' = magic(n)']) end</pre> <p>generates a sequence of 12 matrices named M1 through M12.</p>

# eval

---

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](#)

[lasterr](#)

[Function evaluation](#)

Last error message.

<b>Purpose</b>	Evaluate expression in workspace
<b>Syntax</b>	<code>evalin(ws, 'expression')</code> <code>[X, Y, Z, ...] = evalin(ws, 'expression')</code> <code>evalin(ws, 'try', 'catch')</code>
<b>Description</b>	<code>evalin(ws, 'expression')</code> evaluates <i>expression</i> in the context of the workspace <i>ws</i> . <i>ws</i> can be either 'caller' or 'base'.  <code>[X, Y, Z, ...] = evalin(ws, 'expression')</code> returns output arguments from the expression.  <code>evalin(ws, 'try', 'catch')</code> tries to evaluate the <i>try</i> expression and if that fails it evaluates the <i>catch</i> expression in the specified workspace.  <code>evalin</code> is useful for getting values from another workspace while <code>assignin</code> is useful for depositing values into another workspace.
<b>Limitation</b>	<code>evalin</code> may not be used recursively to evaluate an expression, i.e., a sequence of the form <code>evalin('caller', 'evalin(''caller'', ''expression''))</code> doesn't work.
<b>See Also</b>	<code>assignin</code> Assign variable in workspace. <code>eval</code> Interpret strings containing MATLAB expressions

# exist

---

<b>Purpose</b>	Check if a variable or file exists
<b>Syntax</b>	<pre>a = exist('item') ident = exist('item', kind)</pre>
<b>Description</b>	<p><code>a = exist('item')</code> returns the status of the variable or file <code>item</code>.</p> <ul style="list-style-type: none"><li>0     If <code>item</code> does not exist.</li><li>1     If the variable <code>item</code> exists in the workspace.</li><li>2     If <code>item</code> is an M-file or a file of unknown type.</li><li>3     If <code>item</code> is a MEX-file.</li><li>4     If <code>item</code> is a MDL-file.</li><li>5     If <code>item</code> is a built-in MATLAB function.</li><li>6     If <code>item</code> is a P-file.</li><li>7     If <code>item</code> is a directory.</li></ul> <p><code>exist('item')</code> or <code>exist('item.ext')</code> returns 2 if <code>item</code> is on the MATLAB search path but the filename extension (<code>ext</code>) is not <code>mdl</code>, <code>p</code>, or <code>mex</code>. <code>item</code> may be a MATLABPATH relative partial pathname.</p> <p><code>ident = exist('item', 'kind')</code> returns logical true (1) if an item of the specified <code>kind</code> is found, and returns 0 otherwise. <code>kind</code> may be:</p> <ul style="list-style-type: none"><li>'var'       Checks only for variables.</li><li>'built-in'    Checks only for built-in functions.</li><li>'file'       Checks only for files.</li><li>'dir'       Checks only for directories.</li></ul>

**Examples**

`exist` can check whether a MATLAB function is built-in or a file:

```
i dent = exist('plot')
i dent =
5
```

`plot` is a built-in function.

**See Also**

<code>dir</code>	Directory listing
<code>help</code>	Online help for MATLAB functions and M-files
<code>lookfor</code>	Keyword search through all help entries
<code>what</code>	Directory listing of M-files, MAT-files, and MEX-files
<code>which</code>	Locate functions and files
<code>who</code>	List directory of variables in memory

See also `partialpath`.

# exp

---

<b>Purpose</b>	Exponential								
<b>Syntax</b>	$Y = \exp(X)$								
<b>Description</b>	The <code>exp</code> 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 $z = x + iy$ , it returns the complex exponential: $e^z = e^x(\cos(y) + i\sin(y))$ .								
<b>Remark</b>	Use <code>expm</code> for matrix exponentials.								
<b>See Also</b>	<table><tr><td><code>expm</code></td><td>Matrix exponential</td></tr><tr><td><code>log</code></td><td>Natural logarithm</td></tr><tr><td><code>log10</code></td><td>Common (base 10) logarithm</td></tr><tr><td><code>expint</code></td><td>Exponential integral</td></tr></table>	<code>expm</code>	Matrix exponential	<code>log</code>	Natural logarithm	<code>log10</code>	Common (base 10) logarithm	<code>expint</code>	Exponential integral
<code>expm</code>	Matrix exponential								
<code>log</code>	Natural logarithm								
<code>log10</code>	Common (base 10) logarithm								
<code>expint</code>	Exponential integral								

**Purpose** Exponential integral

**Syntax** `Y = expi nt(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 `expi nt` as follows:

$$\begin{aligned} \text{expi nt}(-x+i*0) &= -Ei(x) - i*pi \\ Ei(x) &= \text{real}(\text{expi nt}(-x)) \end{aligned}$$

**Description** `Y = expi nt(X)` evaluates the exponential integral for each element of  $X$ .

**Algorithm** For elements of  $X$  in the domain  $[-38, 2]$ , `expi nt` 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  $X$ , `expi nt` uses a continued fraction representation (equation 5.1.22 in [1]):

$$E_n(z) = e^{-z} \left( \frac{1}{z+} \frac{n}{1+} \frac{1}{z+} \frac{n+1}{1+} \frac{2}{z+} \dots \right) |angle(z)| < \pi$$

## References

- [1] Abramowitz, M. and I. A. Stegun. *Handbook of Mathematical Functions*. Chapter 5, New York: Dover Publications, 1965.

<b>Purpose</b>	Matrix exponential
<b>Syntax</b>	$Y = \text{expm}(X)$
<b>Description</b>	$Y = \text{expm}(X)$ raises the constant $e$ to the matrix power $X$ . Complex results are produced if $X$ has nonpositive eigenvalues. Use <code>exp</code> for the element-by-element exponential.
<b>Algorithm</b>	The <code>expm</code> function is built-in, but it uses the Padé approximation with scaling and squaring algorithm expressed in the file <code>expm1.m</code> . A second method of calculating the matrix exponential uses a Taylor series approximation. This method is demonstrated in the file <code>expm2.m</code> . 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 <code>expm3.m</code> , 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 $\text{expm}(X)$ . The built-in method, <code>expm1</code> , is essentially method 3 of [2].
<b>Examples</b>	Suppose $A$ is the 3-by-3 matrix $\begin{matrix} 1 & 1 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & -1 \end{matrix}$ then $\text{expm}(A)$ is $\begin{matrix} 2.7183 & 1.7183 & 1.0862 \\ 0 & 1.0000 & 1.2642 \\ 0 & 0 & 0.3679 \end{matrix}$ while $\text{exp}(A)$ is $\begin{matrix} 2.7183 & 2.7183 & 1.0000 \\ 1.0000 & 1.0000 & 7.3891 \\ 1.0000 & 1.0000 & 0.3679 \end{matrix}$

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

<code>exp</code>	Exponential
<code>funm</code>	Evaluate functions of a matrix
<code>logm</code>	Matrix logarithm
<code>sqrtm</code>	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.

<b>Purpose</b>	Identity matrix
<b>Syntax</b>	$Y = \text{eye}(n)$ $Y = \text{eye}(m, n)$ $Y = \text{eye}(\text{size}(A))$
<b>Description</b>	$Y = \text{eye}(n)$ returns the $n$ -by- $n$ identity matrix. $Y = \text{eye}(m, n)$ or $\text{eye}([m n])$ returns an $m$ -by- $n$ matrix with 1's on the diagonal and 0's elsewhere. $Y = \text{eye}(\text{size}(A))$ returns an identity matrix the same size as $A$ .
<b>Limitations</b>	The identity matrix is not defined for higher-dimensional arrays. The assignment $y = \text{eye}([2, 3, 4])$ results in an error.
<b>See Also</b>	<a href="#">ones</a> Create an array of all ones <a href="#">rand</a> Uniformly distributed random numbers and arrays <a href="#">randn</a> Normally distributed random numbers and arrays <a href="#">zeros</a> Create an array of all zeros

**eye**

---

---

<b>Purpose</b>	Prime factors
<b>Syntax</b>	<code>f = factor(n)</code> <code>f = factor(symb)</code>
<b>Description</b>	<code>f = factor(n)</code> returns a row vector containing the prime factors of n.
<b>Examples</b>	<code>f = factor(123)</code> <code>f =</code> 3        41
<b>See Also</b>	<code>isprime</code> True for prime numbers <code>primes</code> Generate list of prime numbers

# fclose

---

<b>Purpose</b>	Close one or more open files																
<b>Syntax</b>	<pre>status = fclose(fid) status = fclose('all')</pre>																
<b>Description</b>	<p><code>status = fclose(fid)</code> closes the specified file, if it is open, returning 0 if successful and -1 if unsuccessful. Argument <code>fid</code> is a file identifier associated with an open file (See <code>fopen</code> for a complete description).</p> <p><code>status = fclose('all')</code> closes all open files, (except standard input, output, and error), returning 0 if successful and -1 if unsuccessful.</p>																
<b>See Also</b>	<table><tr><td><code>ferror</code></td><td>Query MATLAB about errors in file input or output</td></tr><tr><td><code>fopen</code></td><td>Open a file or obtain information about open files</td></tr><tr><td><code>fprintf</code></td><td>Write formatted data to file</td></tr><tr><td><code>fread</code></td><td>Read binary data from file</td></tr><tr><td><code>fscanf</code></td><td>Read formatted data from file</td></tr><tr><td><code>fseek</code></td><td>Set file position indicator</td></tr><tr><td><code>ftell</code></td><td>Get file position indicator</td></tr><tr><td><code>fwrite</code></td><td>Write binary data from a MATLAB matrix to a file</td></tr></table>	<code>ferror</code>	Query MATLAB about errors in file input or output	<code>fopen</code>	Open a file or obtain information about open files	<code>fprintf</code>	Write formatted data to file	<code>fread</code>	Read binary data from file	<code>fscanf</code>	Read formatted data from file	<code>fseek</code>	Set file position indicator	<code>ftell</code>	Get file position indicator	<code>fwrite</code>	Write binary data from a MATLAB matrix to a file
<code>ferror</code>	Query MATLAB about errors in file input or output																
<code>fopen</code>	Open a file or obtain information about open files																
<code>fprintf</code>	Write formatted data to file																
<code>fread</code>	Read binary data from file																
<code>fscanf</code>	Read formatted data from file																
<code>fseek</code>	Set file position indicator																
<code>ftell</code>	Get file position indicator																
<code>fwrite</code>	Write binary data from a MATLAB matrix to a file																

---

<b>Purpose</b>	Test for end-of-file
<b>Syntax</b>	<code>eofstat = feof(fid)</code>
<b>Description</b>	<code>eofstat = feof(fid)</code> tests whether the end-of-file indicator is set for the file with identifier <code>fid</code> . It returns 1 if the end-of-file indicator is set, or 0 if it is not. (See <code>fopen</code> for a complete description of <code>fid</code> .) The end-of-file indicator is set when there is no more input from the file.
<b>See Also</b>	<a href="#">fopen</a> Open a file or obtain information about open files

# ferror

---

<b>Purpose</b>	Query MATLAB about errors in file input or output																
<b>Syntax</b>	<pre>message = ferror(fid) message = ferror(fid, 'clear') [message, errnum] = ferror(...)</pre>																
<b>Description</b>	<p><code>message = ferror(fid)</code> returns the error message <code>message</code>. Argument <code>fid</code> is a file identifier associated with an open file (See <code>fopen</code> for a complete description).</p> <p><code>message = ferror(fid, 'clear')</code> clears the error indicator for the specified file.</p> <p><code>[message, errnum] = ferror(...)</code> returns the error status number <code>errnum</code> of the most recent file I/O operation associated with the specified file.</p> <p>If the most recent I/O operation performed on the specified file was successful, the value of <code>message</code> is empty and <code>ferror</code> returns an <code>errnum</code> value of 0.</p> <p>A nonzero <code>errnum</code> indicates that an error occurred in the most recent file I/O operation. The value of <code>message</code> 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.</p>																
<b>See Also</b>	<table><tr><td><code>fclose</code></td><td>Close one or more open files</td></tr><tr><td><code>fopen</code></td><td>Open a file or obtain information about open files</td></tr><tr><td><code>fprintf</code></td><td>Write formatted data to file</td></tr><tr><td><code>fread</code></td><td>Read binary data from file</td></tr><tr><td><code>fscanf</code></td><td>Read formatted data from file</td></tr><tr><td><code>fseek</code></td><td>Set file position indicator</td></tr><tr><td><code>ftell</code></td><td>Get file position indicator</td></tr><tr><td><code>fwrite</code></td><td>Write binary data from a MATLAB matrix to a file</td></tr></table>	<code>fclose</code>	Close one or more open files	<code>fopen</code>	Open a file or obtain information about open files	<code>fprintf</code>	Write formatted data to file	<code>fread</code>	Read binary data from file	<code>fscanf</code>	Read formatted data from file	<code>fseek</code>	Set file position indicator	<code>ftell</code>	Get file position indicator	<code>fwrite</code>	Write binary data from a MATLAB matrix to a file
<code>fclose</code>	Close one or more open files																
<code>fopen</code>	Open a file or obtain information about open files																
<code>fprintf</code>	Write formatted data to file																
<code>fread</code>	Read binary data from file																
<code>fscanf</code>	Read formatted data from file																
<code>fseek</code>	Set file position indicator																
<code>ftell</code>	Get file position indicator																
<code>fwrite</code>	Write binary data from a MATLAB matrix to a file																

<b>Purpose</b>	Function evaluation
<b>Syntax</b>	$[y_1, y_2, \dots] = \text{feval}(\text{function}, x_1, \dots, x_n)$
<b>Description</b>	$[y_1, y_2, \dots] = \text{feval}(\text{function}, x_1, \dots, x_n)$ If <i>function</i> is a string containing the name of a function (usually defined by an M-file), then $\text{feval}(\text{function}, x_1, \dots, x_n)$ evaluates that function at the given arguments.
<b>Examples</b>	The statements:  <code>[V, D] = feval ('eig', A)</code> <code>[V, D] = eig(A)</code>  are equivalent. <code>feval</code> is useful in functions that accept string arguments specifying function names. For example, the function:  <code>function plotf(fun, x)</code> <code>y = feval(fun, x);</code> <code>plot(x, y)</code>  can be used to graph other functions.
<b>See Also</b>	<code>assignin</code> Assign value to variable in workspace <code>builtin</code> Execute builtin function from overloaded method <code>eval</code> Interpret strings containing MATLAB expressions <code>evalin</code> Evaluate expression in workspace

<b>Purpose</b>	One-dimensional fast Fourier transform
<b>Syntax</b>	$Y = \text{fft}(X)$ $Y = \text{fft}(X, n)$ $Y = \text{fft}(X, [], \text{dim})$ $Y = \text{fft}(X, n, \text{dim})$
<b>Definition</b>	The functions $X = \text{fft}(x)$ and $x = \text{ifft}(X)$ 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 $n$ th root of unity.
<b>Description</b>	<p><math>Y = \text{fft}(X)</math> returns the discrete Fourier transform of vector <math>X</math>, computed with a fast Fourier transform (FFT) algorithm.</p> <p>If <math>X</math> is a matrix, <math>\text{fft}</math> returns the Fourier transform of each column of the matrix.</p> <p>If <math>X</math> is a multidimensional array, <math>\text{fft}</math> operates on the first nonsingleton dimension.</p> <p><math>Y = \text{fft}(X, n)</math> returns the <math>n</math>-point FFT. If the length of <math>X</math> is less than <math>n</math>, <math>X</math> is padded with trailing zeros to length <math>n</math>. If the length of <math>X</math> is greater than <math>n</math>, the sequence <math>X</math> is truncated. When <math>X</math> is a matrix, the length of the columns are adjusted in the same manner.</p> <p><math>Y = \text{fft}(X, [], \text{dim})</math> and <math>Y = \text{fft}(X, n, \text{dim})</math> apply the FFT operation across the dimension <math>\text{dim}</math>.</p>

<b>Remarks</b>	The <code>fft</code> 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.”
<b>Examples</b>	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:
	<pre>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))</pre> <p>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 <code>y</code> is found by taking the 512-point fast Fourier transform (FFT):</p> <pre>Y = fft(y, 512);</pre> <p>The power spectral density, a measurement of the energy at various frequencies, is</p> <pre>Pyy = Y.*conj(Y) / 512;</pre> <p>Graph the first 257 points (the other 255 points are redundant) on a meaningful frequency axis.</p> <pre>f = 1000*(0:256)/512; plot(f, Pyy(1:257))</pre> <p>This represents the frequency content of <code>y</code> in the range from DC up to and including the Nyquist frequency. (The signal produces the strong peaks.)</p>

**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:

<code>fft2</code>	Two-dimensional fast Fourier transform
<code>fftshift</code>	Rearrange the outputs of <code>fft</code> and <code>fft2</code>
<code>ifft</code>	Inverse one-dimensional fast Fourier transform

---

<b>Purpose</b>	Two-dimensional fast Fourier transform
<b>Syntax</b>	$Y = \text{fft2}(X)$ $Y = \text{fft2}(X, m, n)$
<b>Description</b>	$Y = \text{fft2}(X)$ performs the two-dimensional FFT. The result $Y$ is the same size as $X$ .  $Y = \text{fft2}(X, m, n)$ truncates $X$ , or pads $X$ with zeros to create an $m$ -by- $n$ array before doing the transform. The result is $m$ -by- $n$ .
<b>Algorithm</b>	$\text{fft2}(X)$ can be simply computed as  $\text{fft}(\text{fft}(X) . ') . '$  This computes the one-dimensional FFT of each column $X$ , then of each row of the result. The time required to compute $\text{fft2}(X)$ depends strongly on the number of prime factors in $[m, n] = \text{size}(X)$ . It is fastest when $m$ and $n$ are powers of 2.
<b>See Also</b>	<a href="#">fft</a> One-dimensional fast Fourier transform <a href="#">fftshift</a> Rearrange the outputs of fft and fft2 <a href="#">ifft2</a> Inverse two-dimensional fast Fourier transform

# fftn

---

<b>Purpose</b>	Multidimensional fast Fourier transform
<b>Syntax</b>	$Y = \text{fftn}(X)$ $Y = \text{fftn}(X, \text{size})$
<b>Description</b>	$Y = \text{fftn}(X)$ performs the N-dimensional fast Fourier transform. The result $Y$ is the same size as $X$ .  $Y = \text{fftn}(X, \text{size})$ pads $X$ with zeros, or truncates $X$ , to create a multidimensional array of size $\text{size}$ before performing the transform. The size of the result $Y$ is $\text{size}$ .
<b>Algorithm</b>	$\text{fftn}(X)$ is equivalent to <pre>Y = X; for p = 1:length(size(X))     Y = fft(Y, [], p); end</pre> This computes in-place the one-dimensional fast Fourier transform along each dimension of $X$ . The time required to compute $\text{fftn}(X)$ 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.
<b>See Also</b>	<a href="#">fft</a> One-dimensional fast Fourier transform <a href="#">fft2</a> Two-dimensional fast Fourier transform <a href="#">ifftn</a> Inverse multidimensional fast Fourier transform

<b>Purpose</b>	Shift DC component of fast Fourier transform to center of spectrum
<b>Syntax</b>	<code>Y = fftshift(X)</code>
<b>Description</b>	<code>Y = fftshift(X)</code> rearranges the outputs of <code>fft</code> , <code>fft2</code> , and <code>fftn</code> by moving the zero frequency component to the center of the array.  For vectors, <code>fftshift(X)</code> swaps the left and right halves of <code>X</code> . For matrices, <code>fftshift(X)</code> swaps quadrants one and three of <code>X</code> with quadrants two and four. For higher-dimensional arrays, <code>fftshift(X)</code> swaps “half-spaces” of <code>X</code> along each dimension.
<b>Examples</b>	For any matrix <code>X</code>  <code>Y = fft2(X)</code>  has <code>Y(1, 1) = sum(sum(X))</code> ; the DC component of the signal is in the upper-left corner of the two-dimensional FFT. For  <code>Z = fftshift(Y)</code>  this DC component is near the center of the matrix.
<b>See Also</b>	<code>fft</code> One-dimensional fast Fourier transform <code>fft2</code> Two-dimensional fast Fourier transform <code>fftn</code> Multidimensional fast Fourier transform

# fgetl

---

<b>Purpose</b>	Return the next line of a file as a string without line terminator(s)
<b>Syntax</b>	<code>line = fgetl(fd)</code>
<b>Description</b>	<code>line = fgetl(fd)</code> returns the next line of the file with identifier <code>fd</code> . If <code>fgetl</code> encounters the end of a file, it returns <code>-1</code> . (See <code>fopen</code> for a complete description of <code>fd</code> .)  The returned string <code>line</code> does not include the line terminator(s) with the text line (to obtain the line terminator(s), use <code>fgets</code> ).
<b>See Also</b>	<code>fgets</code> Return the next line of a file as a string with line terminator(s)

<b>Purpose</b>	Return the next line of a file as a string with line terminator(s)
<b>Syntax</b>	<code>line = fgets(fd)</code> <code>line = fgets(fd, nchar)</code>
<b>Description</b>	<code>line = fgets(fd)</code> returns the next line for the file with identifier <code>fd</code> . If <code>fgets</code> encounters the end of a file, it returns <code>-1</code> . (See <code>fopen</code> for a complete description of <code>fd</code> .)  The returned string <code>line</code> includes the line terminator(s) associated with the text line (to obtain the string without the line terminator(s), use <code>fgetl</code> ).  <code>line = fgets(fd, nchar)</code> returns at most <code>nchar</code> characters of the next line. No additional characters are read after the line terminator(s) or an end-of-file.
<b>See Also</b>	<code>fgetl</code> Return the next line of a file as a string without line terminator(s)

# fieldnames

---

<b>Purpose</b>	Field names of a structure
<b>Syntax</b>	<code>names = fieldnames(s)</code>
<b>Description</b>	<code>names = fieldnames(s)</code> returns a cell array of strings containing the structure field names associated with the structure <code>s</code> .
<b>Examples</b>	Given the structure:  <code>mystr(1, 1).name = 'alice'; mystr(1, 1).ID = 0; mystr(2, 1).name = 'gertrude'; mystr(2, 1).ID = 1</code>  Then the command <code>n = fieldnames(mystr)</code> yields  <code>n = 'name' 'ID'</code>
<b>See Also</b>	<a href="#">getfield</a> Get field of structure array <a href="#">setfield</a> Set field of structure array

<b>Purpose</b>	Filename parts
<b>Syntax</b>	[path, name, ext, ver] = fileparts( <i>file</i> )
<b>Description</b>	[path, name, ext, ver] = fileparts( <i>file</i> ) returns the path, filename, extension, and version for the specified file. ver will be nonempty only on VMS systems. fileparts is platform dependent.
<b>See Also</b>	<code>fullfile</code> Build full filename from parts

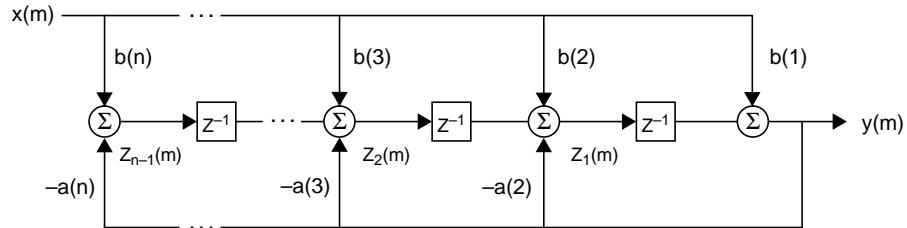
# filter

---

<b>Purpose</b>	Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter
<b>Syntax</b>	<pre>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)</pre>
<b>Description</b>	<p>The <code>filter</code> function filters a data sequence using a digital filter which works for both real and complex inputs. The filter is a <i>direct form II transposed</i> implementation of the standard difference equation (see “Algorithm”).</p> <p><code>y = filter(b, a, X)</code> filters the data in vector <code>X</code> with the filter described by numerator coefficient vector <code>b</code> and denominator coefficient vector <code>a</code>. If <code>a(1)</code> is not equal to 1, <code>filter</code> normalizes the filter coefficients by <code>a(1)</code>. If <code>a(1)</code> equals 0, <code>filter</code> returns an error.</p> <p>If <code>X</code> is a matrix, <code>filter</code> operates on the columns of <code>X</code>. If <code>X</code> is a multidimensional array, <code>filter</code> operates on the first nonsingleton dimension.</p> <p><code>[y, zf] = filter(b, a, X)</code> returns the final conditions, <code>zf</code>, of the filter delays. Output <code>zf</code> is a vector of <code>max(size(a), size(b))</code> or an array of such vectors, one for each column of <code>X</code>.</p> <p><code>[y, zf] = filter(b, a, X, zi)</code> accepts initial conditions and returns the final conditions, <code>zi</code> and <code>zf</code> respectively, of the filter delays. Input <code>zi</code> is a vector (or an array of vectors) of length <code>max(length(a), length(b))-1</code>.</p> <p><code>y = filter(b, a, X, zi, dim)</code> and <code>[...] = filter(b, a, X, [], dim)</code> operate across the dimension <code>dim</code>.</p>

**Algorithm**

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



or

$$y(n) = b(1)*x(n) + b(2)*x(n-1) + \dots + b(nb+1)*x(n-nb) \\ - a(2)*y(n-1) - \dots - a(na+1)*y(n-na)$$

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

$$y(m) = b(1)x(m) + z_1(m-1)$$

$$z_1(m) = b(2)x(m) + z_2(m-1) - a(2)y(m)$$

$$\vdots = \vdots \vdots$$

$$z_{n-2}(m) = b(n-1)x(m) + z_{n-1}(m-1) - a(n-1)y(m)$$

$$z_{n-1}(m) = b(n)x(m) - a(n)y(m)$$

The input-output description of this filtering operation in the  $z$ -transform domain is a rational transfer function,

$$Y(z) = \frac{b(1) + b(2)z^{-1} + \dots + b(nb+1)z^{-nb}}{1 + a(2)z^{-1} + \dots + a(na+1)z^{-na}} X(z)$$

**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

<b>Purpose</b>	Two-dimensional digital filtering				
<b>Syntax</b>	<pre>Y = filter2(h, X) Y = filter2(h, X, shape)</pre>				
<b>Description</b>	<p><code>Y = filter2(h, X)</code> filters the data in <code>X</code> with the two-dimensional FIR filter in the matrix <code>h</code>. It computes the result, <code>Y</code>, using two-dimensional correlation, and returns the central part of the correlation that is the same size as <code>X</code>.</p> <p><code>Y = filter2(h, X, shape)</code> returns the part of <code>Y</code> specified by the <code>shape</code> parameter. <code>shape</code> is a string with one of these values:</p> <ul style="list-style-type: none"><li>• '<code>full</code>' returns the full two-dimensional correlation. In this case, <code>Y</code> is larger than <code>X</code>.</li><li>• '<code>same</code>' (the default) returns the central part of the correlation. In this case, <code>Y</code> is the same size as <code>X</code>.</li><li>• '<code>valid</code>' returns only those parts of the correlation that are computed without zero-padded edges. In this case, <code>Y</code> is smaller than <code>X</code>.</li></ul>				
<b>Remarks</b>	Two-dimensional correlation is equivalent to two-dimensional convolution with the filter matrix rotated 180 degrees. See the Algorithm section for more information about how <code>filter2</code> performs linear filtering.				
<b>Algorithm</b>	<p>Given a matrix <code>X</code> and a two-dimensional FIR filter <code>h</code>, <code>filter2</code> rotates your filter matrix 180 degrees to create a convolution kernel. It then calls <code>conv2</code>, the two-dimensional convolution function, to implement the filtering operation.</p> <p><code>filter2</code> uses <code>conv2</code> to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, <code>filter2</code> then extracts the central part of the convolution that is the same size as the input matrix, and returns this as the result. If the <code>shape</code> parameter specifies an alternate part of the convolution for the result, <code>filter2</code> returns the appropriate part.</p>				
<b>See Also</b>	<table><tr><td><code>conv2</code></td><td>Two-dimensional convolution</td></tr><tr><td><code>filter</code></td><td>Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter</td></tr></table>	<code>conv2</code>	Two-dimensional convolution	<code>filter</code>	Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter
<code>conv2</code>	Two-dimensional convolution				
<code>filter</code>	Filter data with an infinite impulse response (IIR) or finite impulse response (FIR) filter				

<b>Purpose</b>	Find indices and values of nonzero elements
<b>Syntax</b>	<pre>k = find(x) [i,j] = find(X) [i,j,v] = find(X)</pre>
<b>Description</b>	<p><code>k = find(X)</code> returns the indices of the array <code>x</code> that point to nonzero elements. If none is found, <code>find</code> returns an empty matrix.</p> <p><code>[i,j] = find(X)</code> returns the row and column indices of the nonzero entries in the matrix <code>X</code>. This is often used with sparse matrices.</p> <p><code>[i,j,v] = find(X)</code> returns a column vector <code>v</code> of the nonzero entries in <code>X</code>, as well as row and column indices.</p> <p>In general, <code>find(X)</code> regards <code>X</code> as <code>X(:)</code>, which is the long column vector formed by concatenating the columns of <code>X</code>.</p>
<b>Examples</b>	<p><code>[i,j,v] = find(X~=0)</code> produces a vector <code>v</code> with all 1s, and returns the row and column indices.</p> <p>Some operations on a vector</p> <pre>x = [11 0 33 0 55]'; find(x)  ans = 1 3 5  find(x == 0)  ans = 2 4</pre>

# find

---

```
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
3	2	1
2	3	1

## See Also

The relational operators `<`, `<=`, `>`, `>=`, `==`, `~=`, and:

`nonzeros`  
`sparse`

Nonzero matrix elements  
Create sparse matrix

<b>Purpose</b>	Find one string within another						
<b>Syntax</b>	<code>k = findstr(str1, str2)</code>						
<b>Description</b>	<code>k = findstr(str1, str2)</code> finds the starting indices of any occurrences of the shorter string within the longer.						
<b>Examples</b>	<pre>str1 = 'Find the starting indices of the shorter string.'; str2 = 'the'; findstr(str1, str2)</pre> <pre>ans =        6      30</pre>						
<b>See Also</b>	<table><tr><td><code>strcmp</code></td><td>Compare strings</td></tr><tr><td><code>strmatch</code></td><td>Find possible matches for a string</td></tr><tr><td><code>strncmp</code></td><td>Compare the first n characters of two strings</td></tr></table>	<code>strcmp</code>	Compare strings	<code>strmatch</code>	Find possible matches for a string	<code>strncmp</code>	Compare the first n characters of two strings
<code>strcmp</code>	Compare strings						
<code>strmatch</code>	Find possible matches for a string						
<code>strncmp</code>	Compare the first n characters of two strings						

# fix

---

<b>Purpose</b>	Round towards zero
<b>Syntax</b>	<code>B = fix(A)</code>
<b>Description</b>	<code>B = fix(A)</code> rounds the elements of <code>A</code> toward zero, resulting in an array of integers. For complex <code>A</code> , the imaginary and real parts are rounded independently.
<b>Examples</b>	<pre>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</pre>
<b>See Also</b>	<code>ceil</code> Round toward infinity <code>floor</code> Round towards minus infinity <code>round</code> Round to nearest integer

<b>Purpose</b>	Flip array along a specified dimension												
<b>Syntax</b>	$B = \text{flipldim}(A, \text{dim})$												
<b>Description</b>	$B = \text{flipldim}(A, \text{dim})$ returns $A$ with dimension $\text{dim}$ flipped. When the value of $\text{dim}$ is 1, the array is flipped row-wise down. When $\text{dim}$ is 2, the array is flipped columnwise left to right. $\text{flipldim}(A, 1)$ is the same as $\text{fliplr}(A)$ , and $\text{flipldim}(A, 2)$ is the same as $\text{fliplr}(A)$ .												
<b>Examples</b>	$\text{flipldim}(A, 1)$ where $A =$ <table><tr><td>1</td><td>4</td></tr><tr><td>2</td><td>5</td></tr><tr><td>3</td><td>6</td></tr></table> produces <table><tr><td>3</td><td>6</td></tr><tr><td>2</td><td>5</td></tr><tr><td>1</td><td>4</td></tr></table>	1	4	2	5	3	6	3	6	2	5	1	4
1	4												
2	5												
3	6												
3	6												
2	5												
1	4												
<b>See Also</b>	<a href="#">fliplr</a> Flip matrices left-right <a href="#">fliplr</a> Flip matrices up-down <a href="#">permute</a> Rearrange the dimensions of a multidimensional array <a href="#">rot90</a> Rotate matrix 90°												

# **fliplr**

---

<b>Purpose</b>	Flip matrices left-right												
<b>Syntax</b>	<code>B = fliplr(A)</code>												
<b>Description</b>	<code>B = fliplr(A)</code> returns A with columns flipped in the left-right direction, that is, about a vertical axis.												
<b>Examples</b>	<p><code>A =</code></p> <table><tr><td>1</td><td>4</td></tr><tr><td>2</td><td>5</td></tr><tr><td>3</td><td>6</td></tr></table> <p>produces</p> <table><tr><td>4</td><td>1</td></tr><tr><td>5</td><td>2</td></tr><tr><td>6</td><td>3</td></tr></table>	1	4	2	5	3	6	4	1	5	2	6	3
1	4												
2	5												
3	6												
4	1												
5	2												
6	3												
<b>Limitations</b>	Array A must be two dimensional.												
<b>See Also</b>	<table><tr><td><code>flipdim</code></td><td>Flip array along a specified dimension</td></tr><tr><td><code>flipud</code></td><td>Flip matrices up-down</td></tr><tr><td><code>rot90</code></td><td>Rotate matrix 90°</td></tr></table>	<code>flipdim</code>	Flip array along a specified dimension	<code>flipud</code>	Flip matrices up-down	<code>rot90</code>	Rotate matrix 90°						
<code>flipdim</code>	Flip array along a specified dimension												
<code>flipud</code>	Flip matrices up-down												
<code>rot90</code>	Rotate matrix 90°												

---

<b>Purpose</b>	Flip matrices up-down												
<b>Syntax</b>	<code>B = flipud(A)</code>												
<b>Description</b>	<code>B = flipud(A)</code> returns <code>A</code> with rows flipped in the up-down direction, that is, about a horizontal axis.												
<b>Examples</b>	<p><code>A =</code></p> <table><tr><td>1</td><td>4</td></tr><tr><td>2</td><td>5</td></tr><tr><td>3</td><td>6</td></tr></table> <p>produces</p> <table><tr><td>3</td><td>6</td></tr><tr><td>2</td><td>5</td></tr><tr><td>1</td><td>4</td></tr></table>	1	4	2	5	3	6	3	6	2	5	1	4
1	4												
2	5												
3	6												
3	6												
2	5												
1	4												
<b>Limitations</b>	Array <code>A</code> must be two dimensional.												
<b>See Also</b>	<table><tr><td><code>fliplr</code></td><td>Flip array along a specified dimension</td></tr><tr><td><code>fliplr</code></td><td>Flip matrices left-right</td></tr><tr><td><code>rot90</code></td><td>Rotate matrix 90°</td></tr></table>	<code>fliplr</code>	Flip array along a specified dimension	<code>fliplr</code>	Flip matrices left-right	<code>rot90</code>	Rotate matrix 90°						
<code>fliplr</code>	Flip array along a specified dimension												
<code>fliplr</code>	Flip matrices left-right												
<code>rot90</code>	Rotate matrix 90°												

# floor

---

<b>Purpose</b>	Round towards minus infinity
<b>Syntax</b>	<code>B = floor(A)</code>
<b>Description</b>	<code>B = floor(A)</code> 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.
<b>Examples</b>	<pre>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</pre>
<b>See Also</b>	<code>ceil</code> Round toward infinity <code>fix</code> Round towards zero <code>round</code> Round to nearest integer

<b>Purpose</b>	Count floating-point operations
<b>Syntax</b>	<code>f = flops</code> <code>flops(0)</code>
<b>Description</b>	<code>f = flops</code> returns the cumulative number of floating-point operations. <code>flops(0)</code> resets the count to zero.
<b>Examples</b>	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	$2*n^3$
$A^{100}$	$99*(2*n^3)$
$\text{lu}(A)$	$(2/3)*n^3$

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

```

<b>Algorithm</b>	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

---

<b>Purpose</b>	Minimize a function of one variable
<b>Syntax</b>	<pre>x = fmin('fun', x1, x2) x = fmin('fun', x1, x2, options) x = fmin('fun', x1, x2, options, P1, P2, ...) [x, options] = fmin(...)</pre>
<b>Description</b>	<p><code>x = fmin('fun', x1, x2)</code> returns a value of <code>x</code> which is a local minimizer of <code>fun(x)</code> in the interval <math>x_1 &lt; x &lt; x_2</math>.</p> <p><code>x = fmin('fun', x1, x2, options)</code> does the same as the above, but uses <code>options</code> control parameters.</p> <p><code>x = fmin('fun', x1, x2, options, P1, P2, ...)</code> does the same as the above, but passes arguments to the objective function, <code>fun(x, P1, P2, ...)</code>. Pass an empty matrix for <code>options</code> to use the default value.</p> <p><code>[x, options] = fmin(...)</code> returns, in <code>options(10)</code>, a count of the number of steps taken.</p>
<b>Arguments</b>	<p><code>x1, x2</code> Interval over which <i>function</i> is minimized.</p> <p><code>P1, P2, ...</code> Arguments to be passed to <i>function</i>.</p> <p><code>fun</code> A string containing the name of the function to be minimized.</p> <p><code>options</code> A vector of control parameters. Only three of the 18 components of <code>options</code> are referenced by <code>fmin</code>; Optimization Toolbox functions use the others. The three control options used by <code>fmin</code> are:</p> <ul style="list-style-type: none"><li>• <code>options(1)</code> — If this is nonzero, intermediate steps in the solution are displayed. The default value of <code>options(1)</code> is 0.</li><li>• <code>options(2)</code> — This is the termination tolerance. The default value is <code>1.e-4</code>.</li><li>• <code>options(14)</code> — This is the maximum number of steps. The default value is 500.</li></ul>

**Examples**

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

`fmin('cos', 3, 4, [1, 1. e-12])` displays the steps taken to compute  $\pi$  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**

<code>fmins</code>	Minimize a function of several variables
<code>fzero</code>	Zero of a function of one variable
foptions in the Optimization Toolbox (or type <code>help foptions</code> ).	

**References**

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

# fmins

---

<b>Purpose</b>	Minimize a function of several variables						
<b>Syntax</b>	<pre>x = fmins('fun', x0) x = fmins('fun', x0, options) x = fmins('fun', x0, options, [], P1, P2, ...) [x, options] = fmins(...)</pre>						
<b>Description</b>	<p><code>x = fmins('fun', x0)</code> returns a vector <code>x</code> which is a local minimizer of <code>fun(x)</code> near <math>x_0</math>.</p> <p><code>x = fmins('fun', x0, options)</code> does the same as the above, but uses <code>options</code> control parameters.</p> <p><code>x = fmins('fun', x0, options, [], P1, P2, ...)</code> does the same as above, but passes arguments to the objective function, <code>fun(x, P1, P2, ...)</code>. Pass an empty matrix for <code>options</code> to use the default value.</p> <p><code>[x, options] = fmins(...)</code> returns, in <code>options(10)</code>, a count of the number of steps taken.</p>						
<b>Arguments</b>	<table><tr><td><code>x0</code></td><td>Starting vector.</td></tr><tr><td><code>P1, P2...</code></td><td>Arguments to be passed to <code>fun</code>.</td></tr><tr><td><code>[]</code></td><td>Argument needed to provide compatibility with <code>fminu</code> in the Optimization Toolbox.</td></tr></table>	<code>x0</code>	Starting vector.	<code>P1, P2...</code>	Arguments to be passed to <code>fun</code> .	<code>[]</code>	Argument needed to provide compatibility with <code>fminu</code> in the Optimization Toolbox.
<code>x0</code>	Starting vector.						
<code>P1, P2...</code>	Arguments to be passed to <code>fun</code> .						
<code>[]</code>	Argument needed to provide compatibility with <code>fminu</code> in the Optimization Toolbox.						

<i>fun</i>	A string containing the name of the objective function to be minimized. <i>fun(x)</i> is a scalar valued function of a vector variable.
<i>options</i>	A vector of control parameters. Only four of the 18 components of <i>options</i> are referenced by <i>fmmins</i> ; Optimization Toolbox functions use the others. The four control options used by <i>fmmins</i> are: <ul style="list-style-type: none"> <li>• <i>options(1)</i> — If this is nonzero, intermediate steps in the solution are displayed. The default value of <i>options(1)</i> is 0.</li> <li>• <i>options(2)</i> and <i>options(3)</i> — These are the termination tolerances for <i>x</i> and <i>function(x)</i>, respectively. The default values are <math>1 \cdot 10^{-4}</math>.</li> <li>• <i>options(14)</i> — This is the maximum number of steps. The default value is 500.</li> </ul>

## 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] = fmmins('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 and the function values of the simplex vertices differ from the lowest function value by less than the specified tolerance, or the maximum number of function evaluations has been exceeded.

**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] Lagarias, Jeffrey C., James A. Reeds, Margaret H. Wright, and Paul E. Wright, "Convergence Properties of the Nelder-Mead Simplex Algorithm in Low Dimensions", May 1, 1997. To appear in *SIAM Journal of Optimization*.

# fopen

<b>Purpose</b>	Open a file or obtain information about open files
<b>Syntax</b>	<pre>fid = fopen(filename, permission) [fid, message] = fopen(filename, permission, format) fid = fopen('all') [filename, permission, format] = fopen(fid)</pre>
<b>Description</b>	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.

Two fids are predefined and cannot be explicitly opened or closed:

- 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.164' or ' a'	IEEE floating point with little-endian byte ordering and 64-bit long data type
' ieee-be.164' 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 1 and 2 (standard output and standard error). The number of elements in the vector is equal to the number of open files.

# fopen

---

[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
```

# for

---

has the same effect as

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

except j is also set here.

## See Also

break	Terminate execution of for or while loop
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

<b>Purpose</b>	Control the output display format	
<b>Syntax</b>	MATLAB performs all computations in double precision. The <code>format</code> command described below switches among different display formats.	
<b>Description</b>		
Command	Result	Example
<code>format</code>	Default. Same as <code>short</code> .	
<code>format short</code>	5 digit scaled fixed point	3. 1416
<code>format long</code>	15 digit scaled fixed point	3. 14159265358979
<code>format short e</code>	5 digit floating-point	3. 1416e+00
<code>format long e</code>	15 digit floating-point	3. 141592653589793e+0 0
<code>format short g</code>	Best of 5 digit fixed or floating	3. 1416
<code>format long g</code>	Best of 15 digit fixed or floating	3. 14159265358979
<code>format hex</code>	Hexadecimal	400921fb54442d18
<code>format bank</code>	Fixed dollars and cents	3. 14
<code>format rat</code>	Ratio of small integers	355/113
<code>format +</code>	+,-, blank	+
<code>format compact</code>	Suppresses excess line feeds.	
<code>format loose</code>	Add line feeds.	
<b>Algorithms</b>	The command <code>format +</code> displays +, -, and blank characters for positive, negative, and zero elements. <code>format hex</code> displays the hexadecimal representation of a binary double-precision number. <code>format rat</code> uses a continued fraction algorithm to approximate floating-point values by ratios of small integers. See <code>rat.m</code> for the complete code.	
<b>See Also</b>	<code>fprintf</code> , <code>num2str</code> , <code>rat</code> , <code>sprintf</code> , <code>spy</code>	

# fprintf

<b>Purpose</b>	Write formatted data to file
<b>Syntax</b>	<pre>count = fprintf(fid, format, A, . . .) fprintf(format, A, . . .)</pre>
<b>Description</b>	<p><code>count = fprintf(fid, format, A, . . .)</code> formats the data in the real part of matrix <code>A</code> (and in any additional matrix arguments) under control of the specified <code>format</code> string, and writes it to the file associated with file identifier <code>fid</code>. <code>fprintf</code> returns a count of the number of bytes written.</p> <p>Argument <code>fid</code> is an integer file identifier obtained from <code>fopen</code>. (It may also be 1 for standard output (the screen) or 2 for standard error. See <code>fopen</code> for more information.) Omitting <code>fid</code> from <code>fprintf</code>'s argument list causes output to appear on the screen, and is the same as writing to standard output (<code>fid = 1</code>)</p> <p><code>fprintf(format, A, . . .)</code> writes to standard output—the screen.</p> <p>The <code>format</code> 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:</p>
	<p>For more information see “Tables” and “References”.</p>
<b>Remarks</b>	The <code>fprintf</code> function behaves like its ANSI C language <code>fprintf()</code> 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 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 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
\\" or "	Single quotation mark
(two single quotes)	
%%	Percent character

# fprintf

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.
```

# fprintf

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

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

displays on the screen:

```
It's Friday.
```

The commands

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

display the lines:

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

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

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

## See Also

fclose	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.

<b>Purpose</b>	Read binary data from file
<b>Syntax</b>	$[A, \text{count}] = \text{fread}(\text{fid}, \text{size}, \text{precision})$ $[A, \text{count}] = \text{fread}(\text{fid}, \text{size}, \text{precision}, \text{skip})$
<b>Description</b>	<p><math>[A, \text{count}] = \text{fread}(\text{fid}, \text{size}, \text{precision})</math> 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.</p> <p>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:</p> <ul style="list-style-type: none"> <li>n         Reads n elements into a column vector.</li> <li>inf        Reads to the end of the file, resulting in a column vector containing the same number of elements as are in the file.</li> <li>[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.</li> </ul> <p>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.</p> <p>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.</p> <p><math>[A, \text{count}] = \text{fread}(\text{fid}, \text{size}, \text{precision}, \text{skip})</math> includes an optional skip argument that specifies the number of bytes to skip after each precision value is read. With the skip argument present, fread reads in one value and does a skip of input, reads in another value and does a skip of input, etc. for at most size times. This is useful for extracting data in noncontiguous fields from fixed</p>

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'	'unsigned short'	Unsigned integer; 16 bits
'uint'	'unsigned int'	Unsigned integer; 32 bits
'ulong'	'unsigned long'	Unsigned integer; 32 or 64 bits
'float'	'float'	Floating-point; 32 bits
'double'	'double'	Floating-point; 64 bits

Two formats map to an input stream 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

# freqspace

---

<b>Purpose</b>	Determine frequency spacing for frequency response
<b>Syntax</b>	<pre>[f1, f2] = freqspace(n) [f1, f2] = freqspace([m n]) [x1, y1] = freqspace(..., 'meshgrid') f = freqspace(N) f = freqspace(N, 'whole')</pre>
<b>Description</b>	<p><code>freqspace</code> returns the implied frequency range for equally spaced frequency responses. <code>freqspace</code> is useful when creating desired frequency responses for various one- and two-dimensional applications.</p> <p><code>[f1, f2] = freqspace(n)</code> returns the two-dimensional frequency vectors <code>f1</code> and <code>f2</code> for an <code>n</code>-by-<code>n</code> matrix.</p> <p>For <code>n</code> odd, both <code>f1</code> and <code>f2</code> are <math>[-n+1: 2: n-1]/n</math>.</p> <p>For <code>n</code> even, both <code>f1</code> and <code>f2</code> are <math>[-n: 2: n-2]/n</math>.</p> <p><code>[f1, f2] = freqspace([m n])</code> returns the two-dimensional frequency vectors <code>f1</code> and <code>f2</code> for an <code>m</code>-by-<code>n</code> matrix.</p> <p><code>[x1, y1] = freqspace(..., 'meshgrid')</code> is equivalent to</p> <pre>[f1, f2] = freqspace(...); [x1, y1] = meshgrid(f1, f2);</pre> <p><code>f = freqspace(N)</code> returns the one-dimensional frequency vector <code>f</code> assuming <code>N</code> evenly spaced points around the unit circle. For <code>N</code> even or odd, <code>f</code> is <math>(0: 2/N: 1)</math>. For <code>N</code> even, <code>freqspace</code> therefore returns <math>(N+2)/2</math> points. For <code>N</code> odd, it returns <math>(N+1)/2</math> points.</p> <p><code>f = freqspace(N, 'whole')</code> returns <code>N</code> evenly spaced points around the whole unit circle. In this case, <code>f</code> is <math>0: 2/N: 2*(N-1)/N</math>.</p>
<b>See Also</b>	<code>meshgrid</code> Generate X and Y matrices for three-dimensional plots

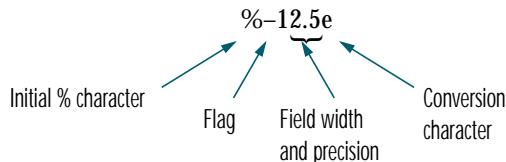
<b>Purpose</b>	Rewind an open file																		
<b>Syntax</b>	<code>frewind(fid)</code>																		
<b>Description</b>	<code>frewind(fid)</code> sets the file position indicator to the beginning of the file specified by <code>fid</code> , an integer file identifier obtained from <code>fopen</code> .																		
<b>Remarks</b>	Rewinding a <code>fid</code> associated with a tape device may not work even though <code>frewind</code> does not generate an error message.																		
<b>See Also</b>	<table><tr><td><code>fclose</code></td><td>Close one or more open files</td></tr><tr><td><code>ferror</code></td><td>Query MATLAB about errors in file input or output</td></tr><tr><td><code>fopen</code></td><td>Open a file or obtain information about open files</td></tr><tr><td><code>fprintf</code></td><td>Write formatted data to file</td></tr><tr><td><code>fread</code></td><td>Read binary data from file</td></tr><tr><td><code>fscanf</code></td><td>Read formatted data from file</td></tr><tr><td><code>fseek</code></td><td>Set file position indicator</td></tr><tr><td><code>ftell</code></td><td>Get file position indicator</td></tr><tr><td><code>fwrite</code></td><td>Write binary data from a MATLAB matrix to a file</td></tr></table>	<code>fclose</code>	Close one or more open files	<code>ferror</code>	Query MATLAB about errors in file input or output	<code>fopen</code>	Open a file or obtain information about open files	<code>fprintf</code>	Write formatted data to file	<code>fread</code>	Read binary data from file	<code>fscanf</code>	Read formatted data from file	<code>fseek</code>	Set file position indicator	<code>ftell</code>	Get file position indicator	<code>fwrite</code>	Write binary data from a MATLAB matrix to a file
<code>fclose</code>	Close one or more open files																		
<code>ferror</code>	Query MATLAB about errors in file input or output																		
<code>fopen</code>	Open a file or obtain information about open files																		
<code>fprintf</code>	Write formatted data to file																		
<code>fread</code>	Read binary data from file																		
<code>fscanf</code>	Read formatted data from file																		
<code>fseek</code>	Set file position indicator																		
<code>ftell</code>	Get file position indicator																		
<code>fwrite</code>	Write binary data from a MATLAB matrix to a file																		

# fscanf

---

<b>Purpose</b>	Read formatted data from file
<b>Syntax</b>	$A = \text{fscanf}(\text{fid}, \text{format})$ $[A, \text{count}] = \text{fscanf}(\text{fid}, \text{format}, \text{size})$
<b>Description</b>	$A = \text{fscanf}(\text{fid}, \text{format})$ reads all the data from the file specified by <code>fid</code> , converts it according to the specified <code>format</code> string, and returns it in matrix <code>A</code> . Argument <code>fid</code> is an integer file identifier obtained from <code>fopen</code> . <code>format</code> is a string specifying the format of the data to be read. See “Remarks” for details. $[A, \text{count}] = \text{fscanf}(\text{fid}, \text{format}, \text{size})$ reads the amount of data specified by <code>size</code> , converts it according to the specified <code>format</code> string, and returns it along with a count of elements successfully read. <code>size</code> is an argument that determines how much data is read. Valid options are: <ul style="list-style-type: none"><li><code>n</code> Read <code>n</code> elements into a column vector.</li><li><code>inf</code> Read to the end of the file, resulting in a column vector containing the same number of elements as are in the file.</li><li><code>[m, n]</code> Read enough elements to fill an <code>m</code>-by-<code>n</code> matrix, filling the matrix in column order. <code>n</code> can be <code>Inf</code>, but not <code>m</code>.</li></ul>
<b>fscanf</b>	<b>fscanf</b> differs from its C language namesakes <code>scanf()</code> and <code>fscanf()</code> in an important respect — it is <i>vectorized</i> in order to return a matrix argument. The <code>format</code> string is cycled through the file until an end-of-file is reached or the amount of data specified by <code>size</code> is read in.
<b>Remarks</b>	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 <code>format</code> 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

<code>fclose</code>	Close one or more open files
<code>ferror</code>	Query MATLAB about errors in file input or output
<code>fopen</code>	Open a file or obtain information about open files
<code>fprintf</code>	Write formatted data to file
<code>fread</code>	Read binary data from file
<code>fseek</code>	Set file position indicator
<code>ftell</code>	Get file position indicator
<code>fwrite</code>	Write binary data from a MATLAB matrix to a file

<b>Purpose</b>	Set file position indicator																				
<b>Syntax</b>	<code>status = fseek(fid, offset, origin)</code>																				
<b>Description</b>	<code>status = fseek(fid, offset, origin)</code> repositions the file position indicator in the file with the given <code>fid</code> to the byte with the specified <code>offset</code> relative to <code>origin</code> .																				
<b>Arguments</b>	<table><tr><td><code>fid</code></td><td>An integer file identifier obtained from <code>fopen</code>.</td></tr><tr><td><code>offset</code></td><td>A value that is interpreted as follows:<table><tr><td><code>offset &gt; 0</code></td><td>Move position indicator <code>offset</code> bytes toward the end of the file.</td></tr><tr><td><code>offset = 0</code></td><td>Do not change position.</td></tr><tr><td><code>offset &lt; 0</code></td><td>Move position indicator <code>offset</code> bytes toward the beginning of the file.</td></tr></table></td></tr><tr><td><code>origin</code></td><td>A string whose legal values are:<table><tr><td>'bof'</td><td>-1: Beginning of file.</td></tr><tr><td>'cof'</td><td>0: Current position in file.</td></tr><tr><td>'eof'</td><td>1: End of file.</td></tr></table></td></tr><tr><td><code>status</code></td><td>A returned value that is 0 if the <code>fseek</code> operation is successful and -1 if it fails. If an error occurs, use the function <code>ferror</code> to get more information about the nature of the error.</td></tr></table>	<code>fid</code>	An integer file identifier obtained from <code>fopen</code> .	<code>offset</code>	A value that is interpreted as follows: <table><tr><td><code>offset &gt; 0</code></td><td>Move position indicator <code>offset</code> bytes toward the end of the file.</td></tr><tr><td><code>offset = 0</code></td><td>Do not change position.</td></tr><tr><td><code>offset &lt; 0</code></td><td>Move position indicator <code>offset</code> bytes toward the beginning of the file.</td></tr></table>	<code>offset &gt; 0</code>	Move position indicator <code>offset</code> bytes toward the end of the file.	<code>offset = 0</code>	Do not change position.	<code>offset &lt; 0</code>	Move position indicator <code>offset</code> bytes toward the beginning of the file.	<code>origin</code>	A string whose legal values are: <table><tr><td>'bof'</td><td>-1: Beginning of file.</td></tr><tr><td>'cof'</td><td>0: Current position in file.</td></tr><tr><td>'eof'</td><td>1: End of file.</td></tr></table>	'bof'	-1: Beginning of file.	'cof'	0: Current position in file.	'eof'	1: End of file.	<code>status</code>	A returned value that is 0 if the <code>fseek</code> operation is successful and -1 if it fails. If an error occurs, use the function <code>ferror</code> to get more information about the nature of the error.
<code>fid</code>	An integer file identifier obtained from <code>fopen</code> .																				
<code>offset</code>	A value that is interpreted as follows: <table><tr><td><code>offset &gt; 0</code></td><td>Move position indicator <code>offset</code> bytes toward the end of the file.</td></tr><tr><td><code>offset = 0</code></td><td>Do not change position.</td></tr><tr><td><code>offset &lt; 0</code></td><td>Move position indicator <code>offset</code> bytes toward the beginning of the file.</td></tr></table>	<code>offset &gt; 0</code>	Move position indicator <code>offset</code> bytes toward the end of the file.	<code>offset = 0</code>	Do not change position.	<code>offset &lt; 0</code>	Move position indicator <code>offset</code> bytes toward the beginning of the file.														
<code>offset &gt; 0</code>	Move position indicator <code>offset</code> bytes toward the end of the file.																				
<code>offset = 0</code>	Do not change position.																				
<code>offset &lt; 0</code>	Move position indicator <code>offset</code> bytes toward the beginning of the file.																				
<code>origin</code>	A string whose legal values are: <table><tr><td>'bof'</td><td>-1: Beginning of file.</td></tr><tr><td>'cof'</td><td>0: Current position in file.</td></tr><tr><td>'eof'</td><td>1: End of file.</td></tr></table>	'bof'	-1: Beginning of file.	'cof'	0: Current position in file.	'eof'	1: End of file.														
'bof'	-1: Beginning of file.																				
'cof'	0: Current position in file.																				
'eof'	1: End of file.																				
<code>status</code>	A returned value that is 0 if the <code>fseek</code> operation is successful and -1 if it fails. If an error occurs, use the function <code>ferror</code> to get more information about the nature of the error.																				
<b>See Also</b>	<table><tr><td><code>fopen</code></td><td>Open a file or obtain information about open files</td></tr><tr><td><code>ftell</code></td><td>Get file position indicator</td></tr></table>	<code>fopen</code>	Open a file or obtain information about open files	<code>ftell</code>	Get file position indicator																
<code>fopen</code>	Open a file or obtain information about open files																				
<code>ftell</code>	Get file position indicator																				

# **f.tell**

---

<b>Purpose</b>	Get file position indicator																
<b>Syntax</b>	<code>position = f.tell(fid)</code>																
<b>Description</b>	<code>position = f.tell(fid)</code> returns the location of the file position indicator for the file specified by <code>fid</code> , an integer file identifier obtained from <code>fopen</code> . The <code>position</code> is a nonnegative integer specified in bytes from the beginning of the file. A returned value of <code>-1</code> for <code>position</code> indicates that the query was unsuccessful; use <code>ferror</code> to determine the nature of the error.																
<b>See Also</b>	<table><tr><td><code>fclose</code></td><td>Close one or more open files</td></tr><tr><td><code>ferror</code></td><td>Query MATLAB about errors in file input or output</td></tr><tr><td><code>fopen</code></td><td>Open a file or obtain information about open files</td></tr><tr><td><code>fprintf</code></td><td>Write formatted data to file</td></tr><tr><td><code>fread</code></td><td>Read binary data from file</td></tr><tr><td><code>fscanf</code></td><td>Read formatted data from file</td></tr><tr><td><code>fseek</code></td><td>Set file position indicator</td></tr><tr><td><code>fwrite</code></td><td>Write binary data from a MATLAB matrix to a file</td></tr></table>	<code>fclose</code>	Close one or more open files	<code>ferror</code>	Query MATLAB about errors in file input or output	<code>fopen</code>	Open a file or obtain information about open files	<code>fprintf</code>	Write formatted data to file	<code>fread</code>	Read binary data from file	<code>fscanf</code>	Read formatted data from file	<code>fseek</code>	Set file position indicator	<code>fwrite</code>	Write binary data from a MATLAB matrix to a file
<code>fclose</code>	Close one or more open files																
<code>ferror</code>	Query MATLAB about errors in file input or output																
<code>fopen</code>	Open a file or obtain information about open files																
<code>fprintf</code>	Write formatted data to file																
<code>fread</code>	Read binary data from file																
<code>fscanf</code>	Read formatted data from file																
<code>fseek</code>	Set file position indicator																
<code>fwrite</code>	Write binary data from a MATLAB matrix to a file																

---

<b>Purpose</b>	Convert sparse matrix to full matrix
<b>Syntax</b>	<code>A = full(S)</code>
<b>Description</b>	<code>A = full(S)</code> converts a sparse matrix <code>S</code> to full storage organization. If <code>S</code> is a full matrix, it is left unchanged. If <code>A</code> is full, <code>issparse(A)</code> is 0.
<b>Remarks</b>	Let <code>X</code> be an $m$ -by- $n$ matrix with $nz = nnz(X)$ nonzero entries. Then <code>full(X)</code> requires space to store $m \cdot n$ real numbers while <code>sparse(X)</code> 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, <code>sparse(X)</code> requires less storage than <code>full(X)</code> if the density, $nz/\text{prod}(\text{size}(X))$ , 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.
<b>Examples</b>	Here is an example of a sparse matrix with a density of about two-thirds. <code>sparse(S)</code> and <code>full(S)</code> require about the same number of bytes of storage.  <pre>S = sparse(rand(200, 200) &lt; 2/3); A = full(S); whos Name      Size        Bytes    Class A         200X200     320000   double array (logical) S         200X200     318432   sparse array (logical)</pre>
<b>See Also</b>	<code>sparse</code> Create sparse matrix

## fullfile

---

<b>Purpose</b>	Build full filename from parts
<b>Syntax</b>	<code>fullfile(dir1, dir2, ..., filename)</code>
<b>Description</b>	<code>fullfile(dir1, dir2, ..., filename)</code> builds a full filename from the directories and filename specified. This is conceptually equivalent to <code>f = [dir1 dirsep dir2 dirsep ... dirsep filename]</code> 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].
<b>Example</b>	<code>fullfile(matlabroot, 'tool box/matlab/general/Contents.m')</code> and <code>fullfile(matlabroot, 'tool box', 'matlab', 'general', 'Contents.m')</code> produce the same result on UNIX, but only the second one works on all platforms.

<b>Purpose</b>	Function M-files
<b>Description</b>	<p>You add new functions to MATLAB's vocabulary by expressing them in terms of existing commands and functions that compose the new function reside in a text file called an <i>M-file</i>.</p> <p>M-files can be either <i>scripts</i> or <i>functions</i>. Scripts are simply files containing a sequence of MATLAB statements. Functions make use of their own local variables and accept input arguments.</p> <p>The name of an M-file begins with an alphabetic character, and has a filename extension of <code>.m</code>. The M-file name, less its extension, is what MATLAB searches for when you try to use the script or function.</p> <p>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 <code>.m</code> extension. For example, the existence of a file on disk called <code>stat.m</code> with</p> <pre>function [ mean, stdev] = stat(x) n = length(x); mean = sum(x)/n; stdev = sqrt(sum((x-mean).^2/n));</pre> <p>defines a new function called <code>stat</code> that calculates the mean and standard deviation of a vector. The variables within the body of the function are all local variables.</p> <p>A <i>subfunction</i>, visible only to the other functions in the same file, is created by defining a new function with the <code>function</code> keyword after the body of the preceding function or subfunction. For example, <code>avg</code> is a subfunction within the file <code>stat.m</code>:</p> <pre>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;</pre>

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 (e.g., 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 prepared 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

<b>Purpose</b>	Evaluate functions of a matrix						
<b>Syntax</b>	<pre>Y = funm(X, 'function') [Y, esterr] = funm(X, 'function')</pre>						
<b>Description</b>	<p><code>Y = funm(X, 'function')</code> evaluates <i>function</i> using Parlett's method [1]. <i>X</i> must be a square matrix, and <i>function</i> any element-wise function.</p> <p>The commands <code>funm(X, 'sqrt')</code> and <code>funm(X, 'log')</code> are equivalent to the commands <code>sqrtm(X)</code> and <code>logm(X)</code>. The commands <code>funm(X, 'exp')</code> and <code>expm(X)</code> compute the same function, but by different algorithms. <code>expm(X)</code> is preferred.</p> <p><code>[Y, esterr] = funm(X, 'function')</code> does not print any message, but returns a very rough estimate of the relative error in the computer result. If <i>X</i> is symmetric or Hermitian, then its Schur form is diagonal, and <code>funm</code> is able to produce an accurate result.</p>						
<b>Examples</b>	<p>The statements</p> <pre>S = funm(X, 'sin'); C = funm(X, 'cos');</pre> <p>produce the same results to within roundoff error as</p> <pre>E = expm(i*X); C = real(E); S = imag(E);</pre> <p>In either case, the results satisfy <math>S^*S+C^*C = I</math>, where <math>I = \text{eye}(\text{size}(X))</math>.</p>						
<b>Algorithm</b>	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.						
<b>See Also</b>	<table border="0"> <tr> <td><code>expm</code></td> <td>Matrix exponential</td> </tr> <tr> <td><code>logm</code></td> <td>Matrix logarithm</td> </tr> <tr> <td><code>sqrtm</code></td> <td>Matrix square root</td> </tr> </table>	<code>expm</code>	Matrix exponential	<code>logm</code>	Matrix logarithm	<code>sqrtm</code>	Matrix square root
<code>expm</code>	Matrix exponential						
<code>logm</code>	Matrix logarithm						
<code>sqrtm</code>	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.

<b>Purpose</b>	Write binary data to a file
<b>Syntax</b>	<pre>count = fwrite(fid, A, precision) count = fwrite(fid, A, precision, skip)</pre>
<b>Description</b>	<p><code>count = fwrite(fid, A, precision)</code> writes the elements of matrix <code>A</code> to the specified file, translating MATLAB values to the specified numeric <code>precision</code>. (See “Remarks” for more information.)</p> <p>The data are written to the file in column order, and a <code>count</code> is kept of the number of elements written successfully. Argument <code>fid</code> is an integer file identifier obtained from <code>fopen</code>.</p> <p><code>count = fwrite(fid, A, precision, skip)</code> includes an optional <code>skip</code> argument that specifies the number of bytes to skip before each <code>precision</code> value is written. With the <code>skip</code> argument present, <code>fwrite</code> skips and writes one value, skips and writes another value, etc. until all of <code>A</code> is written. This is useful for inserting data into noncontiguous fields in fixed-length records. If <code>precision</code> is a bit format like '<code>bitN</code>' or '<code>ubitN</code>', <code>skip</code> is specified in bits.</p>
<b>Remarks</b>	<p>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.</p> <p>The tables below give C-compliant, platform-independent numeric precision string formats that you should use whenever you want your code to be portable.</p> <p>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.</p>

## fwrite

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'	'unsigned short'	Unsigned integer; 16 bits
'uint'	'unsigned int'	Unsigned integer; 32 bits
'ulong'	'unsigned long'	Unsigned integer; 32 or 64 bits
'float'	'float'	Floating-point; 32 bits
'double'	'double'	Floating-point; 64 bits

Two formats map to an input stream 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 ( $1 \leq N \leq 64$ )

## 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

# fzero

<b>Purpose</b>	Zero of a function of one variable				
<b>Syntax</b>	<pre>z = fzero('fun', x) z = fzero('fun', x, tol) z = fzero('fun', x, tol, trace) z = fzero('fun', x, tol, trace, P1, P2, ...)</pre>				
<b>Description</b>	<p><code>fzero('fun', x)</code> finds a zero of <i>fun</i>. <i>fun</i> is a string containing the name of a real-valued function of a single real variable. The value returned is near a point where <i>fun</i> changes sign, or NaN if the search fails.</p> <p><code>fzero('fun', x)</code> where <i>x</i> is a vector of length 2, assumes <i>x</i> is an interval where the sign of <math>f(x(1))</math> differs from the sign of <math>f(x(2))</math>. An error occurs if this is not true. Calling <code>fzero</code> with an interval guarantees <code>fzero</code> will return a value near a point where <i>fun</i> changes sign.</p> <p><code>fzero('fun', x)</code> where <i>x</i> is a scalar value, uses <i>x</i> as a starting point. <code>fzero</code> looks for an interval containing a sign change for <i>fun</i> and containing <i>x</i>. 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.</p> <p><code>fzero('fun', x, tol)</code> returns an answer accurate to within a relative error of <i>tol</i>.</p> <p><code>z = fzero('fun', x, tol, trace)</code> displays information at each iteration.</p> <p><code>z = fzero('fun', x, tol, trace, P1, P2, ...)</code> provides for additional arguments passed to the function <code>fun(x, P1, P2, ...)</code>. Pass an empty matrix for <i>tol</i> or <i>trace</i> to use the default value, for example:</p> <pre>fzero('fun', x, [], [], P1)</pre> <p>For the purposes of this command, zeros are considered to be points where the function actually crosses, not just touches, the <i>x</i>-axis.</p>				
<b>Arguments</b>	<table><tr><td><i>fun</i></td><td>A string containing the name of a file in which an arbitrary function of one variable is defined.</td></tr><tr><td><i>x</i></td><td>Your initial estimate of the <i>x</i>-coordinate of a zero of the function or an interval in which you think a zero is found.</td></tr></table>	<i>fun</i>	A string containing the name of a file in which an arbitrary function of one variable is defined.	<i>x</i>	Your initial estimate of the <i>x</i> -coordinate of a zero of the function or an interval in which you think a zero is found.
<i>fun</i>	A string containing the name of a file in which an arbitrary function of one variable is defined.				
<i>x</i>	Your initial estimate of the <i>x</i> -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  $\pi$  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
fmi n	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.

<b>Purpose</b>	Test matrices
<b>Syntax</b>	$[A, B, C, \dots] = \text{gallery}('tmfun', P1, P2, \dots)$ $\text{gallery}(3)$ a badly conditioned 3-by-3 matrix $\text{gallery}(5)$ an interesting eigenvalue problem
<b>Description</b>	$[A, B, C, \dots] = \text{gallery}('tmfun', P1, P2, \dots)$ returns the test matrices specified by string <i>tmfun</i> . <i>tmfun</i> is the name of a matrix family selected from the table below. <i>P1</i> , <i>P2</i> , ... are input parameters required by the individual matrix family. The number of optional parameters <i>P1</i> , <i>P2</i> , ... 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.

# gallery

Test Matrices			
cauchy	chebspec	chebvand	chow
circul	clement	compar	condex
cyclo	dorr	dramadah	feidler
forsythe	frank	gearmat	grcar
hanowa	house	invhess	invol
ipjfact	jordblloc	kahan	kms
krylov	lauchli	lehmer	lesp
lotkin	minij	moler	neumann
orthog	parte	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  $0 < x(1) < \dots < x(n)$  and  $0 < y(1) < \dots < y(n)$ .

### chebspec—Chebyshev spectral differentiation matrix

`C = gallery('chebspec', n, swi tch)` returns a Chebyshev spectral differentiation matrix of order  $n$ . Argument `swi tch` is a variable that determines the character of the output matrix. By default, `swi tch = 0`.

For `swi tch = 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 `swi tch = 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, al pha, del ta)` returns  $A$  such that  $A = H(\text{al pha}) + \text{del ta} * \text{eye}(n)$ , where  $H_{i,j}(\alpha) = \alpha^{(i-j+1)}$ . Argument  $n$  is the order of the Chow matrix, while `al pha` and `del ta` are scalars with default values 1 and 0, respectively.

$H(\text{al pha})$  has  $p = \text{floor}(n/2)$  eigenvalues that are equal to zero. The rest of the eigenvalues are equal to  $4 * \text{al pha} * \cos(k * \pi / (n+2))^2$ ,  $k=1:n-p$ .

## circul—Circulant matrix

`C = gallery('circul', v)` returns the circulant matrix whose first row is the vector `v`.

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. It is a special Toeplitz matrix in which the diagonals “wrap around.”

If `v` is a scalar, then `C = gallery('circul', 1, v)`.

The eigensystem of `C` (`n`-by-`n`) is known explicitly: If `t` is an `n`th root of unity, then the inner product of `v` with  $w = [1 \ t \ t^2 \dots \ t^n]$  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  $n-1, n-3, n-5, \dots$ , 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) - triu(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 $\theta$ )
$k = 4$	$n \geq 4$	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('cyclo', [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('cyclo', n, k)`, where  $n$  is a scalar, is the same as `gallery('cyclo', [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  $\text{mu}(A) = \text{norm}(\text{inv}(A), 'fro')$  is relatively large, although not necessarily maximal. An anti-Hadamard matrix  $A$  is a matrix with elements 0 or 1 for which  $\text{mu}(A)$  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  $\text{abs}(\det(A)) = 1$ , and  $\text{mu}(A) > c(1.75)^n$ , 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  $\text{abs}(n(i)-n(j))$ . 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  $\text{inv}(A)$  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  $\text{inv}(A)$  is tridiagonal except for nonzero  $(1, n)$  and  $(n, 1)$  elements.

**forsythe—Perturbed Jordan block**

`A = gallery('forsythe', n, al pha, l ambda)` returns the n-by-n matrix equal to the Jordan block with eigenvalue `l ambda`, excepting that `A(n, 1) = al pha`. The default values of scalars `al pha` and `l ambda` 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  $(1, \text{abs}(i))$  position, `sign(j)` in the  $(n, n+1-\text{abs}(j))$  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  $[\sin(w+a), \sin(w+2a), \dots, \sin(w+Na)]$ , 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 -1s on the subdiagonal, 1s on the diagonal, and `k` superdiagonals of 1s. The default is `k = 3`. The eigenvalues are sensitive.

## hanowa—Matrix whose eigenvalues lie on a vertical line in the complex plane

`A = gallery(' hanowa' , n, d)` returns an n-by-n block 2-by-2 matrix of the form:

$$\begin{bmatrix} d*\text{eye}(m) & -d\text{ag}(1:m) \\ d\text{ag}(1:m) & d*\text{eye}(m) \end{bmatrix}$$

Argument n is an even integer  $n=2*m$ . Matrix A has complex eigenvalues of the form  $d \pm k*i$ , for  $1 \leq k \leq m$ . 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  $\text{eye}(n, n) - \beta*v*v'$  is a Householder matrix.

A Householder matrix H satisfies the relationship

$$H*x = -\text{sign}(x(1)) * \text{norm}(x) * e1$$

where e1 is the first column of  $\text{eye}(n, n)$ . Note that if x is complex, then  $\text{sign}(x) = \exp(i*\text{arg}(x))$  (which equals  $x./\text{abs}(x)$  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  $\text{ones}(n, 1)*x'$  and whose strict upper triangle agrees with that of  $[1 y]*\text{ones}(1, n)$ .

The matrix is nonsingular if  $x(1) \neq 0$  and  $x(i+1) \neq y(i)$  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 ( $A^*A = \text{eye}(n)$ ) 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-Szegő Toeplitz matrix**

`A = gallery('kms', n, rho)` returns the  $n$ -by- $n$  Kac-Murdock-Szegő Toeplitz matrix such that  $A(i, j) = \rho^{\lvert i-j \rvert}$ , 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  $L = \text{inv}(\text{triw}(n, -\rho, 1))$ , and  $D(i, i) = (1 - \text{abs}(\rho))^2 * \text{eye}(n)$ , except  $D(1, 1) = 1$ .
- Positive definite if and only if  $0 < \text{abs}(\rho) < 1$ .
- The inverse  $\text{inv}(A)$  is tridiagonal.

## krylov—Krylov matrix

`B = gallery('krylov', A, x, j)` returns the Krylov matrix

$$[x, Ax, A^2x, \dots, A^{(j-1)}x]$$

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

$$[\text{ones}(1, n); \mu * \text{eye}(n)]$$

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  $j \geq i$ .

The Lehmer matrix A has these properties:

- A is totally nonnegative.
- The inverse  $\text{inv}(A)$  is tridiagonal and explicitly known.
- The order  $n \leq \text{cond}(A) \leq 4*n*n$ .

**l esp—Tridiagonal matrix with real, sensitive eigenvalues**

`A = gallery('l esp', n)` returns an  $n$ -by- $n$  matrix whose eigenvalues are real and smoothly distributed in the interval approximately  $[-2\sqrt{n}-3.5, -4.5]$ .

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  $D = \text{diag}(1!, 2!, \dots, n!)$ .

**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  $U^T * U$ , where  $U = \text{triu}(n, alpha)$ .

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 `k`th 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:

- |                     |  |
|---------------------|--|
| <code>k = 1</code>  | $Q(i, j) = \sqrt{2/(n+1)} * \sin(i*j*pi/(n+1))$<br>Symmetric eigenvector matrix for second difference matrix. This is the default.   |
| <code>k = 2</code>  | $Q(i, j) = 2/\sqrt{2*n+1} * \sin(2*i*j*pi/(2*n+1))$<br>Symmetric.  |
| <code>k = 3</code>  | $Q(r, s) = \exp(2*pi*i*(r-1)*(s-1)/n) / \sqrt{n}$<br>Unitary, the Fourier matrix. $Q^4$ is the identity. This is essentially the same matrix as <code>fft(eye(n))/sqrt(n)</code> ! |
| <code>k = 4</code>  | Helmut matrix: a permutation of a lower Hessenberg matrix, whose first row is <code>ones(1: n)/sqrt(n)</code> .  |
| <code>k = 5</code>  | $Q(i, j) = \sin(2*pi*(i-1)*(j-1)/n) + \cos(2*pi*(i-1)*(j-1)/n)$<br>Symmetric matrix arising in the Hartley transform.  |
| <code>k = -1</code> | $Q(i, j) = \cos((i-1)*(j-1)*pi/(n-1))$<br>Chebyshev Vandermonde-like matrix, based on extrema of $T(n-1)$ .  |
| <code>k = -2</code> | $Q(i, j) = \cos((i-1)*(j-1/2)*pi/n)$<br>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  $\text{cond}(A) = \kappa$  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)

- 1 One large singular value
- 4 Arithmetically distributed singular values
- 5 Random singular values with uniformly distributed logarithm
- < 0 If mode is -1, -2, -3, -4, or -5, then randsvd treats mode as abs(mode), except that in the original matrix of singular values the order of the diagonal entries is reversed: small to large instead of large to small.

Condition number kappa defaults to `sqrt(1/eps)`. In the special case where  $\kappa < 0$ , A is a random, full, symmetric, positive definite matrix with  $\text{cond}(A) = -\kappa$  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:

- $(n-\lfloor \log_2(n) \rfloor)-1$  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  $\text{abs}(z) = 1$ ,” 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  $\text{abs}(e(i)) \leq m-1/m$ , where  $m = n+1$ .
- $i \leq e(i) \leq i+1$  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  $n$ th roots of unity; those of `smoke(n)` are the  $n$ th roots of unity times  $2^{(1/n)}$ .

### **toeppd—Symmetric positive definite Toeplitz matrix**

`A = gallery('toeppd', n, m, w, theta)` returns an  $n$ -by- $n$  symmetric, positive semi-definite (SPD) Toeplitz matrix composed of the sum of  $m$  rank 2 (or, for certain `theta`, rank 1) SPD Toeplitz matrices. Specifically,

$$T = w(1)*T(theta(1)) + \dots + w(m)*T(theta(m))$$

where `T(theta(k))` has  $(i, j)$  element  $\cos(2\pi\theta(k)(i-j))$ .

By default:  $m = n$ ,  $w = \text{rand}(m, 1)$ , and  $\theta = \text{rand}(m, 1)$ .

### **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  $2\cos(2t) + 20i\sin(t)$ .

### **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  $\text{length}(d)-1$ .

`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

$$d + 2\sqrt{c \cdot e} \cdot \cos(k\pi / (n+1))$$

where  $k = 1:n$ . (see [1].)

# gallery

---

`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 alphas on the first  $k \geq 0$  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

$$\text{cond}(\text{gallery('triw', n, 2)}) = \cot(\pi / (4*n))^2,$$

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

$$0.25 \leq \text{eig}(\text{inv}(D) * A) \leq 4.5$$

where  $D = \text{diag}(\text{diag}(A))$  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
$n = 3$	$[A, b] = \text{gallery('wilk', 3)}$	Upper triangular system $Ux=b$ illustrating inaccurate solution.
$n = 4$	$[A, b] = \text{gallery('wilk', 4)}$	Lower triangular system $Lx=b$ , ill-conditioned.
$n = 5$	$A = \text{gallery('wilk', 5)}$	$\text{hilb}(6)(1:5, 2:6)*1.8144$ . A symmetric positive definite matrix.
$n = 21$	$A = \text{gallery('wilk', 21)}$	W21+, tridiagonal matrix. Eigenvalue problem.

# gallery

---

## 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 Tool box 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.math.man.ac.uk/pub/narep> or <http://www.math.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.

<b>Purpose</b>	Gamma functions	
<b>Syntax</b>	$Y = \text{gamma}(A)$	Gamma function
	$Y = \text{gammairc}(X, A)$	Incomplete gamma function
	$Y = \text{gammaln}(A)$	Logarithm of gamma function
<b>Definition</b>	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:	
	$\text{gamma}(n+1) = n! = \text{prod}(1:n)$	
	The incomplete gamma function is:	
	$P(x, a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$	
<b>Description</b>	$Y = \text{gamma}(A)$ returns the gamma function at the elements of A. A must be real.	
	$Y = \text{gammairc}(X, A)$ 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).	
	$Y = \text{gammaln}(A)$ returns the logarithm of the gamma function, $\text{gammaln}(A) = \log(\text{gamma}(A))$ . The gammaln command avoids the underflow and overflow that may occur if it is computed directly using $\log(\text{gamma}(A))$ .	
<b>Algorithm</b>	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].	

## gamma, gammairc, gammaln

---

### 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.

<b>Purpose</b>	Greatest common divisor
<b>Syntax</b>	$G = \text{gcd}(A, B)$ $[G, C, D] = \text{gcd}(A, B)$
<b>Description</b>	$G = \text{gcd}(A, B)$ returns an array containing the greatest common divisors of the corresponding elements of integer arrays A and B. By convention, $\text{gcd}(0, 0)$ returns a value of 0; all other inputs return positive integers for G.  $[G, C, D] = \text{gcd}(A, B)$ returns both the greatest common divisor array G, and the arrays C and D, which satisfy the equation: $A(i) \cdot C(i) + B(i) \cdot D(i) = G(i)$ . These are useful for solving Diophantine equations and computing elementary Hermite transformations.
<b>Examples</b>	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 ( <i>a unimodular matrix</i> ) such that:  $E * [a; b] = [g, 0],$ where g is the greatest common divisor of a and b as returned by the command $[g, c, d] = \text{gcd}(a, b).$  The matrix E equals:  $\begin{matrix} c & d \\ -b/g & a/g \end{matrix}$ In the case where a = 2 and b = 4:  $\begin{aligned} [g, c, d] &= \text{gcd}(2, 4) \\ g &= 2 \\ c &= 1 \\ d &= 0 \end{aligned}$

# gcd

---

So that:

$$\begin{array}{rcl} E = & & \\ \begin{array}{cc} 1 & 0 \\ -2 & 1 \end{array} & & \end{array}$$

In the next example, we solve for  $x$  and  $y$  in the Diophantine equation  
 $30x + 56y = 8$ .

$$[g, c, d] = \text{gcd}(30, 56)$$

$$\begin{array}{rcl} g = & & 2 \\ & & \end{array}$$

$$\begin{array}{rcl} c = & & -13 \\ & & \end{array}$$

$$\begin{array}{rcl} d = & & 7 \\ & & \end{array}$$

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:

$$x = (-13*4) = -52; \quad y = (7*4) = 28$$

## 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.

---

<b>Purpose</b>	Macintosh gestalt function
<b>Syntax</b>	<code>gestal tbi ts = gestal t(' selector' )</code>
<b>Description</b>	<code>gestal tbi ts = gestal t(' selector' )</code> passes the four-character string <i>selector</i> to the Macintosh Operating System function <code>gestal t</code> . For details about <code>gestal t</code> , refer to Chapter 1 of <i>Inside Macintosh: Operating System Utilities</i> .
<b>Example</b>	The result, a 32-bit integer, is stored bitwise in <code>gestal tbi ts</code> . Thus, the least significant bit of the result is <code>gestal tbi ts(32)</code> , while the most significant bit is <code>gestal tbi ts(1)</code> .  After executing:  <code>gestal tbi ts = gestal t(' sysa' )</code> <code>gestal tbi ts(32)</code> will be 1 if run from a 680x0-based Macintosh, while <code>gestal tbi ts(31)</code> will be 1 if run from a PowerPC-based Macintosh.

# getfield

---

<b>Purpose</b>	Get field of structure array
<b>Syntax</b>	<pre>f = getfield(s, 'field') f = getfield(s, {i,j}, 'field', {k})</pre>
<b>Description</b>	<p><code>f = getfield(s, 'field')</code>, where <code>s</code> is a 1-by-1 structure, returns the contents of the specified field. This is equivalent to the syntax <code>f = s.field</code>.</p> <p><code>f = getfield(s, {i,j}, 'field', {k})</code> returns the contents of the specified field. This is equivalent to the syntax <code>f = s(i,j).field(k)</code>. All subscripts must be passed as cell arrays—that is, they must be enclosed in curly braces (similar to <code>{i,j}</code> and <code>{k}</code> above). Pass field references as strings.</p>
<b>Examples</b>	<p>Given the structure:</p> <pre>mystr(1, 1).name = 'alice'; mystr(1, 1).ID = 0; mystr(2, 1).name = 'gertrude'; mystr(2, 1).ID = 1</pre> <p>Then the command <code>f = getfield(mystr, {2, 1}, 'name')</code> yields</p> <pre>f = gertrude</pre> <p>To list the contents of all name (or other) fields, embed <code>getfield</code> in a loop:</p> <pre>for i = 1:2     name{i} = getfield(mystr, {i, 1}, 'name'); end name  name = 'alice'      'gertrude'</pre>
<b>See Also</b>	<a href="#">fields</a> <a href="#">Field names of a structure</a> <a href="#">setfield</a> <a href="#">Set field of structure array</a>

<b>Purpose</b>	Define global variables
<b>Syntax</b>	<code>global X Y Z</code>
<b>Description</b>	<code>global X Y Z</code> 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, <i>all</i> 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 <code>global</code> statement, it is initialized 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:
	<ul style="list-style-type: none"> <li>• in the current workspace, a variable with the same name exists.</li> <li>• in an M-file, it has been referenced previously.</li> </ul>
<b>Remarks</b>	Use <code>clear global variable</code> to clear a global variable from the global workspace. Use <code>clear variable</code> 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:
	<pre>ui control ('style', 'pushbutton', 'CallBack', ... 'global MY_GLOBAL, disp(MY_GLOBAL), MY_GLOBAL = MY_GLOBAL+1, clear MY_GLOBAL',... 'string', 'count')</pre>
<b>Examples</b>	Here is the code for the functions <code>tic</code> and <code>toc</code> (some comments abridged), which manipulate a stopwatch-like timer. The global variable <code>TICTOC</code> is shared

# global

---

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 TI CT0C
TI CT0C = 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 TI CT0C
if nargout < 1
    elapsed_time = etime(clock, TI CT0C)
else
    t = etime(clock, TI CT0C);
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  $\text{norm}(b - A^*x) / \text{norm}(b)$  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  $i\text{nv}(M)^*A^*x = i\text{nv}(M)^*b$  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  $x_0$ . If  $x_0$  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, rel res] = gmres(A, b, restart, tol, maxit, M1, M2, x0)` also returns the relative residual  $\text{norm}(b - A^*x) / \text{norm}(b)$ . If `flag` is 0, then  $\text{rel res} \leq \text{tol}$ .

`[x, flag, rel res, 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, rel res, 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  $(i-1)*\text{restart}+j+1$  and  $\text{resvec}(\text{end}) \leq \text{tol} * \text{norm}(b)$ .

## 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);
```

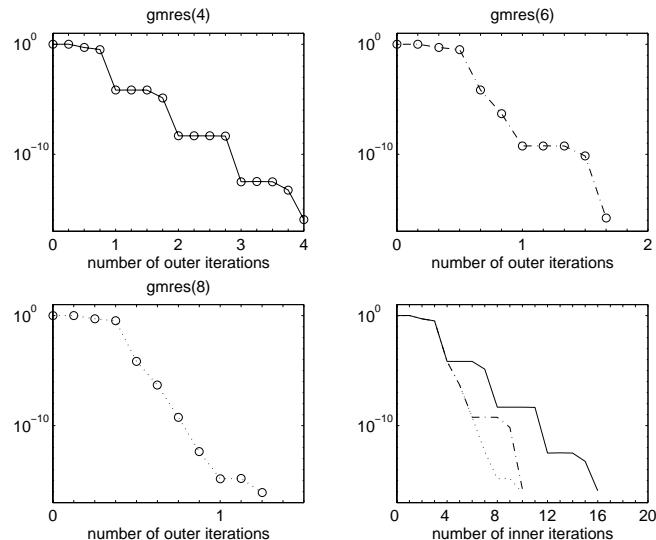
`flag1` is 2 since the upper triangular `U1` has a zero on its diagonal so `gmres(5)` fails in the first iteration when it tries to solve a system such as `U1*y = r` for `y` with backslash.

```
[L2, U2] = luinc(A, 1e-6);
tol = 1e-15;
[x4, flag4, rel res4, iter4, resvec4] = gmres(A, b, 4, tol, 5, L2, U2);
[x6, flag6, rel res6, iter6, resvec6] = gmres(A, b, 6, tol, 3, L2, U2);
[x8, flag8, rel res8, 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

# gmres

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

[bi\\_cg](#)      BiConjugate Gradients method  
[bi\\_cgst\\_ab](#)      BiConjugate Gradients Stabilized method  
[cgs](#)      Conjugate Gradients Squared method  
[l\\_ui\\_nc](#)      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**

- $\text{FX} = \text{gradient}(F)$
- $[\text{FX}, \text{FY}] = \text{gradient}(F)$
- $[\text{Fx}, \text{Fy}, \text{Fz}, \dots] = \text{gradient}(F)$
- $[\dots] = \text{gradient}(F, h)$
- $[\dots] = \text{gradient}(F, h1, h2, \dots)$

**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,  $F(x,y,z,\dots)$ ,

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

**Description**  $\text{FX} = \text{gradient}(F)$  where  $F$  is a vector returns the one-dimensional numerical gradient of  $F$ .  $\text{FX}$  corresponds to  $\partial F / \partial x$ , the differences in the  $x$  direction.

$[\text{FX}, \text{FY}] = \text{gradient}(F)$  where  $F$  is a matrix returns the  $x$  and  $y$  components of the two-dimensional numerical gradient.  $\text{FX}$  corresponds to  $\partial F / \partial x$ , the differences in the  $x$  (column) direction.  $\text{FY}$  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.

$[\text{FX}, \text{FY}, \text{FZ}, \dots] = \text{gradient}(F)$  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 ( $h1, h2, \dots$ ) specify the spacing for each dimension of  $F$ . Scalar spacing parameters specify a constant spacing for each dimension. Vector

# gradient

---

parameters specify the coordinates of the values along corresponding dimensions of F. In this case, the length of the vector must match the size of the corresponding dimension.

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

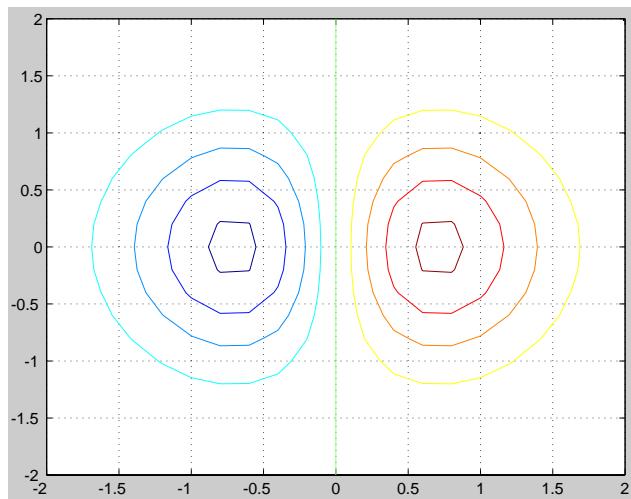
[ . . . ] = gradient(F, h1, h2, . . . ) with N spacing parameters specifies the spacing for each dimension of F.

## Examples

The statements

```
v = -2:0.2:2;
[x, y] = meshgrid(v);
z = x .* exp(-x.^2 - y.^2);
[px, py] = gradient(z, .2, .2);
contour(v, v, z), hold on, quiver(px, py), hold off
```

produce



Given,

```
F(:,:,1) = magic(3); F(:,:,2) = pascal(3);
```

```
gradient(F) takes dx = dy = dz = 1.
```

```
[PX, PY, PZ] = gradient(F, 0.2, 0.1, 0.2) takes dx = 0.2, dy = 0.1, and  
dz = 0.2.
```

## See Also

del2

diff

Discrete Laplacian

Differences and approximate derivatives

# griddata

---

<b>Purpose</b>	Data gridding								
<b>Syntax</b>	<pre>ZI = griddata(x, y, z, XI, YI) [XI, YI, ZI] = griddata(x, y, z, xi, yi) [...] = griddata(..., method)</pre>								
<b>Description</b>	<p><code>ZI = griddata(x, y, z, XI, YI)</code> fits a surface of the form <math>z = f(x, y)</math> to the data in the (usually) nonuniformly spaced vectors <math>(x, y, z)</math>. <code>griddata</code> interpolates this surface at the points specified by <math>(XI, YI)</math> to produce <code>ZI</code>. The surface always passes through the data points. <code>XI</code> and <code>YI</code> usually form a uniform grid (as produced by <code>meshgrid</code>).</p> <p><code>XI</code> can be a row vector, in which case it specifies a matrix with constant columns. Similarly, <code>YI</code> can be a column vector, and it specifies a matrix with constant rows.</p> <p><code>[XI, YI, ZI] = griddata(x, y, z, xi, yi)</code> returns the interpolated matrix <code>ZI</code> as above, and also returns the matrices <code>XI</code> and <code>YI</code> formed from row vector <code>xi</code> and column vector <code>yi</code>. These latter are the same as the matrices returned by <code>meshgrid</code>.</p> <p><code>[...] = griddata(..., method)</code> uses the specified interpolation method:</p> <table><tbody><tr><td>'linear'</td><td>Triangle-based linear interpolation (default)</td></tr><tr><td>'cubic'</td><td>Triangle-based cubic interpolation</td></tr><tr><td>'nearest'</td><td>Nearest neighbor interpolation</td></tr><tr><td>'v4'</td><td>MATLAB 4 <code>griddata</code> method</td></tr></tbody></table> <p>The <code>method</code> defines the type of surface fit to the data. The '<code>cubic</code>' and '<code>v4</code>' methods produce smooth surfaces while '<code>linear</code>' and '<code>nearest</code>' have discontinuities in the first and zero'th derivatives, respectively. All the methods except '<code>v4</code>' are based on a Delaunay triangulation of the data.</p>	'linear'	Triangle-based linear interpolation (default)	'cubic'	Triangle-based cubic interpolation	'nearest'	Nearest neighbor interpolation	'v4'	MATLAB 4 <code>griddata</code> method
'linear'	Triangle-based linear interpolation (default)								
'cubic'	Triangle-based cubic interpolation								
'nearest'	Nearest neighbor interpolation								
'v4'	MATLAB 4 <code>griddata</code> method								
<b>Remarks</b>	<p><code>XI</code> and <code>YI</code> can be matrices, in which case <code>griddata</code> returns the values for the corresponding points <math>(XI(i,j), YI(i,j))</math>. Alternatively, you can pass in the row and column vectors <code>xi</code> and <code>yi</code>, respectively. In this case, <code>griddata</code></p>								

interprets these vectors as if they were matrices produced by the command `meshgrid(xi, yi)`.

## Algorithm

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

## Examples

Sample a function at 100 random points between  $\pm 2$ :

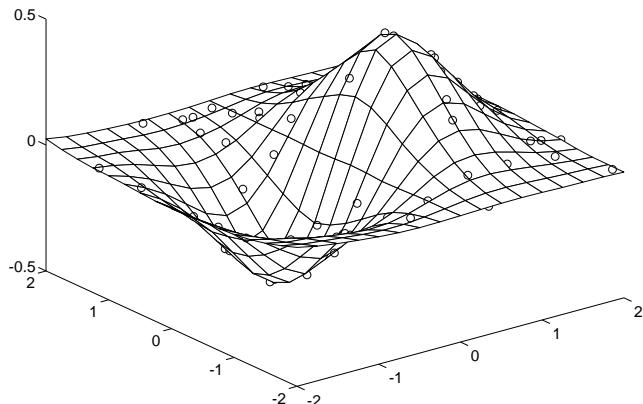
```
rand('seed', 0)
x = rand(100, 1)*4-2; y = rand(100, 1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

`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
```



# griddata

---

**See Also** `de launay`, `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.

<b>Purpose</b>	Generalized singular value decomposition
<b>Syntax</b>	$[U, V, X, C, S] = \text{gsvd}(A, B)$ $[U, V, X, C, S] = \text{gsvd}(A, B, 0)$ $\text{sigma} = \text{gsvd}(A, B)$
<b>Description</b>	<p><math>[U, V, X, C, S] = \text{gsvd}(A, B)</math> returns unitary matrices <math>U</math> and <math>V</math>, a (usually) square matrix <math>X</math>, and nonnegative diagonal matrices <math>C</math> and <math>S</math> so that</p> $\begin{aligned} A &= U * C * X' \\ B &= V * S * X' \\ C' * C + S' * S &= I \end{aligned}$ <p><math>A</math> and <math>B</math> must have the same number of columns, but may have different numbers of rows. If <math>A</math> is <math>m</math>-by-<math>p</math> and <math>B</math> is <math>n</math>-by-<math>p</math>, then <math>U</math> is <math>m</math>-by-<math>m</math>, <math>V</math> is <math>n</math>-by-<math>n</math> and <math>X</math> is <math>p</math>-by-<math>q</math> where <math>q = \min(m+n, p)</math>.</p> <p><math>\text{sigma} = \text{gsvd}(A, B)</math> returns the vector of generalized singular values, <math>\text{sqrt}(\text{diag}(C' * C) ./ \text{diag}(S' * S))</math>.</p> <p>The nonzero elements of <math>S</math> are always on its main diagonal. If <math>m \geq p</math> the nonzero elements of <math>C</math> are also on its main diagonal. But if <math>m &lt; p</math>, the nonzero diagonal of <math>C</math> is <math>\text{diag}(C, p-m)</math>. This allows the diagonal elements to be ordered so that the generalized singular values are nondecreasing.</p> <p><math>\text{gsvd}(A, B, 0)</math>, with three input arguments and either <math>m</math> or <math>n \geq p</math>, produces the “economy-sized” decomposition where the resulting <math>U</math> and <math>V</math> have at most <math>p</math> columns, and <math>C</math> and <math>S</math> have at most <math>p</math> rows. The generalized singular values are <math>\text{diag}(C) ./ \text{diag}(S)</math>.</p> <p>When <math>B</math> is square and nonsingular, the generalized singular values, <math>\text{gsvd}(A, B)</math>, are equal to the ordinary singular values, <math>\text{svd}(A/B)</math>, but they are sorted in the opposite order. Their reciprocals are <math>\text{gsvd}(B, A)</math>.</p> <p>In this formulation of the gsvd, no assumptions are made about the individual ranks of <math>A</math> or <math>B</math>. The matrix <math>X</math> has full rank if and only if the matrix <math>[A; B]</math> has full rank. In fact, <math>\text{svd}(X)</math> and <math>\text{cond}(X)</math> are equal to <math>\text{svd}([A; B])</math> and <math>\text{cond}([A; B])</math>. Other formulations, eg. G. Golub and C. Van Loan [1], require that <math>\text{nul1}(A)</math> and <math>\text{nul1}(B)</math> do not overlap and replace <math>X</math> by <math>\text{inv}(X)</math> or <math>\text{inv}(X')</math>.</p> <p>Note, however, that when <math>\text{nul1}(A)</math> and <math>\text{nul1}(B)</math> do overlap, the nonzero elements of <math>C</math> and <math>S</math> are not uniquely determined.</p>

## gsvd

---

### Examples

In the first example, the matrices have at least as many rows as columns.

```
A = reshape(1:15, 5, 3)
```

```
B = magic(3)
```

A =

1	6	11
2	7	12
3	8	13
4	9	14
5	10	15

B =

8	1	6
3	5	7
4	9	2

The statement

```
[U, V, X, C, S] = gsvd(A, B)
```

produces a 5-by-5 orthogonal U, a 3-by-3 orthogonal V, a 3-by-3 nonsingular X,

X =

-2.8284	9.3761	-6.9346
5.6569	8.3071	-18.3301
-2.8284	7.2381	-29.7256

and

C =

0.0000	0	0
0	0.3155	0
0	0	0.9807
0	0	0
0	0	0

S =

1.0000	0	0
0	0.9489	0
0	0	0.1957

Since A is rank deficient, the first diagonal element of C is zero.

The economy sized decomposition,

`[U, V, X, C, S] = gsvd(A, B, 0)`

produces a 5-by-3 matrix U and a 3-by-3 matrix C.

`U =`

-0.3736	-0.6457	-0.4279
-0.0076	-0.3296	-0.4375
0.8617	-0.0135	-0.4470
-0.2063	0.3026	-0.4566
-0.2743	0.6187	-0.4661

`C =`

0.0000	0	0
0	0.3155	0
0	0	0.9807

The other three matrices, V, X, and S are the same as those obtained with the full decomposition.

The generalized singular values are the ratios of the diagonal elements of C and S.

`sigma = gsvd(A, B)`

`sigma =`

0.0000
0.3325
5.0123

These values are a reordering of the ordinary singular values

`svd(A/B)`

`ans =`

5.0123
0.3325
0.0000

## gsvd

---

In the second example, the matrices have at least as many columns as rows.

```
A = reshape(1: 15, 3, 5)
```

```
B = magic(5)
```

```
A =
```

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

```
B =
```

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

The statement

```
[U, V, X, C, S] = gsvd(A, B)
```

produces a 3-by-3 orthogonal U, a 5-by-5 orthogonal V, a 5-by-5 nonsingular X and

```
C =
```

0	0	0. 0000	0	0
0	0	0	0. 0439	0
0	0	0	0	0. 7432

```
S =
```

1. 0000	0	0	0	0
0	1. 0000	0	0	0
0	0	1. 0000	0	0
0	0	0	0. 9990	0
0	0	0	0	0. 6690

In this situation, the nonzero diagonal of C is `diag(C, 2)`. The generalized singular values include three zeros.

```
sigma = gsvd(A, B)

sigma =
    0
    0
    0.0000
    0.0439
    1.1109
```

Reversing the roles of A and B reciprocates these values, producing three infinities.

```
gsvd(B, A)

ans =
    0.9001
    22.7610
    Inf
    Inf
    Inf
```

## Algorithm

The generalized singular value decomposition uses the C-S decomposition described in [1], as well as the built-in `svd` and `qr` functions. The C-S decomposition is implemented in a subfunction in the `gsvd` M-file.

## Diagnostics

The only warning or error message produced by `gsvd` itself occurs when the two input arguments do not have the same number of columns.

## Reference

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

## See Also

`svd`      Singular value decomposition

# hadamard

---

<b>Purpose</b>	Hadamard matrix
<b>Syntax</b>	$H = \text{hadamard}(n)$
<b>Description</b>	$H = \text{hadamard}(n)$ returns the Hadamard matrix of order $n$ .
<b>Definition</b>	Hadamard matrices are matrices of 1's and -1's whose columns are orthogonal, $H' * H = n*I$ where $[n \ n] = \text{size}(H)$ and $I = \text{eye}(n,n)$ . 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 $\text{rem}(n, 4) = 0$ . This function handles only the cases where $n$ , $n/12$ , or $n/20$ is a power of 2.
<b>Examples</b>	The command $\text{hadamard}(4)$ produces the 4-by-4 matrix: $\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{matrix}$
<b>See Also</b>	<a href="#">compan</a> Companion matrix <a href="#">hankel</a> Hankel matrix <a href="#">toeplitz</a> Toeplitz matrix
<b>References</b>	[1] Ryser, H. J., <i>Combinatorial Mathematics</i> , John Wiley and Sons, 1963. [2] Pratt, W. K., <i>Digital Signal Processing</i> , John Wiley and Sons, 1978.

<b>Purpose</b>	Hankel matrix
<b>Syntax</b>	$H = \text{hankel}(c)$ $H = \text{hankel}(c, r)$
<b>Description</b>	$H = \text{hankel}(c)$ returns the square Hankel matrix whose first column is $c$ and whose elements are zero below the first anti-diagonal. $H = \text{hankel}(c, r)$ 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.
<b>Definition</b>	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 $p = [c \ r(2: \text{end})]$ completely determines the Hankel matrix.
<b>Examples</b>	A Hankel matrix with anti-diagonal disagreement is  $c = 1: 3; r = 7: 10;$ $h = \text{hankel}(c, r)$ $h =$ $\begin{matrix} 1 & 2 & 3 & 8 \\ 2 & 3 & 8 & 9 \\ 3 & 8 & 9 & 10 \end{matrix}$  $p = [1 \ 2 \ 3 \ 8 \ 9 \ 10]$
<b>See Also</b>	<a href="#">hadamard</a> Hadamard matrix <a href="#">toeplitz</a> Toeplitz matrix

# **hdf**

---

<b>Purpose</b>	HDF interface
<b>Syntax</b>	<code>hdf*(functstr, param1, param2, . . .)</code>
<b>Description</b>	MATLAB provides a set of functions that enable you to access the HDF library developed and supported by the National Center for Supercomputing Applications (NCSA). MATLAB supports all or a portion of these HDF interfaces: SD, V, VS, AN, DRF8, DF24, H, HE, and HD.  To use these functions you must be familiar with the HDF library. Documentation for the library is available on the NCSA HDF Web page at <a href="http://hdf.ncsa.uiuc.edu">http://hdf.ncsa.uiuc.edu</a> . MATLAB additionally provides extensive command line help for each of the provided functions.
This table lists the interface-specific HDF functions that MATLAB provides:	
Function	Interface
<code>hdfan</code>	Multifile annotation
<code>hdfdf24</code>	24-bit raster image
<code>hdfdfr8</code>	8-bit raster image
<code>hdfh</code>	HDF H interface
<code>hdfhd</code>	HDF HD interface
<code>hdfhe</code>	HDF HE interface
<code>hdfml</code>	Gateway utilities
<code>hdfsd</code>	Multifile scientific data set
<code>hdfv</code>	Vgroup
<code>hdfvf</code>	Vdata VF functions
<code>hdfvh</code>	Vdata VH functions
<code>hdfvs</code>	Vdata VS functions

**See Also**

`imfinfo`, `imread`, `imwrite`

<b>Purpose</b>	Online help for MATLAB functions and M-files
<b>Syntax</b>	<code>hel p</code> <code>hel p <i>topic c</i></code>
<b>Description</b>	<code>hel p</code> , by itself, lists all primary help topics. Each main help topic corresponds to a directory name on MATLAB's search path.  <code>hel p <i>topic c</i></code> 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, <code>hel p</code> displays information on that function. If it is a directory name, <code>hel p</code> 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.
<b>Remarks</b>	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 <code>hel p</code> , 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 <code>Contents.m</code> within each directory.  The command <code>hel p <i>topic c</i></code> , where <i>topic c</i> is a directory name, displays the comment lines in the <code>Contents.m</code> file located in that directory. If a contents file does not exist, <code>hel p</code> displays the H1 lines of all the files in the directory.  The command <code>hel p <i>topic c</i></code> , where <i>topic c</i> is a function name, displays help on the function by listing the first contiguous comment lines in the M-file <i>topic c.m</i> .  <b>Creating Online Help for Your Own M-Files</b> 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 <i>Applying MATLAB</i> 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**

<code>dir</code>	Directory listing
<code>lookfor</code>	Keyword search through all help entries
<code>more</code>	Control paged output for the command window
<code>path</code>	Control MATLAB's directory search path
<code>what</code>	Directory listing of M-files, MAT-files, and MEX-files
<code>which</code>	Locate functions and files

See also `partialpath`.

---

<b>Purpose</b>	Hessenberg form of a matrix									
<b>Syntax</b>	$[P, H] = \text{hess}(A)$ $H = \text{hess}(A)$									
<b>Description</b>	$H = \text{hess}(A)$ finds $H$ , the Hessenberg form of matrix $A$ .  $[P, H] = \text{hess}(A)$ produces a Hessenberg matrix $H$ and a unitary matrix $P$ so that $A = P * H * P'$ and $P' * P = \text{eye}(\text{size}(A))$ .									
<b>Definition</b>	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.									
<b>Examples</b>	$H$ is a 3-by-3 eigenvalue test matrix:									
	$H =$ <table style="margin-left: 20px; border-collapse: collapse;"> <tr><td>-149</td><td>-50</td><td>-154</td></tr> <tr><td>537</td><td>180</td><td>546</td></tr> <tr><td>-27</td><td>-9</td><td>-25</td></tr> </table>	-149	-50	-154	537	180	546	-27	-9	-25
-149	-50	-154								
537	180	546								
-27	-9	-25								
	Its Hessenberg form introduces a single zero in the (3,1) position:									
	$\text{hess}(H) =$ <table style="margin-left: 20px; border-collapse: collapse;"> <tr><td>-149. 0000</td><td>42. 2037</td><td>-156. 3165</td></tr> <tr><td>-537. 6783</td><td>152. 5511</td><td>-554. 9272</td></tr> <tr><td>0</td><td>0. 0728</td><td>2. 4489</td></tr> </table>	-149. 0000	42. 2037	-156. 3165	-537. 6783	152. 5511	-554. 9272	0	0. 0728	2. 4489
-149. 0000	42. 2037	-156. 3165								
-537. 6783	152. 5511	-554. 9272								
0	0. 0728	2. 4489								
<b>Algorithm</b>	For real matrices, <code>hess</code> 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 <code>hess</code> 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 <i>EISPACK Guide</i> .									
<b>See Also</b>	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 30%;"><code>eig</code></td> <td>Eigenvalues and eigenvectors</td> </tr> <tr> <td><code>qz</code></td> <td>QZ factorization for generalized eigenvalues</td> </tr> <tr> <td><code>schur</code></td> <td>Schur decomposition</td> </tr> </table>	<code>eig</code>	Eigenvalues and eigenvectors	<code>qz</code>	QZ factorization for generalized eigenvalues	<code>schur</code>	Schur decomposition			
<code>eig</code>	Eigenvalues and eigenvectors									
<code>qz</code>	QZ factorization for generalized eigenvalues									
<code>schur</code>	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.

---

<b>Purpose</b>	IEEE hexadecimal to decimal number conversion
<b>Syntax</b>	<code>d = hex2dec('hex_value')</code>
<b>Description</b>	<code>d = hex2dec('hex_value')</code> converts <i>hex_value</i> to its floating-point integer representation. The argument <i>hex_value</i> is a hexadecimal integer stored in a MATLAB string. If <i>hex_value</i> is a character array, each row is interpreted as a hexadecimal string.
<b>Examples</b>	<code>hex2dec('3ff')</code> is 1023. For a character array S
	<pre>S = 0FF 2DE 123  hex2dec(S) ans =     255     734     291</pre>
<b>See Also</b>	<a href="#">dec2hex</a> Decimal to hexadecimal number conversion <a href="#">format</a> Control the output display format <a href="#">hex2num</a> Hexadecimal to double number conversion <a href="#">sprintf</a> Write formatted data to a string

# hex2num

---

<b>Purpose</b>	Hexadecimal to double number conversion						
<b>Syntax</b>	<code>f = hex2num('hex_value')</code>						
<b>Description</b>	<code>f = hex2num('hex_value')</code> converts <i>hex_value</i> 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.						
<b>Examples</b>	<pre>f = hex2num('400921fb54442d18') f =     3.14159265358979</pre>						
<b>Limitations</b>	hex2num only works for IEEE numbers; it does not work for the floating-point representation of the VAX or other non-IEEE computers.						
<b>See Also</b>	<table><tr><td><code>format</code></td><td>Control the output display format</td></tr><tr><td><code>hex2dec</code></td><td>IEEE hexadecimal to decimal number conversion</td></tr><tr><td><code>sprintf</code></td><td>Write formatted data to a string</td></tr></table>	<code>format</code>	Control the output display format	<code>hex2dec</code>	IEEE hexadecimal to decimal number conversion	<code>sprintf</code>	Write formatted data to a string
<code>format</code>	Control the output display format						
<code>hex2dec</code>	IEEE hexadecimal to decimal number conversion						
<code>sprintf</code>	Write formatted data to a string						

---

<b>Purpose</b>	Hilbert matrix
<b>Syntax</b>	<code>H = hilb(n)</code>
<b>Description</b>	<code>H = hilb(n)</code> returns the Hilbert matrix of order $n$ .
<b>Definition</b>	The Hilbert matrix is a notable example of a poorly conditioned matrix [1]. The elements of the Hilbert matrices are $H(i, j) = 1/(i + j - 1)$ .
<b>Examples</b>	Even the fourth-order Hilbert matrix shows signs of poor conditioning.  <code>cond(hilb(4)) =</code> 1. 5514e+04
<b>Algorithm</b>	See the M-file for a good example of efficient MATLAB programming where conventional for loops are replaced by vectorized statements.
<b>See Also</b>	<code>invhilb</code> Inverse of the Hilbert matrix
<b>References</b>	[1] Forsythe, G. E. and C. B. Moler, <i>Computer Solution of Linear Algebraic Systems</i> , Prentice-Hall, 1967, Chapter 19.

**i**

---

<b>Purpose</b>	Imaginary unit								
<b>Syntax</b>	$i$ $a+bi$ $x+i *y$								
<b>Description</b>	As the basic imaginary unit $\text{sqrt}(-1)$ , $i$ 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.								
<b>Examples</b>	$Z = 2+3i$ $Z = x+i *y$ $Z = r*\exp(i *theta)$								
<b>See Also</b>	<table><tr><td><code>conj</code></td><td>Complex conjugate</td></tr><tr><td><code>i mag</code></td><td>Imaginary part of a complex number</td></tr><tr><td><code>j</code></td><td>Imaginary unit</td></tr><tr><td><code>real</code></td><td>Real part of complex number</td></tr></table>	<code>conj</code>	Complex conjugate	<code>i mag</code>	Imaginary part of a complex number	<code>j</code>	Imaginary unit	<code>real</code>	Real part of complex number
<code>conj</code>	Complex conjugate								
<code>i mag</code>	Imaginary part of a complex number								
<code>j</code>	Imaginary unit								
<code>real</code>	Real part of complex number								

---

<b>Purpose</b>	Conditionally execute statements
<b>Syntax</b>	<pre>if expression     statements end if expression1     statements elseif expression2     statements else     statements end</pre>
<b>Description</b>	<p>if conditionally executes statements.</p> <p>The simple form is:</p> <pre>if expression     statements end</pre> <p>More complicated forms use else or elseif. Each if must be paired with a matching end.</p>
<b>Arguments</b>	<p><i>expression</i>    A MATLAB expression, usually consisting of smaller expressions or variables joined by relational operators (==, &lt;, &gt;, &lt;=, &gt;=, or ~=). Two examples are: count &lt; limit and (height - offset) &gt;= 0. Expressions may also include logical functions, as in: isreal(A). Simple expressions can be combined by logical operators (&amp;,  , ~) into compound expressions such as: (count &lt; limit) &amp; ((height - offset) &gt;= 0).</p> <p><i>statements</i>    One or more MATLAB statements to be executed only if the <i>expression</i> is true (or nonzero). See Examples for information about how nonscalar variables are evaluated.</p>

# if

---

## 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:

$$\begin{array}{cc} A = & B = \\ \begin{matrix} 1 & 0 \\ 2 & 3 \end{matrix} & \begin{matrix} 1 & 1 \\ 3 & 4 \end{matrix} \end{array}$$

The expression:

$A < B$	Evaluates as <i>false</i>	Since $A(1, 1)$ is not less than $B(1, 1)$ .
$A < (B+1)$	Evaluates as <i>true</i>	Since no element of A is greater than the corresponding element of B.
$A \& B$	Evaluates as <i>false</i>	Since $A(1, 2) \mid B(1, 2)$ is <i>false</i> .
$5 > B$	Evaluates as <i>true</i>	Since every element of B is less than 5.

## See Also

break	Terminate execution of for or while loop
else	Conditionally execute statements
end	Terminate for, while, switch, try, 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

---

<b>Purpose</b>	Inverse one-dimensional fast Fourier transform
<b>Syntax</b>	$y = \text{ifft}(X)$ $y = \text{ifft}(X, n)$ $y = \text{ifft}(X, [ ], dim)$ $y = \text{ifft}(X, n, dim)$
<b>Description</b>	<p><math>y = \text{ifft}(X)</math> returns the inverse fast Fourier transform of vector <math>X</math>.</p> <p>If <math>X</math> is a matrix, <math>\text{ifft}</math> returns the inverse Fourier transform of each column of the matrix.</p> <p>If <math>X</math> is a multidimensional array, <math>\text{ifft}</math> operates on the first non-singleton dimension.</p> <p><math>y = \text{ifft}(X, n)</math> returns the <math>n</math>-point inverse fast Fourier transform of vector <math>X</math>.</p> <p><math>y = \text{ifft}(X, [ ], dim)</math> and <math>y = \text{ifft}(X, n, dim)</math> return the inverse discrete Fourier transform of <math>X</math> across the dimension <math>dim</math>.</p>
<b>Examples</b>	For any $x$ , $\text{ifft}(\text{fft}(x))$ equals $x$ to within roundoff error. If $x$ is real, $\text{ifft}(\text{fft}(x))$ may have small imaginary parts.
<b>Algorithm</b>	The algorithm for $\text{ifft}(x)$ is the same as the algorithm for $\text{fft}(x)$ , except for a sign change and a scale factor of $n = \text{length}(x)$ . So the execution time is fastest when $n$ is a power of 2 and slowest when $n$ is a large prime.
<b>See Also</b>	dftmtx, freqz, specplot, and spectrum in the Signal Processing Toolbox, and:
fft	One-dimensional fast Fourier transform
fft2	Two-dimensional fast Fourier transform
fftshift	Shift DC component of fast Fourier transform to center of spectrum

## **ifft2**

---

<b>Purpose</b>	Inverse two-dimensional fast Fourier transform						
<b>Syntax</b>	$Y = \text{ifft2}(X)$ $Y = \text{ifft2}(X, m, n)$						
<b>Description</b>	$Y = \text{ifft2}(X)$ returns the two-dimensional inverse fast Fourier transform of matrix $X$ . $Y = \text{ifft2}(X, m, n)$ returns the $m$ -by- $n$ inverse fast Fourier transform of matrix $X$ .						
<b>Examples</b>	For any $X$ , $\text{ifft2}(\text{fft2}(X))$ equals $X$ to within roundoff error. If $X$ is real, $\text{ifft2}(\text{fft2}(X))$ may have small imaginary parts.						
<b>Algorithm</b>	The algorithm for $\text{ifft2}(X)$ is the same as the algorithm for $\text{fft2}(X)$ , except for a sign change and scale factors of $[m, n] = \text{size}(X)$ . The execution time is fastest when $m$ and $n$ are powers of 2 and slowest when they are large primes.						
<b>See Also</b>	dftmtx, freqz, specplot, and spectrum in the Signal Processing Toolbox, and:  <table><tr><td><math>\text{fft2}</math></td><td>Two-dimensional fast Fourier transform</td></tr><tr><td><math>\text{fftshift}</math></td><td>Shift DC component of fast Fourier transform to center of spectrum</td></tr><tr><td><math>\text{ifft}</math></td><td>Inverse one-dimensional fast Fourier transform</td></tr></table>	$\text{fft2}$	Two-dimensional fast Fourier transform	$\text{fftshift}$	Shift DC component of fast Fourier transform to center of spectrum	$\text{ifft}$	Inverse one-dimensional fast Fourier transform
$\text{fft2}$	Two-dimensional fast Fourier transform						
$\text{fftshift}$	Shift DC component of fast Fourier transform to center of spectrum						
$\text{ifft}$	Inverse one-dimensional fast Fourier transform						

---

<b>Purpose</b>	Inverse multidimensional fast Fourier transform						
<b>Syntax</b>	$Y = \text{ifftn}(X)$ $Y = \text{ifftn}(X, \text{size}(\text{X}))$						
<b>Description</b>	$Y = \text{ifftn}(X)$ performs the N-dimensional inverse fast Fourier transform. The result $Y$ is the same size as $X$ . $Y = \text{ifftn}(X, \text{size}(\text{X}))$ pads $X$ with zeros, or truncates $X$ , to create a multidimensional array of size $\text{size}(Y)$ before performing the inverse transform. The size of the result $Y$ is $\text{size}(Y)$ .						
<b>Remarks</b>	For any $X$ , $\text{ifftn}(\text{fft}(X))$ equals $X$ within roundoff error. If $X$ is real, $\text{ifftn}(\text{fft}(X))$ may have small imaginary parts.						
<b>Algorithm</b>	$\text{ifftn}(X)$ is equivalent to						
	<pre> Y = X; for p = 1:length(size(X))     Y = ifft(Y, [], p); end </pre>						
	This computes in-place the one-dimensional inverse fast Fourier transform along each dimension of $X$ . The time required to compute $\text{ifftn}(X)$ 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.						
<b>See Also</b>	<table> <tr> <td><a href="#">fft</a></td><td>One-dimensional fast Fourier transform</td></tr> <tr> <td><a href="#">fft2</a></td><td>Two-dimensional fast Fourier transform</td></tr> <tr> <td><a href="#">fftn</a></td><td>Multidimensional fast Fourier transform</td></tr> </table>	<a href="#">fft</a>	One-dimensional fast Fourier transform	<a href="#">fft2</a>	Two-dimensional fast Fourier transform	<a href="#">fftn</a>	Multidimensional fast Fourier transform
<a href="#">fft</a>	One-dimensional fast Fourier transform						
<a href="#">fft2</a>	Two-dimensional fast Fourier transform						
<a href="#">fftn</a>	Multidimensional fast Fourier transform						

## **ifftshift**

---

<b>Purpose</b>	Inverse FFT shift
<b>Syntax</b>	<code>ifftshift(X)</code>
<b>Description</b>	<code>ifftshift</code> undoes the results of <code>fftshift</code> . If $X$ is a vector, <code>ifftshift(X)</code> swaps the left and right halves of $X$ . For matrices, <code>ifftshift(X)</code> swaps the first quadrant with the third and the second quadrant with the fourth. If $X$ is a multidimensional array, <code>ifftshift(X)</code> swaps half-spaces of $X$ along each dimension.
<b>See Also</b>	<code>fft</code> , <code>fft2</code> , <code>fftn</code> , <code>fftshift</code>

---

<b>Purpose</b>	Imaginary part of a complex number						
<b>Syntax</b>	$Y = \text{imag}(Z)$						
<b>Description</b>	$Y = \text{imag}(Z)$ returns the imaginary part of the elements of array $Z$ .						
<b>Examples</b>	<pre>imag(2+3i) ans = 3</pre>						
<b>See Also</b>	<table><tr><td>conj</td><td>Complex conjugate</td></tr><tr><td>i, j</td><td>Imaginary unit (<math>\sqrt{-1}</math>)</td></tr><tr><td>real</td><td>Real part of complex number</td></tr></table>	conj	Complex conjugate	i, j	Imaginary unit ( $\sqrt{-1}$ )	real	Real part of complex number
conj	Complex conjugate						
i, j	Imaginary unit ( $\sqrt{-1}$ )						
real	Real part of complex number						

# imfinfo

---

<b>Purpose</b>	Return information about a graphics file
<b>Synopsis</b>	<code>info = imfinfo(filename, fmt)</code> <code>info = imfinfo(filename)</code>
<b>Description</b>	<code>info = imfinfo(filename, fmt)</code> returns a structure whose fields contain information about an image in a graphics file. <code>filename</code> is a string that specifies the name of the graphics file, and <code>fmt</code> 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 <code>imfinfo</code> cannot find a file named <code>filename</code> , it looks for a file named <code>filename. fmt</code> .

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
<code>filename</code>	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
<code>fileModDate</code>	A string containing the date when the file was last modified
<code>fileSize</code>	An integer indicating the size of the file in bytes
<code>format</code>	A string containing the file format, as specified by <code>fmt</code> ; for JPEG and TIFF files, the three-letter variant is returned
<code>formatVersion</code>	A string or number describing the version of the format
<code>width</code>	An integer indicating the width of the image in pixels
<code>height</code>	An integer indicating the height of the image in pixels
<code>bitDepth</code>	An integer indicating the number of bits per pixel
<code>colorType</code>	A string indicating the type of image; either ' <code>truecolor</code> ' for a truecolor RGB image, ' <code>grayscale</code> ' for a grayscale intensity image, or ' <code>indexed</code> ' for an indexed image

`info = imfinfo(filename)` attempts to infer the format of the file from its content.

# imfinfo

---

## 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](#)  
[imwrite](#)

Read image from graphics file  
Write an image to a graphics file

**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

---

[...] = 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

# imwrite

<b>Purpose</b>	Write an image to a graphics file
<b>Synopsis</b>	<code>imwrite(A, filename, fmt)</code> <code>imwrite(X, map, filename, fmt)</code> <code>imwrite(..., filename)</code> <code>imwrite(..., Parameter, Value, ...)</code>
<b>Description</b>	<code>imwrite(A, filename, fmt)</code> writes the image in A to <code>filename</code> . <code>filename</code> is a string that specifies the name of the output file, and <code>fmt</code> is a string that specifies the format of the file. If A is a grayscale intensity image or a truecolor (RGB) image of class <code>uint8</code> , <code>imwrite</code> writes the actual values in the array to the file. If A is of class <code>double</code> , <code>imwrite</code> rescales the values in the array before writing, using <code>uint8(round(255*A))</code> . 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
' <code>bmp</code> '	Windows Bitmap (BMP)
' <code>hdf</code> '	Hierarchical Data Format (HDF)
' <code>j pg</code> ' or ' <code>j peg</code> '	Joint Photographers Expert Group (JPEG)
' <code>pcx</code> '	Windows Paintbrush (PCX)
' <code>tif</code> ' or ' <code>tiff</code> '	Tagged Image File Format (TIFF)
' <code>xwd</code> '	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
'Compressi on'	One of these strings: 'none', 'rle', 'jpeg'	'rle'
'Qual i ty'	A number between 0 and 100; parameter applies only if 'Compressi on' 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
'Qual i ty'	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
'Compressi on'	One of these strings: 'none', 'packbits', 'ccitt'; 'ccitt' is valid for binary images only	'ccitt' for binary images; 'packbits' for all other images
'Descripti on'	Any string; fills in the ImageDescription field returned by <code>imfinfo</code>	empty
'Resoluti on'	A scalar value that is used for the XResolution and YResolution tags in the output file	72

This table summarizes the types of images that `imwri te` 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 (Note: indexed images are converted to RGB before writing out JPEG files, because the JPEG format does not support indexed images)
PCX	8-bit images
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', 'Compressi on', 'none', ...
'WriteMode', 'append')
```

**See Also**[imfinfo](#)

Return information about a graphics file

[imread](#)

Read image from graphics file

# ind2sub

<b>Purpose</b>	Subscripts from linear index
<b>Syntax</b>	$[I, J] = \text{ind2sub}(size, I\text{ND})$ $[I_1, I_2, I_3, \dots, I_n] = \text{ind2sub}(size, I\text{ND})$
<b>Description</b>	The <code>ind2sub</code> command determines the equivalent subscript values corresponding to a single index into an array.  $[I, J] = \text{ind2sub}(size, I\text{ND})$ returns the arrays $I$ and $J$ containing the equivalent row and column subscripts corresponding to the index matrix $I\text{ND}$ for a matrix of size $size$ .  For matrices, $[I, J] = \text{ind2sub}(\text{size}(A), \text{find}(A>5))$ returns the same values as $[I, J] = \text{find}(A>5)$ .  $[I_1, I_2, I_3, \dots, I_n] = \text{ind2sub}(size, I\text{ND})$ returns $n$ subscript arrays $I_1, I_2, \dots, I_n$ containing the equivalent multidimensional array subscripts equivalent to $I\text{ND}$ for an array of size $size$ .
<b>Examples</b>	The mapping from linear indexes to subscript equivalents for a 2-by-2-by-2 array is:  <p>A 1x4 vector [1, 3, 2, 4] maps to a 2x2 matrix [1, 3; 2, 4]. This matrix then maps to a 2x2x2 cube where element 1 maps to [1, 1, 1], element 2 to [2, 1, 1], element 3 to [1, 2, 1], and element 4 to [2, 2, 1].</p>
<b>See Also</b>	<a href="#">sub2ind</a> <a href="#">find</a>
	<a href="#">Single index from subscripts</a> <a href="#">Find indices and values of nonzero elements</a>

---

<b>Purpose</b>	Infinity
<b>Syntax</b>	<code>Inf</code>
<b>Description</b>	<code>Inf</code> 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.
<b>Examples</b>	<code>1/0</code> , <code>1. e1000</code> , <code>2^1000</code> , and <code>exp(1000)</code> all produce <code>Inf</code> . <code>log(0)</code> produces <code>-Inf</code> . <code>Inf-Inf</code> and <code>Inf/Inf</code> both produce <code>NaN</code> , Not-a-Number.
<b>See Also</b>	<code>is*</code> Detect state <code>NaN</code> Not-a-Number

# inferiorito

---

<b>Purpose</b>	Inferior class relationship
<b>Syntax</b>	<code>inferiorito('class1', 'class2', ...)</code>
<b>Description</b>	The <code>inferiorito</code> function establishes a hierarchy which determines the order in which MATLAB calls object methods.  <code>inferiorito('class1', 'class2', ...)</code> invoked within a class constructor method (say <code>myclass.m</code> ) indicates that <code>myclass</code> 's method should not be invoked if a function is called with an object of class <code>myclass</code> and one or more objects of class <code>class1</code> , <code>class2</code> , and so on.
<b>Remarks</b>	Suppose A is of class ' <code>class_a</code> ', B is of class ' <code>class_b</code> ' and C is of class ' <code>class_c</code> '. Also suppose the constructor <code>class_c.m</code> contains the statement: <code>inferiorito('class_a')</code> . Then <code>e = fun(a, c)</code> or <code>e = fun(c, a)</code> invokes <code>class_a.fun</code> .  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, <code>fun(b, c)</code> calls <code>class_b.fun</code> , while <code>fun(c, b)</code> calls <code>class_c.fun</code> .
<b>See Also</b>	<a href="#">superiorito</a> Superior class relationship

<b>Purpose</b>	Construct an inline object
<b>Syntax</b>	$g = \text{inline}(expr)$ $g = \text{inline}(expr, arg1, arg2, \dots)$ $g = \text{inline}(expr, n)$
<b>Description</b>	<p><code>inline(expr)</code> constructs an inline function object from the MATLAB expression contained in the string <code>expr</code>. The input argument to the inline function is automatically determined by searching <code>expr</code> 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.</p> <p><code>inline(expr, arg1, arg2, \dots)</code> constructs an inline function whose input arguments are specified by the strings <code>arg1, arg2, \dots</code>. Multicharacter symbol names may be used.</p> <p><code>inline(expr, n)</code>, where n is a scalar, constructs an inline function whose input arguments are x, P1, P2, ...</p>
<b>Remarks</b>	<p>Three commands related to <code>inline</code> allow you to examine an inline function object and determine how it was created.</p> <p><code>char(fun)</code> converts the inline function into a character array. This is identical to <code>formula(fun)</code>.</p> <p><code>argnames(fun)</code> returns the names of the input arguments of the inline object <code>fun</code> as a cell array of strings.</p> <p><code>formula(fun)</code> returns the formula for the inline object <code>fun</code>.</p> <p>A fourth command <code>vectorize(fun)</code> inserts a . before any ^, * or /' in the formula for <code>fun</code>. The result is a vectorized version of the inline function.</p>

## inline

---

### Examples

Create a simple inline function to square a number:

```
g = inline('t^2')
g =
    Inline function:
g(t) = t^2
```

```
char(g)
ans =
t^2
```

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

---

<b>Purpose</b>	Functions in memory
<b>Syntax</b>	<code>M = inmem</code> <code>[M, mex] = inmem</code>
<b>Description</b>	<code>M = inmem</code> returns a cell array of strings containing the names of the M-files that are in the P-code buffer.  <code>[M, mex] = inmem</code> returns a cell array containing the names of the MEX-files that have been loaded.
<b>Examples</b>	<pre>clear all % start with a clean slate erf(.5) M = inmem</pre> <p>lists the M-files that were required to run erf.</p>

**Purpose**

Detect points inside a polygonal region

**Syntax**

```
IN = inpol ygon(X, Y, xv, yv)
```

**Description**

`IN = inpol ygon(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:

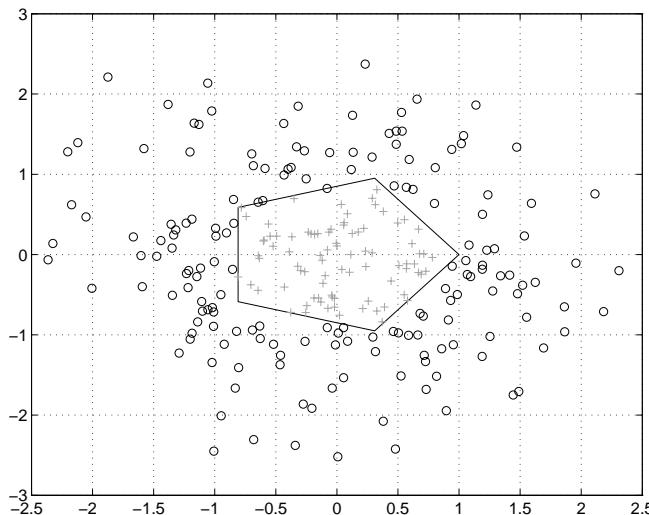
$IN(p, q) = 1$       If  $(X(p, q), Y(p, q))$  is inside the polygonal region

$IN(p, q) = 0.5$       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 = inpol ygon(x, y, xv, yv);
plot(xv, yv, x(in), y(in), 'r+', x(~in), y(~in), 'bo')
```



# input

---

<b>Purpose</b>	Request user input				
<b>Syntax</b>	<pre>user_entry = input('prompt') user_entry = input('prompt', 's')</pre>				
<b>Description</b>	The response to the <code>input</code> prompt can be any MATLAB expression, which is evaluated using the variables in the current workspace.  <code>user_entry = input('prompt')</code> displays <i>prompt</i> as a prompt on the screen, waits for input from the keyboard, and returns the value entered in <code>user_entry</code> .  <code>user_entry = input('prompt', 's')</code> returns the entered string as a text variable rather than as a variable name or numerical value.				
<b>Remarks</b>	If you press the <b>Return</b> key without entering anything, <code>input</code> returns an empty matrix.  The text string for the prompt may contain one or more ' <code>\n</code> ' characters. The ' <code>\n</code> ' means to skip to the next line. This allows the prompt string to span several lines. To display just a backslash, use ' <code>\\"</code> '.				
<b>Examples</b>	Press <b>Return</b> to select a default value by detecting an empty matrix:  <pre>i = input('Do you want more? Y/N [Y]: ', 's'); if isempty(i)     i = 'Y'; end</pre>				
<b>See Also</b>	The <code>ginput</code> and <code>uicontrol</code> commands in the <i>MATLAB Graphics Guide</i> , and:  <table><tr><td><code>keyboard</code></td><td>Invoke the keyboard in an M-file</td></tr><tr><td><code>menu</code></td><td>Generate a menu of choices for user input</td></tr></table>	<code>keyboard</code>	Invoke the keyboard in an M-file	<code>menu</code>	Generate a menu of choices for user input
<code>keyboard</code>	Invoke the keyboard in an M-file				
<code>menu</code>	Generate a menu of choices for user input				

---

<b>Purpose</b>	Input argument name
<b>Syntax</b>	<code>i nputname( <i>argnum</i>)</code>
<b>Description</b>	This command can be used only inside the body of a function.  <code>i nputname( <i>argnum</i>)</code> returns the workspace variable name corresponding to the argument number <i>argnum</i> . If the input argument has no name (for example, if it is an expression instead of a variable), the <code>i nputname</code> command returns the empty string ('').
<b>Examples</b>	Suppose the function <code>myfun.m</code> is defined as:  <code>function c = myfun(a, b) disp(sprintf('First calling variable is "%s". ', inputname(1))</code>  Then  <code>x = 5; y = 3; myfun(x, y)</code> produces  <code>First calling variable is "x".</code>  But  <code>myfun(pi+1, pi-1)</code> produces  <code>First calling variable is "".</code>
<b>See Also</b>	<code>nargin</code> , <code>nargout</code> Number of function arguments <code>nargchk</code> Check number of input arguments

## **int2str**

---

<b>Purpose</b>	Integer to string conversion									
<b>Syntax</b>	<code>str = int2str(N)</code>									
<b>Description</b>	<code>str = int2str(N)</code> converts an integer to a string with integer format. The input <code>N</code> can be a single integer or a vector or matrix of integers. Noninteger inputs are rounded before conversion.									
<b>Examples</b>	<code>int2str(2+3)</code> is the string ' <code>5</code> '. One way to label a plot is <pre>title(['case number ' int2str(n)])</pre> For matrix or vector inputs, <code>int2str</code> returns a string matrix: <pre>int2str(eye(3))</pre> <code>ans =</code>  <table><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td></tr></table>	1	0	0	0	1	0	0	0	1
1	0	0								
0	1	0								
0	0	1								
<b>See Also</b>	<a href="#">fprintf</a> Write formatted data to file <a href="#">num2str</a> Number to string conversion <a href="#">sprintf</a> 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**

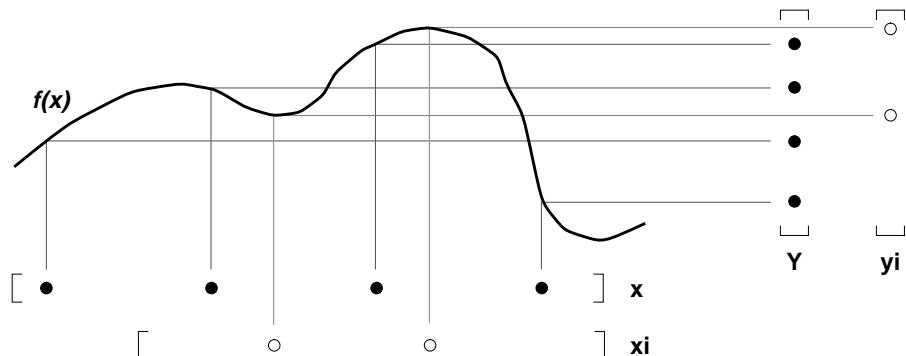
`yi = interp1(x, Y, xi)` returns vector `yi` containing elements corresponding to the elements of `xi` and determined by interpolation within vectors `x` and `Y`. The vector `x` specifies the points at which the data `Y` is given. If `Y` is a matrix, then the interpolation is performed for each column of `Y` and `yi` will be `length(xi)`-by-`size(Y, 2)`. Out of range values are returned as NaNs.

`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`,

# interp1

and, based upon their locations, returns values  $y_i$  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.

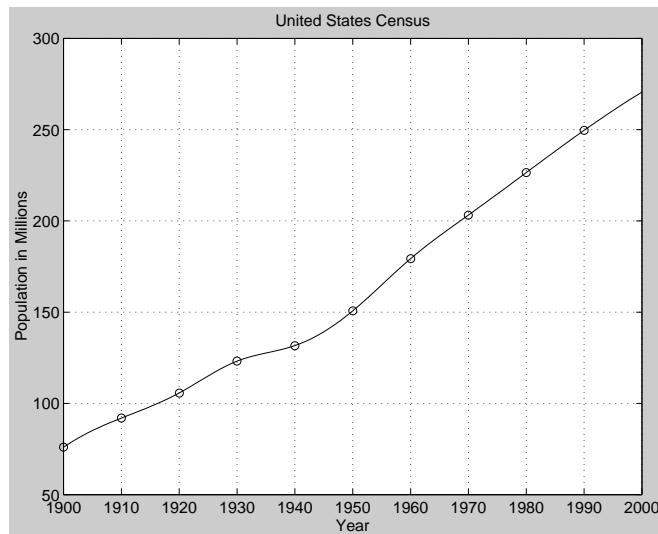
```
t = 1900: 10: 1990;
p = [ 75. 995  91. 972  105. 711  123. 203  131. 669...
      150. 697  179. 323  203. 212  226. 505  249. 633];
```

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

<code>interpft</code>	One-dimensional interpolation using the FFT method.
<code>interp2</code>	Two-dimensional data interpolation (table lookup)
<code>interp3</code>	Three-dimensional data interpolation (table lookup)
<code>interpn</code>	Multidimensional data interpolation (table lookup)
<code>spline</code>	Cubic spline interpolation

## References

[1] de Boor, C. *A Practical Guide to Splines*, Springer-Verlag, 1978.

# interp2

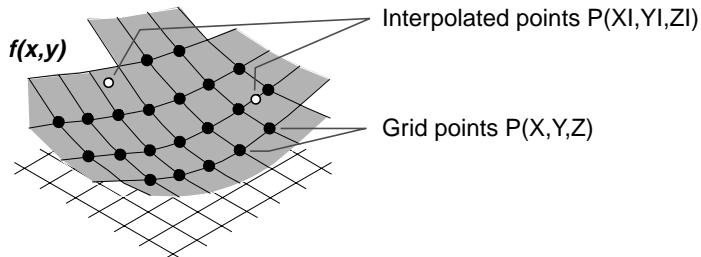
---

<b>Purpose</b>	Two-dimensional data interpolation (table lookup)
<b>Syntax</b>	<pre>ZI = interp2(X, Y, Z, XI, YI) ZI = interp2(Z, XI, YI) ZI = interp2(Z, ntimes) ZI = interp2(X, Y, Z, XI, YI, method)</pre>
<b>Description</b>	<p><code>ZI = interp2(X, Y, Z, XI, YI)</code> returns matrix <code>ZI</code> containing elements corresponding to the elements of <code>XI</code> and <code>YI</code> and determined by interpolation within the two-dimensional function specified by matrices <code>X</code>, <code>Y</code>, and <code>Z</code>. <code>X</code> and <code>Y</code> must be monotonic, and have the same format ("plaid") as if they were produced by <code>meshgrid</code>. Matrices <code>X</code> and <code>Y</code> specify the points at which the data <code>Z</code> is given. Out of range values are returned as NaNs.</p> <p><code>XI</code> and <code>YI</code> can be matrices, in which case <code>interp2</code> returns the values of <code>Z</code> corresponding to the points <math>(XI(i, j), YI(i, j))</math>. Alternatively, you can pass in the row and column vectors <code>xi</code> and <code>yi</code>, respectively. In this case, <code>interp2</code> interprets these vectors as if you issued the command <code>meshgrid(xi, yi)</code>.</p> <p><code>ZI = interp2(Z, XI, YI)</code> assumes that <code>X = 1:n</code> and <code>Y = 1:m</code>, where <code>[m, n] = size(Z)</code>.</p> <p><code>ZI = interp2(Z, ntimes)</code> expands <code>Z</code> by interleaving interpolates between every element, working recursively for <code>ntimes</code>. <code>interp2(Z)</code> is the same as <code>interp2(Z, 1)</code>.</p> <p><code>ZI = interp2(X, Y, Z, XI, YI, method)</code> specifies an alternative interpolation method:</p> <ul style="list-style-type: none"><li>• 'linear' for bilinear interpolation (default)</li><li>• 'nearest' for nearest neighbor interpolation</li><li>• 'spline' for cubic spline interpolation</li><li>• 'cubic' for bicubic interpolation</li></ul> <p>All interpolation methods require that <code>X</code> and <code>Y</code> be monotonic, and have the same format ("plaid") as if they were produced by <code>meshgrid</code>. Variable spacing is handled by mapping the given values in <code>X</code>, <code>Y</code>, <code>XI</code>, and <code>YI</code> to an equally spaced domain before interpolating. For faster interpolation when <code>X</code> and <code>Y</code> are equally</p>

spaced and monotonic, use the methods '*\*linear*', '*\*cubic*', '*\*spline*', or '*\*nearest*'.

## Remarks

The `interp2` command interpolates between data points. It finds values of a two-dimensional function  $f(x,y)$  underlying the data at intermediate points.



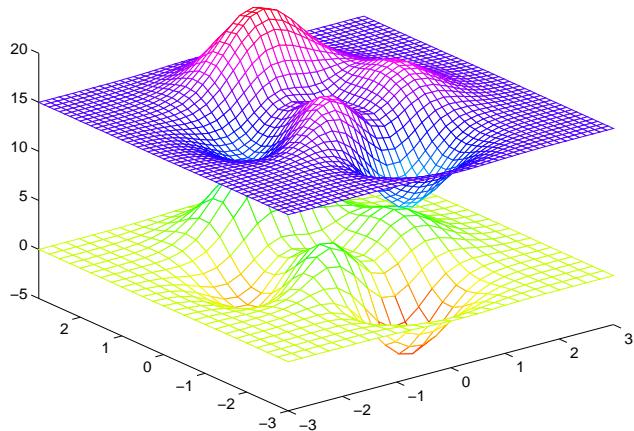
Interpolation is the same operation as table lookup. Described in table lookup terms, the table is `tab = [NaN, Y; X, Z]` and `interp2` looks up the elements of `XI` in `X`, `YI` in `Y`, and, based upon their location, returns values `ZI` interpolated within the elements of `Z`.

## interp2

### Examples

Interpolate the peaks function over a finer grid:

```
[X, Y] = meshgrid(-3: .25: 3);  
Z = peaks(X, Y);  
[XI, YI] = meshgrid(-3: .125: 3);  
ZI = interp2(X, Y, Z, XI, YI);  
mesh(X, Y, Z), hold, mesh(XI, YI, ZI+15)  
hold off  
axis([-3 3 -3 3 -5 20])
```



Given this set of employee data,

```
years = 1950: 10: 1990;  
service = 10: 10: 30;  
wage = [ 150. 697 199. 592 187. 625  
        179. 323 195. 072 250. 287  
        203. 212 179. 092 322. 767  
        226. 505 153. 706 426. 730  
        249. 633 120. 281 598. 243];
```

it is possible to interpolate to find the wage earned in 1975 by an employee with 15 years' service:

```
w = interp2(service, years, wage, 15, 1975)  
w =  
    190. 6287
```

**See Also**

`griddata`  
`interp1`  
`interp3`  
`interpn`  
`meshgrid`

**Data gridding**

One-dimensional data interpolation (table lookup)  
Three-dimensional data interpolation (table lookup)  
Multidimensional data interpolation (table lookup)  
Generation of X and Y arrays for three-dimensional plots.

# interp3

---

<b>Purpose</b>	Three-dimensional data interpolation (table lookup)
<b>Syntax</b>	<pre>VI = interp3(X, Y, Z, V, XI, YI, ZI) VI = interp3(V, XI, YI, ZI) VI = interp3(V, ntimes) VI = interp3(..., method)</pre>
<b>Description</b>	<p><code>VI = interp3(X, Y, Z, V, XI, YI, ZI)</code> interpolates to find <code>VI</code>, the values of the underlying three-dimensional function <code>V</code> at the points in matrices <code>XI</code>, <code>YI</code> and <code>ZI</code>. Matrices <code>X</code>, <code>Y</code> and <code>Z</code> specify the points at which the data <code>V</code> is given. Out of range values are returned as <code>NaN</code>.</p> <p><code>XI</code>, <code>YI</code>, and <code>ZI</code> can be matrices, in which case <code>interp3</code> returns the values of <code>Z</code> corresponding to the points <math>(XI(i, j), YI(i, j), ZI(i, j))</math>. Alternatively, you can pass in the vectors <code>xi</code>, <code>yi</code>, and <code>zi</code>. Vector arguments that are not the same size are interpreted as if you called <code>meshgrid</code>.</p> <p><code>VI = interp3(V, XI, YI, ZI)</code> assumes <math>X=1:N</math>, <math>Y=1:M</math>, <math>Z=1:P</math> where <math>[M, N, P] = \text{size}(V)</math>.</p> <p><code>VI = interp3(V, ntimes)</code> expands <code>V</code> by interleaving interpolates between every element, working recursively for <code>ntimes</code> iterations. The command <code>interp3(V, 1)</code> is the same as <code>interp3(V)</code>.</p> <p><code>VI = interp3(..., method)</code> specifies alternative methods:</p> <ul style="list-style-type: none"><li>• 'linear' for linear interpolation (default)</li><li>• 'cubic' for cubic interpolation</li><li>• 'spline' for cubic spline interpolation</li><li>• 'nearest' for nearest neighbor interpolation</li></ul>
<b>Discussion</b>	All the interpolation methods require that <code>X</code> , <code>Y</code> and <code>Z</code> be monotonic and have the same format ("plaid") as if they were produced by <code>meshgrid</code> . Variable spacing is handled by mapping the given values in <code>X</code> , <code>Y</code> , <code>Z</code> , <code>XI</code> , <code>YI</code> and <code>ZI</code> to an equally spaced domain before interpolating. For faster interpolation when <code>X</code> , <code>Y</code> , and <code>Z</code> are equally spaced and monotonic, use the methods '*linear', '*cubic', '*spline', or '*nearest'.

**Examples**

To generate a coarse approximation of flow and interpolate over a finer mesh:

```
[x, y, z, v] = flow(10);  
[xi, yi, zi] = meshgrid(.1: .25: 10, -3: .25: 3, -3: .25: 3);  
vi = interp3(x, y, z, v, xi, yi, zi); % V is 31-by-41-by-27  
slice(xi, yi, zi, vi, [6 9.5], 2, [-2 .2]) shading flat
```

**See Also**

<code>interp1</code>	One-dimensional data interpolation (table lookup)
<code>interp2</code>	Two-dimensional data interpolation (table lookup)
<code>interpn</code>	Multidimensional data interpolation (table lookup).
<code>meshgrid</code>	Generate X and Y matrices for three-dimensional plots

# interpft

---

<b>Purpose</b>	One-dimensional interpolation using the FFT method
<b>Syntax</b>	$y = \text{interpft}(x, n)$ $y = \text{interpft}(x, n, dim)$
<b>Description</b>	$y = \text{interpft}(x, n)$ returns the vector $y$ that contains the value of the periodic function $x$ resampled to $n$ equally spaced points. If $\text{length}(x) = m$ , 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, $\text{interpft}$ operates on the columns of $X$ , returning a matrix $Y$ with the same number of columns as $X$ , but with $n$ rows. $y = \text{interpft}(x, n, dim)$ operates along the specified dimension.
<b>Algorithm</b>	The $\text{interpft}$ command uses the FFT method. The original vector $x$ is transformed to the Fourier domain using $\text{fft}$ and then transformed back with more points.
<b>See Also</b>	<a href="#">interp1</a> One-dimensional data interpolation (table lookup)

<b>Purpose</b>	Multidimensional data interpolation (table lookup)
<b>Syntax</b>	<pre>VI = interpn(X1, X2, X3, ..., V, Y1, Y2, Y3, ...) VI = interpn(V, Y1, Y2, Y3, ...) VI = interpn(V, ntimes) VI = interpn(..., method)</pre>
<b>Description</b>	<p><code>VI = interpn(X1, X2, X3, ..., V, Y1, Y2, Y3, ...)</code> interpolates to find <code>VI</code>, the values of the underlying multidimensional function <code>V</code> at the points in the arrays <code>Y1</code>, <code>Y2</code>, <code>Y3</code>, etc. For a multidimensional <code>V</code>, you should call <code>interpn</code> with <math>2*N+1</math> arguments, where <code>N</code> is the number of dimensions in <code>V</code>. Arrays <code>X1,X2,X3,...</code> specify the points at which the data <code>V</code> is given. Out of range values are returned as <code>NaN</code>.</p> <p><code>Y1, Y2, Y3,...</code> can be matrices, in which case <code>interpn</code> returns the values of <code>VI</code> corresponding to the points <math>(Y1(i,j), Y2(i,j), Y3(i,j), \dots)</math>. Alternatively, you can pass in the vectors <code>y1, y2, y3,...</code> In this case, <code>interpn</code> interprets these vectors as if you issued the command <code>ndgrid(y1, y2, y3, ...)</code>.</p> <p><code>VI = interpn(V, Y1, Y2, Y3, ...)</code> interpolates as above, assuming <code>X1 = 1:size(V, 1), X2 = 1:size(V, 2), X3 = 1:size(V, 3)</code>, and so on.</p> <p><code>VI = interpn(V, ntimes)</code> expands <code>V</code> by interleaving interpolates between each element, working recursively for <code>ntimes</code> iterations. <code>interpn(V, 1)</code> is the same as <code>interpn(V)</code>.</p> <p><code>VI = interpn(..., method)</code> specifies alternative methods:</p> <ul style="list-style-type: none"> <li>• '<code>linear</code>' for linear interpolation (default)</li> <li>• '<code>cubic</code>' for cubic interpolation</li> <li>• '<code>spline</code>' for cubic spline interpolation</li> <li>• '<code>nearest</code>' for nearest neighbor interpolation</li> </ul>
<b>Discussion</b>	<p>All the interpolation methods require that <code>X,Y</code> and <code>Z</code> be monotonic and have the same format ("plaid") as if they were produced by <code>ndgrid</code>. Variable spacing is handled by mapping the given values in <code>X1,X2,X3,...</code> and <code>Y1,Y2,Y3,...</code> to an equally spaced domain before interpolating. For faster interpolation when <code>X1,X2,Y3, and so on</code> are equally spaced and monotonic, use the methods '<code>*linear</code>', '<code>*cubic</code>', '<code>*spline</code>', or '<code>*nearest</code>'.</p>

# interp

---

## See Also

`interp1`

`interp2`

`ndgrid`

One-dimensional data interpolation (table lookup)

Two-dimensional data interpolation (table lookup)

Generate arrays for multidimensional functions and  
interpolation

---

<b>Purpose</b>	Set intersection of two vectors										
<b>Syntax</b>	<pre>c = intersect(a, b) c = intersect(A, B, 'rows') [c, ia, ib] = intersect(...)</pre>										
<b>Description</b>	<p><code>c = intersect(a, b)</code> returns the values common to both <code>a</code> and <code>b</code>. The resulting vector is sorted in ascending order. In set theoretic terms, this is <math>a \cap b</math>. <code>a</code> and <code>b</code> can be cell arrays of strings.</p> <p><code>c = intersect(A, B, 'rows')</code> when <code>A</code> and <code>B</code> are matrices with the same number of columns returns the rows common to both <code>A</code> and <code>B</code>.</p> <p><code>[c, ia, ib] = intersect(a, b)</code> also returns column index vectors <code>ia</code> and <code>ib</code> such that <code>c = a(ia)</code> and <code>c = b(ib)</code> (or <code>c = a(ia, :)</code> and <code>c = b(ib, :)</code>).</p>										
<b>Examples</b>	<pre>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</pre>										
<b>See Also</b>	<table border="0"> <tr> <td><code>ismember</code></td> <td>True for a set member</td> </tr> <tr> <td><code>setdiff</code></td> <td>Return the set difference of two vectors</td> </tr> <tr> <td><code>setxor</code></td> <td>Set exclusive-or of two vectors</td> </tr> <tr> <td><code>union</code></td> <td>Set union of two vectors</td> </tr> <tr> <td><code>unique</code></td> <td>Unique elements of a vector</td> </tr> </table>	<code>ismember</code>	True for a set member	<code>setdiff</code>	Return the set difference of two vectors	<code>setxor</code>	Set exclusive-or of two vectors	<code>union</code>	Set union of two vectors	<code>unique</code>	Unique elements of a vector
<code>ismember</code>	True for a set member										
<code>setdiff</code>	Return the set difference of two vectors										
<code>setxor</code>	Set exclusive-or of two vectors										
<code>union</code>	Set union of two vectors										
<code>unique</code>	Unique elements of a vector										

# inv

---

<b>Purpose</b>	Matrix inverse
<b>Syntax</b>	$Y = \text{inv}(X)$
<b>Description</b>	$Y = \text{inv}(X)$ 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 $\text{inv}$ arises when solving the system of linear equations $Ax = b$ . One way to solve this is with $x = \text{inv}(A) * b$ . A better way, from both an execution time and numerical accuracy standpoint, is to use the matrix division operator $x = A \backslash b$ . This produces the solution using Gaussian elimination, without forming the inverse. See $\backslash$ and $/$ for further information.
<b>Examples</b>	Here is an example demonstrating the difference between solving a linear system by inverting the matrix with $\text{inv}(A) * b$ and solving it directly with $A \backslash b$ . A matrix $A$ of order 100 has been constructed so that its condition number, $\text{cond}(A)$ , is $1. \text{e}10$ , and its norm, $\text{norm}(A)$ , 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
	<pre>tic, y = inv(A) * b, toc err = norm(y-x) res = norm(A*y-b)</pre> produce  <pre>elapseds_time = 9.6600 err = 2.4321e-07 res = 1.8500e-09</pre> while the statements
	<pre>tic, z = A \ b, toc err = norm(z-x) res = norm(A*z-b)</pre>

---

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  $y = \text{inv}(A) * b$  as with  $z = A \backslash b$ . 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  $A \backslash b$  instead of  $\text{inv}(A) * b$  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**

\  
/  
det  
lu  
rref

Matrix left division (backslash)  
Matrix right division (slash)  
Matrix determinant  
LU matrix factorization  
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.

<b>Purpose</b>	Inverse of the Hilbert matrix
<b>Syntax</b>	<code>H = invhilb(n)</code>
<b>Description</b>	<code>H = invhilb(n)</code> generates the exact inverse of the exact Hilbert matrix for $n$ less than about 15. For larger $n$ , <code>invhilb(n)</code> generates an approximation to the inverse Hilbert matrix.
<b>Limitations</b>	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 <code>invhilb(n)</code> with <code>inv(hilb(n))</code> involves the effects of two or three sets of roundoff errors:
	<ul style="list-style-type: none"> <li>• The errors caused by representing <code>hilb(n)</code></li> <li>• The errors in the matrix inversion process</li> <li>• The errors, if any, in representing <code>invhilb(n)</code></li> </ul> <p>It turns out that the first of these, which involves representing fractions like <math>1/3</math> and <math>1/5</math> in floating-point, is the most significant.</p>
<b>Examples</b>	<code>invhilb(4)</code> is
	$\begin{matrix} 16 & -120 & 240 & -140 \\ -120 & 1200 & -2700 & 1680 \\ 240 & -2700 & 6480 & -4200 \\ -140 & 1680 & -4200 & 2800 \end{matrix}$
<b>See Also</b>	<code>hilb</code> Hilbert matrix
<b>References</b>	[1] Forsythe, G. E. and C. B. Moler, <i>Computer Solution of Linear Algebraic Systems</i> , Prentice-Hall, 1967, Chapter 19.

# ipermute

---

<b>Purpose</b>	Inverse permute the dimensions of a multidimensional array
<b>Syntax</b>	$A = \text{i permute}(B, order)$
<b>Description</b>	$A = \text{i permute}(B, order)$ is the inverse of <code>permute</code> . <code>i permute</code> rearranges the dimensions of $B$ so that <code>permute(A, order)</code> 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.
<b>Remarks</b>	<code>permute</code> and <code>i permute</code> are a generalization of transpose ( <code>.</code> ) for multidimensional arrays.
<b>Examples</b>	Consider the 2-by-2-by-3 array $a$ :
	$a = \text{cat}(3, \text{eye}(2), 2 * \text{eye}(2), 3 * \text{eye}(2))$
	$\begin{array}{cc} a(:,:,1) = & a(:,:,2) = \\ 1 & 0 \\ 0 & 1 \end{array}$
	$\begin{array}{cc} a(:,:,3) = & \\ 3 & 0 \\ 0 & 3 \end{array}$
	Permuting and inverse permuting $a$ in the same fashion restores the array to its original form:
	<pre>B = permute(a, [3 2 1]); C = i permute(B, [3 2 1]); isequal(a, C) ans=</pre>
	1
<b>See Also</b>	<code>permute</code> Rearrange the dimensions of a multidimensional array

<b>Purpose</b>	Detect state	
<b>Syntax</b>	<code>k = iscell(C)</code> <code>k = iscellstr(S)</code> <code>k = ischar(S)</code> <code>k = isempty(A)</code> <code>k = isequal(A, B, ...)</code> <code>k = isfield(S, 'field')</code> <code>TF = isnfinite(A)</code> <code>k = isglobal(NAME)</code> <code>TF = ishandle(H)</code> <code>k = ishold</code> <code>k = isieee</code> <code>TF = isnan(A)</code> <code>TF = isletter('str')</code>	<code>k = islogical(A)</code> <code>TF = isnan(A)</code> <code>k = isnumeric(A)</code> <code>k = isobject(A)</code> <code>k = isppc</code> <code>TF = isprime(A)</code> <code>k = isreal(A)</code> <code>TF = isspace('str')</code> <code>k = issparse(S)</code> <code>k = issstruct(S)</code> <code>k = issstudent</code> <code>k = isunix</code> <code>k = isvms</code>
<b>Description</b>	<p><code>k = iscell(C)</code> returns logical true (1) if C is a cell array and logical false (0) otherwise.</p> <p><code>k = iscellstr(S)</code> 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.</p> <p><code>k = ischar(S)</code> returns logical true (1) if S is a character array and logical false (0) otherwise.</p> <p><code>k = isempty(A)</code> 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.</p> <p><code>k = isequal(A, B, ...)</code> returns logical true (1) if the input arrays are the same type and size and hold the same contents, and logical false (0) otherwise.</p> <p><code>k = isfield(S, 'field')</code> returns logical true (1) if <i>field</i> is the name of a field in the structure array S.</p> <p><code>TF = isnan(A)</code> 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.</p>	

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 NaNs and logical false (0) where they are not.

`k = isnumeric(A)` returns logical true (1) if A is a numeric array and logical false (0) otherwise. For example, sparse arrays, and double precision arrays are numeric while strings, cell arrays, and structure arrays are not.

`k = isobject(A)` returns logical true (1) if A is an object and logical false (0) otherwise.

`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 = issstruct(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

<code>isfinite(1./a) =</code>	<code>[1 1 0 1 1]</code>
<code>isinf(1./a) =</code>	<code>[0 0 1 0 0]</code>
<code>isnan(1./a) =</code>	<code>[0 0 0 0 0]</code>

and

<code>isfinite(0./a) =</code>	<code>[1 1 0 1 1]</code>
<code>isinf(0./a) =</code>	<code>[0 0 0 0 0]</code>
<code>isnan(0./a) =</code>	<code>[0 0 1 0 0]</code>

---

<b>Purpose</b>	Detect an object of a given class
<b>Syntax</b>	<code>K = isa(obj, 'class_name')</code>
<b>Description</b>	<code>K = isa(obj, 'class_name')</code> returns logical true (1) if <code>obj</code> is of class (or a subclass of) <code>class_name</code> , and logical false (0) otherwise.
	The argument <code>class_name</code> is the name of a user-defined or pre-defined class of objects. Predefined MATLAB classes include:
<code>cell</code>	Multidimensional cell array
<code>double</code>	Multidimensional double precision array
<code>sparse</code>	Two-dimensional real (or complex) sparse array
<code>char</code>	Array of alphanumeric characters
<code>struct</code>	Structure
<code>'class_name'</code>	User-defined object class
<b>Examples</b>	<code>isa(rand(3, 4), 'double')</code> returns 1.
<b>See Also</b>	<code>class</code> Create object or return class of object

# ismember

---

<b>Purpose</b>	Detect members of a set										
<b>Syntax</b>	<pre>k = ismember(a, S) k = ismember(A, S, 'rows')</pre>										
<b>Description</b>	<p><code>k = ismember(a, S)</code> returns a vector the same length as <code>a</code> containing logical true (1) where the elements of <code>a</code> are in the set <code>S</code>, and logical false (0) elsewhere. In set theoretic terms, <code>k</code> is 1 where <math>a \in S</math>. <code>a</code> and <code>S</code> can be cell arrays of strings.</p> <p><code>k = ismember(A, S, 'rows')</code> when <code>A</code> and <code>S</code> are matrices with the same number of columns returns a vector containing 1 where the rows of <code>A</code> are also rows of <code>S</code> and 0 otherwise.</p>										
<b>Examples</b>	<pre>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</pre>										
<b>See Also</b>	<table><tr><td><code>intersect</code></td><td>Set intersection of two vectors</td></tr><tr><td><code>setdiff</code></td><td>Return the set difference of two vectors</td></tr><tr><td><code>setxor</code></td><td>Set exclusive-or of two vectors</td></tr><tr><td><code>union</code></td><td>Set union of two vectors</td></tr><tr><td><code>unique</code></td><td>Unique elements of a vector</td></tr></table>	<code>intersect</code>	Set intersection of two vectors	<code>setdiff</code>	Return the set difference of two vectors	<code>setxor</code>	Set exclusive-or of two vectors	<code>union</code>	Set union of two vectors	<code>unique</code>	Unique elements of a vector
<code>intersect</code>	Set intersection of two vectors										
<code>setdiff</code>	Return the set difference of two vectors										
<code>setxor</code>	Set exclusive-or of two vectors										
<code>union</code>	Set union of two vectors										
<code>unique</code>	Unique elements of a vector										

---

<b>Purpose</b>	Detect strings
<b>Description</b>	This MATLAB 4 function has been renamed <code>ischar</code> in MATLAB 5.
<b>See Also</b>	<code>is*</code> Detect state

**isstr**

---



**isstr**

---



# j

---

<b>Purpose</b>	Imaginary unit								
<b>Syntax</b>	$j$ $x+yj$ $x+j *y$								
<b>Description</b>	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.								
<b>Examples</b>	$Z = 2+3j$ $Z = x+j *y$ $Z = r*exp(j *theta)$								
<b>See Also</b>	<table><tr><td><code>conj</code></td><td>Complex conjugate</td></tr><tr><td><code>i</code></td><td>Imaginary unit</td></tr><tr><td><code>i mag</code></td><td>Imaginary part of a complex number</td></tr><tr><td><code>real</code></td><td>Real part of complex number</td></tr></table>	<code>conj</code>	Complex conjugate	<code>i</code>	Imaginary unit	<code>i mag</code>	Imaginary part of a complex number	<code>real</code>	Real part of complex number
<code>conj</code>	Complex conjugate								
<code>i</code>	Imaginary unit								
<code>i mag</code>	Imaginary part of a complex number								
<code>real</code>	Real part of complex number								

**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**

<code>dbstop</code>	Set breakpoints in an M-file function
<code>i nput</code>	Request user input
<code>qui t</code>	Terminate MATLAB
<code>return</code>	Terminate keyboard mode

# kron

---

**Purpose** Kronecker tensor product

**Syntax**  $K = \text{kron}(X, Y)$

**Description**  $K = \text{kron}(X, Y)$  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  $\text{kron}(X, Y)$  is  $m*p$ -by- $n*q$ .

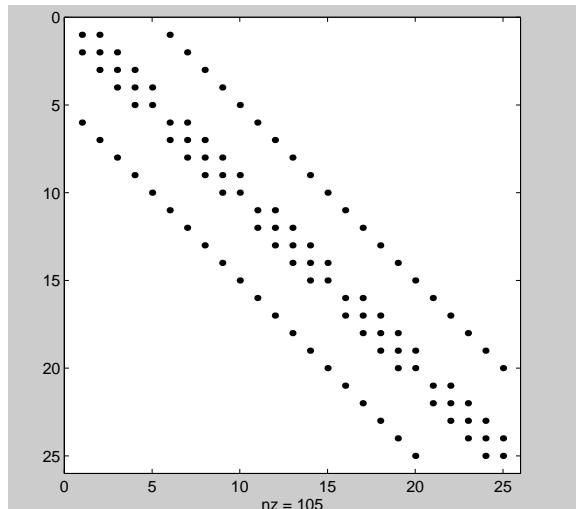
**Examples** If  $X$  is 2-by-3, then  $\text{kron}(X, Y)$  is

$$[ X(1, 1)*Y \quad X(1, 2)*Y \quad X(1, 3)*Y \\ X(2, 1)*Y \quad X(2, 2)*Y \quad X(2, 3)*Y ]$$

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:



<b>Purpose</b>	Last error message
<b>Syntax</b>	<code>str = lasterr</code> <code>lasterr('')</code>
<b>Description</b>	<code>str = lasterr</code> returns the last error message generated by MATLAB.  <code>lasterr('')</code> resets <code>lasterr</code> so it returns an empty matrix until the next error occurs.
<b>Examples</b>	Here is a function that examines the <code>lasterr</code> string and displays its own message based on the error that last occurred. This example deals with two cases, each of which is an error that can result from a matrix multiply.

```
function catchfcn
l = lasterr;
j = findstr(l, 'Inner matrix dimensions');
if 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 ('string', 'catchstr')
```

or the `try ... catch... end` statements. The `catch` action examines the `lasterr` string to determine the cause of the error and takes appropriate action.

The eval function evaluates *string* and returns if no error occurs. If an error occurs, eval executes *catchstr*. Using eval with the catchfcn function above:

```
clear  
A = [1 2 3; 6 7 2; 0 -1 5];  
B = [9 5 6; 0 4 9];  
eval ('A*B', 'catch')
```

MATLAB responds with Wrong dimensions for matrix multiply.

## See Also

<a href="#">error</a>	Display error messages
<a href="#">eval</a>	Interpret strings containing MATLAB expressions

---

<b>Purpose</b>	Last warning message
<b>Syntax</b>	<code>lastwarn</code> <code>lastwarn('')</code> <code>lastwarn('string')</code>
<b>Description</b>	<code>lastwarn</code> returns a string containing the last warning message issued by MATLAB.  <code>lastwarn('')</code> resets the <code>lastwarn</code> function so that it will return an empty string matrix until the next warning is encountered.  <code>lastwarn('string')</code> sets the last warning message to ' <code>string</code> '. The last warning message is updated regardless of whether warning is on or off.
<b>See Also</b>	<code>lasterr</code> , <code>warning</code>

# **Icm**

---

<b>Purpose</b>	Least common multiple
<b>Syntax</b>	$L = \text{lcm}(A, B)$
<b>Description</b>	$L = \text{lcm}(A, B)$ 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).
<b>Examples</b>	<pre>lcm(8, 40) ans =     40</pre> <pre>lcm(pascal(3), magic(3))  ans =     8     1     6     3    10    21     4     9     6</pre>
<b>See Also</b>	<a href="#">gcd</a> 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  $n$ :

$$P_n(x) = \frac{1}{2^n n!} \left[ \frac{d^n}{dx^n} (x^2 - 1)^n \right]$$

The Schmidt seminormalized associated Legendre functions are related to the nonnormalized associated Legendre functions  $P_n^m(x)$  by:

$$S_n^m(x) = \sqrt{\frac{2(n-m)!}{(n+m)!}} P_n^m(x)$$

**Description**  $P = \text{legendre}(n, X)$  computes the associated Legendre functions of degree  $n$  and order  $m = 0, 1, \dots, n$ , evaluated at  $X$ . Argument  $n$  must be a scalar integer less than 256, and  $X$  must contain real values in the domain  $-1 \leq x \leq 1$ .

The returned array  $P$  has one more dimension than  $X$ , and each element  $P(m+1, d1, d2, \dots)$  contains the associated Legendre function of degree  $n$  and order  $m$  evaluated at  $X(d1, d2, \dots)$ .

If  $X$  is a vector, then  $P$  is a matrix of the form:

$$\begin{array}{cccc} P_2^0(x(1)) & P_2^0(x(2)) & P_2^0(x(3)) & \dots \\ P_2^1(x(1)) & P_2^1(x(2)) & P_2^1(x(3)) & \dots \\ P_2^2(x(1)) & P_2^2(x(2)) & P_2^2(x(3)) & \dots \end{array}$$

## legendre

---

`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:

	<b>x = 0</b>	<b>x = 0.1</b>	<b>x = 0.2</b>
<b>m = 0</b>	0. 5000	0. 4850	0. 4400
<b>m = 1</b>	0	0. 2985	0. 5879
<b>m = 2</b>	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))`.

---

<b>Purpose</b>	Length of vector
<b>Syntax</b>	<code>n = length(X)</code>
<b>Description</b>	The statement <code>length(X)</code> is equivalent to <code>max(size(X))</code> for nonempty arrays and 0 for empty arrays.
	<code>n = length(X)</code> returns the size of the longest dimension of <code>X</code> . If <code>X</code> is a vector, this is the same as its length.
<b>Examples</b>	<pre>x = ones(1, 8); n = length(x) n =     8</pre> <pre>x = rand(2, 10, 3); n = length(x) n =     10</pre>
<b>See Also</b>	<a href="#">ndims</a> Number of array dimensions <a href="#">size</a> Array dimensions

## lin2mu

---

<b>Purpose</b>	Linear to mu-law conversion
<b>Syntax</b>	<code>mu = lin2mu(y)</code>
<b>Description</b>	<code>mu = lin2mu(y)</code> converts linear audio signal amplitudes in the range $-1 \leq Y \leq 1$ to mu-law encoded “flints” in the range $0 \leq mu \leq 255$ .
<b>See Also</b>	<code>auwrite</code> Write NeXT/SUN (.au) sound file <code>mu2lin</code> Mu-law to linear 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**

<code>:</code> (Colon)	Create vectors, matrix subscripting, and for iterations
<code>logspace</code>	Generate logarithmically spaced vectors

# load

---

<b>Purpose</b>	Retrieve variables from disk
<b>Syntax</b>	<code>load</code> <code>load <i>filename</i></code> <code>load (<i>filename</i>)</code> <code>load <i>filename</i>.ext</code> <code>load <i>filename</i> -ascii</code> <code>load <i>filename</i> -mat</code> <code>S = load(...)</code>
<b>Description</b>	<p>The <code>load</code> and <code>save</code> commands retrieve and store MATLAB variables on disk.</p> <p><code>load</code> by itself, loads all the variables saved in the file '<code>matlab.mat</code>' .</p> <p><code>load <i>filename</i></code> retrieves the variables from '<code>filename.mat</code>' given a full pathname or a MATLABPATH relative partial pathname .</p> <p><code>load (<i>filename</i>)</code> loads a file whose name is stored in <code>filename</code>. The statements:</p> <pre>str = 'filename.mat'; load (str)</pre> <p>retrieve the variables from the binary file '<code>filename.mat</code>' .</p> <p><code>load <i>filename</i>.ext</code> reads ASCII files that contain rows of space separated values. The resulting data is placed into a variable with the same name as the file (without the extension). ASCII files may contain MATLAB comments (lines that begin with %).</p> <p><code>load <i>filename</i> -ascii</code> or <code>load <i>filename</i> -mat</code> can be used to force <code>load</code> to treat the file as either an ASCII file or a MAT file.</p> <p><code>S = load(...)</code> returns the contents of a MAT-file as a structure instead of directly loading the file into the workspace. The field names in <code>S</code> match the names of the variables that were retrieved. When the file is ASCII, <code>S</code> is a double-precision array.</p>
<b>Remarks</b>	MAT-files are double-precision binary MATLAB format files created by the <code>save</code> command and readable by the <code>load</code> command. They can be created on one machine and later read by MATLAB on another machine with a different

floating-point format, retaining as much accuracy and range as the disparate formats allow. They can also be manipulated by other programs, external to MATLAB.

The Application Program Interface Libraries contain C and Fortran callable routines to read and write MAT-files from external programs.

**See Also**

<code>fprintf</code>	Write formatted data to file
<code>fscanf</code>	Read formatted data from file
<code>save</code>	Save workspace variables on disk
<code>spconvert</code>	Import matrix from sparse matrix external format
See also <code>partialpath</code> .	

# log

---

<b>Purpose</b>	Natural logarithm								
<b>Syntax</b>	$Y = \log(X)$								
<b>Description</b>	The <code>log</code> 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:								
	$\log(z) = \log(\text{abs}(z)) + i * \text{atan2}(y, x)$								
<b>Examples</b>	The statement <code>abs(log(-1))</code> is a clever way to generate $\pi$ :								
	<pre>ans = 3.1416</pre>								
<b>See Also</b>	<table><tr><td><code>exp</code></td><td>Exponential</td></tr><tr><td><code>log10</code></td><td>Common (base 10) logarithm</td></tr><tr><td><code>log2</code></td><td>Base 2 logarithm and dissect floating-point numbers into exponent and mantissa</td></tr><tr><td><code>logm</code></td><td>Matrix logarithm</td></tr></table>	<code>exp</code>	Exponential	<code>log10</code>	Common (base 10) logarithm	<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa	<code>logm</code>	Matrix logarithm
<code>exp</code>	Exponential								
<code>log10</code>	Common (base 10) logarithm								
<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa								
<code>logm</code>	Matrix logarithm								

<b>Purpose</b>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa																					
<b>Syntax</b>	$Y = \log2(X)$ $[F, E] = \log2(X)$																					
<b>Description</b>	$Y = \log2(X)$ 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 $0.5 \leq \text{abs}(F) < 1$ . For real $X$ , $F$ satisfies the equation: $X = F \cdot 2^E$ . Argument $E$ is an array of integers that, for real $X$ , satisfy the equation: $X = F \cdot 2^E$ .																					
<b>Remarks</b>	This function corresponds to the ANSI C function <code>frexp()</code> and the IEEE floating-point standard function <code>logb()</code> . Any zeros in $X$ produce $F = 0$ and $E = 0$ .																					
<b>Examples</b>	For IEEE arithmetic, the statement $[F, E] = \log2(X)$ yields the values:																					
	<table border="0"> <thead> <tr> <th><b>X</b></th> <th><b>F</b></th> <th><b>E</b></th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1/2</td> <td>1</td> </tr> <tr> <td>pi</td> <td>pi / 4</td> <td>2</td> </tr> <tr> <td>-3</td> <td>-3/4</td> <td>2</td> </tr> <tr> <td>eps</td> <td>1/2</td> <td>-51</td> </tr> <tr> <td>real max</td> <td>1-eps/2</td> <td>1024</td> </tr> <tr> <td>real min</td> <td>1/2</td> <td>-1021</td> </tr> </tbody> </table>	<b>X</b>	<b>F</b>	<b>E</b>	1	1/2	1	pi	pi / 4	2	-3	-3/4	2	eps	1/2	-51	real max	1-eps/2	1024	real min	1/2	-1021
<b>X</b>	<b>F</b>	<b>E</b>																				
1	1/2	1																				
pi	pi / 4	2																				
-3	-3/4	2																				
eps	1/2	-51																				
real max	1-eps/2	1024																				
real min	1/2	-1021																				
<b>See Also</b>	<a href="#">log</a> Natural logarithm <a href="#">pow2</a> Base 2 power and scale floating-point numbers																					

# log10

---

<b>Purpose</b>	Common (base 10) logarithm								
<b>Syntax</b>	$Y = \log10(X)$								
<b>Description</b>	The <code>log10</code> 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 $X$ .								
<b>Examples</b>	<p>On a computer with IEEE arithmetic</p> <p><code>log10(realmax)</code> is 308.2547</p> <p>and</p> <p><code>log10(eps)</code> is -15.6536</p>								
<b>See Also</b>	<table><tr><td><code>exp</code></td><td>Exponential</td></tr><tr><td><code>log</code></td><td>Natural logarithm</td></tr><tr><td><code>log2</code></td><td>Base 2 logarithm and dissect floating-point numbers into exponent and mantissa</td></tr><tr><td><code>logm</code></td><td>Matrix logarithm</td></tr></table>	<code>exp</code>	Exponential	<code>log</code>	Natural logarithm	<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa	<code>logm</code>	Matrix logarithm
<code>exp</code>	Exponential								
<code>log</code>	Natural logarithm								
<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa								
<code>logm</code>	Matrix logarithm								

<b>Purpose</b>	Convert numeric values to logical
<b>Syntax</b>	<code>K = logical(A)</code>
<b>Description</b>	<code>K = logical(A)</code> returns an array that can be used for logical indexing or logical tests.  <code>A(B)</code> , where <code>B</code> is a logical array, returns the values of <code>A</code> at the indices where the real part of <code>B</code> is nonzero. <code>B</code> must be the same size as <code>A</code> .
<b>Remarks</b>	Logical arrays are also created by the relational operators ( <code>==,&lt;,&gt;,~</code> , etc.) and functions like <code>any</code> , <code>all</code> , <code>isnan</code> , <code>isinf</code> , and <code>isfinite</code> .
<b>Examples</b>	Given <code>A = [1 2 3; 4 5 6; 7 8 9]</code> , the statement <code>B = logical(eye(3))</code> 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

---

<b>Purpose</b>	Matrix logarithm
<b>Syntax</b>	$Y = \text{logm}(X)$ $[Y, \text{esterr}] = \text{logm}(X)$
<b>Description</b>	$Y = \text{logm}(X)$ returns the matrix logarithm: the inverse function of $\text{expm}(X)$ . Complex results are produced if $X$ has negative eigenvalues. A warning message is printed if the computed $\text{expm}(Y)$ is not close to $X$ .  $[Y, \text{esterr}] = \text{logm}(X)$ does not print any warning message, but returns an estimate of the relative residual, $\text{norm}(\text{expm}(Y) - X) / \text{norm}(X)$ .
<b>Remarks</b>	If $X$ is real symmetric or complex Hermitian, then so is $\text{logm}(X)$ . Some matrices, like $X = [0 \ 1; \ 0 \ 0]$ , do not have any logarithms, real or complex, and $\text{logm}$ cannot be expected to produce one.
<b>Limitations</b>	For most matrices:  $\text{logm}(\text{expm}(X)) = X = \text{expm}(\text{logm}(X))$  These identities may fail for some $X$ . For example, if the computed eigenvalues of $X$ include an exact zero, then $\text{logm}(X)$ generates infinity. Or, if the elements of $X$ are too large, $\text{expm}(X)$ may overflow.
<b>Examples</b>	Suppose $A$ is the 3-by-3 matrix  $\begin{matrix} 1 & 1 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & -1 \end{matrix}$ and $X = \text{expm}(A)$ is  $X = \begin{matrix} 2.7183 & 1.7183 & 1.0862 \\ 0 & 1.0000 & 1.2642 \\ 0 & 0 & 0.3679 \end{matrix}$

Then  $A = \text{logm}(X)$  produces the original matrix A.

$A =$

$$\begin{matrix} 1.0000 & 1.0000 & 0.0000 \\ 0 & 0 & 2.0000 \\ 0 & 0 & -1.0000 \end{matrix}$$

But  $\text{log}(X)$  involves taking the logarithm of zero, and so produces

$\text{ans} =$

$$\begin{matrix} 1.0000 & 0.5413 & 0.0826 \\ -\text{Inf} & 0 & 0.2345 \\ -\text{Inf} & -\text{Inf} & -1.0000 \end{matrix}$$

## 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

<code>expm</code>	Matrix exponential
<code>funm</code>	Evaluate functions of a matrix
<code>sqrtm</code>	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

---

<b>Purpose</b>	Generate logarithmically spaced vectors
<b>Syntax</b>	$y = \text{logspace}(a, b)$ $y = \text{logspace}(a, b, n)$ $y = \text{logspace}(a, \pi)$
<b>Description</b>	The <code>logspace</code> function generates logarithmically spaced vectors. Especially useful for creating frequency vectors, it is a logarithmic equivalent of <code>linspace</code> and the ":" or colon operator.
	$y = \text{logspace}(a, b)$ generates a row vector $y$ of 50 logarithmically spaced points between decades $10^a$ and $10^b$ .
	$y = \text{logspace}(a, b, n)$ generates $n$ points between decades $10^a$ and $10^b$ .
	$y = \text{logspace}(a, \pi)$ 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.
<b>Remarks</b>	All the arguments to <code>logspace</code> must be scalars.
<b>See Also</b>	<code>:</code> (Colon)      Create vectors, matrix subscripting, and for iterations <code>linspace</code> Generate linearly spaced vectors

<b>Purpose</b>	Keyword search through all help entries
<b>Syntax</b>	<code>lookfor topic</code> <code>lookfor topic -all</code>
<b>Description</b>	<code>lookfor topic</code> searches for the string <i>topic</i> 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, <code>lookfor</code> displays the H1 line.  <code>lookfor topic -all</code> searches the entire first comment block of an M-file looking for <i>topic</i> .
<b>Examples</b>	For example  <code>lookfor inverse</code>  finds at least a dozen matches, including H1 lines containing “inverse hyperbolic cosine,” “two-dimensional inverse FFT,” and “pseudoinverse.” Contrast this with  <code>which inverse</code>  or  <code>what inverse</code>  These commands run more quickly, but probably fail to find anything because MATLAB does not ordinarily have a function <code>inverse</code> .  In summary, <code>what</code> lists the functions in a given directory, <code>which</code> finds the directory containing a given function or file, and <code>lookfor</code> finds all functions in all directories that might have something to do with a given keyword.
<b>See Also</b>	<code>dir</code> Directory listing <code>help</code> Online help for MATLAB functions and M-files <code>what</code> Directory listing of M-files, MAT-files, and MEX-files <code>which</code> Locate functions and files <code>who</code> List directory of variables in memory

# **lower**

---

<b>Purpose</b>	Convert string to lower case
<b>Syntax</b>	<code>t = lower('str')</code> <code>B = lower(A)</code>
<b>Description</b>	<code>t = lower('str')</code> returns the string formed by converting any upper-case characters in <i>str</i> to the corresponding lower-case characters and leaving all other characters unchanged.  <code>B = lower(A)</code> when <i>A</i> is a cell array of strings, returns a cell array the same size as <i>A</i> containing the result of applying <code>lower</code> to each string within <i>A</i> .
<b>Examples</b>	<code>lower('MathWorks')</code> is <code>mathworks</code> .
<b>Remarks</b>	Character sets supported: <ul style="list-style-type: none"><li>• Mac: Standard Roman</li><li>• PC: Windows Latin-1</li><li>• Other: ISO Latin-1 (ISO 8859-1)</li></ul>
<b>See Also</b>	<a href="#">upper</a> Convert string to upper case

<b>Purpose</b>	Least squares solution in the presence of known covariance
<b>Syntax</b>	$x = \text{lscov}(A, b, V)$ $[x, dx] = \text{lscov}(A, b, V)$
<b>Description</b>	$x = \text{lscov}(A, b, V)$ 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 $V$ .
	$[x, dx] = \text{lscov}(A, b, V)$ returns the standard errors of $x$ in $dx$ . The standard statistical formula for the standard error of the coefficients is:
	$\text{mse} = B' * (\text{inv}(V) - \text{inv}(V) * A * \text{inv}(A' * \text{inv}(V) * A) * A' * \text{inv}(V)) * B. / (m-n)$ $dx = \text{sqrt}(\text{diag}(\text{inv}(A' * \text{inv}(V) * A) * \text{mse}))$
<b>Algorithm</b>	The vector $x$ minimizes the quantity $(A^*x-b)' * \text{inv}(V) * (A^*x-b)$ . The classical linear algebra solution to this problem is $x = \text{inv}(A' * \text{inv}(V) * A) * A' * \text{inv}(V) * b$ but the <code>lscov</code> function instead computes the QR decomposition of $A$ and then modifies $Q$ by $V$ .
<b>See Also</b>	<code>\</code> Matrix left division (backslash) <code>nnl s</code> Nonnegative least squares <code>qr</code> Orthogonal-triangular decomposition
<b>Reference</b>	Strang, G., <i>Introduction to Applied Mathematics</i> , Wellesley-Cambridge, 1986, p. 398.

# lu

---

<b>Purpose</b>	LU matrix factorization
<b>Syntax</b>	$[L, U] = \text{lu}(X)$ $[L, U, P] = \text{lu}(X)$ $\text{lu}(X)$
<b>Description</b>	The <code>lu</code> 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 <i>LU</i> , or sometimes the <i>LR</i> , factorization.  $[L, U] = \text{lu}(X)$ 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 $X = L^*U$ .  $[L, U, P] = \text{lu}(X)$ returns an upper triangular matrix in $U$ , a lower triangular matrix in $L$ , and a permutation matrix in $P$ , so that $L^*U = P^*X$ .  $\text{lu}(X)$ returns the output from the LINPACK routine ZGEFA.
<b>Remarks</b>	Most of the algorithms for computing LU factorization are variants of Gaussian elimination. The factorization is a key step in obtaining the inverse with <code>iinv</code> and the determinant with <code>det</code> . It is also the basis for the linear equation solution or matrix division obtained with <code>\</code> and <code>/</code> .
<b>Arguments</b>	<p><code>L</code>      A factor of <math>X</math>. Depending on the form of the function, <math>L</math> is either lower triangular, or else the product of a lower triangular matrix with a permutation matrix <math>P</math>.</p> <p><code>U</code>      An upper triangular matrix that is a factor of <math>X</math>.</p> <p><code>P</code>      The permutation matrix satisfying the equation <math>L^*U = P^*X</math>.</p>
<b>Examples</b>	Start with
	$A = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{matrix}$

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,  $X = \text{inv}(A)$ , is actually computed from the inverses of the triangular factors:

$$X = \text{inv}(U) * \text{inv}(L)$$

The determinant of the example matrix is

$$d = \det(A)$$

which gives

$$\begin{aligned} d &= \\ &27 \end{aligned}$$

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:

$$x = A \backslash b$$

The solution is actually computed by solving two triangular systems:

$$y = L \backslash b, \quad x = U \backslash y$$

## 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
i nv	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**

```
lui nc(X, '0')
[L, U] = lui nc(X, '0')
[L, U, P] = lui nc(X, '0')
lui nc(X, droptol)
lui nc(X, options)
[L, U] = lui nc(X, options)
[L, U] = lui nc(X, droptol)
[L, U, P] = lui nc(X, options)
[L, U, P] = lui nc(X, droptol)
```

**Description**

`lui nc` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`lui nc(X, '0')` computes the incomplete LU factorization of level 0 of a square sparse matrix. The triangular factors have the same sparsity pattern as the permutation of the original sparse matrix `X`, and their product agrees with the permuted `X` over its sparsity pattern. `lui nc(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(lui nc(X, '0')) = nnz(X)`, with the possible exception of some zeros due to cancellation.

`[L, U] = lui nc(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:

$$\text{nnz}(L) + \text{nnz}(U) = \text{nnz}(X) + n, \text{ where } X \text{ is } n\text{-by-}n.$$

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] = lui nc(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`

$$\text{spones}(L) = \text{spones}(\text{tril}(P^*X))$$

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.

`lui nc(X, droptol)` computes the incomplete LU factorization of any sparse matrix using a drop tolerance. `droptol` must be a non-negative scalar.

`lui nc(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.

`lui nc(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, `lui nc` 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 described in greater detail in [l u](#).

`l u i nc(X, options)` is the same as `l u i nc(X, droptol)` if `options` has `droptol` as its only field.

`[L, U] = l u i nc(X, options)` returns a permutation of a unit lower triangular matrix in `L` and an upper triangular matrix in `U`. The product `L*U` is an approximation to `X`. `l u i nc(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] = l u i nc(X, options)` is the same as `l u i nc(X, droptol)` if `options` has `droptol` as its only field.

`[L, U, P] = l u i nc(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

$$\text{abs}(U(i, j)) \geq \text{droptol} * \text{norm}(X(:, j)),$$

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`

$$\text{abs}(L(i, j)) \geq \text{droptol} * \text{norm}(X(:, j)) / U(j, j).$$

The product `L*U` is an approximation to the permuted `P*X`.

`[L, U, P] = l u i nc(X, options)` is the same as `[L, U, P] = l u i nc(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 `udi ag` 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.

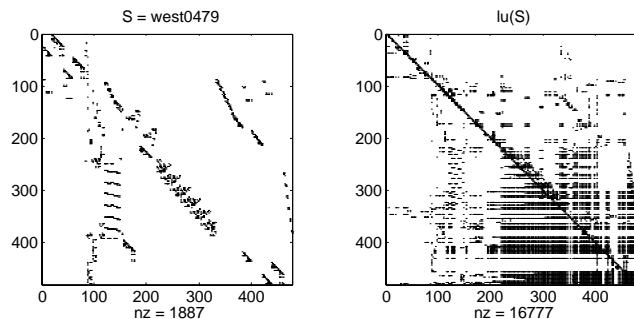
## Limitations

`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);
```

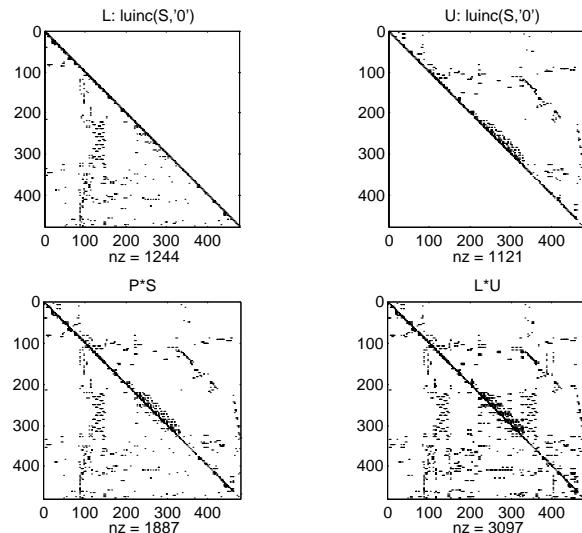


Compute the incomplete LU factorization of level 0.

```
[L, U, P] = luinc(S, '0');
D = (L*U) .* spones(P*S) - P*S;
```

`spones(U)` and `spones(triu(P*S))` are identical.

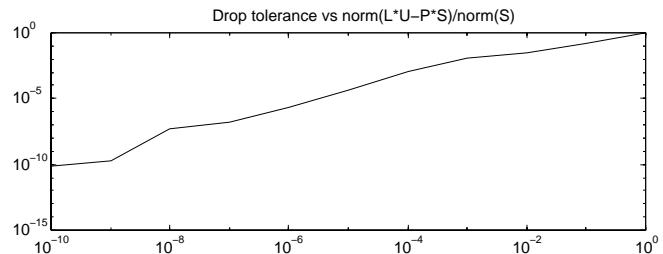
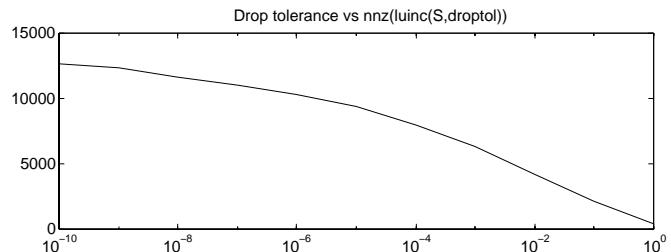
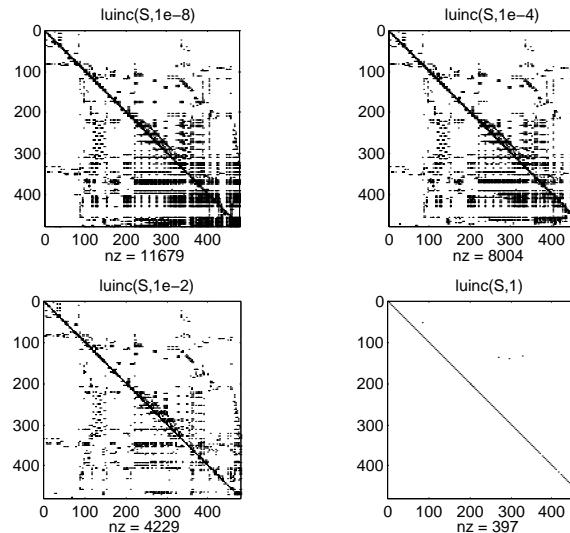
spones(L) and spones(tril(P\*S)) disagree at 73 places on the diagonal, where L is 1 and P\*S is 0, and also at position (206,113), where L is 0 due to cancellation, and P\*S is -1. D has entries of the order of eps.



[ ILO, IU0, IPO ] = 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  $\text{norm}(L^*U - P^*S, 1) / \text{norm}(S, 1)$  in second figure below.



**Algorithm**

`lui nc(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.

`lui nc(X, droptol)` and `lui nc(X, options)` are based on the column-oriented lu for sparse matrices.

**See Also**

<code>lu</code>	LU matrix factorization
<code>chol nc</code>	Sparse Incomplete Cholesky and Cholesky-Infinity factorizations
<code>bi cg</code>	BiConjugate Gradients method

**References**

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

# magic

---

<b>Purpose</b>	Magic square
<b>Syntax</b>	$M = \text{magic}(n)$
<b>Description</b>	$M = \text{magic}(n)$ 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.
<b>Remarks</b>	A magic square, scaled by its magic sum, is doubly stochastic.
<b>Examples</b>	The magic square of order 3 is $\begin{matrix} M = \text{magic}(3) \\ M = \\ \begin{matrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{matrix} \end{matrix}$ <p>This is called a magic square because the sum of the elements in each column is the same.</p> $\begin{matrix} \text{sum}(M) = \\ 15 & 15 & 15 \end{matrix}$ <p>And the sum of the elements in each row, obtained by transposing twice, is the same.</p> $\begin{matrix} \text{sum}(M')' = \\ 15 \\ 15 \\ 15 \end{matrix}$ <p>This is also a special magic square because the diagonal elements have the same sum.</p> $\begin{matrix} \text{sum}(\text{diag}(M)) = \\ 15 \end{matrix}$ <p>The value of the characteristic sum for a magic square of order <math>n</math> is</p> $\text{sum}(1:n^2)/n$ <p>which, when <math>n = 3</math>, is 15.</p>

**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**

<code>ones</code>	Create an array of all ones
<code>rand</code>	Uniformly distributed random numbers and arrays

## mat2str

---

<b>Purpose</b>	Convert a matrix into a string						
<b>Syntax</b>	<pre>str = mat2str(A) str = mat2str(A, n)</pre>						
<b>Description</b>	<p><code>str = mat2str(A)</code> converts matrix A into a string, suitable for input to the <code>eval</code> function, using full precision.</p> <p><code>str = mat2str(A, n)</code> converts matrix A using n digits of precision.</p>						
<b>Limitations</b>	The <code>mat2str</code> function is intended to operate on scalar, vector, or rectangular array inputs only. An error will result if A is a multidimensional array.						
<b>Examples</b>	<p>Consider the matrix:</p> <pre>A =     1     2     3     4</pre> <p>The statement</p> <pre>b = mat2str(A)</pre> <p>produces:</p> <pre>b = [ 1 2 ; 3 4 ]</pre> <p>where b is a string of 11 characters, including the square brackets, spaces, and a semicolon.</p> <p><code>eval(mat2str(A))</code> reproduces A.</p>						
<b>See Also</b>	<table><tr><td><code>int2str</code></td><td>Integer to string conversion</td></tr><tr><td><code>sprintf</code></td><td>Write formatted data to a string</td></tr><tr><td><code>str2num</code></td><td>String to number conversion</td></tr></table>	<code>int2str</code>	Integer to string conversion	<code>sprintf</code>	Write formatted data to a string	<code>str2num</code>	String to number conversion
<code>int2str</code>	Integer to string conversion						
<code>sprintf</code>	Write formatted data to a string						
<code>str2num</code>	String to number conversion						

**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

## **matlabroot**

---

<b>Purpose</b>	Root directory of MATLAB installation
<b>Syntax</b>	<code>rd = matlabroot</code>
<b>Description</b>	<code>rd = matlabroot</code> returns the name of the directory in which the MATLAB software is installed.
<b>Example</b>	<code>fullfile(matlabroot, 'tool box', 'matlab', 'general', '')</code> produces a full path to the <code>tool box/matlab/general</code> directory that is correct for the platform it is executed on.

<b>Purpose</b>	Maximum elements of an array										
<b>Syntax</b>	$C = \text{max}(A)$ $C = \text{max}(A, B)$ $C = \text{max}(A, [], dim)$ $[C, I] = \text{max}(\dots)$										
<b>Description</b>	<p><math>C = \text{max}(A)</math> returns the largest elements along different dimensions of an array.</p> <p>If A is a vector, <math>\text{max}(A)</math> returns the largest element in A.</p> <p>If A is a matrix, <math>\text{max}(A)</math> treats the columns of A as vectors, returning a row vector containing the maximum element from each column.</p> <p>If A is a multidimensional array, <math>\text{max}(A)</math> treats the values along the first non-singleton dimension as vectors, returning the maximum value of each vector.</p> <p><math>C = \text{max}(A, B)</math> returns an array the same size as A and B with the largest elements taken from A or B.</p> <p><math>C = \text{max}(A, [], dim)</math> returns the largest elements along the dimension of A specified by scalar <i>dim</i>. For example, <math>\text{max}(A, [], 1)</math> produces the maximum values along the first dimension (the rows) of A.</p> <p><math>[C, I] = \text{max}(\dots)</math> 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.</p>										
<b>Remarks</b>	For complex input A, $\text{max}$ returns the complex number with the largest modulus, computed with $\text{max}(\text{abs}(A))$ . The $\text{max}$ function ignores NaNs.										
<b>See Also</b>	<table border="0"> <tr> <td><code>iisnan</code></td> <td>Detect Not-A-Number (NaN)</td> </tr> <tr> <td><code>mean</code></td> <td>Average or mean values of array</td> </tr> <tr> <td><code>median</code></td> <td>Median values of array</td> </tr> <tr> <td><code>min</code></td> <td>Minimum elements of an array</td> </tr> <tr> <td><code>sort</code></td> <td>Sort elements in ascending order</td> </tr> </table>	<code>iisnan</code>	Detect Not-A-Number (NaN)	<code>mean</code>	Average or mean values of array	<code>median</code>	Median values of array	<code>min</code>	Minimum elements of an array	<code>sort</code>	Sort elements in ascending order
<code>iisnan</code>	Detect Not-A-Number (NaN)										
<code>mean</code>	Average or mean values of array										
<code>median</code>	Median values of array										
<code>min</code>	Minimum elements of an array										
<code>sort</code>	Sort elements in ascending order										

# mean

---

<b>Purpose</b>	Average or mean value of arrays
<b>Syntax</b>	$M = \text{mean}(A)$ $M = \text{mean}(A, dim)$
<b>Description</b>	$M = \text{mean}(A)$ returns the mean values of the elements along different dimensions of an array. If A is a vector, $\text{mean}(A)$ returns the mean value of A. If A is a matrix, $\text{mean}(A)$ treats the columns of A as vectors, returning a row vector of mean values. If A is a multidimensional array, $\text{mean}(A)$ treats the values along the first non-singleton dimension as vectors, returning an array of mean values. $M = \text{mean}(A, dim)$ returns the mean values for elements along the dimension of A specified by scalar <i>dim</i> .
<b>Examples</b>	<pre>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</pre>
<b>See Also</b>	<a href="#">corrcoef</a> Correlation coefficients <a href="#">cov</a> Covariance matrix <a href="#">max</a> Maximum elements of an array <a href="#">median</a> Median value of arrays <a href="#">min</a> Minimum elements of an array <a href="#">std</a> Standard deviation

<b>Purpose</b>	Median value of arrays												
<b>Syntax</b>	$M = \text{median}(A)$ $M = \text{median}(A, dim)$												
<b>Description</b>	<p><math>M = \text{median}(A)</math> returns the median values of the elements along different dimensions of an array.</p> <p>If <math>A</math> is a vector, <math>\text{median}(A)</math> returns the median value of <math>A</math>.</p> <p>If <math>A</math> is a matrix, <math>\text{median}(A)</math> treats the columns of <math>A</math> as vectors, returning a row vector of median values.</p> <p>If <math>A</math> is a multidimensional array, <math>\text{median}(A)</math> treats the values along the first nonsingleton dimension as vectors, returning an array of median values.</p> <p><math>M = \text{median}(A, dim)</math> returns the median values for elements along the dimension of <math>A</math> specified by scalar <math>dim</math>.</p>												
<b>Examples</b>	<pre>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</pre>												
<b>See Also</b>	<table> <tr> <td><code>corrcoef</code></td> <td>Correlation coefficients</td> </tr> <tr> <td><code>cov</code></td> <td>Covariance matrix</td> </tr> <tr> <td><code>max</code></td> <td>Maximum elements of an array</td> </tr> <tr> <td><code>mean</code></td> <td>Average or mean value of arrays</td> </tr> <tr> <td><code>min</code></td> <td>Minimum elements of an array</td> </tr> <tr> <td><code>std</code></td> <td>Standard deviation</td> </tr> </table>	<code>corrcoef</code>	Correlation coefficients	<code>cov</code>	Covariance matrix	<code>max</code>	Maximum elements of an array	<code>mean</code>	Average or mean value of arrays	<code>min</code>	Minimum elements of an array	<code>std</code>	Standard deviation
<code>corrcoef</code>	Correlation coefficients												
<code>cov</code>	Covariance matrix												
<code>max</code>	Maximum elements of an array												
<code>mean</code>	Average or mean value of arrays												
<code>min</code>	Minimum elements of an array												
<code>std</code>	Standard deviation												

# menu

---

<b>Purpose</b>	Generate a menu of choices for user input
<b>Syntax</b>	<code>k = menu('mttitle', 'opt1', 'opt2', ..., 'optn')</code>
<b>Description</b>	<code>k = menu('mttitle', 'opt1', 'opt2', ..., 'optn')</code> displays the menu whose title is in the string variable <code>'mttitle'</code> and whose choices are string variables <code>'opt1'</code> , <code>'opt2'</code> , and so on. <code>menu</code> returns the value you entered.
<b>Remarks</b>	To call <code>menu</code> from another ui-object, set that object's <code>Interruptible</code> property to <code>'yes'</code> . For more information, see the <i>MATLAB Graphics Guide</i> .
<b>Examples</b>	<code>k = menu('Choose a color', 'Red', 'Green', 'Blue')</code> displays



After input is accepted, use `k` to control the color of a graph.

```
color = [ 'r', 'g', 'b' ]  
plot(t, s, color(k))
```

<b>See Also</b>	The <code>ui control</code> command in the <i>MATLAB Graphics Guide</i> , and:
<code>input</code>	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**

$[X, Y] = \text{meshgrid}(x, y)$  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  $y$ .

$[X, Y] = \text{meshgrid}(x)$  is the same as  $[X, Y] = \text{meshgrid}(x, x)$ .

$[X, Y, Z] = \text{meshgrid}(x, y, z)$  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

$[X, Y, Z] = \text{meshgrid}(x, y, z)$

produces the same result as

$[Y, X, Z] = \text{ndgrid}(y, x, z)$

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

$[X, Y] = \text{meshgrid}(1: 3, 10: 14)$

# meshgrid

---

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

`griddata`, `mesh`, `ndgrid`, `slice`, `surf`

<b>Purpose</b>	Display method names						
<b>Syntax</b>	<pre>methods <i>class_name</i> n = methods(' <i>class_name</i>')</pre>						
<b>Description</b>	<p><code>methods <i>class_name</i></code> displays the names of the methods for the class with the name <i>class_name</i>.</p> <p><code>n = methods(' <i>class_name</i>')</code> returns the method names in a cell array of strings.</p>						
<b>See Also</b>	<table><tr><td><code>help</code></td><td>Online help for MATLAB functions and M-files</td></tr><tr><td><code>what</code></td><td>List M-, MAT- and MEX-files</td></tr><tr><td><code>which</code></td><td>Locate functions and files</td></tr></table>	<code>help</code>	Online help for MATLAB functions and M-files	<code>what</code>	List M-, MAT- and MEX-files	<code>which</code>	Locate functions and files
<code>help</code>	Online help for MATLAB functions and M-files						
<code>what</code>	List M-, MAT- and MEX-files						
<code>which</code>	Locate functions and files						

## **mexext**

---

**Purpose**      Return the MEX-filename extension

**Syntax**      `ext = mexext`

**Description**      `ext = mexext` returns the filename extension for the current platform.

<b>Purpose</b>	The name of the currently running M-file
<b>Syntax</b>	<code>mfilename</code>
<b>Description</b>	<code>mfilename</code> 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, <code>mfilename</code> returns an empty matrix.

# min

---

<b>Purpose</b>	Minimum elements of an array
<b>Syntax</b>	$C = \text{min}(A)$ $C = \text{min}(A, B)$ $C = \text{min}(A, [], dim)$ $[C, I] = \text{min}(\dots)$
<b>Description</b>	$C = \text{min}(A)$ returns the smallest elements along different dimensions of an array. If A is a vector, $\text{min}(A)$ returns the smallest element in A. If A is a matrix, $\text{min}(A)$ treats the columns of A as vectors, returning a row vector containing the minimum element from each column. If A is a multidimensional array, $\text{min}$ operates along the first nonsingleton dimension. $C = \text{min}(A, B)$ returns an array the same size as A and B with the smallest elements taken from A or B. $C = \text{min}(A, [], dim)$ returns the smallest elements along the dimension of A specified by scalar <i>dim</i> . For example, $\text{min}(A, [], 1)$ produces the minimum values along the first dimension (the rows) of A. $[C, I] = \text{min}(\dots)$ 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.
<b>Remarks</b>	For complex input A, $\text{min}$ returns the complex number with the smallest modulus, computed with $\text{min}(\text{abs}(A))$ . The $\text{min}$ function ignores NaNs.
<b>See Also</b>	<a href="#">max</a> Maximum elements of an array <a href="#">mean</a> Average or mean values of array <a href="#">median</a> Median values of array <a href="#">sort</a> Sort elements in ascending order

<b>Purpose</b>	True if M-file cannot be cleared
<b>Syntax</b>	<code>mislocked</code> <code>mislocked(fun)</code>
<b>Description</b>	<code>mislocked</code> by itself is 1 if the currently running M-file is locked and 0 otherwise. <code>mislocked(fun)</code> is 1 if the function named <i>fun</i> is locked in memory and 0 otherwise. Locked M-files cannot be removed with the <code>clear</code> function.
<b>See Also</b>	<code>mlock</code> , <code>munlock</code>

# **mkdir**

---

<b>Purpose</b>	Make directory
<b>Syntax</b>	<code>mkdir(<i>dirname</i>)</code> <code>mkdir(<i>parentdir, newdir</i>)</code> <code>status = mkdir(...)</code> <code>[status, msg] = mkdir(...)</code>
<b>Description</b>	<code>mkdir(<i>dirname</i>)</code> creates the directory <i>dirname</i> in the current directory.  <code>mkdir(<i>parentdir, newdir</i>)</code> creates the directory <i>newdir</i> in the existing directory <i>parentdir</i> .  <code>status = mkdir(...)</code> returns 1 if the new directory is created successfully, 2 if it already exists, and 0 otherwise.  <code>[status, msg] = mkdir(...)</code> returns a non-empty error message string when an error occurs.
<b>See Also</b>	<code>copyfile</code> Copy file

---

<b>Purpose</b>	Prevent M-file clearing
<b>Syntax</b>	<code>ml ock</code> <code>ml ock(<i>fun</i>)</code>
<b>Description</b>	<code>ml ock</code> , by itself, locks the currently running M-file so that subsequent <code>clear</code> commands do not remove it.  <code>ml ock(<i>fun</i>)</code> locks the M-file named <i>fun</i> in memory  Use the command <code>munl ock</code> or <code>munl ock(<i>fun</i>)</code> to return the M-file to its normal removable state.
<b>See Also</b>	<code>munl ock</code>

# mod

---

<b>Purpose</b>	Modulus (signed remainder after division)
<b>Syntax</b>	$M = \text{mod}(X, Y)$
<b>Definition</b>	$\text{mod}(x, y)$ is $x \bmod y$ .
<b>Description</b>	$M = \text{mod}(X, Y)$ returns the remainder $X - Y \cdot \lfloor X/Y \rfloor$ for nonzero $Y$ , and returns $X$ otherwise. $\text{mod}(X, Y)$ always differs from $X$ by a multiple of $Y$ .
<b>Remarks</b>	So long as operands $X$ and $Y$ are of the same sign, the function $\text{mod}(X, Y)$ returns the same result as does $\text{rem}(X, Y)$ . However, for positive $X$ and $Y$ ,
	$\text{mod}(-x, y) = \text{rem}(-x, y) + y$
	The $\text{mod}$ function is useful for congruence relationships: $x$ and $y$ are congruent (mod $m$ ) if and only if $\text{mod}(x, m) == \text{mod}(y, m)$ .
<b>Examples</b>	$\text{mod}(13, 5)$  ans = 3  $\text{mod}([1:5], 3)$  ans = 1      2      0      1      2  $\text{mod}(\text{magic}(3), 3)$  ans = 2      1      0 0      2      1 1      0      2
<b>Limitations</b>	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.
<b>See Also</b>	<a href="#">rem</a> 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** `di ary` `Save session in a disk file`

# mu2lin

<b>Purpose</b>	Mu-law to linear conversion
<b>Syntax</b>	<code>y = mu2lin(mu)</code>
<b>Description</b>	<code>y = mu2lin(mu)</code> converts mu-law encoded 8-bit audio signals, stored as "flints" in the range $0 \leq \text{mu} \leq 255$ , to linear signal amplitude in the range $-s < Y < s$ where $s = 32124/32768 \approx .9803$ . The input <code>mu</code> is often obtained using <code>fread(..., 'uchar')</code> to read byte-encoded audio files. "Flints" are MATLAB's integers – floating-point numbers whose values are integers.
<b>See Also</b>	<a href="#">auread</a> Read NeXT/SUN (.au) sound file <a href="#">lin2mu</a> Linear to mu-law conversion

<b>Purpose</b>	Allow M-file clearing
<b>Syntax</b>	<code>munlock</code> <code>munlock(<i>fun</i>)</code>
<b>Description</b>	<code>munlock</code> , by itself, unlocks the currently running M-file so that subsequent <code>clear</code> commands can remove it.  <code>munlock(<i>fun</i>)</code> unlocks the M-file named <i>fun</i> from memory. M-files are by default unlocked so that changes to the M-file are picked up. Calls to <code>unlock</code> are only needed to unlock M-files that have been locked with the command <code>lock</code> .
<b>See Also</b>	<code>lock</code>

# NaN

---

<b>Purpose</b>	Not-a-Number
<b>Syntax</b>	NaN
<b>Description</b>	NaN returns the IEEE arithmetic representation for Not-a-Number (NaN). These result from operations which have undefined numerical results.
<b>Examples</b>	<p>These operations produce NaN:</p> <ul style="list-style-type: none"><li>• Any arithmetic operation on a NaN, such as <code>sqrt(NaN)</code></li><li>• Addition or subtraction, such as magnitude subtraction of infinities as <code>(+Inf) + (-Inf)</code></li><li>• Multiplication, such as <code>0*Inf</code></li><li>• Division, such as <code>0/0</code> and <code>Inf/Inf</code></li><li>• Remainder, such as <code>rem(x, y)</code> where <code>y</code> is zero or <code>x</code> is infinity</li></ul>
<b>Remarks</b>	Logical operations involving NaNs always return false, except <code>~=</code> (not equal). Consequently, the statement <code>NaN ~= NaN</code> is true while the statement <code>NaN == NaN</code> is false.
<b>See Also</b>	<a href="#">Inf</a> <a href="#">Infinity</a>

<b>Purpose</b>	Check number of input arguments
<b>Syntax</b>	<code>msg = nargchk(<i>low</i>, <i>high</i>, <i>number</i>)</code>
<b>Description</b>	The <code>nargchk</code> function often is used inside an M-file to check that the correct number of arguments have been passed.  <code>msg = nargchk(<i>low</i>, <i>high</i>, <i>number</i>)</code> returns an error message if <i>number</i> is less than <i>low</i> or greater than <i>high</i> . If <i>number</i> is between <i>low</i> and <i>high</i> (inclusive), <code>nargchk</code> returns an empty matrix.
<b>Arguments</b>	<i>low</i> , <i>high</i> The minimum and maximum number of input arguments that should be passed.  <i>number</i> The number of arguments actually passed, as determined by the <code>nargin</code> function.
<b>Examples</b>	Given the function <code>foo</code> :  <pre>function f = foo(x, y, z) error(nargchk(2, 3, nargin))</pre> Then typing <code>foo(1)</code> produces:  Not enough input arguments.
<b>See Also</b>	<code>nargin</code> , <code>nargout</code> Number of function arguments

## **nargin, nargout**

---

<b>Purpose</b>	Number of function arguments
<b>Syntax</b>	<pre>n = nargin n = nargin('fun') n = nargout n = nargout('fun')</pre>
<b>Description</b>	<p>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.</p> <p><code>nargin</code> returns the number of input arguments specified for a function.</p> <p><code>nargin('fun')</code> returns the number of declared inputs for the M-file function <i>fun</i> or <code>-1</code> if the function has a variable of input arguments.</p> <p><code>nargout</code> returns the number of output arguments specified for a function.</p> <p><code>nargout('fun')</code> returns the number of declared outputs for the M-file function <i>fun</i>.</p>

## Examples

This example shows portions of the code for a function called `myplot`, which accepts an optional number of input and output arguments:

```
function [x0,y0] = myplot(fname,lims,npts,angl,subdiv)
% MYPLOT Plot a function.
% MYPLOT(fname,lims,npts,angl,subdiv)
%     The first two input arguments are
%     required; the other three have default values.
%
if nargin < 5, subdiv = 20; end
if nargin < 4, angl = 10; end
if nargin < 3, npts = 25; end
%
if nargout == 0
    plot(x,y)
else
    x0 = x;
    y0 = y;
end
```

## See Also

`inputname`  
`nargchk`

Input argument name  
Check number of input arguments

# nchoosek

---

<b>Purpose</b>	Binomial coefficient or all combinations																				
<b>Syntax</b>	$C = \text{nchoosek}(n, k)$ $C = \text{nchoosek}(v, k)$																				
<b>Description</b>	$C = \text{nchoosek}(n, k)$ where $n$ and $k$ are nonnegative integers, returns $n!/(n-k)! k!$ . This is the number of combinations of $n$ things taken $k$ at a time. $C = \text{nchoosek}(v, k)$ , 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.																				
<b>Examples</b>	The command <code>nchoosek(2: 2: 10, 4)</code> returns the even numbers from two to ten, taken four at a time:																				
	<table><tr><td>2</td><td>4</td><td>6</td><td>8</td></tr><tr><td>2</td><td>4</td><td>6</td><td>10</td></tr><tr><td>2</td><td>4</td><td>8</td><td>10</td></tr><tr><td>2</td><td>6</td><td>8</td><td>10</td></tr><tr><td>4</td><td>6</td><td>8</td><td>10</td></tr></table>	2	4	6	8	2	4	6	10	2	4	8	10	2	6	8	10	4	6	8	10
2	4	6	8																		
2	4	6	10																		
2	4	8	10																		
2	6	8	10																		
4	6	8	10																		
<b>Limitations</b>	This function is only practical for situations where $n$ is less than about 15.																				
<b>See Also</b>	<a href="#">perms</a> All possible permutations																				

<b>Purpose</b>	Generate arrays for multidimensional functions and interpolation
<b>Syntax</b>	$[X_1, X_2, X_3, \dots] = \text{ndgrid}(x_1, x_2, x_3, \dots)$ $[X_1, X_2, \dots] = \text{ndgrid}(x)$
<b>Description</b>	$[X_1, X_2, X_3, \dots] = \text{ndgrid}(x_1, x_2, x_3, \dots)$ transforms the domain specified by vectors $x_1, x_2, x_3, \dots$ into arrays $X_1, X_2, X_3, \dots$ that can be used for the evaluation of functions of multiple variables and multidimensional interpolation. The $i$ th dimension of the output array $X_i$ are copies of elements of the vector $x_i$ .
	$[X_1, X_2, \dots] = \text{ndgrid}(x)$ is the same as $[X_1, X_2, \dots] = \text{ndgrid}(x, x, \dots)$ .
<b>Examples</b>	To evaluate the function $x_1 e^{-x_1^2 - x_2^2}$ over the range $-2 < x_1 < 2 ; -2 < x_2 < 2$ :
	<pre>[X1, X2] = ndgrid(-2:2:2, -2:2:2); Z = X1 .* exp(-X1.^2 - X2.^2); mesh(Z)</pre>
<b>Remarks</b>	The <code>ndgrid</code> function is like <code>meshgrid</code> except that the order of the first two input arguments are switched. That is, the statement
	$[X_1, X_2, X_3] = \text{ndgrid}(x_1, x_2, x_3)$
	produces the same result as
	$[X_2, X_1, X_3] = \text{meshgrid}(x_2, x_1, x_3)$ .
	Because of this, <code>ndgrid</code> is better suited to multidimensional problems that aren't spatially based, while <code>meshgrid</code> is better suited to problems in two- or three-dimensional Cartesian space.
<b>See Also</b>	<code>meshgrid</code> <code>interpn</code> Generate X and Y matrices for three-dimensional plots Multidimensional data interpolation (table lookup).

## **ndims**

---

<b>Purpose</b>	Number of array dimensions
<b>Syntax</b>	<code>n = ndims(A)</code>
<b>Description</b>	<code>n = ndims(A)</code> 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 <code>size(A, dim) = 1</code> .
<b>Algorithm</b>	<code>ndims(x) is length(size(x)).</code>
<b>See Also</b>	<code>size</code> Array dimensions

---

<b>Purpose</b>	Next power of two						
<b>Syntax</b>	<code>p = nextpow2(A)</code>						
<b>Description</b>	<p><code>p = nextpow2(A)</code> returns the smallest power of two that is greater than or equal to the absolute value of <code>A</code>. (That is, <code>p</code> that satisfies <math>2^p \geq \text{abs}(A)</math>).</p> <p>This function is useful for optimizing FFT operations, which are most efficient when sequence length is an exact power of two.</p> <p>If <code>A</code> is non-scalar, <code>nextpow2</code> returns the smallest power of two greater than or equal to <code>length(A)</code>.</p>						
<b>Examples</b>	<p>For any integer <code>n</code> in the range from 513 to 1024, <code>nextpow2(n)</code> is 10.</p> <p>For a 1-by-30 vector <code>A</code>, <code>length(A)</code> is 30 and <code>nextpow2(A)</code> is 5.</p>						
<b>See Also</b>	<table><tr><td><code>fft</code></td><td>One-dimensional fast Fourier transform</td></tr><tr><td><code>log2</code></td><td>Base 2 logarithm and dissect floating-point numbers into exponent and mantissa</td></tr><tr><td><code>pow2</code></td><td>Base 2 power and scale floating-point numbers</td></tr></table>	<code>fft</code>	One-dimensional fast Fourier transform	<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa	<code>pow2</code>	Base 2 power and scale floating-point numbers
<code>fft</code>	One-dimensional fast Fourier transform						
<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa						
<code>pow2</code>	Base 2 power and scale floating-point numbers						

## nnls

---

<b>Purpose</b>	Nonnegative least squares																
<b>Syntax</b>	$x = \text{nnls}(A, b)$ $x = \text{nnls}(A, b, tol)$ $[x, w] = \text{nnls}(A, b)$ $[x, w] = \text{nnls}(A, b, tol)$																
<b>Description</b>	$x = \text{nnls}(A, b)$ solves the system of equations $Ax = b$ in a least squares sense, subject to the constraint that the solution vector $x$ has nonnegative elements: $x_j \geq 0$ , $j = 1, 2, \dots, n$ . The solution $x$ minimizes $\ (Ax - b)\ $ subject to $x \geq 0$ .  $x = \text{nnls}(A, b, tol)$ solves the system of equations, and specifies a tolerance $tol$ . By default, $tol$ is: $\max(\text{size}(A)) * \text{norm}(A, 1) * \text{eps}$ .  $[x, w] = \text{nnls}(A, b)$ also returns the dual vector $w$ , where $w_i \leq 0$ when $x_i = 0$ and $w_i \geq 0$ when $x_i > 0$ .  $[x, w] = \text{nnls}(A, b, tol)$ solves the system of equations, returns the dual vector $w$ , and specifies a tolerance $tol$ .																
<b>Examples</b>	Compare the unconstrained least squares solution to the $\text{nnls}$ solution for a 4-by-2 problem:  $A =$ <table><tr><td>0.0372</td><td>0.2869</td></tr><tr><td>0.6861</td><td>0.7071</td></tr><tr><td>0.6233</td><td>0.6245</td></tr><tr><td>0.6344</td><td>0.6170</td></tr></table> $b =$ <table><tr><td>0.8587</td></tr><tr><td>0.1781</td></tr><tr><td>0.0747</td></tr><tr><td>0.8405</td></tr></table> $[A \backslash b \text{ nnls}(A, b)] =$ <table><tr><td>-2.5627</td><td>0</td></tr><tr><td>3.1108</td><td>0.6929</td></tr></table>	0.0372	0.2869	0.6861	0.7071	0.6233	0.6245	0.6344	0.6170	0.8587	0.1781	0.0747	0.8405	-2.5627	0	3.1108	0.6929
0.0372	0.2869																
0.6861	0.7071																
0.6233	0.6245																
0.6344	0.6170																
0.8587																	
0.1781																	
0.0747																	
0.8405																	
-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.

## **nnz**

---

**Purpose** Number of nonzero matrix elements

**Syntax** `n = nnz(X)`

**Description** `n = nnz(X)` returns the number of nonzero elements in matrix `X`.

The density of a sparse matrix is  $\text{nnz}(X) / \text{prod}(\text{size}(X))$ .

**Examples** The matrix

```
w = sparse(wilkinson(21));
```

is a tridiagonal matrix with 20 nonzeros on each of three diagonals, so  
`nnz(w) = 60`.

<b>See Also</b>	<code>find</code>	Find indices and values of nonzero elements
	<code>nonzeros</code>	Nonzero matrix elements
	<code>nzmax</code>	Amount of storage allocated for nonzero matrix elements
	<code>size</code>	Array dimensions
	<code>whos</code>	List directory of variables in memory
	<code>isa</code>	Detect an object of a given class

**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,

$$\text{length}(s) = \text{nnz}(A) \leq \text{nzmax}(A) \leq \text{prod}(\text{size}(A))$$

<b>See Also</b>	<code>find</code>	Find indices and values of nonzero elements
	<code>nnz</code>	Number of nonzero matrix elements
	<code>nzmax</code>	Amount of storage allocated for nonzero matrix elements
	<code>size</code>	Array dimensions
	<code>whos</code>	List directory of variables in memory
	<code>isa</code>	Detect an object of a given class

# norm

<b>Purpose</b>	Vector and matrix norms
<b>Syntax</b>	$n = \text{norm}(A)$ $n = \text{norm}(A, p)$
<b>Description</b>	The <i>norm</i> of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. The <i>norm</i> function calculates several different types of matrix norms:
	$n = \text{norm}(A)$ returns the largest singular value of $A$ , $\max(\text{svd}(A))$ .
	$n = \text{norm}(A, p)$ returns a different kind of norm, depending on the value of $p$ :
If $p$ is...	Then <i>norm</i> returns...
1	The 1-norm, or largest column sum of $A$ , $\max(\sum(\text{abs}(A)))$ .
2	The largest singular value (same as <i>norm(A)</i> ).
<i>i nf</i>	The infinity norm, or largest row sum of $A$ , $\max(\sum(\text{abs}(A')))$ .
'fro'	The Frobenius-norm of matrix $A$ , $\sqrt{\sum(\text{diag}(A' * A))}$ .
When $A$ is a vector, slightly different rules apply:	
$\text{norm}(A, p)$	Returns $\sum(\text{abs}(A) . ^p)^{(1/p)}$ , for any $1 \leq p \leq \infty$ .
$\text{norm}(A)$	Returns $\text{norm}(A, 2)$ .
$\text{norm}(A, \text{i nf})$	Returns $\max(\text{abs}(A))$ .
$\text{norm}(A, -\text{i nf})$	Returns $\min(\text{abs}(A))$ .
<b>Remarks</b>	To obtain the root-mean-square (RMS) value, use $\text{norm}(A) / \sqrt(n)$ . Note that $\text{norm}(A)$ , where $A$ is an $n$ -element vector, is the length of $A$ .
<b>See Also</b>	<b>cond</b> Condition number with respect to inversion <b>normest</b> 2-norm estimate <b>svd</b> 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('willkison', 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**

<code>cond</code>	Condition number with respect to inversion
<code>condest</code>	1-norm matrix condition number estimate
<code>norm</code>	Vector and matrix norms
<code>svd</code>	Singular value decomposition

## **now**

---

<b>Purpose</b>	Current date and time
<b>Syntax</b>	<code>t = now</code>
<b>Description</b>	<code>t = now</code> returns the current date and time as a serial date number. To return the time only, use <code>rem(now, 1)</code> . To return the date only, use <code>floor(now)</code> .
<b>Examples</b>	<pre>t1 = now, t2 = rem(now, 1)</pre> <code>t1 =</code>  <code>7.2908e+05</code>  <code>t2 =</code>  <code>0.4013</code>
<b>See Also</b>	<code>clock</code> Current time as a date vector <code>date</code> Current date string <code>datenum</code> Serial date number

**Purpose** Null space of a matrix

**Syntax**  $B = \text{nul1}(A)$

**Description**  $B = \text{nul1}(A)$  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**

---

<b>Purpose</b>	Convert a numeric array into a cell array
<b>Syntax</b>	<code>c = num2cell(A)</code> <code>c = num2cell(A, dims)</code>
<b>Description</b>	<code>c = num2cell(A)</code> 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.  <code>c = num2cell(A, dims)</code> 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.
<b>Examples</b>	The statement  <code>num2cell(A, 2)</code> places the rows of A into separate cells. Similarly  <code>num2cell(A, [1 3])</code> places the column-depth pages of A into separate cells.
<b>See Also</b>	<a href="#">cat</a> Concatenate arrays

<b>Purpose</b>	Number to string conversion
<b>Syntax</b>	<pre>str = num2str(A) str = num2str(A, precision) str = num2str(A, format)</pre>
<b>Description</b>	<p>The <code>num2str</code> function converts numbers to their string representations. This function is useful for labeling and titling plots with numeric values.</p> <p><code>str = num2str(a)</code> converts array A into a string representation str with roughly four digits of precision and an exponent if required.</p> <p><code>str = num2str(a, precision)</code> converts the array A into a string representation str with maximum precision specified by <i>precision</i>. Argument <i>precision</i> specifies the number of digits the output string is to contain. The default is four.</p> <p><code>str = num2str(A, format)</code> converts array A using the supplied <i>format</i>. By default, this is '<code>%11.4g</code>', which signifies four significant digits in exponential or fixed-point notation, whichever is shorter. (See <code>fprintf</code> for format string details).</p>
<b>Examples</b>	<p><code>num2str(pi)</code> is 3.142.</p> <p><code>num2str(eps)</code> is 2.22e-16.</p> <p><code>num2str(magic(2))</code> produces the string matrix</p> <pre>1 3 4 2</pre>
<b>See Also</b>	<p><code>fprintf</code> Write formatted data to file <code>int2str</code> Integer to string conversion <code>sprintf</code> Write formatted data to a string</p>

# **nzmax**

---

<b>Purpose</b>	Amount of storage allocated for nonzero matrix elements
<b>Syntax</b>	<code>n = nzmax(S)</code>
<b>Description</b>	<code>n = nzmax(S)</code> returns the amount of storage allocated for nonzero elements.  If S is a sparse matrix... <code>nzmax(S)</code> is the number of storage locations allocated for the nonzero elements in S. If S is a full matrix... <code>nzmax(S) = prod(size(S))</code> .
	Often, <code>nnz(S)</code> and <code>nzmax(S)</code> 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 <code>nzmax(S)</code> reflects this. Alternatively, <code>sparse(i, j, s, m, n, nzmax)</code> or its simpler form, <code>spalloc(m, n, nzmax)</code> , can set <code>nzmax</code> in anticipation of later fill-in.
<b>See Also</b>	<code>find</code> Find indices and values of nonzero elements <code>nnz</code> Number of nonzero matrix elements <code>nonzeros</code> Nonzero matrix elements <code>size</code> Array dimensions <code>whos</code> List directory of variables in memory <code>isa</code> Detect an object of a given class

# ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb

<b>Purpose</b>	Solve differential equations
<b>Syntax</b>	<pre>[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, ...)</pre>
<b>Arguments</b>	<p><b>F</b> Name of the ODE file, a MATLAB function of <math>t</math> and <math>y</math> returning a column vector. All solvers can solve systems of equations in the form <math>y' = F(t, y)</math>. <code>ode15s</code>, <code>ode23s</code>, <code>ode23t</code>, and <code>ode23tb</code> can solve equations of the form <math>My' = F(t, y)</math>. Of these four solvers all but <code>ode23s</code> can solve equations in the form <math>M(t)y' = F(t, y)</math>. For information about ODE file syntax, see the <code>odefile</code> reference page.</p> <p><b>tspan</b> A vector specifying the interval of integration <math>[t_0 \ t_{final}]</math>. To obtain solutions at specific times (all increasing or all decreasing), use <math>tspan = [t_0, t_1, \dots, t_{final}]</math>.</p> <p><b>y0</b> A vector of initial conditions.</p> <p><b>options</b> Optional integration argument created using the <code>odeset</code> function. See <code>odeset</code> for details.</p> <p><b>p1, p2...</b> Optional parameters to be passed to <code>F</code>.</p> <p><b>T, Y</b> Solution matrix <math>Y</math>, where each row corresponds to a time returned in column vector <math>T</math>.</p>
<b>Description</b>	<p><code>[T, Y] = solver('F', tspan, y0)</code> with <math>tspan = [t_0 \ t_{final}]</math> integrates the system of differential equations <math>y' = F(t, y)</math> from time <math>t_0</math> to <math>t_{final}</math> with initial conditions <math>y_0</math>. '<code>F</code>' is a string containing the name of an ODE file. Function <code>F(t, y)</code> must return a column vector. Each row in solution array <math>y</math> corresponds to a time returned in column vector <math>t</math>. To obtain solutions at the specific times <math>t_0, t_1, \dots, t_{final}</math> (all increasing or all decreasing), use <math>tspan = [t_0 \ t_1 \ \dots \ t_{final}]</math>.</p> <p><code>[T, Y] = solver('F', tspan, y0, options)</code> solves as above with default integration parameters replaced by property values specified in <code>options</code>, an</p>

## **ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb**

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- or three-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 `JPattern` to '`on`' if  $\partial F / \partial y$  is a sparse matrix and the ODE file is coded so that `F([], [], 'j pattern')` returns a sparsity pattern matrix of 1's and 0's showing the nonzeros of  $\partial F / \partial y$ . Set `Jacbian` to '`on`' if the ODE file is coded so that `F(t, y, 'jacbian')` returns  $\partial F / \partial y$ .

`ode15s`, `ode23s`, `ode23t`, and `ode23tb` can solve problems  $M y' = F(t, y)$  with a constant mass matrix `M` that is nonsingular and (usually) sparse. Set `MassConstant` to '`on`' if the ODE file is coded so that `F([], [], 'mass')` returns `M` (see `fem2ode`). Of these four solvers all but `ode23s` 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`).

## ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb

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.
ode23t	Moderately Stiff	Low	If the problem is only moderately stiff and you need a solution without numerical damping.
ode23tb	Stiff	Low	If using crude error tolerances to solve stiff systems or there is a 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 2-511 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)
```

# **ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb**

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 *Using MATLAB*.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
RelTol, AbsTol	✓	✓	✓	✓	✓	✓	✓
OutputFcns, 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:

$$\begin{aligned}y'_1 &= y_2 \ y_3 & y_1(0) &= 0 \\y'_2 &= -y_1 \ y_3 & y_2(0) &= 1 \\y'_3 &= -0.51 \ y_1 \ y_2 & y_3(0) &= 1\end{aligned}$$

## ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb

To simulate this system, create a function M-file `rigid` containing the equations:

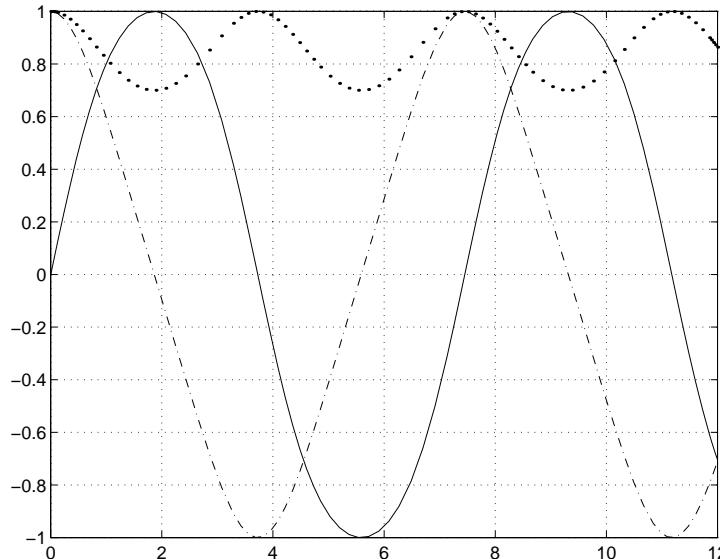
```
function dy = rigid(t, y)
dy = zeros(3, 1); % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

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

## **ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb**

the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$\begin{aligned}y'_1 &= y_2 \\y'_2 &= 1000(1 - y_1^2)y_2 - y_1\end{aligned}\quad \begin{aligned}y_1(0) &= 0 \\y_2(0) &= 1\end{aligned}$$

To simulate this system, create a function M-file vdp1000 containing the equations:

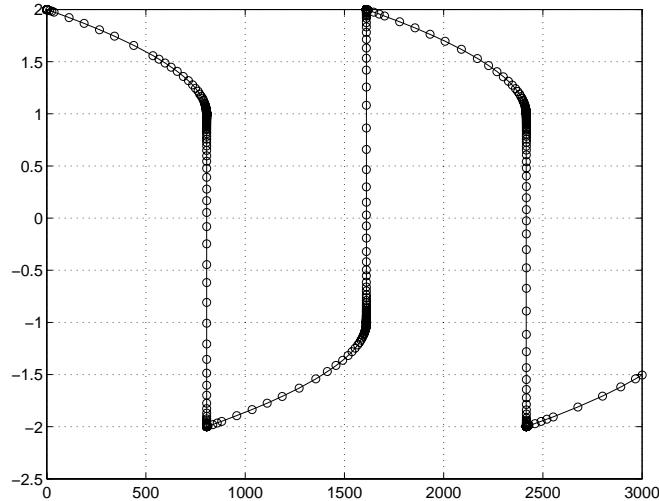
```
function dy = vdp1000(t, y)
dy = zeros(2, 1); % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

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');
```



## **ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb**

### **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-Basforth-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 below.

ode15s is a variable order solver based on the numerical differentiation formulas, NDFs. Optionally, it uses the backward differentiation formulas, BDFs (also known as Gear’s method) that are usually less efficient. Like ode113, ode15s is a multistep solver. 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]

ode23t is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping.

ode23tb is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like ode23s, this solver may be more efficient than ode15s at crude tolerances. [8, 9]

### **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).
- [8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics* 20, 1996.
- [9] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp 436-451

<b>Purpose</b>	Define a differential equation problem for ODE solvers
<b>Description</b>	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 <code>odefile</code> , 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 solvers `ode15s`, `ode23t`, and `ode23tb` can solve  $M(t)y' = F(t, y)$ . `ode15s`, `ode23s`, `ode23t`, and `ode23tb` 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 2-517).

**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.

## **odefile**

---

- Edit the file to eliminate any cases not applicable to your IVP.

- Insert the appropriate information where indicated. The definition of the ODE system is required information.

```

switch flag
case '' % Return dy/dt = f(t, y).
varargout{1} = f(t, y, p1, p2);
case 'init' % Return default [tspan, y0, options].
[varargout{1:3}] = init(p1, p2);
case 'jacobian' % Return Jacobian matrix df/dy.
varargout{1} = jacobian(t, y, p1, p2);
case 'j pattern' % Return sparsity pattern matrix S.
varargout{1} = j pattern(t, y, p1, p2);
case 'mass' % Return mass matrix M(t) or M
varargout{1} = mass(t, y, p1, p2);
case 'events' % Return [value, isterminal, direction].
[varargout{1:3}] = events(t, y, p1, p2);
otherwise
error(['Unknown flag ''', flag, '''']);
end
% -----
function dydt = f(t, y, p1, p2)
dydt = < Insert a function of t and/or y, p1, and p2 here. >
% -----
function [tspan, y0, options] = init(p1, p2)
tspan = < Insert tspan here. >;
y0 = < Insert y0 here. >;
options = < Insert options = odeset(...) or [] here. >;
% -----
function dfdy = jacobian(t, y, p1, p2)
dfdy = < Insert Jacobian matrix here. >;
% -----
function S = j pattern(t, y, p1, p2)
S = < Insert Jacobian matrix sparsity pattern here. >;
% -----
function M = mass(t, y, p1, p2)
M = < Insert mass matrix here. >;
% -----
function [value, isterminal, direction] = events(t, y, p1, p2)
value = < Insert event function vector here. >

```

```
isterminal = < Insert logical ISTERMINAL vector here. >;  
direction = < Insert DIRECTION vector here. >;
```

## Notes

- 1 The ODE file must accept  $t$  and  $y$  vectors from the ODE solvers and must return a column vector the same length as  $y$ . The optional input argument `flag` determines the type of output (mass matrix, Jacobian, etc.) returned by the ODE file.
- 2 The solvers repeatedly call the ODE file to evaluate the system of differential equations at various times. *This is required information*—you must define the ODE system to be solved.
- 3 The `switch` statement determines the type of output required, so that the ODE file can pass the appropriate information to the solver. (See 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 '`jacobi an`' 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 '`j pattern`' 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  $M\dot{y} = F(t, y)$  or  $M(t)\dot{y} = F(t, y)$  .
- 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 *Using MATLAB* and also the examples `bal1ode` and `orbi1ode`.
- 9 An unrecognized `flag` generates an error.

**Examples**

The van der Pol equation,  $y''_1 - \mu(1 - y_1^2)y'_1 + y_1 = 0$ , is equivalent to a system of coupled first-order differential equations:

$$\begin{aligned}y'_1 &= y_2 \\y''_2 &= \mu(1 - y_1^2)y_2 - y_1\end{aligned}$$

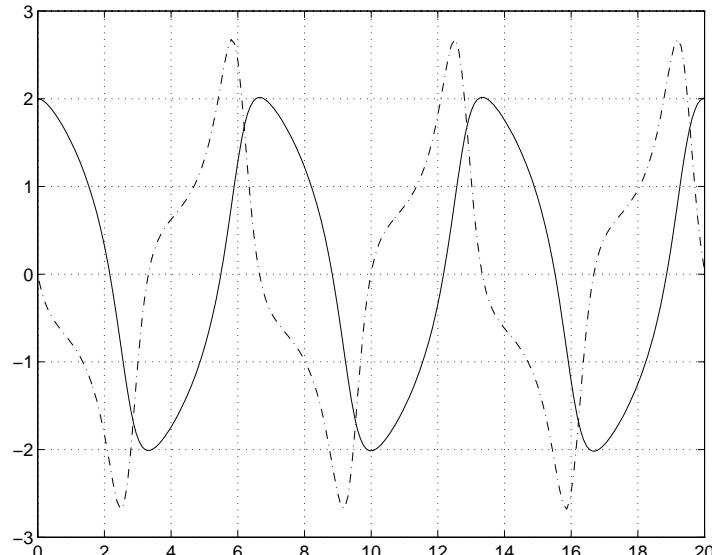
**The M-file**

```
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), '-.')
```



## odefile

---

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, vdode, 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

**Purpose** Extract properties from options structure created with odeset

**Syntax**

```
o = odeget(options, 'name')
o = odeget(options, 'name', default)
```

**Description** `o = odeget(options, 'name')` extracts the value of the property specified by string 'name' from integrator options structure `options`, returning an empty matrix if the property value is not specified in `options`. It is only necessary to type the leading characters that uniquely identify the property name. Case is ignored for property names. The empty matrix [] is a valid options argument.

`o = odeget(options, 'name', default)` returns `o = default` if the named property is not specified in `options`.

**Example** Having constructed an ODE options structure,

```
options = odeset('RelTol', 1e-4, 'AbsTol', [1e-3 2e-3 3e-3]);
```

you can view these property settings with `odeget`:

```
odeget(options, 'RelTol')
ans =
```

1.0000e-04

```
odeget(options, 'AbsTol')
ans =
```

0.0010    0.0020    0.0030

**See Also** `odeset`

# odeset

---

<b>Purpose</b>	Create or alter options structure for input to ODE solvers																														
<b>Syntax</b>	<pre>options = odeset('name1', value1, 'name2', value2, ...) options = odeset(olopts, 'name1', value1, ...) options = odeset(olopts, newopts) odeset</pre>																														
<b>Description</b>	<p>The <code>odeset</code> function lets you adjust the integration parameters of the ODE solvers. See below for information about the integration parameters.</p> <p><code>options = odeset('name1', value1, 'name2', value2, ...)</code> creates an integrator options structure in which the named properties have the specified values. The <code>odeset</code> function sets any unspecified properties to the empty matrix <code>[]</code>.</p> <p>It is sufficient to type only the leading characters that uniquely identify the property name. Case is ignored for property names.</p> <p><code>options = odeset(olopts, 'name1', value1, ...)</code> alters an existing options structure with the values supplied.</p> <p><code>options = odeset(olopts, newopts)</code> alters an existing options structure <code>olopts</code> by combining it with a new options structure <code>newopts</code>. Any new options not equal to the empty matrix overwrite corresponding options in <code>olopts</code>. For example:</p> <p style="text-align: center;"><code>olopts</code> <table border="1"><tr><td>F</td><td>1</td><td>[]</td><td>4</td><td>'s'</td><td>'s'</td><td>[]</td><td>[]</td><td>[]</td><td>...</td></tr></table> <code>newopts</code> <table border="1"><tr><td>T</td><td>3</td><td>F</td><td>[]</td><td>''</td><td>[]</td><td>[]</td><td>[]</td><td>[]</td><td>...</td></tr></table> <code>odeset(olopts, newopts)</code> <table border="1"><tr><td>T</td><td>3</td><td>F</td><td>4</td><td>''</td><td>'s'</td><td>[]</td><td>[]</td><td>[]</td><td>...</td></tr></table></p>	F	1	[]	4	's'	's'	[]	[]	[]	...	T	3	F	[]	''	[]	[]	[]	[]	...	T	3	F	4	''	's'	[]	[]	[]	...
F	1	[]	4	's'	's'	[]	[]	[]	...																						
T	3	F	[]	''	[]	[]	[]	[]	...																						
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`

## odeset

---

**Table 2-1: 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 2-2: Solver Output Properties**

Property	Value	Description
OutputFcn	String	The name of an installable output function (for example, <code>odeplot</code> , <code>odephas2</code> , <code>odephas3</code> , and <code>odeprint</code> ). The ODE solvers call <code>outputfcn(TSPAN, Y0, 'init')</code> before beginning the integration, to initialize the output function. Subsequently, the solver calls <code>status = outputfcn(T, Y)</code> after computing each output point $(T, Y)$ . The status return value should be 1 if integration should be halted (e.g., a <b>STOP</b> button has been pressed) and 0 otherwise. When the integration is complete, the solver calls <code>outputfcn([], [], 'done')</code> .
OutputSel	Vector of indices	Specifies which components of the solution vector are to be passed to the output function.

**Table 2-2: 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 2-3: 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 ([], [], 'j pattern') by returning a sparse matrix containing 1's showing the nonzeros of $\partial F/\partial y$ (see brussode).

**Table 2-3: 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 $F(t, [y_1 \ y_2 \ \dots])$ returns $[F(t, y_1) \ F(t, y_2) \ \dots]$ . 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 2-4: 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 2-5: 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 ODE file is coded so that $F(t, [], 'mass')$ returns a constant mass matrix $M$ (see odefile).

**Table 2-6: 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 2-7: `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**

`odef file`, `odeget`, `ode45`, `ode23`, `ode23t`, `ode23tb`, `ode113`, `ode15s`, `ode23s`

# ones

---

<b>Purpose</b>	Create an array of all ones								
<b>Syntax</b>	$Y = \text{ones}(n)$ $Y = \text{ones}(m, n)$ $Y = \text{ones}([m\ n])$ $Y = \text{ones}(d_1, d_2, d_3, \dots)$ $Y = \text{ones}([d_1\ d_2\ d_3, \dots])$ $Y = \text{ones}(\text{size}(A))$								
<b>Description</b>	$Y = \text{ones}(n)$ returns an $n$ -by- $n$ matrix of 1s. An error message appears if $n$ is not a scalar.  $Y = \text{ones}(m, n)$ or $Y = \text{ones}([m\ n])$ returns an $m$ -by- $n$ matrix of ones.  $Y = \text{ones}(d_1, d_2, d_3, \dots)$ or $Y = \text{ones}([d_1\ d_2\ d_3, \dots])$ returns an array of 1s with dimensions $d_1$ -by- $d_2$ -by- $d_3$ -by-....  $Y = \text{ones}(\text{size}(A))$ returns an array of 1s that is the same size as $A$ .								
<b>See Also</b>	<table><tr><td><code>eye</code></td><td>Identity matrix</td></tr><tr><td><code>rand</code></td><td>Uniformly distributed random numbers and arrays</td></tr><tr><td><code>randn</code></td><td>Normally distributed random numbers and arrays</td></tr><tr><td><code>zeros</code></td><td>Create an array of all zeros</td></tr></table>	<code>eye</code>	Identity matrix	<code>rand</code>	Uniformly distributed random numbers and arrays	<code>randn</code>	Normally distributed random numbers and arrays	<code>zeros</code>	Create an array of all zeros
<code>eye</code>	Identity matrix								
<code>rand</code>	Uniformly distributed random numbers and arrays								
<code>randn</code>	Normally distributed random numbers and arrays								
<code>zeros</code>	Create an array of all zeros								

**Purpose** Range space of a matrix

**Syntax**  $B = \text{orth}(A)$

**Description**  $B = \text{orth}(A)$  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  $B' * B = \text{eye}(\text{rank}(A))$ . The number of columns of  $B$  is the rank of  $A$ .

**See Also** [nul1](#) Null space of a matrix  
[svd](#) Singular value decomposition  
[rank](#) Rank of a matrix

# otherwise

---

<b>Purpose</b>	Default part of switch statement
<b>Description</b>	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).

<b>Examples</b>	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.

<b>See Also</b>	<a href="#">switch</a>	Switch among several cases based on expression
-----------------	------------------------	--

<b>Purpose</b>	Consolidate workspace memory
<b>Syntax</b>	<code>pack</code> <code>pack <i>filename</i></code>
<b>Description</b>	<code>pack</code> , by itself, frees up needed space by compressing information into the minimum memory required.  <code>pack <i>filename</i></code> accepts an optional <i>filename</i> for the temporary file used to hold the variables. Otherwise it uses the file named <code>pack. tmp</code> .
<b>Remarks</b>	The <code>pack</code> 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 <code>pack</code> command may find you some free memory without forcing you to delete variables.  The <code>pack</code> command frees space by: <ul style="list-style-type: none"><li>• Saving all variables on disk in a temporary file called <code>pack. tmp</code>.</li><li>• Clearing all variables and functions from memory.</li><li>• Reloading the variables back from <code>pack. tmp</code>.</li><li>• Deleting the temporary file <code>pack. tmp</code>.</li></ul> If you use <code>pack</code> 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: <ul style="list-style-type: none"><li>• <b>MS-Windows:</b> Increase the swap space by opening the Control Panel, double-clicking on the 386 Enhanced icon, and pressing the <b>Virtual Memory</b> button.</li><li>• <b>Macintosh:</b> Change the application memory size by using <b>Get Info</b> on the program icon. You may also want to turn on virtual memory via the Memory Control Panel.</li></ul>

# **pack**

---

- **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](#)

<b>Purpose</b>	Partial pathname
<b>Description</b>	<p>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.</p> <p>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/clone.mat are valid partial pathnames. Specifying the @ in method directory names is optional, so funfun/inline/formula is also a valid partial pathname.</p> <p>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.</p>

# pascal

---

<b>Purpose</b>	Pascal matrix																
<b>Syntax</b>	$A = \text{pascal}(n)$ $A = \text{pascal}(n, 1)$ $A = \text{pascal}(n, 2)$																
<b>Description</b>	$A = \text{pascal}(n)$ 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.  $A = \text{pascal}(n, 1)$ returns the lower triangular Cholesky factor (up to the signs of the columns) of the Pascal matrix. It is <i>involuntary</i> , that is, it is its own inverse.  $A = \text{pascal}(n, 2)$ returns a transposed and permuted version of $\text{pascal}(n, 1)$ . $A$ is a cube root of the identity matrix.																
<b>Examples</b>	$\text{pascal}(4)$ returns																
	<table><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>3</td><td>6</td><td>10</td></tr><tr><td>1</td><td>4</td><td>10</td><td>20</td></tr></table>	1	1	1	1	1	2	3	4	1	3	6	10	1	4	10	20
1	1	1	1														
1	2	3	4														
1	3	6	10														
1	4	10	20														
	$A = \text{pascal}(3, 2)$ produces																
	$A =$ <table><tr><td>0</td><td>0</td><td>-1</td></tr><tr><td>0</td><td>-1</td><td>2</td></tr><tr><td>-1</td><td>-1</td><td>1</td></tr></table>	0	0	-1	0	-1	2	-1	-1	1							
0	0	-1															
0	-1	2															
-1	-1	1															
<b>See Also</b>	<a href="#">chol</a> <a href="#">Cholesky factorization</a>																

<b>Purpose</b>	Control MATLAB's directory search path								
<b>Syntax</b>	<pre>path p = path path('newpath') path(path, 'newpath') path('newpath', path)</pre>								
<b>Description</b>	<p><code>path</code> prints out the current setting of MATLAB's search path. On all platforms except the Macintosh, the path resides in <code>pathdef.m</code> (in tool box/local). The Macintosh stores its path in the Matlab Settings File (usually in the Preferences folder).</p> <p><code>p = path</code> returns the current search path in string variable <code>p</code>.</p> <p><code>path('newpath')</code> changes the path to the string '<code>newpath</code>'.</p> <p><code>path(path, 'newpath')</code> appends a new directory to the current path.</p> <p><code>path('newpath', path)</code> prepends a new directory to the current path.</p>								
<b>Remarks</b>	<p>MATLAB has a <i>search path</i>. If you enter a name, such as <code>fox</code>, the MATLAB interpreter:</p> <ol style="list-style-type: none"> <li>1 Looks for <code>fox</code> as a variable.</li> <li>2 Checks for <code>fox</code> as a built-in function.</li> <li>3 Looks in the current directory for <code>fox.mex</code> and <code>fox.m</code>.</li> <li>4 Searches the directories specified by <code>path</code> for <code>fox.mex</code> and <code>fox.m</code>.</li> </ol>								
<b>Examples</b>	<p>Add a new directory to the search path on various operating systems:</p> <table> <tbody> <tr> <td>UNIX:</td> <td><code>path(path, '/home/myfriend/goodstuff')</code></td> </tr> <tr> <td>VMS:</td> <td><code>path(path, 'DISKS1:[MYFRIEND.GOODSTUFF]')</code></td> </tr> <tr> <td>MS-DOS:</td> <td><code>path(path, 'TOOLS\GOODSTUFF')</code></td> </tr> <tr> <td>Macintosh:</td> <td><code>path(path, 'Tools:GoodStuff')</code></td> </tr> </tbody> </table>	UNIX:	<code>path(path, '/home/myfriend/goodstuff')</code>	VMS:	<code>path(path, 'DISKS1:[MYFRIEND.GOODSTUFF]')</code>	MS-DOS:	<code>path(path, 'TOOLS\GOODSTUFF')</code>	Macintosh:	<code>path(path, 'Tools:GoodStuff')</code>
UNIX:	<code>path(path, '/home/myfriend/goodstuff')</code>								
VMS:	<code>path(path, 'DISKS1:[MYFRIEND.GOODSTUFF]')</code>								
MS-DOS:	<code>path(path, 'TOOLS\GOODSTUFF')</code>								
Macintosh:	<code>path(path, 'Tools:GoodStuff')</code>								

# path

---

## See Also

<code>addpath</code>	Add directories to MATLAB's search path
<code>cd</code>	Change working directory
<code>dir</code>	Directory listing
<code>rmpath</code>	Remove directories from MATLAB's search path
<code>what</code>	Directory listing of M-files, MAT-files, and MEX-files

---

<b>Purpose</b>	Halt execution temporarily
<b>Syntax</b>	<code>pause</code> <code>pause(n)</code> <code>pause on</code> <code>pause off</code>
<b>Description</b>	<p><code>pause</code>, by itself, causes M-files to stop and wait for you to press any key before continuing.</p> <p><code>pause(n)</code> pauses execution for <math>n</math> seconds before continuing.</p> <p><code>pause on</code> allows subsequent pause commands to pause execution.</p> <p><code>pause off</code> ensures that any subsequent pause or <code>pause(n)</code> statements do not pause execution. This allows normally interactive scripts to run unattended.</p>
<b>See Also</b>	The <code>drawnow</code> command in the <i>MATLAB Graphics Guide</i> .

**Purpose** Preconditioned Conjugate Gradients method

**Syntax**

```
x = pcg(A, b)
pcg(A, b, tol)
pcg(A, b, tol, maxi t)
pcg(A, b, tol, maxi t, M)
pcg(A, b, tol, maxi t, M1, M2)
pcg(A, b, tol, maxi t, M1, M2, x0)
x = pcg(A, b, tol, maxi t, M1, M2, x0)
[x, flag] = pcg(A, b, tol, maxi t, M1, M2, x0)
[x, flag, rel res] = pcg(A, b, tol, maxi t, M1, M2, x0)
[x, flag, rel res, iter] = pcg(A, b, tol, maxi t, M1, M2, x0)
[x, flag, rel res, iter, resvec] = pcg(A, b, tol, maxi t, 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  $\text{norm}(b - A^*x) / \text{norm}(b)$  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, maxi t)` additionally specifies the maximum number of iterations, `maxi t`.

`pcg(A, b, tol, maxi t, M)` and `pcg(A, b, tol, maxi t, M1, M2)` use left preconditioner  $M$  or  $M=M1*M2$  and effectively solve the system  $\text{inv}(M)^*A^*x = \text{inv}(M)^*b$  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, maxi t, M)` with:

```
R = chol (M);
pcg(A, b, tol, maxi t, R', R).
```

The preconditioner `M` must be symmetric and positive definite.

`pcg(A, b, tol, maxi t, 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, maxi t, 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, maxi t, M1, M2, x0)` returns a solution `x` and a flag which describes the convergence of `pcg`:

Flag	Convergence
0	<code>pcg</code> converged to the desired tolerance <code>tol</code> within <code>maxi t</code> iterations without failing for any reason.
1	<code>pcg</code> iterated <code>maxi t</code> times but did not converge.
2	One of the systems of equations of the form <code>M*y = r</code> 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 <code>pcg</code> 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  $\text{norm}(b - A^*x) / \text{norm}(b)$ . If flag is 0, then relres  $\leq$  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  $0 \leq \text{iter} \leq \text{maxit}$ .

[ $x$ , 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  $\text{resvec}(1) = \text{norm}(b - A^*x_0)$ . If flag is 0, resvec is of length iter+1 and  $\text{resvec}(\text{end}) \leq \text{tol} * \text{norm}(b)$ .

## Examples

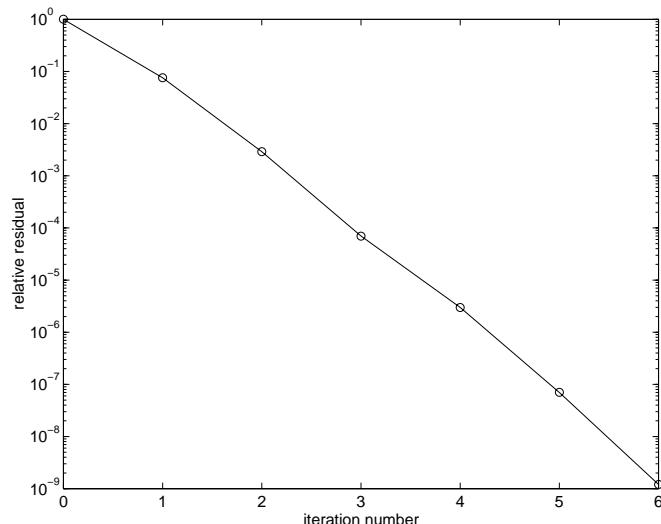
```
A = delsq(numgrid('C', 25))
b = ones(length(A), 1)
[x, flag] = pcg(A, b)
```

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

<code>bicg</code>	BiConjugate Gradients method
<code>bicgstab</code>	BiConjugate Gradients Stabilized method
<code>cgs</code>	Conjugate Gradients Squared method
<code>cholinc</code>	Sparse Incomplete Cholesky and Cholesky-Infinity factorizations
<code>gmres</code>	Generalized Minimum Residual method (with restarts)
<code>qmr</code>	Quasi-Minimal Residual method
<code>\</code>	Matrix left division

## References

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

## pcode

---

<b>Purpose</b>	Create prepared pseudocode file (P-file)
<b>Syntax</b>	<code>pcode <i>fun</i></code> <code>pcode *.m</code> <code>pcode <i>fun1 fun2 ...</i></code> <code>pcode... -i npl ace</code>
<b>Description</b>	<p><code>pcode <i>fun</i></code> parses the M-file <code>fun.m</code> into the P-file <code>fun.p</code> and puts it into the current directory. The original M-file can be anywhere on the search path.</p> <p><code>pcode *.m</code> creates P-files for all the M-files in the current directory.</p> <p><code>pcode <i>fun1 fun2 ...</i></code> creates P-files for the listed functions.</p> <p><code>pcode... -i npl ace</code> creates P-files in the same directory as the M-files. An error occurs if the files can't be created.</p>

---

<b>Purpose</b>	All possible permutations																		
<b>Syntax</b>	<code>P = perms(v)</code>																		
<b>Description</b>	<code>P = perms(v)</code> , where <code>v</code> is a row vector of length <code>n</code> , creates a matrix whose rows consist of all possible permutations of the <code>n</code> elements of <code>v</code> . Matrix <code>P</code> contains $n!$ rows and <code>n</code> columns.																		
<b>Examples</b>	The command <code>perms(2:2:6)</code> returns <i>all</i> the permutations of the numbers 2, 4, and 6:																		
	<table><tr><td>6</td><td>4</td><td>2</td></tr><tr><td>4</td><td>6</td><td>2</td></tr><tr><td>6</td><td>2</td><td>4</td></tr><tr><td>2</td><td>6</td><td>4</td></tr><tr><td>4</td><td>2</td><td>6</td></tr><tr><td>2</td><td>4</td><td>6</td></tr></table>	6	4	2	4	6	2	6	2	4	2	6	4	4	2	6	2	4	6
6	4	2																	
4	6	2																	
6	2	4																	
2	6	4																	
4	2	6																	
2	4	6																	
<b>Limitations</b>	This function is only practical for situations where <code>n</code> is less than about 15.																		
<b>See Also</b>	<table><tr><td><code>nchoosek</code></td><td>Binomial coefficient or all combinations</td></tr><tr><td><code>permute</code></td><td>Rearrange the dimensions of a multidimensional array</td></tr><tr><td><code>randperm</code></td><td>Random permutation</td></tr></table>	<code>nchoosek</code>	Binomial coefficient or all combinations	<code>permute</code>	Rearrange the dimensions of a multidimensional array	<code>randperm</code>	Random permutation												
<code>nchoosek</code>	Binomial coefficient or all combinations																		
<code>permute</code>	Rearrange the dimensions of a multidimensional array																		
<code>randperm</code>	Random permutation																		

# permute

---

<b>Purpose</b>	Rearrange the dimensions of a multidimensional array
<b>Syntax</b>	<code>B = permute(A, order)</code>
<b>Description</b>	<code>B = permute(A, order)</code> rearranges the dimensions of A so that they are in the order specified by the vector <i>order</i> . B has the same values of A but the order of the subscripts needed to access any particular element is rearranged as specified by <i>order</i> . All the elements of <i>order</i> must be unique.
<b>Remarks</b>	<code>permute</code> and <code>i permute</code> are a generalization of transpose ( <code>'</code> ) for multidimensional arrays.
<b>Examples</b>	Given any matrix A, the statement <code>permute(A, [2 1])</code> is the same as <code>A'</code> . For example: <pre>A = [1 2; 3 4]; permute(A, [2 1]) ans =     1     3     2     4</pre> The following code permutes a three-dimensional array: <pre>X = rand(12, 13, 14); Y = permute(X, [2 3 1]); size(Y) ans =     13     14     12</pre>
<b>See Also</b>	<code>i permute</code> Inverse permute the dimensions of a multidimensional array

---

<b>Purpose</b>	Define persistent variable
<b>Syntax</b>	<code>persistent X Y Z</code>
<b>Description</b>	<p><code>persistent X Y Z</code> defines X, Y, and Z as persistent in scope, so that X, Y, and Z maintain their values from one call to the next. <code>persistent</code> can be used within a function only.</p> <p>Persistent variables are cleared when the M-file is cleared from memory or when the M-file is changed. To keep an M-file in memory until MATLAB quits, use <code>unlock</code>. If the persistent variable does not exist the first time you issue the <code>persistent</code> statement, it is initialized to the empty matrix.</p> <p>It is an error to declare a variable <code>persistent</code> if a variable with the same name exists in the current workspace.</p> <p>By convention, persistent variable names are often long with all capital letters (not required).</p>
<b>See Also</b>	<code>clear</code> , <code>global</code> , <code>mislocked</code> , <code>unlock</code>

# pi

---

<b>Purpose</b>	Ratio of a circle's circumference to its diameter, $\pi$												
<b>Syntax</b>	<code>pi</code>												
<b>Description</b>	<code>pi</code> returns the floating-point number nearest the value of $\pi$ . The expressions <code>4*atan(1)</code> and <code>imag(log(-1))</code> provide the same value.												
<b>Examples</b>	The expression <code>sin(pi)</code> is not exactly zero because <code>pi</code> is not exactly $\pi$ :  <code>sin(pi)</code> ans = 1. 2246e-16												
<b>See Also</b>	<table><tr><td><code>ans</code></td><td>The most recent answer</td></tr><tr><td><code>eps</code></td><td>Floating-point relative accuracy</td></tr><tr><td><code>i</code></td><td>Imaginary unit</td></tr><tr><td><code>Inf</code></td><td>Infinity</td></tr><tr><td><code>j</code></td><td>Imaginary unit</td></tr><tr><td><code>NaN</code></td><td>Not-a-Number</td></tr></table>	<code>ans</code>	The most recent answer	<code>eps</code>	Floating-point relative accuracy	<code>i</code>	Imaginary unit	<code>Inf</code>	Infinity	<code>j</code>	Imaginary unit	<code>NaN</code>	Not-a-Number
<code>ans</code>	The most recent answer												
<code>eps</code>	Floating-point relative accuracy												
<code>i</code>	Imaginary unit												
<code>Inf</code>	Infinity												
<code>j</code>	Imaginary unit												
<code>NaN</code>	Not-a-Number												

<b>Purpose</b>	Moore-Penrose pseudoinverse of a matrix
<b>Syntax</b>	$B = \text{pinv}(A)$ $B = \text{pinv}(A, \text{tol})$
<b>Definition</b>	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 $\text{svd}(A)$ and any singular values less than $\text{tol}$ are treated as zero.
<b>Description</b>	$B = \text{pinv}(A)$ returns the Moore-Penrose pseudoinverse of $A$ .  $B = \text{pinv}(A, \text{tol})$ returns the Moore-Penrose pseudoinverse and overrides the default tolerance, $\max(\text{size}(A)) * \text{norm}(A) * \text{eps}$ .
<b>Examples</b>	<p>If <math>A</math> is square and not singular, then <math>\text{pinv}(A)</math> is an expensive way to compute <math>\text{inv}(A)</math>. If <math>A</math> is not square, or is square and singular, then <math>\text{inv}(A)</math> does not exist. In these cases, <math>\text{pinv}(A)</math> has some of, but not all, the properties of <math>\text{inv}(A)</math>.</p> <p>If <math>A</math> has more rows than columns and is not of full rank, then the overdetermined least squares problem</p> $\text{minimize } \text{norm}(A*x - b)$ <p>does not have a unique solution. Two of the infinitely many solutions are</p> $x = \text{pinv}(A)*b$ <p>and</p> $y = A\b$ <p>These two are distinguished by the facts that <math>\text{norm}(x)</math> is smaller than the norm of any other solution and that <math>y</math> has the fewest possible nonzero components.</p>

## pinv

---

For example, the matrix generated by

```
A = magic(8); A = A(:, 1:6)
```

is an 8-by-6 matrix that happens to have  $\text{rank}(A) = 3$ .

$A =$

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  $b = 260 * \text{ones}(8, 1)$ ,

$b =$

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

$$\begin{aligned}x = \\1.1538 \\1.4615 \\1.3846 \\1.3846 \\1.4615 \\1.1538\end{aligned}$$

and

$$y = A \setminus b$$

which is

$$\begin{aligned}y = \\3.0000 \\4.0000 \\0 \\0 \\1.0000 \\0\end{aligned}$$

Both of these are exact solutions in the sense that  $\text{norm}(A*x - b)$  and  $\text{norm}(A*y - b)$  are on the order of roundoff error. The solution  $x$  is special because

$$\text{norm}(x) = 3.2817$$

is smaller than the norm of any other solution, including

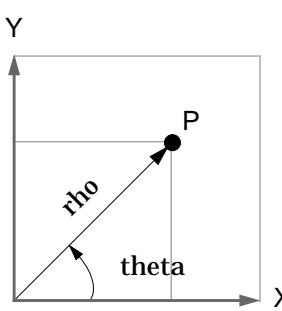
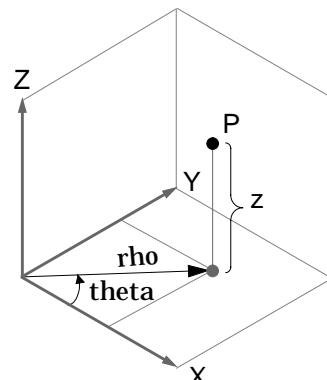
$$\text{norm}(y) = 5.0990$$

On the other hand, the solution  $y$  is special because it has only three nonzero components.

## See Also

<code>i nv</code>	Matrix inverse
<code>qr</code>	Orthogonal-triangular decomposition
<code>rank</code>	Rank of a matrix
<code>svd</code>	Singular value decomposition

# pol2cart

<b>Purpose</b>	Transform polar or cylindrical coordinates to Cartesian	
<b>Syntax</b>	$[X, Y] = \text{pol2cart}(\text{THETA}, \text{RHO})$ $[X, Y, Z] = \text{pol2cart}(\text{THETA}, \text{RHO}, Z)$	
<b>Description</b>	$[X, Y] = \text{pol2cart}(\text{THETA}, \text{RHO})$ transforms the polar coordinate data stored in corresponding elements of THETA and RHO to two-dimensional Cartesian, or <i>xy</i> , coordinates. The arrays THETA and RHO must be the same size (or either can be scalar). The values in THETA must be in radians.  $[X, Y, Z] = \text{pol2cart}(\text{THETA}, \text{RHO}, Z)$ transforms the cylindrical coordinate data stored in corresponding elements of THETA, RHO, and Z to three-dimensional Cartesian, or <i>xyz</i> , 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.	
<b>Algorithm</b>	The mapping from polar and cylindrical coordinates to Cartesian coordinates is:	
	 <p>Polar to Cartesian Mapping</p> <pre>theta = atan2(y, x) rho = sqrt(x.^2 + y.^2)</pre>  <p>Cylindrical to Cartesian Mapping</p> <pre>theta = atan2(y, x) rho = sqrt(x.^2 + y.^2) z = z</pre>	
<b>See Also</b>	<a href="#">cart2pol</a> <a href="#">cart2sph</a>	Transform Cartesian coordinates to polar or cylindrical Transform Cartesian coordinates to spherical

sph2cart

Transform spherical coordinates to Cartesian

# poly

---

<b>Purpose</b>	Polynomial with specified roots
<b>Syntax</b>	$p = \text{poly}(A)$ $p = \text{poly}(r)$
<b>Description</b>	$p = \text{poly}(A)$ 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 $c_1 s^n + \dots + c_n s + c_{n+1}$ . $p = \text{poly}(r)$ where $r$ is a vector returns a row vector whose elements are the coefficients of the polynomial whose roots are the elements of $r$ .
<b>Remarks</b>	Note the relationship of this command to $r = \text{roots}(p)$ which returns a column vector whose elements are the roots of the polynomial specified by the coefficients row vector $p$ . For vectors, $\text{roots}$ and $\text{poly}$ are inverse functions of each other, up to ordering, scaling, and roundoff error.
<b>Examples</b>	MATLAB displays polynomials as row vectors containing the coefficients ordered by descending powers. The characteristic equation of the matrix $A = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{matrix}$ is returned in a row vector by $\text{poly}$ : $p = \text{poly}(A)$ $p = \begin{matrix} 1 & -6 & -72 & -27 \end{matrix}$

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

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

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.

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

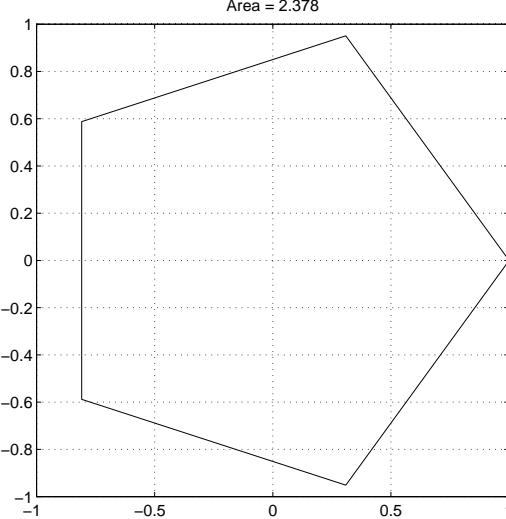
It is possible to prove that poly(A) produces the coefficients in the characteristic polynomial of a matrix within roundoff error of A. This is true even if the eigenvalues of A are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial, which do not use the eigenvalues,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

# poly

---

## See Also

conv	Convolution and polynomial multiplication
polyval	Polynomial evaluation
residue	Convert between partial fraction expansion and polynomial coefficients
roots	Polynomial roots

<b>Purpose</b>	Area of polygon
<b>Syntax</b>	$A = \text{polyarea}(X, Y)$ $A = \text{polyarea}(X, Y, \text{dim})$
<b>Description</b>	<p><math>A = \text{polyarea}(X, Y)</math> returns the area of the polygon specified by the vertices in the vectors X and Y.</p> <p>If X and Y are matrices of the same size, then polyarea returns the area of polygons defined by the columns X and Y.</p> <p>If X and Y are multidimensional arrays, polyarea returns the area of the polygons in the first nonsingleton dimension of X and Y.</p> <p><math>A = \text{polyarea}(X, Y, \text{dim})</math> operates along the dimension specified by scalar dim.</p>
<b>Examples</b>	<pre>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</pre> 
<b>See Also</b>	<a href="#">convhull</a> <b>Convex hull</b> <a href="#">inpolygon</a> Detect points inside a polygonal region

# polyder

---

<b>Purpose</b>	Polynomial derivative
<b>Syntax</b>	$k = \text{polyder}(p)$ $k = \text{polyder}(a, b)$ $[q, d] = \text{polyder}(b, a)$
<b>Description</b>	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.  $k = \text{polyder}(p)$ returns the derivative of the polynomial p.  $k = \text{polyder}(a, b)$ returns the derivative of the product of the polynomials a and b.  $[q, d] = \text{polyder}(b, a)$ returns the numerator q and denominator d of the derivative of the polynomial quotient b/a.
<b>Examples</b>	The derivative of the product  $(3x^2 + 6x + 9)(x^2 + 2x)$ is obtained with  $a = [3 6 9];$ $b = [1 2 0];$ $k = \text{polyder}(a, b)$ $k =$ $12 \quad 36 \quad 42 \quad 18$  This result represents the polynomial  $12x^3 + 36x^2 + 42x + 18$
<b>See Also</b>	<a href="#">conv</a> Convolution and polynomial multiplication <a href="#">deconv</a> Deconvolution and polynomial division

<b>Purpose</b>	Polynomial eigenvalue problem				
<b>Syntax</b>	$[X, e] = \text{polyeig}(A_0, A_1, \dots, A_p)$				
<b>Description</b>	$[X, e] = \text{polyeig}(A_0, A_1, \dots, A_p)$ solves the polynomial eigenvalue problem of degree p: $(A_0 + \lambda A_1 + \dots + \lambda^P A_p)x = 0$ <p>where polynomial degree p is a non-negative integer, and <math>A_0, A_1, \dots, A_p</math> 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.</p>				
<b>Remarks</b>	Based on the values of p and n, <code>polyeig</code> handles several special cases:				
	<ul style="list-style-type: none"> <li>• p = 0, or <math>\text{polyeig}(A)</math> is the standard eigenvalue problem: <math>\text{eig}(A)</math>.</li> <li>• p = 1, or <math>\text{polyeig}(A, B)</math> is the generalized eigenvalue problem: <math>\text{eig}(A, -B)</math>.</li> <li>• n = 1, or <math>\text{polyeig}(a_0, a_1, \dots, a_p)</math> for scalars <math>a_0, a_1, \dots, a_p</math> is the standard polynomial problem: <math>\text{roots}([a_p \dots a_1 a_0])</math>.</li> </ul>				
<b>Algorithm</b>	If both $A_0$ and $A_p$ 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. <code>polyeig</code> attempts to detect this situation and display an appropriate warning message. If either one, but not both, of $A_0$ and $A_p$ is singular, the problem is well posed but some of the eigenvalues may be zero or infinite (Inf).  The <code>polyeig</code> 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 <code>eig</code> and <code>qz</code> for more on this, as well as the <i>EISPACK Guide</i> .				
<b>See Also</b>	<table border="0"> <tr> <td><code>eig</code></td> <td>Eigenvalues and eigenvectors</td> </tr> <tr> <td><code>qz</code></td> <td>QZ factorization for generalized eigenvalues</td> </tr> </table>	<code>eig</code>	Eigenvalues and eigenvectors	<code>qz</code>	QZ factorization for generalized eigenvalues
<code>eig</code>	Eigenvalues and eigenvectors				
<code>qz</code>	QZ factorization for generalized eigenvalues				

# polyfit

---

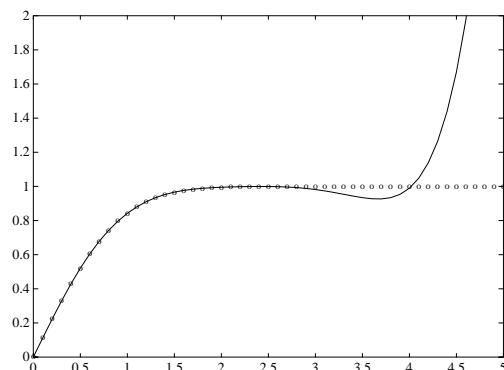
<b>Purpose</b>	Polynomial curve fitting
<b>Syntax</b>	<pre>p = polyfit(x, y, n) [p, s] = polyfit(x, y, n)</pre>
<b>Description</b>	<p><code>p = polyfit(x, y, n)</code> finds the coefficients of a polynomial <math>p(x)</math> of degree <math>n</math> that fits the data, <math>p(x(i))</math> to <math>y(i)</math>, in a least squares sense. The result <math>p</math> is a row vector of length <math>n+1</math> containing the polynomial coefficients in descending powers:</p> $p(x) = p_1x^n + p_2x^{n-1} + \dots + p_nx + p_{n+1}$ <p><code>[p, s] = polyfit(x, y, n)</code> returns the polynomial coefficients <math>p</math> and a structure <math>S</math> for use with <code>polyval</code> to obtain error estimates or predictions. If the errors in the data <math>Y</math> are independent normal with constant variance; <code>polyval</code> will produce error bounds that contain at least 50% of the predictions.</p>
<b>Examples</b>	<p>This example involves fitting the error function, <math>\text{erf}(x)</math>, by a polynomial in <math>x</math>. This is a risky project because <math>\text{erf}(x)</math> is a bounded function, while polynomials are unbounded, so the fit might not be very good.</p> <p>First generate a vector of <math>x</math>-points, equally spaced in the interval <math>[0, 2.5]</math>; then evaluate <math>\text{erf}(x)</math> at those points.</p> <pre>x = (0: 0.1: 2.5)'; y = erf(x);</pre> <p>The coefficients in the approximating polynomial of degree 6 are</p> <pre>p = polyfit(x, y, 6) p =     0.0084   -0.0983    0.4217   -0.7435   0.1471    1.1064   0.0004</pre> <p>There are seven coefficients and the polynomial is</p> $0.0084x^6 - 0.0983x^5 + 0.4217x^4 - 0.7435x^3 + 0.1471x^2 + 1.1064x + 0.0004$ <p>To see how good the fit is, evaluate the polynomial at the data points with</p> <pre>f = polyval(p, x);</pre>

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])
```



# polyfit

---

## Algorithm

The M-file forms the Vandermonde matrix,  $V$ , whose elements are powers of  $x$ .

$$v_{i,j} = x_i^{n-j}$$

It then uses the backslash operator,  $\backslash$ , to solve the least squares problem

$$V_p \equiv y$$

The M-file can be modified to use other functions of  $x$  as the basis functions.

## See Also

[polyval](#)  
[roots](#)

[Polynomial evaluation](#)  
[Polynomial roots](#)

---

<b>Purpose</b>	Polynomial evaluation
<b>Syntax</b>	$y = \text{polyval}(p, x)$ $[y, \text{delta}] = \text{polyval}(p, x, S)$
<b>Description</b>	$y = \text{polyval}(p, x)$ 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, $\text{polyval}$ evaluates $p$ at each element of $x$ . $[y, \text{delta}] = \text{polyval}(p, x, S)$ uses the optional output structure $S$ generated by $\text{polyfit}$ to generate error estimates, $y \pm \text{delta}$ . If the errors in the data input to $\text{polyfit}$ are independent normal with constant variance, $y \pm \text{delta}$ contains at least 50% of the predictions.
<b>Remarks</b>	The $\text{polyvalm}(p, x)$ function, with $x$ a matrix, evaluates the polynomial in a matrix sense. See $\text{polyvalm}$ for more information.
<b>Examples</b>	The polynomial $p(x) = 3x^2 + 2x + 1$ is evaluated at $x = 5, 7$ , and $9$ with $\begin{aligned} p &= [3 2 1]; \\ \text{polyval}(p, [5 7 9]) \end{aligned}$ which results in $\begin{aligned} \text{ans} &= \\ 86 &\quad 162 & 262 \end{aligned}$ For another example, see $\text{polyfit}$ .
<b>See Also</b>	<a href="#">polyfit</a> <b>Polynomial curve fitting</b> <a href="#">polyvalm</a> <b>Matrix polynomial evaluation</b>

# polyvalm

---

<b>Purpose</b>	Matrix polynomial evaluation
<b>Syntax</b>	$Y = \text{polyvalm}(p, X)$
<b>Description</b>	$Y = \text{polyvalm}(p, X)$ 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.
<b>Examples</b>	The Pascal matrices are formed from Pascal's triangle of binomial coefficients. Here is the Pascal matrix of order 4.  $X = \text{pascal}(4)$ $X = \begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 6 & 10 \\ 1 & 4 & 10 & 20 \end{matrix}$ Its characteristic polynomial can be generated with the <code>poly</code> function.  $p = \text{poly}(X)$ $p = \begin{matrix} 1 & -29 & 72 & -29 & 1 \end{matrix}$ 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](#)

# pow2

---

<b>Purpose</b>	Base 2 power and scale floating-point numbers																					
<b>Syntax</b>	$X = \text{pow2}(Y)$ $X = \text{pow2}(F, E)$																					
<b>Description</b>	$X = \text{pow2}(Y)$ returns an array X whose elements are 2 raised to the power Y. $X = \text{pow2}(F, E)$ 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.																					
<b>Remarks</b>	This function corresponds to the ANSI C function <code>l dexp()</code> and the IEEE floating-point standard function <code>scal bn()</code> .																					
<b>Examples</b>	For IEEE arithmetic, the statement $X = \text{pow2}(F, E)$ yields the values:																					
	<table><thead><tr><th>F</th><th>E</th><th>X</th></tr></thead><tbody><tr><td>1/2</td><td>1</td><td>1</td></tr><tr><td><math>\pi / 4</math></td><td>2</td><td><math>\pi</math></td></tr><tr><td>-3/4</td><td>2</td><td>-3</td></tr><tr><td>1/2</td><td>-51</td><td><math>\text{eps}</math></td></tr><tr><td><math>1 - \text{eps} / 2</math></td><td>1024</td><td><math>\text{real max}</math></td></tr><tr><td>1/2</td><td>-1021</td><td><math>\text{real min}</math></td></tr></tbody></table>	F	E	X	1/2	1	1	$\pi / 4$	2	$\pi$	-3/4	2	-3	1/2	-51	$\text{eps}$	$1 - \text{eps} / 2$	1024	$\text{real max}$	1/2	-1021	$\text{real min}$
F	E	X																				
1/2	1	1																				
$\pi / 4$	2	$\pi$																				
-3/4	2	-3																				
1/2	-51	$\text{eps}$																				
$1 - \text{eps} / 2$	1024	$\text{real max}$																				
1/2	-1021	$\text{real min}$																				
<b>See Also</b>	<table><tbody><tr><td><code>log2</code></td><td>Base 2 logarithm and dissect floating-point numbers into exponent and mantissa</td></tr><tr><td><code>^</code></td><td>Matrix power</td></tr><tr><td><code>.^</code></td><td>Array power</td></tr><tr><td><code>exp</code></td><td>Exponential</td></tr><tr><td><code>hex2num</code></td><td>Hexadecimal to double number conversion</td></tr><tr><td><code>real max</code></td><td>Largest positive floating-point number</td></tr><tr><td><code>real min</code></td><td>Smallest positive floating-point number</td></tr></tbody></table>	<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa	<code>^</code>	Matrix power	<code>.^</code>	Array power	<code>exp</code>	Exponential	<code>hex2num</code>	Hexadecimal to double number conversion	<code>real max</code>	Largest positive floating-point number	<code>real min</code>	Smallest positive floating-point number							
<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa																					
<code>^</code>	Matrix power																					
<code>.^</code>	Array power																					
<code>exp</code>	Exponential																					
<code>hex2num</code>	Hexadecimal to double number conversion																					
<code>real max</code>	Largest positive floating-point number																					
<code>real min</code>	Smallest positive floating-point number																					

---

<b>Purpose</b>	Generate list of prime numbers
<b>Syntax</b>	<code>p = primes(n)</code>
<b>Description</b>	<code>p = primes(n)</code> 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.
<b>Examples</b>	<code>p = primes(37)</code>  <code>p =</code>  2        3        5        7        11        13        17        19        23        29        31        37
<b>See Also</b>	<a href="#">factor</a> Prime factors

# prod

---

<b>Purpose</b>	Product of array elements
<b>Syntax</b>	$B = \text{prod}(A)$ $B = \text{prod}(A, dim)$
<b>Description</b>	$B = \text{prod}(A)$ returns the products along different dimensions of an array. If A is a vector, $\text{prod}(A)$ returns the product of the elements. If A is a matrix, $\text{prod}(A)$ treats the columns of A as vectors, returning a row vector of the products of each column. If A is a multidimensional array, $\text{prod}(A)$ treats the values along the first non-singleton dimension as vectors, returning an array of row vectors. $B = \text{prod}(A, dim)$ takes the products along the dimension of A specified by scalar $dim$ .
<b>Examples</b>	The magic square of order 3 is <pre>M = magic(3) M =     8     1     6     3     5     7     4     9     2</pre> The product of the elements in each column is <pre>prod(M) =     96     45     84</pre> The product of the elements in each row can be obtained by: <pre>prod(M, 2) =     48     105     72</pre>
<b>See Also</b>	<a href="#">cumprod</a> Cumulative product <a href="#">diff</a> Difference <a href="#">sum</a> Sum of array elements

<b>Purpose</b>	Measure and display M-file execution profiles
<b>Syntax</b>	<code>profile function</code> <code>profile report</code> <code>profile report n</code> <code>profile report frac</code> <code>profile on</code> <code>profile off</code> <code>profile done</code> <code>profile reset</code> <code>info = profile</code>
<b>Description</b>	<p>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.</p> <p><code>profile function</code> starts the profiler for <i>function</i>. <i>function</i> must be the name of an M-file function or a MATLABPATH relative partial pathname.</p> <p><code>profile report</code> displays a profile summary report for the M-file currently being profiled.</p> <p><code>profile report n</code>, where <i>n</i> is an integer, displays a report showing the <i>n</i> lines that take the most time.</p> <p><code>profile report frac</code>, where <i>frac</i> is a number between 0.0 and 1.0, displays a report of each line that accounts for more than <i>frac</i> of the total time.</p> <p><code>profile on</code> and <code>profile off</code> enable and disable profiling, respectively.</p> <p><code>profile done</code> turns off the profiler and clears its data.</p> <p><code>profile reset</code> erases the bin contents without disabling profiling or changing the M-file under inspection.</p>

# profile

---

`info = profile` returns a structure with the fields:

<code>file</code>	Full path to the function being profiled.
<code>function</code>	Name of function being profiled.
<code>interval</code>	Sampling interval in seconds.
<code>count</code>	Vector of sample counts
<code>state</code>	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`.

<b>Purpose</b>	Quasi-Minimal Residual method
<b>Syntax</b>	<pre>x = qmr(A, b) qmr(A, b, tol) qmr(A, b, tol, maxi t) qmr(A, b, tol, maxi t, M1) qmr(A, b, tol, maxi t, M1, M2) qmr(A, b, tol, maxi t, M1, M2, x0) x = qmr(A, b, tol, maxi t, M1, M2, x0) [x, flag] = qmr(A, b, tol, maxi t, M1, M2, x0) [x, flag, rel res] = qmr(A, b, tol, maxi t, M1, M2, x0) [x, flag, rel res, iter] = qmr(A, b, tol, maxi t, M1, M2, x0) [x, flag, rel res, iter, resvec] = qmr(A, b, tol, maxi t, M1, M2, x0)</pre>
<b>Description</b>	<p><code>x = qmr(A, b)</code> attempts to solve the system of linear equations <math>A^*x=b</math> for <math>x</math>. The coefficient matrix <math>A</math> must be square and the right hand side (column) vector <math>b</math> must have length <math>n</math>, where <math>A</math> is <math>n</math>-by-<math>n</math>. <code>qmr</code> will start iterating from an initial estimate that by default is an all zero vector of length <math>n</math>. Iterates are produced until the method either converges, fails, or has computed the maximum number of iterations. Convergence is achieved when an iterate <math>x</math> has relative residual <math>\text{norm}(b-A^*x)/\text{norm}(b)</math> less than or equal to the tolerance of the method. The default tolerance is <math>1e-6</math>. The default maximum number of iterations is the minimum of <math>n</math> and 20. No preconditioning is used.</p> <p><code>qmr(A, b, tol)</code> specifies the tolerance of the method, <code>tol</code>.</p> <p><code>qmr(A, b, tol, maxi t)</code> additionally specifies the maximum number of iterations, <code>maxi t</code>.</p> <p><code>qmr(A, b, tol, maxi t, M1)</code> and <code>qmr(A, b, tol, maxi t, M1, M2)</code> use left and right preconditioners <math>M1</math> and <math>M2</math> and effectively solve the system <math>\text{inv}(M1)^*A^*\text{inv}(M2)^*y = \text{inv}(M1)^*b</math> for <math>y</math>, where <math>x = \text{inv}(M2)^*y</math>. If <math>M1</math> or <math>M2</math> 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 <math>M1^*y = r</math> are solved using backslash within <code>qmr</code>, it is wise to factor</p>

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	<code>qmr</code> converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations without failing for any reason.
1	<code>qmr</code> iterated <code>maxit</code> 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 <code>qmr</code> 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  $\text{norm}(b - A^*x) / \text{norm}(b)$ . If `flag` is 0, then  $\text{relres} \leq \text{tol}$ .

`[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  $0 \leq \text{iter} \leq \text{maxit}$ .

`[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) \leq 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.

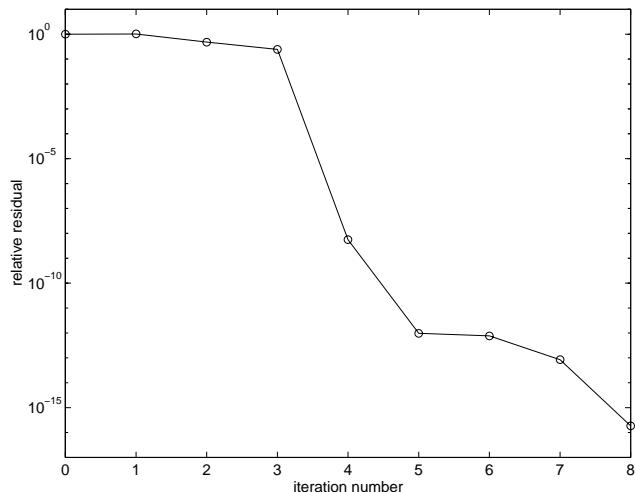
```
[L1, U1] = luinc(A, 1e-5)
[x1, flag1] = qmr(A, b, 1e-6, 20, L1, U1)
```

`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

`bi cg` BiConjugate Gradients method  
`bi cgst ab` BiConjugate Gradients Stabilized method  
`cgs` Conjugate Gradients Squared method  
`gmres` Generalized Minimum Residual method (with restarts)  
`l ui nc` 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.

<b>Purpose</b>	Orthogonal-triangular decomposition
<b>Syntax</b>	$[Q, R] = qr(X)$ $[Q, R, E] = qr(X)$ $[Q, R] = qr(X, 0)$ $[Q, R, E] = qr(X, 0)$ $A = qr(X)$
<b>Description</b>	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 $\text{abs}(\text{diag}(R))$ 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 $\text{abs}(\text{diag}(R))$ is decreasing.
	$A = qr(X)$ returns the output of the LINPACK subroutine ZQRDC. triu( $qr(X)$ ) is $R$ .
<b>Examples</b>	Start with
	$A = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{matrix}$

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:

$$\begin{aligned} [Q, R] &= qr(A) \\ Q = \\ \begin{matrix} -0.0776 & -0.8331 & 0.5444 & 0.0605 \\ -0.3105 & -0.4512 & -0.7709 & 0.3251 \\ -0.5433 & -0.0694 & -0.0913 & -0.8317 \\ -0.7762 & 0.3124 & 0.3178 & 0.4461 \end{matrix} \\ R = \\ \begin{matrix} -12.8841 & -14.5916 & -16.2992 \\ 0 & -1.0413 & -2.0826 \\ 0 & 0 & 0.0000 \\ 0 & 0 & 0 \end{matrix} \end{aligned}$$

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

$$\begin{aligned} b = \\ \begin{matrix} 1 \\ 3 \\ 5 \\ 7 \end{matrix} \end{aligned}$$

The linear system  $Ax = b$  represents four equations in only three unknowns. The best solution in a least squares sense is computed by

$$x = A \setminus b$$

which produces

Warning: Rank deficient, rank = 2, tol = 1.4594E-014

$$\begin{aligned} x = \\ \begin{matrix} 0.5000 \\ 0 \\ 0.1667 \end{matrix} \end{aligned}$$

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
nul	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.

# qrdelete

---

<b>Purpose</b>	Delete column from QR factorization				
<b>Syntax</b>	$[Q, R] = \text{qrdelete}(Q, R, j)$				
<b>Description</b>	$[Q, R] = \text{qrdelete}(Q, R, j)$ 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 $[Q, R] = \text{qr}(A)$ . Argument j specifies the column to be removed from matrix A.				
<b>Algorithm</b>	The qrdelete function uses a series of Givens rotations to zero out the appropriate elements of the factorization.				
<b>See Also</b>	<table><tr><td><code>qr</code></td><td>Orthogonal-triangular decomposition</td></tr><tr><td><code>qrinsert</code></td><td>Insert column in QR factorization</td></tr></table>	<code>qr</code>	Orthogonal-triangular decomposition	<code>qrinsert</code>	Insert column in QR factorization
<code>qr</code>	Orthogonal-triangular decomposition				
<code>qrinsert</code>	Insert column in QR factorization				

<b>Purpose</b>	Insert column in QR factorization
<b>Syntax</b>	$[Q, R] = \text{qrinsert}(Q, R, j, x)$
<b>Description</b>	$[Q, R] = \text{qrinsert}(Q, R, j, x)$ 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 $[Q, R] = \text{qr}(A)$ . Argument x is the column vector to be inserted into matrix A. Argument j specifies the column before which x is inserted.
<b>Algorithm</b>	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.
<b>See Also</b>	<a href="#">qr</a> Orthogonal-triangular decomposition <a href="#">qrdelete</a> Delete column from QR factorization

# qrupdate

---

<b>Description</b>	Rank 1 update to QR factorization
<b>Syntax</b>	$[Q1, R1] = \text{qrupdate}(Q, R, u, v)$
<b>Description</b>	$[Q1, R1] = \text{qrupdate}(Q, R, u, v)$ when $[Q, R] = \text{qr}(A)$ is the original QR factorization of A, returns the QR factorization of $A + u^*v'$ , where u and v are column vectors of appropriate lengths.
<b>Remarks</b>	<code>qrupdate</code> works only for full matrices.
<b>Examples</b>	The matrix

```
mu = sqrt(eps)
mu =
    1.4901e-08
A = [ones(1, 4); mu*eye(4)];
```

is a well-known example in least squares that indicates the dangers of forming  $A^* * A$ . Instead, we work with the QR factorization – orthonormal Q and upper triangular R.

```
[Q, R] = qr(A);
```

As we expect, R is upper triangular.

```
R =
-1.0000   -1.0000   -1.0000   -1.0000
  0         0.0000    0.0000    0.0000
  0         0         0.0000    0.0000
  0         0         0         0.0000
  0         0         0         0
```

In this case, the upper triangular entries of R, excluding the first row, are on the order of `sqrt(eps)`.

Consider the update vectors

```
u = [-1 0 0 0 0]'; v = ones(4, 1);
```

Instead of computing the rather trivial QR factorization of this rank one update to A from scratch with

$$[QT, RT] = qr(A + u^*v')$$

$QT =$

$$\begin{matrix} 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \end{matrix}$$

$RT =$

$$\begin{matrix} 1.0e-07 * \\ -0.1490 & 0 & 0 & 0 \\ 0 & -0.1490 & 0 & 0 \\ 0 & 0 & -0.1490 & 0 \\ 0 & 0 & 0 & -0.1490 \\ 0 & 0 & 0 & 0 \end{matrix}$$

we may use qrupdate.

$$[Q1, R1] = qrupdate(Q, R, u, v)$$

$Q1 =$

$$\begin{matrix} -0.0000 & -0.0000 & -0.0000 & -0.0000 & 1.0000 \\ 1.0000 & -0.0000 & -0.0000 & -0.0000 & 0.0000 \\ -0.0000 & 1.0000 & -0.0000 & -0.0000 & 0.0000 \\ -0.0000 & -0.0000 & 1.0000 & -0.0000 & 0.0000 \\ 0 & 0 & 0 & 1.0000 & 0.0000 \end{matrix}$$

$R1 =$

$$\begin{matrix} 1.0e-07 * \\ 0.1490 & 0.0000 & 0.0000 & 0.0000 \\ 0 & 0.1490 & -0.0000 & -0.0000 \\ 0 & 0 & 0.1490 & -0.0000 \\ 0 & 0 & 0 & 0.1490 \\ 0 & 0 & 0 & 0 \end{matrix}$$

Note that both factorizations are correct, even though they are different.

## Algorithm

qrupdate uses the algorithm in section 12.5.1 of the third edition of *Matrix Computations* by Golub and van Loan. qrupdate is useful since, if we take

# qrupdate

---

$N = \max(m, n)$ , then computing the new QR factorization from scratch is roughly an  $O(N^3)$  algorithm, while simply updating the existing factors in this way is an  $O(N^2)$  algorithm.

## References

Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

## See Also

`chol update`  
`qr`

Rank 1 update to Cholesky factorization  
Orthogonal-triangular decomposition

---

<b>Purpose</b>	Write QuickTime movie file to disk
<b>Syntax</b>	<code>qtwrite(d, size, map, 'filename')</code> <code>qtwrite(mov, map, 'filename')</code> <code>qtwrite(..., options)</code>
<b>Description</b>	<p><code>qtwrite(d, size, map, 'filename')</code> writes the indexed image deck <code>d</code> with size <code>size</code> and colormap <code>map</code> to the QuickTime movie file '<code>filename</code>'. If '<code>filename</code>' already exists, it will be replaced.</p> <p><code>qtwrite(mov, map, 'filename')</code> writes the MATLAB movie matrix <code>mov</code> with colormap <code>map</code> to the QuickTime movie file '<code>filename</code>'.</p> <p><code>qtwrite(..., options)</code> can be used to set the frame rate, spacial quality, and compressor type:</p> <p style="padding-left: 2em;">options(1) : frame rate (frames per second) (10 fps default)</p> <p style="padding-left: 2em;">options(2) : compressor type:</p> <p style="padding-left: 2em;">1 - video (default), 2 - jpeg, 3 - animation</p> <p style="padding-left: 2em;">options(3) : spacial quality:</p> <p style="padding-left: 2em;">1 - minimum, 2 - low, 3 - normal (default), 4 - high, 5 - maximum, 6 - lossless</p> <p><code>qtwrite</code> requires QuickTime and works only on the Macintosh.</p>

# quad, quad8

---

**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  $1. \text{e}{-}3$ . 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  $\pi$ :

```
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  $q = \text{inf}$ .

**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.

# **quit**

---

<b>Purpose</b>	Terminate MATLAB
<b>Syntax</b>	<code>qui t</code>
<b>Description</b>	<code>qui t</code> terminates MATLAB without saving the workspace. To save your workspace variables, use the <code>save</code> command before quitting.
<b>See Also</b>	<code>save</code> Save workspace variables on disk <code>startup</code> MATLAB startup M-file

---

<b>Purpose</b>	QZ factorization for generalized eigenvalues
<b>Syntax</b>	$[AA, BB, Q, Z, V] = qz(A, B)$
<b>Description</b>	The <code>qz</code> 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 <code>AA</code> and <code>BB</code> , and matrices <code>Q</code> and <code>Z</code> containing the products of the left and right transformations, such that $\begin{aligned} Q*A*Z &= AA \\ Q*B*Z &= BB \end{aligned}$ The <code>qz</code> function also returns the generalized eigenvector matrix <code>V</code> . The generalized eigenvalues are the diagonal elements of <code>AA</code> and <code>BB</code> so that $A*V*di ag(BB) = B*V*di ag(AA)$
<b>Arguments</b>	<p><code>A, B</code>      Square matrices.</p> <p><code>AA, BB</code>    Upper triangular matrices.</p> <p><code>Q, Z</code>       Transformation matrices.</p> <p><code>V</code>          Matrix whose columns are eigenvectors.</p>
<b>Algorithm</b>	Complex generalizations of the EISPACK routines <code>QZHES</code> , <code>QZIT</code> , <code>QZVAL</code> , and <code>QZVEC</code> implement the <code>QZ</code> algorithm.
<b>See Also</b>	<code>eig</code> Eigenvalues and eigenvectors
<b>References</b>	[1] Moler, C. B. and G.W. Stewart, "An Algorithm for Generalized Matrix Eigenvalue Problems", <i>SIAM J. Numer. Anal.</i> , Vol. 10, No. 2, April 1973.

# rand

---

<b>Purpose</b>	Uniformly distributed random numbers and arrays								
<b>Syntax</b>	<pre>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')</pre>								
<b>Description</b>	<p>The <code>rand</code> function generates arrays of random numbers whose elements are uniformly distributed in the interval (0,1).</p> <p><code>Y = rand(n)</code> returns an <math>n</math>-by-<math>n</math> matrix of random entries. An error message appears if <math>n</math> is not a scalar.</p> <p><code>Y = rand(m, n)</code> or <code>Y = rand([m n])</code> returns an <math>m</math>-by-<math>n</math> matrix of random entries.</p> <p><code>Y = rand(m, n, p, ...)</code> or <code>Y = rand([m n p...])</code> generates random arrays.</p> <p><code>Y = rand(size(A))</code> returns an array of random entries that is the same size as <math>A</math>.</p> <p><code>rand</code>, by itself, returns a scalar whose value changes each time it's referenced.</p> <p><code>s = rand('state')</code> returns a 35-element vector containing the current state of the uniform generator. To change the state of the generator:</p> <table><tr><td><code>rand('state', s)</code></td><td>Resets the state to <math>s</math>.</td></tr><tr><td><code>rand('state', 0)</code></td><td>Resets the generator to its initial state.</td></tr><tr><td><code>rand('state', j)</code></td><td>For integer <math>j</math>, resets the generator to its <math>j</math>-th state.</td></tr><tr><td><code>rand('state', sum(100*clock))</code></td><td>Resets it to a different state each time.</td></tr></table>	<code>rand('state', s)</code>	Resets the state to $s$ .	<code>rand('state', 0)</code>	Resets the generator to its initial state.	<code>rand('state', j)</code>	For integer $j$ , resets the generator to its $j$ -th state.	<code>rand('state', sum(100*clock))</code>	Resets it to a different state each time.
<code>rand('state', s)</code>	Resets the state to $s$ .								
<code>rand('state', 0)</code>	Resets the generator to its initial state.								
<code>rand('state', j)</code>	For integer $j$ , resets the generator to its $j$ -th state.								
<code>rand('state', sum(100*clock))</code>	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**

<code>randn</code>	Normally distributed random numbers and arrays
<code>randperm</code>	Random permutation
<code>sprand</code>	Sparse uniformly distributed random matrix
<code>sprandn</code>	Sparse normally distributed random matrix

# randn

<b>Purpose</b>	Normally distributed random numbers and arrays								
<b>Syntax</b>	<pre>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')</pre>								
<b>Description</b>	<p>The <code>randn</code> function generates arrays of random numbers whose elements are normally distributed with mean 0 and variance 1.</p> <p><code>Y = randn(n)</code> returns an <math>n</math>-by-<math>n</math> matrix of random entries. An error message appears if <math>n</math> is not a scalar.</p> <p><code>Y = randn(m, n)</code> or <code>Y = randn([m n])</code> returns an <math>m</math>-by-<math>n</math> matrix of random entries.</p> <p><code>Y = randn(m, n, p, ...)</code> or <code>Y = randn([m n p...])</code> generates random arrays.</p> <p><code>Y = randn(size(A))</code> returns an array of random entries that is the same size as <math>A</math>.</p> <p><code>randn</code>, by itself, returns a scalar whose value changes each time it's referenced.</p> <p><code>s = randn('state')</code> returns a 2-element vector containing the current state of the normal generator. To change the state of the generator:</p> <table><tbody><tr><td><code>randn('state', s)</code></td><td>Resets the state to <math>s</math>.</td></tr><tr><td><code>randn('state', 0)</code></td><td>Resets the generator to its initial state.</td></tr><tr><td><code>randn('state', j)</code></td><td>For integer <math>j</math>, resets the generator to its <math>j</math>th state.</td></tr><tr><td><code>randn('state', sum(100*clock))</code></td><td>Resets it to a different state each time.</td></tr></tbody></table>	<code>randn('state', s)</code>	Resets the state to $s$ .	<code>randn('state', 0)</code>	Resets the generator to its initial state.	<code>randn('state', j)</code>	For integer $j$ , resets the generator to its $j$ th state.	<code>randn('state', sum(100*clock))</code>	Resets it to a different state each time.
<code>randn('state', s)</code>	Resets the state to $s$ .								
<code>randn('state', 0)</code>	Resets the generator to its initial state.								
<code>randn('state', j)</code>	For integer $j$ , resets the generator to its $j$ th state.								
<code>randn('state', sum(100*clock))</code>	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**

<code>rand</code>	Uniformly distributed random numbers and arrays
<code>randperm</code>	Random permutation
<code>sprand</code>	Sparse uniformly distributed random matrix
<code>sprandn</code>	Sparse normally distributed random matrix

# randperm

---

<b>Purpose</b>	Random permutation
<b>Syntax</b>	<code>p = randperm(n)</code>
<b>Description</b>	<code>p = randperm(n)</code> returns a random permutation of the integers 1: n.
<b>Remarks</b>	The <code>randperm</code> function calls <code>rand</code> and therefore changes <code>rand</code> 's seed value.
<b>Examples</b>	<code>randperm(6)</code> might be the vector <code>[3 2 6 4 1 5]</code> or it might be some other permutation of 1: 6.
<b>See Also</b>	<code>permute</code> Rearrange the dimensions of a multidimensional array

<b>Purpose</b>	Rank of a matrix
<b>Syntax</b>	<code>k = rank(A)</code> <code>k = rank(A, tol)</code>
<b>Description</b>	The rank function provides an estimate of the number of linearly independent rows or columns of a matrix.  <code>k = rank(A)</code> returns the number of singular values of A that are larger than the default tolerance, <code>max(size(A))*norm(A)*eps</code> .  <code>k = rank(A, tol)</code> returns the number of singular values of A that are larger than <code>tol</code> .
<b>Algorithm</b>	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 <i>LINPACK Users' Guide</i> . The SVD algorithm is the most time consuming, but also the most reliable.
	The rank algorithm is  <code>s = svd(A);</code> <code>tol = max(size(A))*s(1)*eps;</code> <code>r = sum(s &gt; tol);</code>
<b>References</b>	[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, <i>LINPACK Users' Guide</i> , SIAM, Philadelphia, 1979.

## **rat, rats**

---

<b>Purpose</b>	Rational fraction approximation
<b>Syntax</b>	<code>[N, D] = rat(X)</code> <code>[N, D] = rat(X, tol)</code> <code>rat(...)</code> <code>S = rats(X, strl en)</code> <code>S = rats(X)</code>
<b>Description</b>	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 <code>rat</code> function attempts to do this. Rational approximations are generated by truncating continued fraction expansions. The <code>rats</code> function calls <code>rat</code> , and returns strings.
	<code>[N, D] = rat(X)</code> returns arrays <code>N</code> and <code>D</code> so that <code>N./D</code> approximates <code>X</code> to within the default tolerance, <code>1. e-6*norm(X(:), 1)</code> .
	<code>[N, D] = rat(X, tol)</code> returns <code>N./D</code> approximating <code>X</code> to within <code>tol</code> .
	<code>rat(X)</code> , with no output arguments, simply displays the continued fraction.
	<code>S = rats(X, strl en)</code> returns a string containing simple rational approximations to the elements of <code>X</code> . Asterisks are used for elements that cannot be printed in the allotted space, but are not negligible compared to the other elements in <code>X</code> . <code>strl en</code> is the length of each string element returned by the <code>rats</code> function. The default is <code>strl en = 13</code> , which allows 6 elements in 78 spaces.
	<code>S = rats(X)</code> returns the same results as those printed by MATLAB with <code>format rat</code> .
<b>Examples</b>	Ordinarily, the statement  <code>s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7</code> produces  <code>s =</code> <code>0.7595</code>

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  $\pi$  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`

## rat, rats

---

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:

$$\frac{n}{d} = d_1 + \cfrac{1}{d_2 + \cfrac{1}{\left(d_3 + \dots + \cfrac{1}{d_k}\right)}}$$

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  $X = \text{sqrt}(2)$ . For  $x = \text{sqrt}(2)$ , 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.

---

<b>Purpose</b>	Matrix reciprocal condition number estimate												
<b>Syntax</b>	<code>c = rcond(A)</code>												
<b>Description</b>	<code>c = rcond(A)</code> returns an estimate for the reciprocal of the condition of A in 1-norm using the LINPACK condition estimator. If A is well conditioned, <code>rcond(A)</code> is near 1.0. If A is badly conditioned, <code>rcond(A)</code> is near 0.0. Compared to <code>cond</code> , <code>rcond</code> is a more efficient, but less reliable, method of estimating the condition of a matrix.												
<b>Algorithm</b>	The <code>rcond</code> function uses the condition estimator from the LINPACK routine ZGEC0.												
<b>See Also</b>	<table><tr><td><code>cond</code></td><td>Condition number with respect to inversion</td></tr><tr><td><code>condest</code></td><td>1-norm matrix condition number estimate</td></tr><tr><td><code>norm</code></td><td>Vector and matrix norms</td></tr><tr><td><code>normest</code></td><td>2-norm estimate</td></tr><tr><td><code>rank</code></td><td>Rank of a matrix</td></tr><tr><td><code>svd</code></td><td>Singular value decomposition</td></tr></table>	<code>cond</code>	Condition number with respect to inversion	<code>condest</code>	1-norm matrix condition number estimate	<code>norm</code>	Vector and matrix norms	<code>normest</code>	2-norm estimate	<code>rank</code>	Rank of a matrix	<code>svd</code>	Singular value decomposition
<code>cond</code>	Condition number with respect to inversion												
<code>condest</code>	1-norm matrix condition number estimate												
<code>norm</code>	Vector and matrix norms												
<code>normest</code>	2-norm estimate												
<code>rank</code>	Rank of a matrix												
<code>svd</code>	Singular value decomposition												
<b>References</b>	[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, <i>LINPACK Users' Guide</i> , SIAM, Philadelphia, 1979.												

## readsnd

---

**Purpose**      Read snd resources and files

**Syntax**      `[y, Fs] = readsnd(filename)`

**Description**      `[y, Fs] = readsnd(filename)` reads the sound data from the first 'snd' resource in the file *filename*. The sampled sound data is returned in *y*, while the frequency of the sampled sound is placed in *Fs*.

**Example**      `[y, Fs] = readsnd('gong.snd')`

---

<b>Purpose</b>	Real part of complex number										
<b>Syntax</b>	X = real (Z)										
<b>Description</b>	X = real (Z) returns the real part of the elements of the complex array Z.										
<b>Examples</b>	real (2+3*i) is 2.										
<b>See Also</b>	<table><tr><td>abs</td><td>Absolute value and complex magnitude</td></tr><tr><td>angle</td><td>Phase angle</td></tr><tr><td>conj</td><td>Complex conjugate</td></tr><tr><td>i, j</td><td>Imaginary unit (<math>\sqrt{-1}</math>)</td></tr><tr><td>imag</td><td>Imaginary part of a complex number</td></tr></table>	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
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										

# realmax

---

<b>Purpose</b>	Largest positive floating-point number				
<b>Syntax</b>	<code>n = realmax</code>				
<b>Description</b>	<code>n = realmax</code> returns the largest floating-point number representable on a particular computer. Anything larger overflows.				
<b>Examples</b>	On machines with IEEE floating-point format, <code>realmax</code> is one bit less than $2^{1024}$ or about <code>1.7977e+308</code> .				
<b>Algorithm</b>	The <code>realmax</code> function is equivalent to <code>pow2(2-eps, maxexp)</code> , where <code>maxexp</code> is the largest possible floating-point exponent. Execute type <code>realmax</code> to see <code>maxexp</code> for various computers.				
<b>See Also</b>	<table><tr><td><code>eps</code></td><td>Floating-point relative accuracy</td></tr><tr><td><code>realmin</code></td><td>Smallest positive floating-point number</td></tr></table>	<code>eps</code>	Floating-point relative accuracy	<code>realmin</code>	Smallest positive floating-point number
<code>eps</code>	Floating-point relative accuracy				
<code>realmin</code>	Smallest positive floating-point number				

<b>Purpose</b>	Smallest positive floating-point number
<b>Syntax</b>	<code>n = realmin</code>
<b>Description</b>	<code>n = realmin</code> returns the smallest positive normalized floating-point number on a particular computer. Anything smaller underflows or is an IEEE “denormal.”
<b>Examples</b>	On machines with IEEE floating-point format, <code>realmin</code> is $2^{-1022}$ or about $2.2251e-308$ .
<b>Algorithm</b>	The <code>realmin</code> function is equivalent to <code>pow2(1, minexp)</code> where <code>minexp</code> is the smallest possible floating-point exponent. Execute type <code>realmin</code> to see <code>minexp</code> for various computers.
<b>See Also</b>	<code>eps</code> Floating-point relative accuracy <code>realmax</code> Largest positive floating-point number

# recordsound

---

<b>Purpose</b>	Record sound
<b>Syntax</b>	<code>y = recordsound(seconds)</code> <code>y = recordsound(seconds, sampleRate)</code> <code>y = recordsound(seconds, numChannels)</code> <code>y = recordsound(seconds, sampleRate, numChannels)</code>
<b>Description</b>	<p><code>y = recordsound(seconds)</code> records a monophonic sound for seconds number of seconds at the lowest sampling rate (usually 11 or 22 kHz) and highest resolution (usually 8 or 16 bits) that the Macintosh supports.</p> <p><code>y = recordsound(seconds, sampleRate)</code> records a sound at a sampling rate greater than or equal to <code>sampleRate</code>, or at the maximum sampling rate that the Macintosh supports.</p> <p><code>y = recordsound(seconds, numChannels)</code> records a sound with <code>numChannels</code> (usually 1 or 2) channels. If <code>numChannels</code> is 2 and the Macintosh does not support stereo recording, a monophonic sound is recorded instead.</p> <p><code>y = recordsound(seconds, sampleRate, numChannels)</code> records a sound at the specified sampling rate and with the specified number of channels.</p>
<b>Examples</b>	<code>y = recordsound(10)</code> <code>y = recordsound(5, 22050)</code> <code>y = recordsound(5, 2)</code> <code>y = recordsound(5, 44100, 2)</code>

<b>Purpose</b>	Remainder after division
<b>Syntax</b>	$R = \text{rem}(X, Y)$
<b>Description</b>	$R = \text{rem}(X, Y)$ returns $X - \text{fix}(X / Y) * Y$ , where $\text{fix}(X / Y)$ is the integer part of the quotient, $X / Y$ .
<b>Remarks</b>	So long as operands X and Y are of the same sign, the statement $\text{rem}(X, Y)$ returns the same result as does $\text{mod}(X, Y)$ . However, for positive X and Y, $\text{rem}(-x, y) = \text{mod}(-x, y) - y$ The <code>rem</code> function returns a result that is between 0 and $\text{sign}(X) * \text{abs}(Y)$ . If Y is zero, <code>rem</code> returns NaN.
<b>Limitations</b>	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.
<b>See Also</b>	<a href="#">mod</a> Modulus (signed remainder after division)

# repmat

---

<b>Purpose</b>	Replicate and tile an array																																																
<b>Syntax</b>	$B = \text{repmat}(A, m, n)$ $B = \text{repmat}(A, [m n])$ $B = \text{repmat}(A, [m n p \dots])$ $\text{repmat}(A, m, n)$																																																
<b>Description</b>	$B = \text{repmat}(A, m, n)$ creates a large matrix $B$ consisting of an $m$ -by- $n$ tiling of copies of $A$ . The statement $\text{repmat}(A, n)$ creates an $n$ -by- $n$ tiling.  $B = \text{repmat}(A, [m n])$ accomplishes the same result as $\text{repmat}(A, m, n)$ .  $B = \text{repmat}(A, [m n p \dots])$ produces a multidimensional ( $m$ -by- $n$ -by- $p$ -by-...) array composed of copies of $A$ . $A$ may be multidimensional.  $\text{repmat}(A, m, n)$ when $A$ is a scalar, produces an $m$ -by- $n$ matrix filled with $A$ 's value. This can be much faster than $a^*\text{ones}(m, n)$ when $m$ or $n$ is large.																																																
<b>Examples</b>	In this example, $\text{repmat}$ replicates 12 copies of the second-order identity matrix, resulting in a “checkerboard” pattern.  $B = \text{repmat}(\text{eye}(2), 3, 4)$  $B =$ <table><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	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
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  $N = \text{repmat}(\text{NaN}, [2 3])$  creates a 2-by-3 matrix of NaNs.

<b>Purpose</b>	Reshape array						
<b>Syntax</b>	<pre>B = reshape(A, m, n) B = reshape(A, m, n, p, ...) B = reshape(A, [m n p...]) B = reshape(A, size)</pre>						
<b>Description</b>	<p><code>B = reshape(A, m, n)</code> returns the <math>m</math>-by-<math>n</math> matrix <code>B</code> whose elements are taken column-wise from <code>A</code>. An error results if <code>A</code> does not have <math>m \times n</math> elements.</p> <p><code>B = reshape(A, m, n, p, ...)</code> or <code>B = reshape(A, [m n p...])</code> returns an N-D array with the same elements as <code>X</code> but reshaped to have the size <math>m</math>-by-<math>n</math>-by-<math>p</math>-by-... . <math>m \times n \times p \times \dots</math> must be the same as <code>prod(size(x))</code>.</p> <p><code>B = reshape(A, size)</code> returns an N-D array with the same elements as <code>A</code>, but reshaped to <code>size</code>, a vector representing the dimensions of the reshaped array. The quantity <code>prod(size)</code> must be the same as <code>prod(size(A))</code>.</p>						
<b>Examples</b>	<p>Reshape a 3-by-4 matrix into a 2-by-6 matrix:</p> <pre>A = 1   4   7   10 2   5   8   11 3   6   9   12</pre> <p><code>B = reshape(A, 2, 6)</code></p> <pre>B = 1   3   5   7   9   11 2   4   6   8   10  12</pre>						
<b>See Also</b>	<table border="0"> <tr> <td><code>:</code> (colon)</td> <td>Colon :</td> </tr> <tr> <td><code>shiftdim</code></td> <td>Shift dimensions</td> </tr> <tr> <td><code>squeeze</code></td> <td>Remove singleton dimensions</td> </tr> </table>	<code>:</code> (colon)	Colon :	<code>shiftdim</code>	Shift dimensions	<code>squeeze</code>	Remove singleton dimensions
<code>:</code> (colon)	Colon :						
<code>shiftdim</code>	Shift dimensions						
<code>squeeze</code>	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:

$$\frac{b(s)}{a(s)} = \frac{b_1 + b_2 s^{-1} + b_3 s^{-2} + \dots + b_{m+1} s^{-m}}{a_1 + a_2 s^{-1} + a_3 s^{-2} + \dots + a_{n+1} s^{-n}}$$

[ 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:

$$\frac{b(s)}{a(s)} = \frac{r_1}{s - p_1} + \frac{r_2}{s - p_2} + \dots + \frac{r_n}{s - p_n} + k(s)$$

The number of poles n is

$$n = \text{length}(a) - 1 = \text{length}(r) = \text{length}(p)$$

The direct term coefficient vector is empty if  $\text{length}(b) < \text{length}(a)$  ; otherwise

$$\text{length}(k) = \text{length}(b) - \text{length}(a) + 1$$

If  $p(j) = \dots = p(j+m-1)$  is a pole of multiplicity m, then the expansion includes terms of the form

$$\frac{r_j}{s - p_j} + \frac{r_{j+1}}{(s - p_j)^2} + \dots + \frac{r_{j+m-1}}{(s - p_j)^m}$$

<b>Arguments</b>	b, a r p k	Vectors that specify the coefficients of the polynomials in descending powers of $s$ Column vector of residues Column vector of poles Row vector of direct terms
<b>Algorithm</b>		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 resi 2 computes the residues at the repeated root locations.
<b>Limitations</b>		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.
<b>See Also</b>	deconv poly roots	Deconvolution and polynomial division Polynomial with specified roots Polynomial roots
<b>References</b>		[1] Oppenheim, A.V. and R.W. Schafer, <i>Digital Signal Processing</i> , Prentice-Hall, 1975, p. 56.

# return

---

<b>Purpose</b>	Return to the invoking function																		
<b>Syntax</b>	<code>return</code>																		
<b>Description</b>	<code>return</code> causes a normal return to the invoking function or to the keyboard. It also terminates keyboard mode.																		
<b>Examples</b>	If the determinant function were an M-file, it might use a <code>return</code> statement in handling the special case of an empty matrix as follows:																		
	<pre>function d = det(A) %DET det(A) is the determinant of A. if isempty(A)     d = 1;     return else     ... end</pre>																		
<b>See Also</b>	<table><tr><td><code>break</code></td><td>Terminate execution of <code>for</code> or <code>while</code> loop</td></tr><tr><td><code>disp</code></td><td>Display text or array</td></tr><tr><td><code>end</code></td><td>Terminate <code>for</code>, <code>while</code>, <code>switch</code>, <code>try</code>, and <code>if</code> statements or indicate last index</td></tr><tr><td><code>error</code></td><td>Display error messages</td></tr><tr><td><code>for</code></td><td>Repeat statements a specific number of times</td></tr><tr><td><code>if</code></td><td>Conditionally execute statements</td></tr><tr><td><code>keyboard</code></td><td>Invoke the keyboard in an M-file</td></tr><tr><td><code>switch</code></td><td>Switch among several cases based on expression</td></tr><tr><td><code>while</code></td><td>Repeat statements an indefinite number of times</td></tr></table>	<code>break</code>	Terminate execution of <code>for</code> or <code>while</code> loop	<code>disp</code>	Display text or array	<code>end</code>	Terminate <code>for</code> , <code>while</code> , <code>switch</code> , <code>try</code> , and <code>if</code> statements or indicate last index	<code>error</code>	Display error messages	<code>for</code>	Repeat statements a specific number of times	<code>if</code>	Conditionally execute statements	<code>keyboard</code>	Invoke the keyboard in an M-file	<code>switch</code>	Switch among several cases based on expression	<code>while</code>	Repeat statements an indefinite number of times
<code>break</code>	Terminate execution of <code>for</code> or <code>while</code> loop																		
<code>disp</code>	Display text or array																		
<code>end</code>	Terminate <code>for</code> , <code>while</code> , <code>switch</code> , <code>try</code> , and <code>if</code> statements or indicate last index																		
<code>error</code>	Display error messages																		
<code>for</code>	Repeat statements a specific number of times																		
<code>if</code>	Conditionally execute statements																		
<code>keyboard</code>	Invoke the keyboard in an M-file																		
<code>switch</code>	Switch among several cases based on expression																		
<code>while</code>	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**

<code>fields</code>	Field names of a structure
<code>getfield</code>	Get field of structure array
<code>setfield</code>	Set field of structure array
<code>strvcat</code>	Vertical concatenation of strings

# rmpath

---

<b>Purpose</b>	Remove directories from MATLAB's search path				
<b>Syntax</b>	<code>rmpath directory</code>				
<b>Description</b>	<code>rmpath directory</code> removes the specified directory from MATLAB's current search path.				
<b>Remarks</b>	The function syntax form is also acceptable: <code>rmpath('directory')</code>				
<b>Examples</b>	<code>rmpath /usr/local/matlab/mytools</code>				
<b>See Also</b>	<table><tr><td><code>addpath</code></td><td>Add directories to MATLAB's search path</td></tr><tr><td><code>path</code></td><td>Control MATLAB's directory search path</td></tr></table>	<code>addpath</code>	Add directories to MATLAB's search path	<code>path</code>	Control MATLAB's directory search path
<code>addpath</code>	Add directories to MATLAB's search path				
<code>path</code>	Control MATLAB's directory search path				

<b>Purpose</b>	Polynomial roots
<b>Syntax</b>	<code>r = roots(c)</code>
<b>Description</b>	<p><code>r = roots(c)</code> returns a column vector whose elements are the roots of the polynomial <code>c</code>.</p> <p>Row vector <code>c</code> contains the coefficients of a polynomial, ordered in descending powers. If <code>c</code> has <math>n+1</math> components, the polynomial it represents is</p> $c_1 s^n + \dots + c_n s + c_{n+1}.$
<b>Remarks</b>	<p>Note the relationship of this function to <code>p = poly(r)</code>, which returns a row vector whose elements are the coefficients of the polynomial. For vectors, <code>roots</code> and <code>poly</code> are inverse functions of each other, up to ordering, scaling, and roundoff error.</p>
<b>Examples</b>	<p>The polynomial <math>s^3 - 6s^2 - 72s - 27</math> is represented in MATLAB as</p> <pre>p = [1 -6 -72 -27]</pre> <p>The roots of this polynomial are returned in a column vector by</p> <pre>r = roots(p) r =     12.1229    -5.7345    -0.3884</pre>
<b>Algorithm</b>	<p>The algorithm simply involves computing the eigenvalues of the companion matrix:</p> <pre>A = diag(ones(n-2, 1), -1); A(1, :) = -c(2:n-1) ./ c(1); eig(A)</pre> <p>It is possible to prove that the results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix <code>A</code>, but this does not mean that they are the exact roots of a polynomial with coefficients within roundoff error of those in <code>c</code>.</p>

# roots

---

## See Also

fzero	Zero of a function of one variable
poly	Polynomial with specified roots
residue	Convert between partial fraction expansion and polynomial coefficients

---

<b>Purpose</b>	Rotate matrix 90°
<b>Syntax</b>	<pre>B = rot90(A) B = rot90(A, k)</pre>
<b>Description</b>	<p><code>B = rot90(A)</code> rotates matrix A counterclockwise by 90 degrees.</p> <p><code>B = rot90(A, k)</code> rotates matrix A counterclockwise by <math>k \cdot 90</math> degrees, where <math>k</math> is an integer.</p>
<b>Examples</b>	<p>The matrix</p> <pre>X = 1   2   3 4   5   6 7   8   9</pre> <p>rotated by 90 degrees is</p> <pre>Y = rot90(X) Y = 3   6   9 2   5   8 1   4   7</pre>
<b>See Also</b>	<p><code>fliplr</code></p> <p><code>fliplr</code></p> <p><code>fliplr</code></p> <p><code>Flip array along a specified dimension</code></p> <p><code>Flip matrices left-right</code></p> <p><code>Flip matrices up-down</code></p>

# round

---

<b>Purpose</b>	Round to nearest integer						
<b>Syntax</b>	<code>Y = round(X)</code>						
<b>Description</b>	<code>Y = round(X)</code> rounds the elements of <code>X</code> to the nearest integers. For complex <code>X</code> , the imaginary and real parts are rounded independently.						
<b>Examples</b>	<pre>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</pre>						
<b>See Also</b>	<table><tr><td><code>ceil</code></td><td>Round toward infinity</td></tr><tr><td><code>fix</code></td><td>Round towards zero</td></tr><tr><td><code>floor</code></td><td>Round towards minus infinity</td></tr></table>	<code>ceil</code>	Round toward infinity	<code>fix</code>	Round towards zero	<code>floor</code>	Round towards minus infinity
<code>ceil</code>	Round toward infinity						
<code>fix</code>	Round towards zero						
<code>floor</code>	Round towards minus infinity						

<b>Purpose</b>	Reduced row echelon form
<b>Syntax</b>	$R = \text{rref}(A)$ $[R, jb] = \text{rref}(A)$ $[R, jb] = \text{rref}(A, tol)$ $\text{rrefmovie}(A)$
<b>Description</b>	<p><math>R = \text{rref}(A)</math> produces the reduced row echelon form of <math>A</math> using Gauss Jordan elimination with partial pivoting. A default tolerance of <math>(\max(\text{size}(A)) * \text{eps} * \text{norm}(A, \text{inf}))</math> tests for negligible column elements.</p> <p><math>[R, jb] = \text{rref}(A)</math> also returns a vector <math>jb</math> so that:</p> <ul style="list-style-type: none"> <li>• <math>r = \text{length}(jb)</math> is this algorithm's idea of the rank of <math>A</math>,</li> <li>• <math>x(jb)</math> are the bound variables in a linear system <math>Ax = b</math>,</li> <li>• <math>A(:, jb)</math> is a basis for the range of <math>A</math>,</li> <li>• <math>R(1:r, jb)</math> is the <math>r</math>-by-<math>r</math> identity matrix.</li> </ul> <p><math>[R, jb] = \text{rref}(A, tol)</math> uses the given tolerance in the rank tests.</p> <p>Roundoff errors may cause this algorithm to compute a different value for the rank than <code>rank</code>, <code>orth</code> and <code>nul1</code>.</p> <p><code>rrefmovie(A)</code> shows a movie of the algorithm working.</p>
<b>Examples</b>	<p>Use <code>rref</code> on a rank-deficient magic square:</p> <pre> 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 </pre>

## rref, rrefmovie

---

### See Also

[inv](#)  
[lu](#)  
[rank](#)

[Matrix inverse](#)  
[LU matrix factorization](#)  
[Rank of a matrix](#)

**Purpose** Convert real Schur form to complex Schur form

**Syntax**  $[U, T] = \text{rsf2csf}(U, T)$

**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.

$[U, T] = \text{rsf2csf}(U, T)$  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  $U'*U = \text{eye}(\text{size}(A))$ . See schur for details.

**Examples** Given matrix A,

$$\begin{matrix} 1 & 1 & 1 & 3 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 3 & 1 \\ -2 & 1 & 1 & 4 \end{matrix}$$

with the eigenvalues

$$1.9202 - 1.4742i \quad 1.9202 + 1.4742i \quad 4.8121 \quad 1.3474$$

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.

$U =$

$$\begin{matrix} -0.4576 + 0.3044i & 0.5802 - 0.4934i & -0.0197 & -0.3428 \\ 0.1616 + 0.3556i & 0.4235 + 0.0051i & 0.1666 & 0.8001 \\ 0.3963 + 0.2333i & 0.1718 + 0.2458i & 0.7191 & -0.4260 \\ -0.4759 - 0.3278i & -0.2709 - 0.2778i & 0.6743 & 0.2466 \end{matrix}$$

## rsf2csf

---

```
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

---

<b>Purpose</b>	Save workspace variables on disk
<b>Syntax</b>	<code>save</code> <code>save <i>filename</i></code> <code>save <i>filename variables</i></code> <code>save <i>filename options</i></code> <code>save <i>filename variables options</i></code>
<b>Description</b>	<p><code>save</code>, by itself, stores all workspace variables in a binary format in the file named <code>matlab.mat</code>. The data can be retrieved with <code>load</code>.</p> <p><code>save <i>filename</i></code> stores all workspace variables in <code>filename.mat</code> instead of the default <code>matlab.mat</code>. If <code>filename</code> is the special string <code>stdio</code>, the <code>save</code> command sends the data as standard output.</p> <p><code>save <i>filename variables</i></code> saves only the workspace <code>variables</code> you list after the <code>filename</code>.</p>
<b>Options</b>	The forms of the <code>save</code> command that use <code>options</code> are:  <code>save <i>filename options</i></code>  <code>save <i>filename variables options</i>,</code>  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:
<code>-ascii</code>	8-digit ASCII format
<code>-ascii -double</code>	16-digit ASCII format
<code>-ascii -tabs</code>	8-digit ASCII format, tab-separated
<code>-ascii -double -tabs</code>	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 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.

# **save**

---

## **See Also**

`fprintf`      Write formatted data to file  
`fwrite`      Write binary data to a file  
`load`      Retrieve variables from disk

<b>Purpose</b>	Schur decomposition
<b>Syntax</b>	$[U, T] = \text{schur}(A)$ $T = \text{schur}(A)$
<b>Description</b>	The <code>schur</code> command computes the Schur form of a matrix.
	$[U, T] = \text{schur}(A)$ produces a Schur matrix $T$ , and a unitary matrix $U$ so that $A = U*T*U'$ and $U'*U = \text{eye}(\text{size}(A))$ .
	$T = \text{schur}(A)$ returns just the Schur matrix $T$ .
<b>Remarks</b>	<p>The <i>complex Schur form</i> of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The <i>real Schur form</i> has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.</p> <p>If the matrix <math>A</math> is real, <code>schur</code> returns the real Schur form. If <math>A</math> is complex, <code>schur</code> returns the complex Schur form. The function <code>rsf2csf</code> converts the real form to the complex form.</p>
<b>Examples</b>	<p><math>H</math> is a 3-by-3 eigenvalue test matrix:</p> $H = \begin{matrix} -149 & -50 & -154 \\ 537 & 180 & 546 \\ -27 & -9 & -25 \end{matrix}$ <p>Its Schur form is</p> $\text{schur}(H) = \begin{matrix} 1.0000 & 7.1119 & 815.8706 \\ 0 & 2.0000 & -55.0236 \\ 0 & 0 & 3.0000 \end{matrix}$ <p>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.</p>
<b>Algorithm</b>	For real matrices, <code>schur</code> uses the EISPACK routines ORTRAN, ORTHES, and HQR2. <code>ORTHES</code> 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

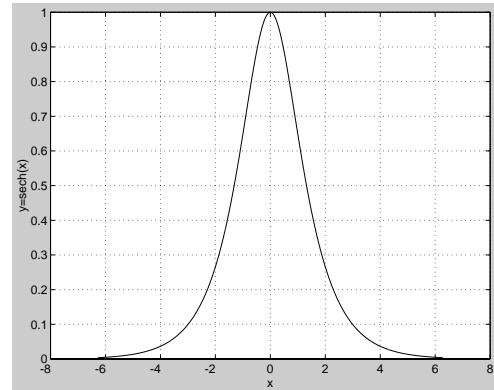
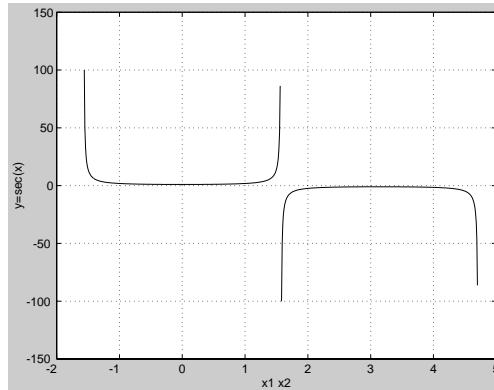
- [1] Garbow, B. S., J. M. Boyle, J. J. Dongarra, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide Extension*, Lecture Notes in Computer Science, Vol. 51, Springer-Verlag, 1977.
- [2] Moler, C.B. and G. W. Stewart, “An Algorithm for Generalized Matrix Eigenvalue Problems,” *SIAM J. Numer. 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.

<b>Purpose</b>	Script M-files						
<b>Description</b>	<p>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.</p> <p>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.</p> <p>Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters.</p> <p>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.</p>						
<b>See Also</b>	<table><tr><td>echo</td><td>Echo M-files during execution</td></tr><tr><td>function</td><td>Function M-files</td></tr><tr><td>type</td><td>List file</td></tr></table>	echo	Echo M-files during execution	function	Function M-files	type	List file
echo	Echo M-files during execution						
function	Function M-files						
type	List file						

# sec, sech

Purpose	Secant and hyperbolic secant
Syntax	$Y = \sec(X)$ $Y = \operatorname{sech}(X)$
Description	The <code>sec</code> and <code>sech</code> 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 $X$ .
	$Y = \operatorname{sech}(X)$ 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 $-2\pi \leq x \leq 2\pi$ .

```
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  $\text{sec}(\pi / 2)$  does not evaluate as infinite but as the reciprocal of the floating-point accuracy  $\text{eps}$ , because  $\pi$  is a floating-point approximation to the exact value of  $\pi$ .

**Algorithm**

$$\text{sec}(z) = \frac{1}{\cos(z)} \quad \text{sech}(z) = \frac{1}{\cosh(z)}$$

**See Also**

asec, asech

Inverse secant and inverse hyperbolic secant

# setdiff

---

<b>Purpose</b>	Return the set difference of two vectors										
<b>Syntax</b>	<pre>c = setdiff(a, b) c = setdiff(A, B, 'rows') [c, i] = setdiff(...)</pre>										
<b>Description</b>	<p><code>c = setdiff(a, b)</code> returns the values in <code>a</code> that are not in <code>b</code>. The resulting vector is sorted in ascending order. In set theoretic terms, <math>c = a - b</math>. <code>a</code> and <code>b</code> can be cell arrays of strings.</p> <p><code>c = (A, B, 'rows')</code> when <code>A</code> and <code>B</code> are matrices with the same number of columns returns the rows from <code>A</code> that are not in <code>B</code>.</p> <p><code>[c, i] = setdiff(...)</code> also returns an index vector <code>i</code> ndex such that <math>c = a(i)</math> or <math>c = a(i, :)</math>.</p>										
<b>Examples</b>	<pre>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</pre>										
<b>See Also</b>	<table><tr><td><code>intersect</code></td><td>Set intersection of two vectors</td></tr><tr><td><code>ismember</code></td><td>True for a set member</td></tr><tr><td><code>setxor</code></td><td>Set exclusive-or of two vectors</td></tr><tr><td><code>union</code></td><td>Set union of two vectors</td></tr><tr><td><code>unique</code></td><td>Unique elements of a vector</td></tr></table>	<code>intersect</code>	Set intersection of two vectors	<code>ismember</code>	True for a set member	<code>setxor</code>	Set exclusive-or of two vectors	<code>union</code>	Set union of two vectors	<code>unique</code>	Unique elements of a vector
<code>intersect</code>	Set intersection of two vectors										
<code>ismember</code>	True for a set member										
<code>setxor</code>	Set exclusive-or of two vectors										
<code>union</code>	Set union of two vectors										
<code>unique</code>	Unique elements of a vector										

---

<b>Purpose</b>	Set field of structure array				
<b>Syntax</b>	<pre>s = setfield(s, 'field', v) s = setfield(s, {i,j}, 'field', {k}, v)</pre>				
<b>Description</b>	<p><code>s = setfield(s, 'field', v)</code>, where <code>s</code> is a 1-by-1 structure, sets the contents of the specified field to the value <code>v</code>. This is equivalent to the syntax <code>s.field = v</code>.</p> <p><code>s = setfield(s, {i,j}, 'field', {k}, v)</code> sets the contents of the specified field to the value <code>v</code>. This is equivalent to the syntax <code>s(i,j).field(k) = v</code>. All subscripts must be passed as cell arrays—that is, they must be enclosed in curly braces (similar to <code>{i,j}</code> and <code>{k}</code> above). Pass field references as strings.</p>				
<b>Examples</b>	<p>Given the structure:</p> <pre>mystr(1, 1).name = 'alice'; mystr(1, 1).ID = 0; mystr(2, 1).name = 'gertrude'; mystr(2, 1).ID = 1</pre> <p>Then the command <code>mystr = setfield(mystr, {2, 1}, 'name', 'ted')</code> yields</p> <pre>mystr = 2x1 struct array with fields:     name     ID</pre>				
<b>See Also</b>	<table border="0"> <tr> <td><code>fields</code></td> <td>Field names of a structure</td> </tr> <tr> <td><code>getfield</code></td> <td>Get field of structure array</td> </tr> </table>	<code>fields</code>	Field names of a structure	<code>getfield</code>	Get field of structure array
<code>fields</code>	Field names of a structure				
<code>getfield</code>	Get field of structure array				

## **setstr**

---

**Purpose** Set string flag

**Description** This MATLAB 4 function has been renamed `char` in MATLAB 5.

**See Also** `char` Create character array (string)

**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. `a` and `b` can be cell arrays of strings.

`c = setxor(A, B, 'rows')` when `A` and `B` are matrices with the same number of columns returns the rows that are not in the intersection of `A` and `B`.

`[c, ia, ib] = setxor(...)` also returns index vectors `ia` and `ib` such that `c` is a sorted combination of the elements `c = a(ia)` and `c = b(ib)` or, for row combinations, `c = a(ia, :)` and `c = b(ib, :)`.

### Examples

```
a = [-1 0 1 Inf -Inf NaN];
b = [-2 pi 0 Inf];
c = setxor(a, b)

c =
          -Inf      -2.0000      -1.0000      1.0000      3.1416      NaN
```

### See Also

<code>intersect</code>	Set intersection of two vectors
<code>ismember</code>	True for a set member
<code>setdiff</code>	Set difference of two vectors
<code>union</code>	Set union of two vectors
<code>unique</code>	Unique elements of a vector

# shiftdim

---

<b>Purpose</b>	Shift dimensions
<b>Syntax</b>	$B = \text{shiftdim}(X, n)$ $[B, \text{nshifts}] = \text{shiftdim}(X)$
<b>Description</b>	$B = \text{shiftdim}(X, n)$ shifts the dimensions of $X$ by $n$ . When $n$ is positive, $\text{shiftdim}$ shifts the dimensions to the left and wraps the $n$ leading dimensions to the end. When $n$ is negative, $\text{shiftdim}$ shifts the dimensions to the right and pads with singletons.  $[B, \text{nshifts}] = \text{shiftdim}(X)$ 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 $\text{size}(A, \text{dim}) = 1$ . $\text{nshifts}$ is the number of dimensions that are removed.
	If $X$ is a scalar, $\text{shiftdim}$ has no effect.
<b>Examples</b>	The $\text{shiftdim}$ command is handy for creating functions that, like $\text{sum}$ or $\text{diff}$ , work along the first nonsingleton dimension.
	<pre>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-3.</pre>
<b>See Also</b>	<a href="#">reshape</a> Reshape array <a href="#">squeeze</a> Remove singleton dimensions

---

<b>Purpose</b>	Signum function								
<b>Syntax</b>	$Y = \text{sign}(X)$								
<b>Description</b>	$Y = \text{sign}(X)$ returns an array $Y$ the same size as $X$ , where each element of $Y$ is: <ul style="list-style-type: none"><li>• 1 if the corresponding element of <math>X</math> is greater than zero</li><li>• 0 if the corresponding element of <math>X</math> equals zero</li><li>• -1 if the corresponding element of <math>X</math> is less than zero</li></ul> For nonzero complex $X$ , $\text{sign}(X) = X ./ \text{abs}(X)$ .								
<b>See Also</b>	<table><tr><td><code>abs</code></td><td>Absolute value and complex magnitude</td></tr><tr><td><code>conj</code></td><td>Complex conjugate</td></tr><tr><td><code>i mag</code></td><td>Imaginary part of a complex number</td></tr><tr><td><code>real</code></td><td>Real part of complex number</td></tr></table>	<code>abs</code>	Absolute value and complex magnitude	<code>conj</code>	Complex conjugate	<code>i mag</code>	Imaginary part of a complex number	<code>real</code>	Real part of complex number
<code>abs</code>	Absolute value and complex magnitude								
<code>conj</code>	Complex conjugate								
<code>i mag</code>	Imaginary part of a complex number								
<code>real</code>	Real part of complex number								

# **sin, sinh**

**Purpose** Sine and hyperbolic sine

**Syntax**

```
Y = sin(X)
Y = sinh(X)
```

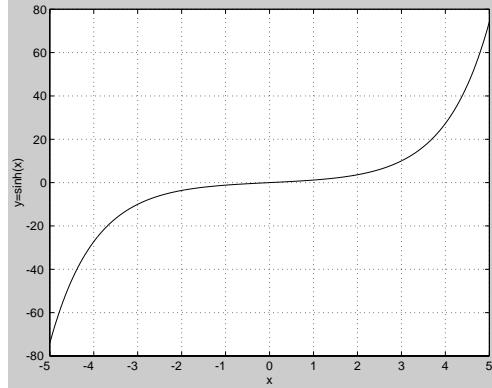
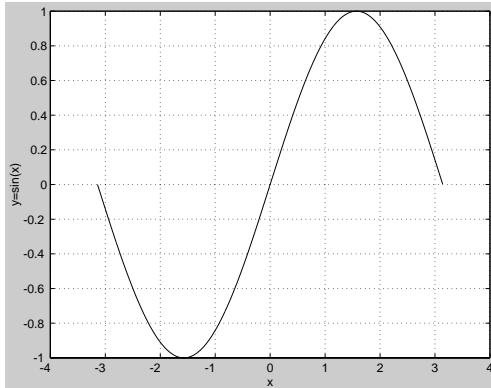
**Description** The `sin` and `sinh` commands operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

`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  $-\pi \leq x \leq \pi$ , and the hyperbolic sine function over the domain  $-5 \leq x \leq 5$ .

```
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  $\pi$ .

**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**

asi n, asi nh

Inverse sine and inverse hyperbolic sine

# size

---

<b>Purpose</b>	Array dimensions
<b>Syntax</b>	$d = \text{size}(X)$ $[m, n] = \text{size}(X)$ $m = \text{size}(X, \text{dim})$ $[d_1, d_2, d_3, \dots, d_n] = \text{size}(X)$
<b>Description</b>	$d = \text{size}(X)$ returns the sizes of each dimension of array X in a vector d with $\text{ndims}(X)$ elements.  $[m, n] = \text{size}(X)$ returns the size of matrix X in variables m and n.  $m = \text{size}(X, \text{dim})$ returns the size of the dimension of X specified by scalar dim.  $[d_1, d_2, d_3, \dots, d_n] = \text{size}(X)$ returns the sizes of the various dimensions of array X in separate variables.  If the number of output arguments n does not equal $\text{ndims}(X)$ , then:  If $n > \text{ndims}(X)$ Ones are returned in the “extra” variables $d_{\text{ndims}(X)+1}$ through $d_n$ .  If $n < \text{ndims}(X)$ The final variable $d_n$ contains the product of the sizes of all the “remaining” dimensions of X, that is, dimensions $n+1$ through $\text{ndims}(X)$ .
<b>Examples</b>	The size of the second dimension of $\text{rand}(2, 3, 4)$ is 3.  $m = \text{size}(\text{rand}(2, 3, 4), 2)$  $m =$ 3  Here the size is output as a single vector.  $d = \text{size}(\text{rand}(2, 3, 4))$  $d =$ 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

# sort

---

<b>Purpose</b>	Sort elements in ascending order
<b>Syntax</b>	$B = \text{sort}(A)$ $[B, \text{INDEX}] = \text{sort}(A)$ $B = \text{sort}(A, \text{dim})$
<b>Description</b>	<p><math>B = \text{sort}(A)</math> sorts the elements along different dimensions of an array, and arranges those elements in ascending order. <math>A</math> can be a cell array of strings.</p> <p>Real, complex, and string elements are permitted. For identical values in <math>A</math>, the location in the input array determines location in the sorted list. When <math>A</math> is complex, the elements are sorted by magnitude, and where magnitudes are equal, further sorted by phase angle on the interval <math>[-\pi, \pi]</math>. If <math>A</math> includes any NaN elements, sort places these at the end.</p> <p>If <math>A</math> is a vector, <math>\text{sort}(A)</math> arranges those elements in ascending order.</p> <p>If <math>A</math> is a matrix, <math>\text{sort}(A)</math> treats the columns of <math>A</math> as vectors, returning sorted columns.</p> <p>If <math>A</math> is a multidimensional array, <math>\text{sort}(A)</math> treats the values along the first non-singleton dimension as vectors, returning an array of sorted vectors.</p> <p><math>[B, \text{INDEX}] = \text{sort}(A)</math> also returns an array of indices. <math>\text{INDEX}</math> is an array of size(<math>A</math>), each column of which is a permutation vector of the corresponding column of <math>A</math>. If <math>A</math> has repeated elements of equal value, indices are returned that preserve the original relative ordering.</p> <p><math>B = \text{sort}(A, \text{dim})</math> sorts the elements along the dimension of <math>A</math> specified by scalar <math>\text{dim}</math>.</p> <p>If <math>\text{dim}</math> is a vector, sort works iteratively on the specified dimensions. Thus, <math>\text{sort}(A, [1 2])</math> is equivalent to <math>\text{sort}(\text{sort}(A, 2), 1)</math>.</p>
<b>See Also</b>	<a href="#">max</a> Maximum elements of an array <a href="#">mean</a> Average or mean value of arrays <a href="#">median</a> Median value of arrays <a href="#">min</a> Minimum elements of an array <a href="#">sortrows</a> Sort rows in ascending order

<b>Purpose</b>	Sort rows in ascending order												
<b>Syntax</b>	<pre>B = sortrows(A) B = sortrows(A, column) [B, index] = sortrows(A)</pre>												
<b>Description</b>	<p><code>B = sortrows(A)</code> sorts the rows of A as a group in ascending order. Argument A must be either a matrix or a column vector.</p> <p>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 <math>[-\pi, \pi]</math>.</p> <p><code>B = sortrows(A, column)</code> sorts the matrix based on the columns specified in the vector column. For example, <code>sortrows(A, [2 3])</code> sorts the rows of A by the second column, and where these are equal, further sorts by the third column.</p> <p><code>[B, index] = sortrows(A)</code> also returns an index vector index.</p> <p>If A is a column vector, then <code>B = A(index)</code>.</p> <p>If A is an m-by-n matrix, then <code>B = A(index, :)</code>.</p>												
<b>Examples</b>	<p>Given the 5-by-5 string matrix,</p> <pre>A = ['one' ; 'two' ; 'three' ; 'four' ; 'five'];</pre> <p>The commands <code>B = sortrows(A)</code> and <code>C = sortrows(A, 1)</code> yield</p> <table style="margin-left: 20px;"> <tr> <td><code>B =</code></td> <td><code>C =</code></td> </tr> <tr> <td>five</td> <td>four</td> </tr> <tr> <td>four</td> <td>five</td> </tr> <tr> <td>one</td> <td>one</td> </tr> <tr> <td>three</td> <td>two</td> </tr> <tr> <td>two</td> <td>three</td> </tr> </table>	<code>B =</code>	<code>C =</code>	five	four	four	five	one	one	three	two	two	three
<code>B =</code>	<code>C =</code>												
five	four												
four	five												
one	one												
three	two												
two	three												
<b>See Also</b>	<a href="#">sort</a> Sort elements in ascending order												

# sound

---

<b>Purpose</b>	Convert vector into sound
<b>Syntax</b>	<code>sound(y, Fs)</code> <code>sound(y)</code> <code>sound(y, Fs, bits)</code>
<b>Description</b>	<code>sound(y, Fs)</code> , sends the signal in vector <i>y</i> (with sample frequency <i>Fs</i> ) to the speaker on PC, Macintosh, and most UNIX platforms. Values in <i>y</i> are assumed to be in the range $-1.0 \leq y \leq 1.0$ . Values outside that range are clipped. Stereo sound is played on platforms that support it when <i>y</i> is an n-by-2 matrix.  <code>sound(y)</code> plays the sound at the default sample rate or 8192 Hz.  <code>sound(y, Fs, bits)</code> plays the sound using <i>bits</i> bits/sample if possible. Most platforms support <i>bits</i> = 8 or <i>bits</i> = 16.
<b>Remarks</b>	MATLAB supports all Windows-compatible sound devices.
<b>See Also</b>	<code>auread</code> Read NeXT/SUN (.au) sound file <code>auwrite</code> Write NeXT/SUN (.au) sound file <code>soundsc</code> Scale data and play as sound <code>wavread</code> Read Microsoft WAVE (.wav) sound file <code>wavwrite</code> Write Microsoft WAVE (.wav) sound file

---

<b>Purpose</b>	Sound capabilities
<b>Syntax</b>	<code>soundcap</code>
<b>Description</b>	<code>soundcap</code> prints the computer's sound capabilities, including whether or not the computer can play stereo sound and record sound, the sampling rates supported for recording, and the resolution supported for recording and playback.

## soundsc

---

<b>Purpose</b>	Scale data and play as sound
<b>Syntax</b>	<code>soundsc(y, Fs)</code> <code>soundsc(y)</code> <code>soundsc(y, Fs, bits)</code> <code>soundsc(y, ..., slim)</code>
<b>Description</b>	<code>soundsc(y, Fs)</code> sends the signal in vector <code>y</code> (with sample frequency <code>Fs</code> ) to the speaker on PC, Macintosh, and most UNIX platforms. The signal <code>y</code> is scaled to the range $-1.0 \leq y \leq 1.0$ before it is played, resulting in a sound that is played as loud as possible without clipping.  <code>soundsc(y)</code> plays the sound at the default sample rate or 8192 Hz.  <code>soundsc(y, Fs, bits)</code> plays the sound using <code>bits</code> bits/sample if possible. Most platforms support <code>bits = 8</code> or <code>bits = 16</code> .  <code>soundsc(y, ..., slim)</code> where <code>slim = [slow shigh]</code> maps the values in <code>y</code> between <code>slow</code> and <code>shigh</code> to the full sound range. The default value is <code>slim = [min(y) max(y)]</code> .
<b>Remarks</b>	MATLAB supports all Windows-compatible sound devices.
<b>See Also</b>	<code>auread</code> Read NeXT/SUN (.au) sound file <code>auwrite</code> Write NeXT/SUN (.au) sound file <code>sound</code> Convert vector into sound <code>wavread</code> Read Microsoft WAVE (.wav) sound file <code>wavwrite</code> Write Microsoft WAVE (.wav) sound file

---

<b>Purpose</b>	Allocate space for sparse matrix
<b>Syntax</b>	<code>S = spalloc(m, n, nzmax)</code>
<b>Description</b>	<code>S = spalloc(m, n, nzmax)</code> 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.
	<code>spalloc(m, n, nzmax)</code> is shorthand for <code>sparse([], [], [], m, n, nzmax)</code>
<b>Examples</b>	To generate efficiently a sparse matrix that has an average of at most three nonzero elements per column
	<pre>S = spalloc(n, n, 3*n); for j = 1:n     S(:,j) = [zeros(n-3, 1)' round(rand(3, 1))' ]'; end</pre>

# sparse

---

<b>Purpose</b>	Create sparse matrix
<b>Syntax</b>	<pre>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)</pre>
<b>Description</b>	<p>The <code>sparse</code> function generates matrices in MATLAB's sparse storage organization.</p> <p><code>S = sparse(A)</code> converts a full matrix to sparse form by squeezing out any zero elements. If <code>S</code> is already sparse, <code>sparse(S)</code> returns <code>S</code>.</p> <p><code>S = sparse(i,j,s,m,n,nzmax)</code> uses vectors <code>i</code>, <code>j</code>, and <code>s</code> to generate an <code>m</code>-by-<code>n</code> sparse matrix with space allocated for <code>nzmax</code> nonzeros. Any elements of <code>s</code> that are zero are ignored, along with the corresponding values of <code>i</code> and <code>j</code>. Vectors <code>i</code>, <code>j</code>, and <code>s</code> are all the same length. Any elements of <code>s</code> that have duplicate values of <code>i</code> and <code>j</code> are added together.</p> <p>To simplify this six-argument call, you can pass scalars for the argument <code>s</code> and one of the arguments <code>i</code> or <code>j</code>—in which case they are expanded so that <code>i</code>, <code>j</code>, and <code>s</code> all have the same length.</p> <p><code>S = sparse(i,j,s,m,n)</code> uses <code>nzmax = length(s)</code>.</p> <p><code>S = sparse(i,j,s)</code> uses <code>m = max(i)</code> and <code>n = max(j)</code>. The maxima are computed before any zeros in <code>s</code> are removed, so one of the rows of <code>[i j s]</code> might be <code>[m n 0]</code>.</p> <p><code>S = sparse(m,n)</code> abbreviates <code>sparse([],[],[],m,n,0)</code>. This generates the ultimate sparse matrix, an <code>m</code>-by-<code>n</code> all zero matrix.</p>
<b>Remarks</b>	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:

<code>diag</code>	Diagonal matrices and diagonals of a matrix
<code>find</code>	Find indices and values of nonzero elements
<code>full</code>	Convert sparse matrix to full matrix
<code>nnz</code>	Number of nonzero matrix elements
<code>nonzeros</code>	Nonzero matrix elements
<code>nzmax</code>	Amount of storage allocated for nonzero matrix elements
<code>spones</code>	Replace nonzero sparse matrix elements with ones
<code>sprandn</code>	Sparse normally distributed random matrix
<code>sprandsym</code>	Sparse symmetric random matrix
<code>spy</code>	Visualize sparsity pattern

# spconvert

---

<b>Purpose</b>	Import matrix from sparse matrix external format
<b>Syntax</b>	<code>S = spconvert(D)</code>
<b>Description</b>	<code>spconvert</code> is used to create sparse matrices from a simple sparse format easily produced by non-MATLAB sparse programs. <code>spconvert</code> is the second step in the process: <ol style="list-style-type: none"><li>1 Load an ASCII data file containing <code>[i, j, v]</code> or <code>[i, j, re, im]</code> as rows into a MATLAB variable.</li><li>2 Convert that variable into a MATLAB sparse matrix.</li></ol>
	<code>S = spconvert(D)</code> converts a matrix <code>D</code> with rows containing <code>[i, j, s]</code> or <code>[i, j, r, s]</code> to the corresponding sparse matrix. <code>D</code> must have an <code>nnz</code> or <code>nnz+1</code> 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 <code>[m n 0]</code> or <code>[m n 0 0]</code> anywhere in <code>D</code> can be used to specify <code>size(S)</code> . If <code>D</code> is already sparse, no conversion is done, so <code>spconvert</code> can be used after <code>D</code> is loaded from either a MAT-file or an ASCII file.
<b>Examples</b>	Suppose the ASCII file <code>uphill.dat</code> contains <pre>1     1     1.0000000000000000 1     2     0.5000000000000000 2     2     0.3333333333333333 1     3     0.3333333333333333 2     3     0.2500000000000000 3     3     0.2000000000000000 1     4     0.2500000000000000 2     4     0.2000000000000000 3     4     0.1666666666666667 4     4     0.142857142857143 4     4     0.0000000000000000</pre>

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.

# spdiags

---

<b>Purpose</b>	Extract and create sparse band and diagonal matrices
<b>Syntax</b>	$[B, d] = \text{spdiags}(A)$ $B = \text{spdiags}(A, d)$ $A = \text{spdiags}(B, d, A)$ $A = \text{spdiags}(B, d, m, n)$
<b>Description</b>	The <code>spdiags</code> function generalizes the function <code>diag</code> . Four different operations, distinguished by the number of input arguments, are possible:  $[B, d] = \text{spdiags}(A)$ 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 $A$ .  $B = \text{spdiags}(A, d)$ extracts the diagonals specified by $d$ .  $A = \text{spdiags}(B, d, A)$ replaces the diagonals specified by $d$ with the columns of $B$ . The output is sparse.  $A = \text{spdiags}(B, d, m, n)$ creates an $m$ -by- $n$ sparse matrix by taking the columns of $B$ and placing them along the diagonals specified by $d$ .
<b>Remarks</b>	If a column of $B$ is longer than the diagonal it's replacing, <code>spdiags</code> takes elements from $B$ 's tail.
<b>Arguments</b>	The <code>spdiags</code> function deals with three matrices, in various combinations, as both input and output: <ul style="list-style-type: none"><li>A An <math>m</math>-by-<math>n</math> matrix, usually (but not necessarily) sparse, with its nonzero or specified elements located on <math>p</math> diagonals.</li><li>B A <math>\min(m, n)</math>-by-<math>p</math> matrix, usually (but not necessarily) full, whose columns are the diagonals of <math>A</math>.</li><li>d A vector of length <math>p</math> whose integer components specify the diagonals in <math>A</math>.</li></ul>

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]
```

## **spdiags**

---

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

---

<b>Purpose</b>	Speak text string
<b>Syntax</b>	<code>speak(y)</code> <code>speak(y, voice)</code>
<b>Description</b>	<code>speak(y)</code> speaks the text string <i>y</i> using the default voice.  <code>speak(y, voice)</code> speaks the text string <i>y</i> using the voice specified by <i>voice</i> . speak requires the Speech Manager and works only on the Macintosh.
<b>Examples</b>	<code>speak('I like math.')</code> <code>speak('I really like matlab', 'good news')</code>

# speye

---

<b>Purpose</b>	Sparse identity matrix
<b>Syntax</b>	$S = \text{speye}(m, n)$ $S = \text{speye}(n)$
<b>Description</b>	$S = \text{speye}(m, n)$ forms an $m$ -by- $n$ sparse matrix with 1s on the main diagonal. $S = \text{speye}(n)$ abbreviates $\text{speye}(n, n)$ .
<b>Examples</b>	$I = \text{speye}(1000)$ 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 $I = \text{sparse}(\text{eye}(1000, 1000))$ , but the latter requires eight megabytes for temporary storage for the full representation.
<b>See Also</b>	<a href="#">spalloc</a> Allocate space for sparse matrix <a href="#">spones</a> Replace nonzero sparse matrix elements with ones <a href="#">spdiags</a> Extract and create sparse band and diagonal matrices <a href="#">sprand</a> Sparse uniformly distributed random matrix <a href="#">sprandn</a> Sparse normally distributed random matrix

**Purpose** Apply function to nonzero sparse matrix elements

**Syntax**  $f = \text{spfun}('function', S)$

**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).

$f = \text{spfun}('function', S)$  evaluates  $\text{function}(S)$  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
```

$f = \text{spfun}('exp', S)$  has the same sparsity pattern as  $S$ :

```
f =
(1, 1)      2. 7183
(2, 2)      7. 3891
(3, 3)      20. 0855
(4, 4)      54. 5982
```

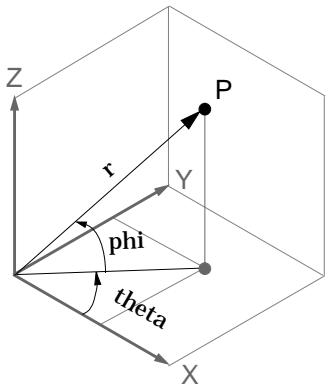
whereas  $\text{exp}(S)$  has 1s where  $S$  has 0s.

```
full(exp(S))
```

```
ans =
2. 7183    1. 0000    1. 0000    1. 0000
1. 0000    7. 3891    1. 0000    1. 0000
1. 0000    1. 0000    20. 0855   1. 0000
1. 0000    1. 0000    1. 0000    54. 5982
```

# sph2cart

---

<b>Purpose</b>	Transform spherical coordinates to Cartesian	
<b>Syntax</b>	$[x, y, z] = \text{sph2cart}(\text{THETA}, \text{PHI}, R)$	
<b>Description</b>	$[x, y, z] = \text{sph2cart}(\text{THETA}, \text{PHI}, R)$ transforms the corresponding elements of spherical coordinate arrays to Cartesian, or <i>xyz</i> , coordinates. <i>THETA</i> , <i>PHI</i> , and <i>R</i> must all be the same size. <i>THETA</i> and <i>PHI</i> are angular displacements in radians from the positive <i>x</i> -axis and from the <i>x</i> - <i>y</i> plane, respectively.	
<b>Algorithm</b>	The mapping from spherical coordinates to three-dimensional Cartesian coordinates is:	
	 $x = r \cdot \cos(\phi) \cdot \cos(\theta)$ $y = r \cdot \cos(\phi) \cdot \sin(\theta)$ $z = r \cdot \sin(\phi)$	
<b>See Also</b>	<a href="#">cart2pol</a> <a href="#">cart2sph</a> <a href="#">pol2cart</a>	Transform Cartesian coordinates to polar or cylindrical Transform Cartesian coordinates to spherical Transform polar or cylindrical coordinates to Cartesian

**Purpose** Cubic spline interpolation

**Syntax**

```
yy = spline(x, y, xx)
pp = spline(x, y)
```

**Description** The `spline` function constructs a spline function which takes the value  $y(:, j)$  at the point  $x(j)$ , all  $j$ . In particular, the given values may be vectors, in which case the spline function describes a curve that passes through the point sequence  $y(:, 1), y(:, 2), \dots$ .

`yy = spline(x, y, xx)` returns the value at `xx` of the interpolating cubic spline. If `xx` is a refinement of the mesh `x`, then `yy` provides a corresponding refinement of `y`.

`pp = spline(x, y)` returns the `pp`-form of the cubic spline interpolant, for later use with `ppval` (and with functions available in the Spline Toolbox).

Ordinarily, the ‘not-a-knot’ end conditions are used. However, if `y` contains exactly two more values than `x` has entries, then `y(:, 1)` and `y(:, end)` are used as the endslopes for the cubic spline.

**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
```

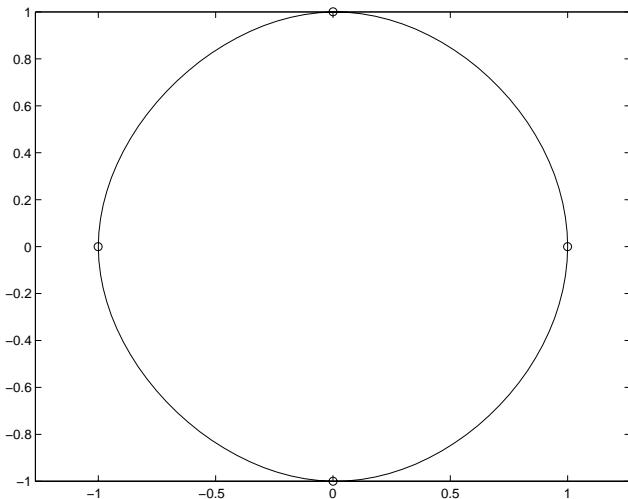
# spline

---

The statements

```
x = pi*[0:.5:2]; y = [0 1 0 -1 0 1 0; 1 0 1 0 -1 0 1];
pp = spline(x, y);
yy = ppval(pp, linspace(0, 2*pi, 101));
plot(yy(1, :), yy(2, :), '-b', y(1, 2:5), y(2, 2:5), 'or'), axis equal
```

generate the plot of a circle, with the five data points  $y(:, 2)$ , ...,  $y(:, 6)$  marked with o's. Note that this y contains two more values (i.e., two more columns) than does x, hence  $y(:, 1)$  and  $y(:, \text{end})$  are used as endslopes.



## Algorithm

A tridiagonal linear system (with, possibly, several right sides) is being solved for the information needed to describe the coefficients of the various cubic polynomials which make up the interpolating spline. `spline` uses the functions `ppval`, `mkpp`, and `unmkpp`. These routines form a small suite of functions for working with piecewise polynomials. `spline` uses these functions in a fairly simple fashion to perform cubic spline interpolation. For access to the more advanced features, see the on-line help for these M-files and the Spline Toolbox.

**See Also**

`interp1` One-dimensional data interpolation (table lookup)  
`interp2` Two-dimensional data interpolation (table lookup)  
`interp3` Three-dimensional data interpolation (table lookup)  
`interpn` Multidimensional data interpolation (table lookup)  
`ppval` Evaluate piecewise polynomial

**References**

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

# spones

---

<b>Purpose</b>	Replace nonzero sparse matrix elements with ones
<b>Syntax</b>	<code>R = spones(S)</code>
<b>Description</b>	<code>R = spones(S)</code> generates a matrix <code>R</code> with the same sparsity structure as <code>S</code> , but with 1's in the nonzero positions.
<b>Examples</b>	<code>c = sum(spones(S))</code> is the number of nonzeros in each column. <code>r = sum(spones(S'))'</code> is the number of nonzeros in each row. <code>sum(c)</code> and <code>sum(r)</code> are equal, and are equal to <code>nnz(S)</code> .
<b>See Also</b>	<code>nnz</code> Number of nonzero matrix elements <code>spalloc</code> Allocate space for sparse matrix <code>spfun</code> Apply function to nonzero sparse matrix elements

<b>Purpose</b>	Set parameters for sparse matrix routines
<b>Syntax</b>	<pre>spparms('key', value) spparms values = spparms [keys, values] = spparms spparms(values) value = spparms('key') spparms('default') spparms('tight')</pre>
<b>Description</b>	<p><code>spparms('key', value)</code> sets one or more of the <i>tunable</i> parameters used in the sparse linear equation operators, \ and /, and the minimum degree orderings, <code>colmmd</code> and <code>symmmd</code>. In ordinary use, you should never need to deal with this function.</p> <p>The meanings of the key parameters are</p> <ul style="list-style-type: none"> <li>'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.</li> <li>'thr_rel',      Minimum degree threshold is <code>thr_rel * mndegree + thr_abs</code>.</li> <li>'thr_abs'</li> <li>'exact_d'      Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.</li> <li>'supernd'      If positive, minimum degree amalgamates the supernodes every supernd stages.</li> <li>'rreduce'      If positive, minimum degree does row reduction every rreduce stages.</li> <li>'wh_frac'      Rows with density &gt; <code>wh_frac</code> are ignored in <code>colmmd</code>.</li> </ul>

## spparms

---

'autommd' Nonzero to use minimum degree orderings with \ and /.

'aug\_rel', Residual scaling parameter for augmented equations is

'aug\_abs'  $\text{aug\_rel} * \max(\max(\text{abs}(A))) + \text{aug\_abs}$ .

For example,  $\text{aug\_rel} = 0$ ,  $\text{aug\_abs} = 1$  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

	<b>Keyword</b>	<b>Default</b>	<b>Tight</b>
val ues(1)	' spumoni '	0.0	
val ues(2)	' thr_rel '	1.1	1.0
val ues(3)	' thr_abs'	1.0	0.0
val ues(4)	' exact_d'	0.0	1.0
val ues(5)	' supernd'	3.0	1.0
val ues(6)	' rreduce'	3.0	1.0
val ues(7)	' wh_frac'	0.5	0.5
val ues(8)	' autommd'	1.0	
val ues(9)	' aug_rel '	0.001	
val ues(10)	' aug_abs'	0.0	

**See Also**

\ Matrix left division (backslash)  
 col mmd 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.

# sprand

---

<b>Purpose</b>	Sparse uniformly distributed random matrix
<b>Syntax</b>	$R = \text{sprand}(S)$ $R = \text{sprand}(m, n, \text{density})$ $R = \text{sprand}(m, n, \text{density}, rc)$
<b>Description</b>	$R = \text{sprand}(S)$ has the same sparsity structure as $S$ , but uniformly distributed random entries.  $R = \text{sprand}(m, n, \text{density})$ is a random, $m$ -by- $n$ , sparse matrix with approximately $\text{density} \times m \times n$ uniformly distributed nonzero entries ( $0 \leq \text{density} \leq 1$ ).  $R = \text{sprand}(m, n, \text{density}, rc)$ 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 $1:r$ , where $1:r \leq \min(m, n)$ , then $R$ has $rc$ as its first $1:r$ 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.
<b>See Also</b>	<a href="#">sprandn</a> Sparse normally distributed random matrix <a href="#">sprandsym</a> Sparse symmetric random matrix

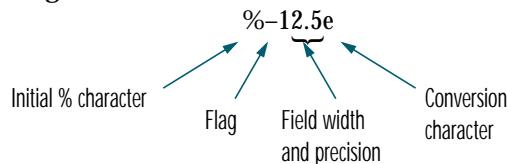
<b>Purpose</b>	Sparse normally distributed random matrix
<b>Syntax</b>	$R = \text{sprandn}(S)$ $R = \text{sprandn}(m, n, density)$ $R = \text{sprandn}(m, n, density, rc)$
<b>Description</b>	$R = \text{sprandn}(S)$ has the same sparsity structure as $S$ , but normally distributed random entries with mean 0 and variance 1.  $R = \text{sprandn}(m, n, density)$ is a random, $m$ -by- $n$ , sparse matrix with approximately $\text{density} \times m \times n$ normally distributed nonzero entries ( $0 \leq \text{density} \leq 1$ ).  $R = \text{sprandn}(m, n, density, rc)$ 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 $1 \leq \min(m, n)$ , then $R$ has $rc$ as its first $1 \times r$ 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.
<b>See Also</b>	<a href="#">sprand</a> Sparse uniformly distributed random matrix <a href="#">sprandn</a> Sparse normally distributed random matrix

# sprandsym

---

<b>Purpose</b>	Sparse symmetric random matrix
<b>Syntax</b>	$R = \text{sprandsym}(S)$ $R = \text{sprandsym}(n, \text{density})$ $R = \text{sprandsym}(n, \text{density}, rc)$ $R = \text{sprandsym}(n, \text{density}, rc, kind)$
<b>Description</b>	<p><math>R = \text{sprandsym}(S)</math> returns a symmetric random matrix whose lower triangle and diagonal have the same structure as <math>S</math>. Its elements are normally distributed, with mean 0 and variance 1.</p> <p><math>R = \text{sprandsym}(n, \text{density})</math> returns a symmetric random, <math>n</math>-by-<math>n</math>, sparse matrix with approximately <math>\text{density} \cdot n \cdot n</math> nonzeros; each entry is the sum of one or more normally distributed random samples, and <math>(0 \leq \text{density} \leq 1)</math>.</p> <p><math>R = \text{sprandsym}(n, \text{density}, rc)</math> returns a matrix with a reciprocal condition number equal to <math>rc</math>. The distribution of entries is nonuniform; it is roughly symmetric about 0; all are in <math>[-1, 1]</math>.</p> <p>If <math>rc</math> is a vector of length <math>n</math>, then <math>R</math> has eigenvalues <math>rc</math>. Thus, if <math>rc</math> is a positive (nonnegative) vector then <math>R</math> is a positive definite matrix. In either case, <math>R</math> 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.</p> <p><math>R = \text{sprandsym}(n, \text{density}, rc, kind)</math> returns a positive definite matrix.</p> <p>Argument <math>kind</math> can be:</p> <ul style="list-style-type: none"><li>• 1 to generate <math>R</math> by random Jacobi rotation of a positive definite diagonal matrix. <math>R</math> has the desired condition number exactly.</li><li>• 2 to generate an <math>R</math> that is a shifted sum of outer products. <math>R</math> has the desired condition number only approximately, but has less structure.</li><li>• 3 to generate an <math>R</math> that has the same structure as the matrix <math>S</math> and approximate condition number <math>1/rc</math>. <math>\text{density}</math> is ignored.</li></ul>
<b>See Also</b>	<a href="#">sprand</a> Sparse uniformly distributed random matrix <a href="#">sprandn</a> Sparse normally distributed random matrix

<b>Purpose</b>	Write formatted data to a string
<b>Syntax</b>	<code>s = sprintf(format, A, ...)</code> <code>[s, errmsg] = sprintf(format, A, ...)</code>
<b>Description</b>	<code>s = sprintf(format, A, ...)</code> formats the data in matrix A (and in any additional matrix arguments) under control of the specified <i>format</i> string, and returns it in the MATLAB string variable s. <code>sprintf</code> is the same as <code>fprintf</code> except that it returns the data in a MATLAB string variable rather than writing it to a file.
	The <i>format</i> 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, errmsg] = sprintf(format, A, ...)` returns an error message string `errmsg` if an error occurred or an empty matrix if an error did not occur.

<b>Remarks</b>	The <code>sprintf</code> function behaves like its ANSI C language <code>sprintf()</code> 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 " (two single quotes)	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.

# sprintf

## 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' , ' hel l o')	hel l o
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.

---

<b>Purpose</b>	Visualize sparsity pattern
<b>Syntax</b>	<code>spy(S)</code> <code>spy(S, markersize)</code> <code>spy(S, 'LineSpec')</code> <code>spy(S, 'LineSpec', markersize)</code>
<b>Description</b>	<p><code>spy(S)</code> plots the sparsity pattern of any matrix <code>S</code>.</p> <p><code>spy(S, markersize)</code>, where <code>markersize</code> is an integer, plots the sparsity pattern using markers of the specified point size.</p> <p><code>spy(S, 'LineSpec')</code>, where <code>LineSpec</code> is a string, uses the specified plot marker type and color.</p> <p><code>spy(S, 'LineSpec', markersize)</code> uses the specified type, color, and size for the plot markers.</p> <p><code>S</code> is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.</p> <p><code>spy</code> replaces <code>format +</code>, which takes much more space to display essentially the same information.</p>

**See Also** The `gplot` and `LineSpec` reference entries in the *MATLAB Graphics Guide*, and:

<code>find</code>	Find indices and values of nonzero elements
<code>symmmd</code>	Sparse symmetric minimum degree ordering
<code>symrcm</code>	Sparse reverse Cuthill-McKee ordering

# **sqrt**

---

<b>Purpose</b>	Square root
<b>Syntax</b>	<code>B = sqrt(A)</code>
<b>Description</b>	<code>B = sqrt(A)</code> returns the square root of each element of the array X. For the elements of X that are negative or complex, <code>sqrt(X)</code> produces complex results.
<b>Remarks</b>	See <code>sqrtm</code> for the matrix square root.
<b>Examples</b>	<pre>sqrt((-2:2)') ans =     0 + 1.4142i     0 + 1.0000i     0     1.0000     1.4142</pre>
<b>See Also</b>	<code>sqrtm</code> Matrix square root

**Purpose**

Matrix square root

**Syntax**

$Y = \text{sqrtm}(X)$   
 $[Y, \text{esterr}] = \text{sqrtm}(X)$

**Description**

$Y = \text{sqrtm}(X)$  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  $X$ .

$[Y, \text{esterr}] = \text{sqrtm}(X)$  does not print any warning message, but returns an estimate of the relative residual,  $\text{norm}(Y^*Y - X) / \text{norm}(X)$ .

**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

$$X = \begin{matrix} & 5 & -4 & 1 & 0 & 0 \\ 5 & & -4 & 6 & -4 & 1 \\ -4 & 6 & & -4 & 1 & 0 \\ 1 & -4 & 6 & & -4 & 1 \\ 0 & 1 & -4 & 6 & & -4 \\ 0 & 0 & 1 & -4 & 5 & \end{matrix}$$

This matrix is symmetric and positive definite. Its unique positive definite square root,  $Y = \text{sqrtm}(X)$ , is a representation of the second difference operator.

$$Y = \begin{matrix} & 2 & -1 & -0 & 0 & -0 \\ 2 & & 2 & -1 & -0 & -0 \\ -1 & 2 & & -1 & -0 & -0 \\ -0 & -1 & 2 & & -1 & 0 \\ 0 & -0 & -1 & 2 & & -1 \\ -0 & -0 & 0 & -1 & 2 & \end{matrix}$$

The matrix

$$X = \begin{matrix} & 7 & 10 \\ 7 & & \\ 15 & 22 & \end{matrix}$$

has four square roots. Two of them are

$$Y_1 = \begin{matrix} & 1.5667 & 1.7408 \\ 1.5667 & & \\ 2.6112 & 4.1779 & \end{matrix}$$

and

$$Y_2 = \begin{matrix} & 1 & 2 \\ 1 & & \\ 3 & 4 & \end{matrix}$$

The other two are  $-Y_1$  and  $-Y_2$ . All four can be obtained from the eigenvalues and vectors of  $X$ .

$$\begin{aligned} [V, D] &= \text{eig}(X); \\ D &= \begin{matrix} & 0.1386 & 0 \\ 0 & & 28.8614 \end{matrix} \end{aligned}$$

The four square roots of the diagonal matrix  $D$  result from the four choices of sign in

$$S = \begin{matrix} & \pm 0.3723 & 0 \\ \pm 0.3723 & & \\ 0 & & \pm 5.3723 \end{matrix}$$

All four  $Y$ s are of the form

$$Y = V^* S V$$

The `sqrtm` function chooses the two plus signs and produces  $Y_1$ , even though  $Y_2$  is more natural because its entries are integers.

Finally, the matrix

$$X = \begin{matrix} & 0 & 1 \\ 0 & & 0 \\ 0 & 0 & \end{matrix}$$

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

<code>expm</code>	Matrix exponential
<code>funm</code>	Evaluate functions of a matrix
<code>logm</code>	Matrix logarithm

# squeeze

---

<b>Purpose</b>	Remove singleton dimensions
<b>Syntax</b>	<code>B = squeeze(A)</code>
<b>Description</b>	<code>B = squeeze(A)</code> returns an array B with the same elements as A, but with all singleton dimensions removed. A singleton dimension is any dimension for which <code>size(A, dim) = 1</code> .
<b>Examples</b>	Consider the 2-by-1-by-3 array <code>Y = rand(2, 1, 3)</code> . This array has a singleton column dimension — that is, there's only one column per page.  <code>Y =</code>  <code>Y(:,:,1) =</code> <code>Y(:,:,2) =</code> 0.5194                  0.0346 0.8310                  0.0535  <code>Y(:,:,3) =</code> 0.5297 0.6711
	The command <code>Z = squeeze(Y)</code> yields a 2-by-3 matrix:  <code>Z =</code> 0.5194    0.0346    0.5297 0.8310    0.0535    0.6711
<b>See Also</b>	<code>reshape</code> Reshape array <code>shiftdim</code> Shift dimensions

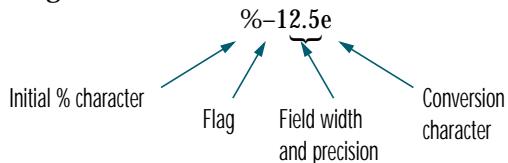
<b>Purpose</b>	Read string under format control
<b>Syntax</b>	$A = \text{sscanf}(s, format)$ $A = \text{sscanf}(s, format, size)$ $[A, count, errmsg, nextindex] = \text{sscanf}(\dots)$
<b>Description</b>	<p><math>A = \text{sscanf}(s, format)</math> reads data from the MATLAB string variable <math>s</math>, converts it according to the specified <math>format</math> string, and returns it in matrix <math>A</math>. <math>format</math> is a string specifying the format of the data to be read. See “Remarks” for details. <math>\text{sscanf}</math> is the same as <math>\text{fscanf}</math> except that it reads the data from a MATLAB string variable rather than reading it from a file.</p> <p><math>A = \text{sscanf}(s, format, size)</math> reads the amount of data specified by <math>size</math> and converts it according to the specified <math>format</math> string. <math>size</math> is an argument that determines how much data is read. Valid options are:</p> <ul style="list-style-type: none"> <li><math>n</math> Read <math>n</math> elements into a column vector.</li> <li><math>inf</math> Read to the end of the file, resulting in a column vector containing the same number of elements as are in the file.</li> <li><math>[m, n]</math> Read enough elements to fill an <math>m</math>-by-<math>n</math> matrix, filling the matrix in column order. <math>n</math> can be <math>Inf</math>, but not <math>m</math>.</li> </ul> <p>If the matrix <math>A</math> results from using character conversions only and <math>size</math> is not of the form <math>[M, N]</math>, a row vector is returned.</p> <p><math>\text{sscanf}</math> differs from its C language namesakes <math>\text{scanf}()</math> and <math>\text{fscanf}()</math> in an important respect — it is <i>vectorized</i> in order to return a matrix argument. The <math>format</math> string is cycled through the file until an end-of-file is reached or the amount of data specified by <math>size</math> is read in.</p> <p><math>[A, count, errmsg, nextindex] = \text{sscanf}(\dots)</math> reads data from MATLAB string variable <math>s</math>, converts it according to the specified <math>format</math> string, and returns it in matrix <math>A</math>. <math>count</math> is an optional output argument that returns the number of elements successfully read. <math>errmsg</math> 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. <math>nextindex</math> is an optional output argument specifying one more than the number of characters scanned in <math>s</math>.</p>

## sscanf

### 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

<code>eval</code>	Interpret strings containing MATLAB expressions
<code>sprintf</code>	Write formatted data to a string

# startup

---

<b>Purpose</b>	MATLAB startup M-file										
<b>Syntax</b>	<code>startup</code>										
<b>Description</b>	<p>At startup time, MATLAB automatically executes the master M-file <code>matlabrc.m</code> and, if it exists, <code>startup.m</code>. On multiuser or networked systems, <code>matlabrc.m</code> is reserved for use by the system manager. The file <code>matlabrc.m</code> invokes the file <code>startup.m</code> if it exists on MATLAB's search path.</p> <p>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.</p>										
<b>Algorithm</b>	<p>Only <code>matlabrc.m</code> is actually invoked by MATLAB at startup. However, <code>matlabrc.m</code> contains the statements</p> <pre>if exist('startup') == 2     startup end</pre> <p>that invoke <code>startup.m</code>. You can extend this process to create additional startup M-files, if required.</p>										
<b>See Also</b>	<table><tr><td><code>!</code></td><td>Operating system command</td></tr><tr><td><code>exist</code></td><td>Check if a variable or file exists</td></tr><tr><td><code>matlabrc</code></td><td>MATLAB startup M-file</td></tr><tr><td><code>path</code></td><td>Control MATLAB's directory search path</td></tr><tr><td><code>quit</code></td><td>Terminate MATLAB</td></tr></table>	<code>!</code>	Operating system command	<code>exist</code>	Check if a variable or file exists	<code>matlabrc</code>	MATLAB startup M-file	<code>path</code>	Control MATLAB's directory search path	<code>quit</code>	Terminate MATLAB
<code>!</code>	Operating system command										
<code>exist</code>	Check if a variable or file exists										
<code>matlabrc</code>	MATLAB startup M-file										
<code>path</code>	Control MATLAB's directory search path										
<code>quit</code>	Terminate MATLAB										

<b>Purpose</b>	Standard deviation
<b>Syntax</b>	$s = \text{std}(X)$ $s = \text{std}(X, \text{flag})$ $s = \text{std}(X, \text{flag}, \text{dim})$
<b>Definition</b>	There are two common textbook definitions for the standard deviation $s$ of a data vector $X$ :
	(1) $s = \left( \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^{\frac{1}{2}}$ and      (2) $s = \left( \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^{\frac{1}{2}}$
	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.
<b>Description</b>	$s = \text{std}(X)$ , 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 <i>unbiased</i> estimate of its variance.  If $X$ is a matrix, $\text{std}(X)$ returns a row vector containing the standard deviation of the elements of each column of $X$ . If $X$ is a multidimensional array, $\text{std}(X)$ is the standard deviation of the elements along the first nonsingleton dimension of $X$ .
	$s = \text{std}(X, \text{flag})$ for $\text{flag} = 0$ , is the same as $\text{std}(X)$ . For $\text{flag} = 1$ , $\text{std}(X, 1)$ returns the standard deviation using (2) above, producing the second moment of the sample about its mean.
	$s = \text{std}(X, \text{flag}, \text{dim})$ computes the standard deviations along the dimension of $X$ specified by scalar $\text{dim}$ .

## std

---

### 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

---

<b>Purpose</b>	String to number conversion												
<b>Syntax</b>	<code>x = str2num('str')</code>												
<b>Description</b>	<code>x = str2num('str')</code> converts the string <i>str</i> , which is an ASCII character representation of a numeric value, to MATLAB's numeric representation. The string can contain:												
	<ul style="list-style-type: none"> <li>• Digits</li> <li>• A decimal point</li> <li>• A leading + or – sign</li> <li>• A letter e preceding a power of 10 scale factor</li> <li>• A letter i indicating a complex or imaginary number.</li> </ul>												
	The <code>str2num</code> function can also convert string matrices.												
<b>Examples</b>	<p><code>str2num('3.14159e0')</code> is approximately <math>\pi</math>.</p> <p>To convert a string matrix:</p> <pre>str2num(['1 2'; '3 4'])</pre> <pre>ans =     1     2     3     4</pre>												
<b>See Also</b>	<table border="0"> <tr> <td><code>[]</code> (special characters)</td> <td>Build arrays</td> </tr> <tr> <td><code>;</code> (special characters)</td> <td>End array rows; suppress printing; separate statements.</td> </tr> <tr> <td><code>hex2num</code></td> <td>Hexadecimal to double number conversion</td> </tr> <tr> <td><code>num2str</code></td> <td>Number to string conversion</td> </tr> <tr> <td><code>sparse</code></td> <td>Create sparse matrix</td> </tr> <tr> <td><code>sscanf</code></td> <td>Read string under format control</td> </tr> </table>	<code>[]</code> (special characters)	Build arrays	<code>;</code> (special characters)	End array rows; suppress printing; separate statements.	<code>hex2num</code>	Hexadecimal to double number conversion	<code>num2str</code>	Number to string conversion	<code>sparse</code>	Create sparse matrix	<code>sscanf</code>	Read string under format control
<code>[]</code> (special characters)	Build arrays												
<code>;</code> (special characters)	End array rows; suppress printing; separate statements.												
<code>hex2num</code>	Hexadecimal to double number conversion												
<code>num2str</code>	Number to string conversion												
<code>sparse</code>	Create sparse matrix												
<code>sscanf</code>	Read string under format control												

# strcat

---

<b>Purpose</b>	String concatenation
<b>Syntax</b>	<code>t = strcat(s1, s2, s3, ...)</code>
<b>Description</b>	<p><code>t = strcat(s1, s2, s3, ...)</code> horizontally concatenates corresponding rows of the character arrays <code>s1, s2, s3</code>, 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.</p> <p>When any of the inputs is a cell array of strings, <code>strcat</code> returns a cell array of strings formed by concatenating corresponding elements of <code>s1, s2</code>, 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.</p>
<b>Examples</b>	<p>Given two 1-by-2 cell arrays <code>a</code> and <code>b</code>,</p> <pre>a =          b =     'abcde'      'fghi'           'j kl'      'mn'</pre> <p>the command <code>t = strcat(a, b)</code> yields:</p> <pre>t =     'abcdej kl'      'fghi mn'</pre> <p>Given the 1-by-1 cell array <code>c = {'Q'}</code>, the command <code>t = strcat(a, b, c)</code> yields:</p> <pre>t =     'abcdej kl Q'      'fghi mnQ'</pre>
<b>Remarks</b>	<p><code>strcat</code> and matrix operation are different for strings that contain trailing spaces:</p> <pre>a = 'hel l o ' b = 'goodby' strcat(a, b) ans = hel l o goodby [a b] ans = hel l o  goodby</pre>

**See Also**[cat](#)[cellstr](#)[strvcat](#)[Concatenate arrays](#)[Create cell array of strings from character array](#)[Vertical concatenation of strings](#)

# strcmp

---

<b>Purpose</b>	Compare strings
<b>Syntax</b>	<pre>k = strcmp('str1', 'str2') TF = strcmp(S, T)</pre>
<b>Description</b>	<p><code>k = strcmp('str1', 'str2')</code> compares the strings <code>str1</code> and <code>str2</code> and returns logical true (1) if the two are identical, and logical false (0) otherwise.</p> <p><code>TF = strcmp(S, T)</code> where either <code>S</code> or <code>T</code> is a cell array of strings, returns an array <code>TF</code> the same size as <code>S</code> and <code>T</code> containing 1 for those elements of <code>S</code> and <code>T</code> that match, and 0 otherwise. <code>S</code> and <code>T</code> 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.</p>
<b>Remarks</b>	<p>Note that the value returned by <code>strcmp</code> is not the same as the C language convention. In addition, the <code>strcmp</code> function is case sensitive; any leading and trailing blanks in either of the strings are explicitly included in the comparison.</p>

**Examples**

```
strcmp('Yes', 'No') =
    0
strcmp('Yes', 'Yes') =
    1

A =
    ' MATLAB'           ' SIMULINK'
    ' Tool boxes'       ' The MathWorks'

B =
    ' Handle Graphics' ' Real Time Workshop'
    ' Tool boxes'       ' 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**

<a href="#">findstr</a>	Find one string within another
<a href="#">strcmpi</a>	Compare strings ignoring case
<a href="#">strncmp</a>	Compare the first n characters of two strings
<a href="#">strmatch</a>	Find possible matches for a string

# strcmpi

---

<b>Purpose</b>	Compare strings ignoring case
<b>Syntax</b>	<code>strcmpi(str1, str2)</code> <code>strcmpi(S, T)</code>
<b>Description</b>	<code>strcmpi(str1, str2)</code> returns 1 if strings <i>str1</i> and <i>str2</i> are the same except for case and 0 otherwise.
	<code>strcmpi(S, T)</code> when either S or T is a cell array of strings, returns an array the same size as S and T containing 1 for those elements of S and T that match except for case, 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.
	<code>strcmpi</code> supports international character sets.
<b>See Also</b>	<code>findstr</code> , <code>strcmp</code> , <code>strmatch</code> , <code>strncmpi</code>

<b>Purpose</b>	MATLAB string handling				
<b>Syntax</b>	<pre>S = 'Any Characters' S = string(X) X = numeric(S)</pre>				
<b>Description</b>	<p><code>S = 'Any Characters'</code> 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 <code>S</code> is the number of characters. A quote within the string is indicated by two quotes.</p> <p><code>S = string(X)</code> can be used to convert an array that contains positive integers representing numeric codes into a MATLAB character array.</p> <p><code>X = double(S)</code> converts the string to its equivalent numeric codes.</p> <p><code>isstr(S)</code> tells if <code>S</code> is a string variable.</p> <p>Use the <code>strcat</code> function for concatenating cell arrays of strings, for arrays of multiple strings, and for padded character arrays. For concatenating two single strings, it is more efficient to use square brackets, as shown in the example, than to use <code>strcat</code>.</p>				
<b>Example</b>	<pre>s = ['It is 1 o''clock', 7]</pre>				
<b>See Also</b>	<table><tr><td><code>char</code></td><td>Create character array (string)</td></tr><tr><td><code>strcat</code></td><td>String concatenation</td></tr></table>	<code>char</code>	Create character array (string)	<code>strcat</code>	String concatenation
<code>char</code>	Create character array (string)				
<code>strcat</code>	String concatenation				

# strjust

---

<b>Purpose</b>	Justify a character array
<b>Syntax</b>	<code>T = strjust(S)</code> <code>T = strjust(S, 'right')</code> <code>T = strjust(S, 'left')</code> <code>T = strjust(S, 'center')</code>
<b>Description</b>	<code>T = strjust(S)</code> or <code>T = strjust(S, 'right')</code> returns a right-justified version of the character array <code>S</code> . <code>T = strjust(S, 'left')</code> returns a left-justified version of <code>S</code> . <code>T = strjust(S, 'center')</code> returns a center-justified version of <code>S</code> .
<b>See Also</b>	<code>deblank</code> Strip trailing blanks from the end of a string

<b>Purpose</b>	Find possible matches for a string								
<b>Syntax</b>	<pre>i = strmatch('str', STRS) i = strmatch('str', STRS, 'exact')</pre>								
<b>Description</b>	<p><code>i = strmatch('str', STRS)</code> looks through the rows of the character array or cell array of strings <code>STRS</code> to find strings that begin with string <code>str</code>, returning the matching row indices. <code>strmatch</code> is fastest when <code>STRS</code> is a character array.</p> <p><code>i = strmatch('str', STRS, 'exact')</code> returns only the indices of the strings in <code>STRS</code> matching <code>str</code> exactly.</p>								
<b>Examples</b>	<p>The statement</p> <pre>i = strmatch('max', strvcat('max', 'mini max', 'maximum'))</pre> <p>returns <code>i = [1; 3]</code> since rows 1 and 3 begin with '<code>max</code>'. The statement</p> <pre>i = strmatch('max', strvcat('max', 'mini max', 'maximum'), 'exact')</pre> <p>returns <code>i = 1</code>, since only row 1 matches '<code>max</code>' exactly.</p>								
<b>See Also</b>	<table><tr><td><code>findstr</code></td><td>Find one string within another</td></tr><tr><td><code>strcmp</code></td><td>Compare strings</td></tr><tr><td><code>strncmp</code></td><td>Compare the first n characters of two strings</td></tr><tr><td><code>strvcat</code></td><td>Vertical concatenation of strings</td></tr></table>	<code>findstr</code>	Find one string within another	<code>strcmp</code>	Compare strings	<code>strncmp</code>	Compare the first n characters of two strings	<code>strvcat</code>	Vertical concatenation of strings
<code>findstr</code>	Find one string within another								
<code>strcmp</code>	Compare strings								
<code>strncmp</code>	Compare the first n characters of two strings								
<code>strvcat</code>	Vertical concatenation of strings								

# strcmp

---

<b>Purpose</b>	Compare the first n characters of two strings										
<b>Syntax</b>	<pre>k = strcmp('str1', 'str2', n) TF = strcmp(S, T, n)</pre>										
<b>Description</b>	<p><code>k = strcmp('str1', 'str2', n)</code> returns logical true (1) if the first n characters of the strings <code>str1</code> and <code>str2</code> are the same, and returns logical false (0) otherwise. Arguments <code>str1</code> and <code>str2</code> may also be cell arrays of strings.</p> <p><code>TF = strcmp(S, T, N)</code> 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.</p>										
<b>Remarks</b>	The command <code>strcmp</code> is case sensitive. Any leading and trailing blanks in either of the strings are explicitly included in the comparison.										
<b>See Also</b>	<table><tr><td><code>findstr</code></td><td>Find one string within another</td></tr><tr><td><code>strcmp</code></td><td>Compare strings</td></tr><tr><td><code>strcmpi</code></td><td>Compare strings ignoring case</td></tr><tr><td><code>strmatch</code></td><td>Find possible matches for a string</td></tr><tr><td><code>strncmpi</code></td><td>Compare first n characters of strings ignoring case</td></tr></table>	<code>findstr</code>	Find one string within another	<code>strcmp</code>	Compare strings	<code>strcmpi</code>	Compare strings ignoring case	<code>strmatch</code>	Find possible matches for a string	<code>strncmpi</code>	Compare first n characters of strings ignoring case
<code>findstr</code>	Find one string within another										
<code>strcmp</code>	Compare strings										
<code>strcmpi</code>	Compare strings ignoring case										
<code>strmatch</code>	Find possible matches for a string										
<code>strncmpi</code>	Compare first n characters of strings ignoring case										

---

<b>Purpose</b>	Compare first n characters of strings ignoring case
<b>Syntax</b>	<code>strcmpi ('str1', 'str2', n)</code> <code>TF = strcmpi (S, T, n)</code>
<b>Description</b>	<code>strcmpi ('str1', 'str2', n)</code> returns 1 if the first n characters of the strings <i>str1</i> and <i>str2</i> are the same except for case, and 0 otherwise.  <code>TF = strcmpi (S, T, n)</code> when either S or T is a cell array of strings, returns an array the same size as S and T containing 1 for those elements of S and T that match except for case (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.  <code>strcmpi</code> supports international character sets.
<b>See Also</b>	<code>findstr</code> , <code>strmatch</code> , <code>strcmp</code> , <code>strcmpi</code>

# strrep

---

<b>Purpose</b>	String search and replace
<b>Syntax</b>	<code>str = strrep(str1, str2, str3)</code>
<b>Description</b>	<code>str = strrep(str1, str2, str3)</code> replaces all occurrences of the string <i>str2</i> within string <i>str1</i> with the string <i>str3</i> .
	<code>strrep(str1, str2, str3)</code> , when any of <i>str1</i> , <i>str2</i> , or <i>str3</i> is a cell array of strings, returns a cell array the same size as <i>str1</i> , <i>str2</i> and <i>str3</i> obtained by performing a <code>strrep</code> 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.
<b>Examples</b>	<pre>s1 = 'This is a good example.'; str = strrep(s1, 'good', 'great') str = This is a great example.  A =     ' MATLAB'           ' SIMULINK'     ' Tool boxes'       ' The MathWorks'  B =     ' Handle Graphics' ' Real Time Workshop'     ' Tool boxes'       ' The MathWorks'  C =     ' Signal Processing' ' Image Processing'     ' MATLAB'           ' SIMULINK'  <code>strrep(A, B, C)</code> ans =     ' MATLAB'           ' SIMULINK'     ' MATLAB'           ' SIMULINK'</pre>
<b>See Also</b>	<a href="#">findstr</a> Find one string within another

<b>Purpose</b>	First token in string
<b>Syntax</b>	<pre>token = strtok('str', delimiter) token = strtok('str') [token, rem] = strtok(...)</pre>
<b>Description</b>	<p><code>token = strtok('str', delimiter)</code> returns the first token in the text string <i>str</i>, that is, the first set of characters before a delimiter is encountered. The vector <code>delimiter</code> contains valid delimiter characters.</p> <p><code>token = strtok('str')</code> uses the default delimiters, the white space characters. These include tabs (ASCII 9), carriage returns (ASCII 13), and spaces (ASCII 32).</p> <p><code>[token, rem] = strtok(...)</code> returns the remainder <code>rem</code> of the original string. The remainder consists of all characters from the first delimiter on.</p>
<b>Examples</b>	<pre>s = 'This is a good example.'; [token, rem] = strtok(s) token = This rem =     is a good example.</pre>
<b>See Also</b>	<p><a href="#">findstr</a>      Find one string within another <a href="#">strmatch</a>      Find possible matches for a string</p>

# struct

---

<b>Purpose</b>	Create structure array
<b>Syntax</b>	<code>s = struct('field1', values1, 'field2', values2, ...)</code>
<b>Description</b>	<code>s = struct('field1', values1, 'field2', values2, ...)</code> creates a structure array with the specified fields and values. The value arrays <code>values1</code> , <code>values2</code> , 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 cell arrays or 1-by-1 if none of the values is a cell.
<b>Examples</b>	The command  <code>s = struct('type', {'big', 'little'}, 'color', {'red'}, 'x', {3 4})</code> produces a structure array <code>s</code> :  <code>s = 1x2 struct array with fields:     type     color     x</code>  The value arrays have been distributed among the fields of <code>s</code> :  <code>s(1) ans =     type: 'big'     color: 'red'     x: 3</code>  <code>s(2) ans =     type: 'little'     color: 'red'     x: 4</code>
<b>See Also</b>	<code>fieldnames</code> Field names of a structure <code>getfield</code> Get field of structure array <code>rmfield</code> Remove structure fields <code>setfield</code> 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

---

<b>Purpose</b>	Vertical concatenation of strings
<b>Syntax</b>	<code>S = strvcat(t1, t2, t3, ...)</code>
<b>Description</b>	<code>S = strvcat(t1, t2, t3, ...)</code> 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.
<b>Remarks</b>	If each text parameter, $t_i$ , is itself a character array, strvcat appends them vertically to create arbitrarily large string matrices.
<b>Examples</b>	The command <code>strvcat('Hello', 'Yes')</code> is the same as <code>['Hello'; 'Yes']</code> , except that strvcat performs the padding automatically.  <code>t1 = 'first'; t2 = 'string'; t3 = 'matrix'; t4 = 'second';</code>  <code>S1 = strvcat(t1, t2, t3)</code> <code>S2 = strvcat(t4, t2, t3)</code>  <code>S1 =</code> <code>S2 =</code>  <code>first</code> <code>second</code> <code>string</code> <code>string</code> <code>matrix</code> <code>matrix</code>  <code>S3 = strvcat(S1, S2)</code>  <code>S3 =</code> <code>first</code> <code>string</code> <code>matrix</code> <code>second</code> <code>string</code> <code>matrix</code>
<b>See Also</b>	<code>cat</code> Concatenate arrays <code>int2str</code> Integer to string conversion <code>mat2str</code> Convert a matrix into a string <code>num2str</code> Number to string conversion <code>string</code> Convert numeric values to string

**Purpose** Single index from subscripts

**Syntax**

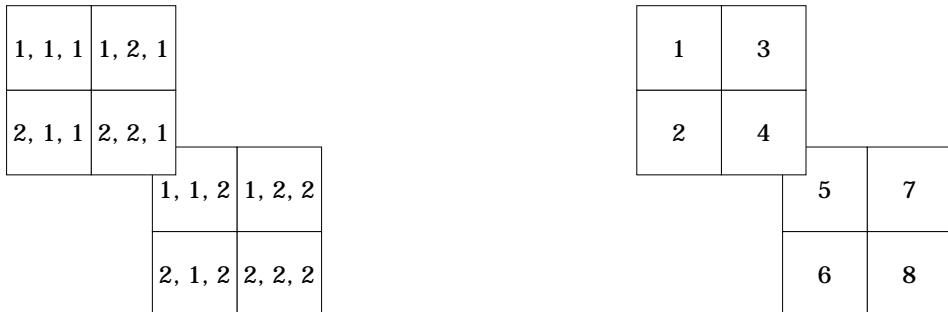
```
IND = sub2ind(sz, I, J)
IND = sub2ind(sz, I1, I2, ..., In)
```

**Description** The `sub2ind` command determines the equivalent single index corresponding to a set of subscript values.

`IND = sub2ind(sz, I, J)` returns the linear index equivalent to the row and column subscripts in the arrays `I` and `J` for an matrix of size `sz`.

`IND = sub2ind(sz, I1, I2, ..., In)` returns the linear index equivalent to the `n` subscripts in the arrays `I1, I2, ..., In` for an array of size `sz`.

**Examples** The mapping from subscripts to linear index equivalents for a 2-by-2-by-2 array is:



**See Also** `ind2sub`, `find`, Subscripts from linear index, Find indices and values of nonzero elements

# subsasgn

---

<b>Purpose</b>	Overloaded method for $A(i) = B$ , $A\{i\} = B$ , and $A.\text{field} = B$
<b>Syntax</b>	$A = \text{subsasgn}(A, S, B)$
<b>Description</b>	$A = \text{subsasgn}(A, S, B)$ is called for the syntax $A(i) = B$ , $A\{i\} = B$ , or $A.\text{field} = B$ when $A$ is an object. $S$ is a structure array with the fields:
	<ul style="list-style-type: none"><li>• type: A string containing '<math>()</math>', '<math>\{\}</math>', or '<math>.</math>', where '<math>()</math>' specifies integer subscripts; '<math>\{\}</math>' specifies cell array subscripts, and '<math>.</math>' specifies subscripted structure fields.</li><li>• subs: A cell array or string containing the actual subscripts.</li></ul>
<b>Examples</b>	The syntax $A(1:2,:) = B$ calls $A = \text{subsasgn}(A, S, B)$ where $S$ is a 1-by-1 structure with $S.\text{type} = '()$ and $S.\text{subs} = \{1:2, ':'\}$ . A colon used as a subscript is passed as the string ' $:$ '. The syntax $A\{1:2\} = B$ calls $A = \text{subsasgn}(A, S, B)$ where $S.\text{type} = '\{\}'$ . The syntax $A.\text{field} = B$ calls $\text{subsasgn}(A, S, B)$ where $S.\text{type} = '.'$ and $S.\text{subs} = '\text{field}'$ . These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases $\text{length}(S)$ is the number of subscripting levels. For instance, $A(1,2).\text{name}(3:5) = B$ calls $A = \text{subsasgn}(A, S, B)$ where $S$ is 3-by-1 structure array with the following values: $\begin{array}{lll} S(1).\text{type} = '()' & S(2).\text{type} = '.' & S(3).\text{type} = '()' \\ S(1).\text{subs} = \{1, 2\} & S(2).\text{subs} = '\text{name}' & S(3).\text{subs} = \{3:5\} \end{array}$
<b>See Also</b>	<a href="#">subsref</a> Overloaded method for $A(i)$ , $A\{i\}$ and $A.\text{field}$ See <i>Using MATLAB</i> for more information about overloaded methods and $\text{subsasgn}$ .

---

<b>Purpose</b>	Overloaded method for X(A)
<b>Syntax</b>	<code>i = subsindex(A)</code>
<b>Description</b>	<p><code>i = subsindex(A)</code> is called for the syntax '<code>X(A)</code>' when <code>A</code> is an object.</p> <p><code>subsindex</code> must return the value of the object as a zero-based integer index (<code>i</code> must contain integer values in the range 0 to <code>prod(size(X))-1</code>). <code>subsindex</code> is called by the default <code>subsref</code> and <code>subsasgn</code> functions, and you can call it if you overload these functions.</p>
<b>See Also</b>	<p><code>subsasgn</code>      Overloaded method for <code>A(i)=B</code>, <code>A{i}=B</code>, and <code>A.field=B</code></p> <p><code>subsref</code>      Overloaded method for <code>A(i)</code>, <code>A{i}</code> and <code>A.field</code></p>

# subsref

---

<b>Purpose</b>	Overloaded method for A(I), A{I} and A.field						
<b>Syntax</b>	<code>B = subsref(A, S)</code>						
<b>Description</b>	<code>B = subsref(A, S)</code> is called for the syntax <code>A(i)</code> , <code>A{i}</code> , or <code>A.i</code> when <code>A</code> is an object. <code>S</code> is a structure array with the fields:						
	<ul style="list-style-type: none"><li>• type: A string containing '<code>()</code>', '<code>{}</code>', or '<code>.</code>', where '<code>()</code>' specifies integer subscripts; '<code>{}</code>' specifies cell array subscripts, and '<code>.</code>' specifies subscripted structure fields.</li><li>• subs: A cell array or string containing the actual subscripts.</li></ul>						
<b>Examples</b>	<p>The syntax <code>A(1:2, :)</code> calls <code>subsref(A, S)</code> where <code>S</code> is a 1-by-1 structure with <code>S.type='()'</code> and <code>S.subs = {1:2, ':'}</code>. A colon used as a subscript is passed as the string '<code>:</code>'.</p> <p>The syntax <code>A{1:2}</code> calls <code>subsref(A, S)</code> where <code>S.type='{}'</code>.</p> <p>The syntax <code>A.field</code> calls <code>subsref(A, S)</code> where <code>S.type='.'</code> and <code>S.subs='field'</code>.</p> <p>These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases <code>length(S)</code> is the number of subscripting levels. For instance, <code>A(1, 2).name(3:5)</code> calls <code>subsref(A, S)</code> where <code>S</code> is 3-by-1 structure array with the following values:</p> <table><tr><td><code>S(1).type='()'</code></td><td><code>S(2).type='.'</code></td><td><code>S(3).type='()'</code></td></tr><tr><td><code>S(1).subs={1, 2}</code></td><td><code>S(2).subs='name'</code></td><td><code>S(3).subs={3:5}</code></td></tr></table>	<code>S(1).type='()'</code>	<code>S(2).type='.'</code>	<code>S(3).type='()'</code>	<code>S(1).subs={1, 2}</code>	<code>S(2).subs='name'</code>	<code>S(3).subs={3:5}</code>
<code>S(1).type='()'</code>	<code>S(2).type='.'</code>	<code>S(3).type='()'</code>					
<code>S(1).subs={1, 2}</code>	<code>S(2).subs='name'</code>	<code>S(3).subs={3:5}</code>					

**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`.

---

<b>Purpose</b>	Angle between two subspaces
<b>Syntax</b>	<code>theta = subspace(A, B)</code>
<b>Description</b>	<code>theta = subspace(A, B)</code> 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 $\arccos(A' * B)$ .
<b>Remarks</b>	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, <code>subspace(A, B)</code> gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.
<b>Examples</b>	Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.  <code>H = hadamard(8);</code> <code>A = H(:, 2: 4);</code> <code>B = H(:, 5: 8);</code>  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.  <code>theta = subspace(A, B)</code> <code>theta =</code> <code>1. 5708</code>  That A and B are orthogonal is shown by the fact that theta is equal to $\pi/2$ .  <code>theta = pi / 2</code> <code>ans =</code> <code>0</code>

# sum

---

<b>Purpose</b>	Sum of array elements
<b>Syntax</b>	$B = \text{sum}(A)$ $B = \text{sum}(A, \text{dim})$
<b>Description</b>	$B = \text{sum}(A)$ returns sums along different dimensions of an array. If $A$ is a vector, $\text{sum}(A)$ returns the sum of the elements. If $A$ is a matrix, $\text{sum}(A)$ treats the columns of $A$ as vectors, returning a row vector of the sums of each column. If $A$ is a multidimensional array, $\text{sum}(A)$ treats the values along the first non-singleton dimension as vectors, returning an array of row vectors. $B = \text{sum}(A, \text{dim})$ sums along the dimension of $A$ specified by scalar $\text{dim}$ .
<b>Remarks</b>	$\text{sum}(\text{diag}(X))$ is the trace of $X$ .
<b>Examples</b>	The magic square of order 3 is  $M = \text{magic}(3)$ $M =$ $\begin{matrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{matrix}$  This is called a magic square because the sums of the elements in each column are the same.  $\text{sum}(M) =$ $\begin{matrix} 15 & 15 & 15 \end{matrix}$ as are the sums of the elements in each row, obtained by transposing:  $\text{sum}(M') =$ $\begin{matrix} 15 & 15 & 15 \end{matrix}$
<b>See Also</b>	<a href="#">cumsum</a> <a href="#">diff</a> <a href="#">prod</a> <a href="#">trace</a>  <a href="#">Cumulative sum</a> <a href="#">Differences and approximate derivatives</a> <a href="#">Product of array elements</a> <a href="#">Sum of diagonal elements</a>

<b>Purpose</b>	Superior class relationship
<b>Syntax</b>	<code>superioro('class1', 'class2', ...)</code>
<b>Description</b>	The <code>superioro</code> function establishes a hierarchy that determines the order in which MATLAB calls object methods.  <code>superioro('class1', 'class2', ...)</code> invoked within a class constructor method (say <code>myclass.m</code> ) indicates that <code>myclass</code> 's method should be invoked if a function is called with an object of class <code>myclass</code> and one or more objects of class <code>class1</code> , <code>class2</code> , and so on.
<b>Remarks</b>	Suppose A is of class ' <code>class_a</code> ', B is of class ' <code>class_b</code> ' and C is of class ' <code>class_c</code> '. Also suppose the constructor <code>class_c.m</code> contains the statement: <code>superioro('class_a')</code> . Then <code>e = fun(a, c)</code> or <code>e = fun(c, a)</code> invokes <code>class_c.fun</code> .  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, <code>fun(b, c)</code> calls <code>class_b.fun</code> , while <code>fun(c, b)</code> calls <code>class_c.fun</code> .
<b>See Also</b>	<code>inferioro</code> Inferior class relationship

## svd

---

<b>Purpose</b>	Singular value decomposition
<b>Syntax</b>	$s = \text{svd}(X)$ $[U, S, V] = \text{svd}(X)$ $[U, S, V] = \text{svd}(X, 0)$
<b>Description</b>	The svd command computes the matrix singular value decomposition.  $s = \text{svd}(X)$ returns a vector of singular values.  $[U, S, V] = \text{svd}(X)$ 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 $X = U*S*V'$ .  $[U, S, V] = \text{svd}(X, 0)$ 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- $n$ .
<b>Examples</b>	For the matrix  $X = \begin{matrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{matrix}$ the statement  $[U, S, V] = \text{svd}(X)$ produces  $U = \begin{matrix} 0.1525 & 0.8226 & -0.3945 & -0.3800 \\ 0.3499 & 0.4214 & 0.2428 & 0.8007 \\ 0.5474 & 0.0201 & 0.6979 & -0.4614 \\ 0.7448 & -0.3812 & -0.5462 & 0.0407 \end{matrix}$

$$S = \begin{matrix} & \\ 14.2691 & 0 \\ 0 & 0.6268 \\ 0 & 0 \\ 0 & 0 \end{matrix}$$

$$V = \begin{matrix} & \\ 0.6414 & -0.7672 \\ 0.7672 & 0.6414 \end{matrix}$$

The economy size decomposition generated by

`[U, S, V] = svd(X, 0)`

produces

$$U = \begin{matrix} & \\ 0.1525 & 0.8226 \\ 0.3499 & 0.4214 \\ 0.5474 & 0.0201 \\ 0.7448 & -0.3812 \end{matrix}$$

$$S = \begin{matrix} & \\ 14.2691 & 0 \\ 0 & 0.6268 \end{matrix}$$

$$V = \begin{matrix} & \\ 0.6414 & -0.7672 \\ 0.7672 & 0.6414 \end{matrix}$$

## 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

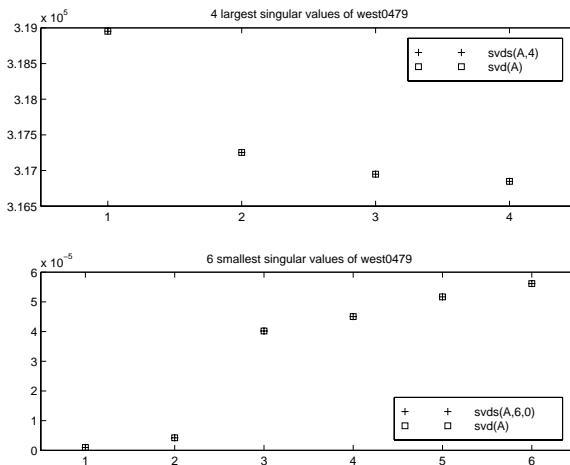
[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

# svds

---

<b>Purpose</b>	A few singular values
<b>Syntax</b>	<pre>s = svds(A) s = svds(A, k) s = svds(A, k, 0) [U, S, V] = svds(A, ...)</pre>
<b>Description</b>	<p><code>svds(A)</code> computes the five largest singular values and associated singular vectors of the matrix A.</p> <p><code>svds(A, k)</code> computes the k largest singular values and associated singular vectors of the matrix A.</p> <p><code>svds(A, k, 0)</code> computes the k smallest singular values and associated singular vectors.</p> <p>With one output argument, s is a vector of singular values. With three output arguments and if A is m-by-n:</p> <ul style="list-style-type: none"><li>• U is m-by-k with orthonormal columns</li><li>• S is k-by-k diagonal</li><li>• V is n-by-k with orthonormal columns</li><li>• <math>U^*S^*V^*</math> is the closest rank k approximation to A</li></ul>
<b>Algorithm</b>	<p><code>svds(A, k)</code> uses eigensolvers to find the k largest magnitude eigenvalues and corresponding eigenvectors of <math>B = [0 \ A; \ A' \ 0]</math>.</p> <p><code>svds(A, k, 0)</code> uses eigensolvers to find the 2k smallest magnitude eigenvalues and corresponding eigenvectors of <math>B = [0 \ A; \ A' \ 0]</math>, and then selects the k positive eigenvalues and their eigenvectors.</p>
<b>Example</b>	<p><code>west0479</code> is a real 479-by-479 sparse matrix. <code>svd</code> calculates all 479 singular values. <code>svds</code> picks out the largest and smallest singular values.</p> <pre>load west0479 s = svd(full(west0479)) sl = svds(west0479, 4) ss = svds(west0479, 6, 0)</pre>

These plots show some of the singular values of west0479 as computed by svd and svds.



The largest singular value of west0479 can be computed a few different ways:

```
svds(west0479, 1) =
3.189517598808622e+05
```

```
max(svd(full(west0479))) =
3.18951759880862e+05
```

```
norm(full(west0479)) =
3.189517598808623e+05
```

and estimated:

```
normest(west0479) =
3.189385666549991e+05
```

## See Also

[svd](#)  
[eig](#)

Singular value decomposition  
Find a few eigenvalues and eigenvectors

# switch

---

**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
```

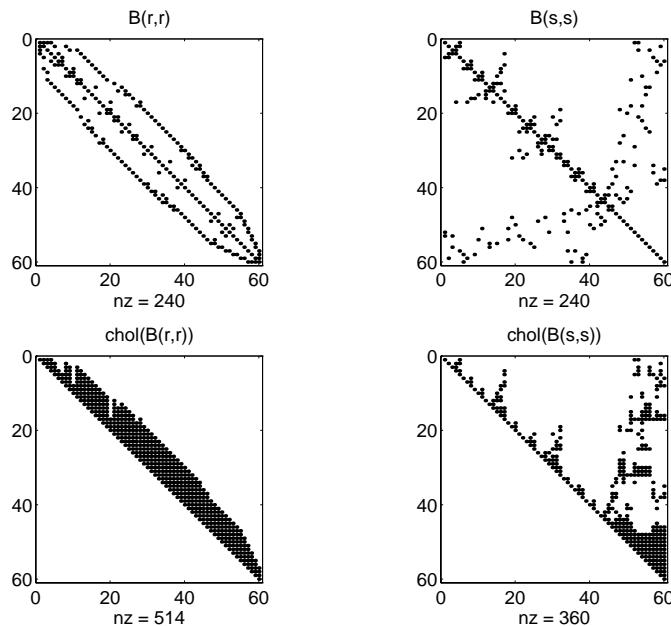
## See Also

case, end, if, otherwise, while

# **symmmd**

---

<b>Purpose</b>	Sparse symmetric minimum degree ordering
<b>Syntax</b>	<code>p = symmmd(S)</code>
<b>Description</b>	<code>p = symmmd(S)</code> returns a symmetric minimum degree ordering of <code>S</code> . For a symmetric positive definite matrix <code>S</code> , this is a permutation <code>p</code> such that <code>S(p, p)</code> tends to have a sparser Cholesky factor than <code>S</code> . Sometimes <code>symmmd</code> works well for symmetric indefinite matrices too.
<b>Remarks</b>	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 <code>spparms</code> .
<b>Algorithm</b>	The symmetric minimum degree algorithm is based on the column minimum degree algorithm. In fact, <code>symmmd(A)</code> just creates a nonzero structure <code>K</code> such that <code>K' * K</code> has the same nonzero structure as <code>A</code> and then calls the column minimum degree code for <code>K</code> .
<b>Examples</b>	Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the <code>symrcm</code> reference page.  <pre>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))')</pre>



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

<code>col mmd</code>	Sparse column minimum degree permutation
<code>col perm</code>	Sparse column permutation based on nonzero count
<code>symrcm</code>	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.

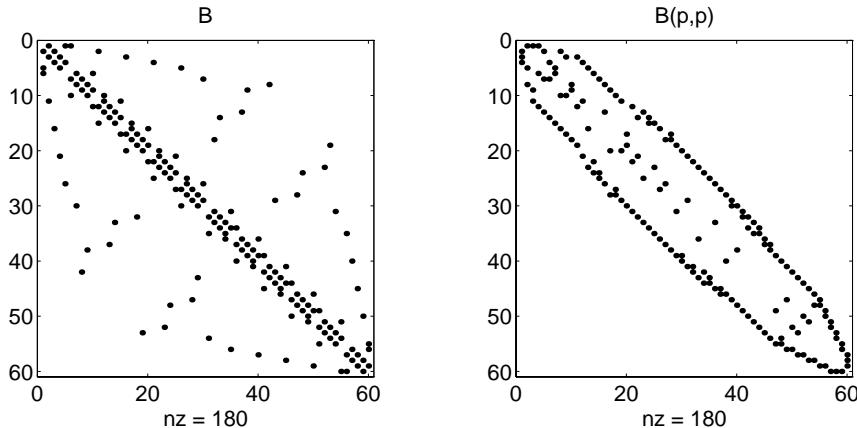
# **symrcm**

---

<b>Purpose</b>	Sparse reverse Cuthill-McKee ordering
<b>Syntax</b>	<code>r = symrcm(S)</code>
<b>Description</b>	<code>r = symrcm(S)</code> 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 $\text{eig}(S(r, r))$ probably takes less time to compute than $\text{eig}(S)$ .
<b>Algorithm</b>	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.
<b>Examples</b>	<p>The statement</p> <pre>B = bucky</pre> <p>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 <code>bucky</code>), 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 <code>spy</code> plot shows</p> <pre>subplot(1, 2, 1), spy(B), title('B')</pre> <p>The reverse Cuthill-McKee ordering is obtained with</p> <pre>p = symrcm(B); R = B(p, p);</pre>

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

<code>colmmd</code>	Sparse column minimum degree permutation
<code>colperm</code>	Sparse column permutation based on nonzero count
<code>symmmd</code>	Sparse symmetric minimum degree ordering

## References

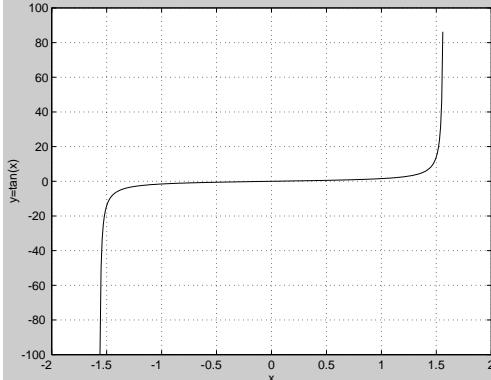
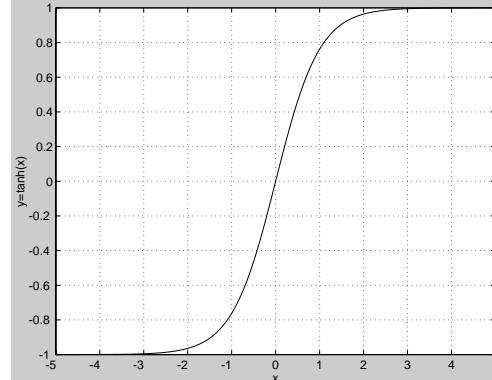
- [1] George, Alan and Joseph Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.
- [2] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," to appear in *SIAM Journal on Matrix Analysis*, 1992. A slightly expanded version is also available as a technical report from the Xerox Palo Alto Research Center.

**symrcm**

---



# **tan, tanh**

<b>Purpose</b>	Tangent and hyperbolic tangent
<b>Syntax</b>	$Y = \tan(X)$ $Y = \tanh(X)$
<b>Description</b>	The <code>tan</code> and <code>tanh</code> 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 $X$ .
<b>Examples</b>	Graph the tangent function over the domain $-\pi/2 < x < \pi/2$ , and the hyperbolic tangent function over the domain $-5 \leq x \leq 5$ .
	<pre>x = (-pi/2)+0.01:0.01:(pi/2)-0.01; plot(x,tan(x)) x = -5:0.01:5; plot(x,tanh(x))</pre>
	 
	The expression $\tan(\pi/2)$ does not evaluate as infinite but as the reciprocal of the floating point accuracy $\text{eps}$ since $\pi$ is only a floating-point approximation to the exact value of $\pi$ .
<b>Algorithm</b>	$\tan(z) = \frac{\sin(z)}{\cos(z)}$ $\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$

**See Also**

[atan](#), [atan2](#)

## **tempdir**

---

<b>Purpose</b>	Return the name of the system's temporary directory
<b>Syntax</b>	<code>tmp_dir = tempdir</code>
<b>Description</b>	<code>tmp_dir = tempdir</code> returns the name of the system's temporary directory, if one exists. This function does not create a new directory.
<b>See Also</b>	<code>tempname</code> Unique name for temporary file

<b>Purpose</b>	Unique name for temporary file
<b>Syntax</b>	<code>tempname</code>
<b>Description</b>	<code>tempname</code> returns a unique string beginning with the characters tp. This string is useful as a name for a temporary file.
<b>See Also</b>	<code>tempdir</code> Return the name of the system's temporary directory

# **tic, toc**

---

<b>Purpose</b>	Stopwatch timer						
<b>Syntax</b>	<pre>tic     <i>any statements</i> toc t = toc</pre>						
<b>Description</b>	<p><code>tic</code> starts a stopwatch timer.</p> <p><code>toc</code> prints the elapsed time since <code>tic</code> was used.</p> <p><code>t = toc</code> returns the elapsed time in <code>t</code>.</p>						
<b>Examples</b>	This example measures how the time required to solve a linear system varies with the order of a matrix.						
	<pre>for n = 1:100     A = rand(n,n);     b = rand(n,1);     tic     x = A\b;     t(n) = toc; end plot(t)</pre>						
<b>See Also</b>	<table><tr><td><code>clock</code></td><td>Current time as a date vector</td></tr><tr><td><code>cputime</code></td><td>Elapsed CPU time</td></tr><tr><td><code>etime</code></td><td>Elapsed time</td></tr></table>	<code>clock</code>	Current time as a date vector	<code>cputime</code>	Elapsed CPU time	<code>etime</code>	Elapsed time
<code>clock</code>	Current time as a date vector						
<code>cputime</code>	Elapsed CPU time						
<code>etime</code>	Elapsed time						

---

<b>Purpose</b>	Toeplitz matrix
<b>Syntax</b>	$T = \text{toeplitz}(c, r)$ $T = \text{toeplitz}(r)$
<b>Description</b>	A <i>Toeplitz</i> matrix is defined by one row and one column. A <i>symmetric Toeplitz</i> matrix is defined by just one row. <code>toeplitz</code> generates Toeplitz matrices given just the row or row and column description.
	$T = \text{toeplitz}(c, r)$ 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.
	$T = \text{toeplitz}(r)$ returns the symmetric or Hermitian Toeplitz matrix formed from vector $r$ , where $r$ defines the first row of the matrix.
<b>Examples</b>	A Toeplitz matrix with diagonal disagreement is  <pre>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</pre>
<b>See Also</b>	<a href="#">hankel</a> <a href="#">Hankel matrix</a>

## trace

---

<b>Purpose</b>	Sum of diagonal elements
<b>Syntax</b>	<code>b = trace(A)</code>
<b>Description</b>	<code>b = trace(A)</code> is the sum of the diagonal elements of the matrix A.
<b>Algorithm</b>	trace is a single-statement M-file. <code>t = sum(diag(A));</code>
<b>See Also</b>	<code>det</code> Matrix determinant <code>eig</code> Eigenvalues and eigenvectors

---

<b>Purpose</b>	Trapezoidal numerical integration
<b>Syntax</b>	$Z = \text{trapz}(Y)$ $Z = \text{trapz}(X, Y)$ $Z = \text{trapz}(\dots, \text{dim})$
<b>Description</b>	<p><math>Z = \text{trapz}(Y)</math> computes an approximation of the integral of <math>Y</math> via the trapezoidal method (with unit spacing). To compute the integral for spacing other than one, multiply <math>Z</math> by the spacing increment.</p> <p>If <math>Y</math> is a vector, <math>\text{trapz}(Y)</math> is the integral of <math>Y</math>.</p> <p>If <math>Y</math> is a matrix, <math>\text{trapz}(Y)</math> is a row vector with the integral over each column.</p> <p>If <math>Y</math> is a multidimensional array, <math>\text{trapz}(Y)</math> works across the first nonsingleton dimension.</p> <p><math>Z = \text{trapz}(X, Y)</math> computes the integral of <math>Y</math> with respect to <math>X</math> using trapezoidal integration.</p> <p>If <math>X</math> is a column vector and <math>Y</math> an array whose first nonsingleton dimension is <math>\text{length}(X)</math>, <math>\text{trapz}(X, Y)</math> operates across this dimension.</p> <p><math>Z = \text{trapz}(\dots, \text{dim})</math> integrates across the dimension of <math>Y</math> specified by scalar <math>\text{dim}</math>. The length of <math>X</math>, if given, must be the same as <math>\text{size}(Y, \text{dim})</math>.</p>
<b>Examples</b>	<p>The exact value of <math>\int_0^\pi \sin(x) dx</math> is 2.</p> <p>To approximate this numerically on a uniformly spaced grid, use</p> <pre>X = 0: pi / 100: pi ; Y = sin(x);</pre> <p>Then both</p> <pre>Z = trapz(X, Y)</pre> <p>and</p> <pre>Z = pi / 100 * trapz(Y)</pre>

# **trapz**

---

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**

<b>cumsum</b>	Cumulative sum
<b>cumtrapz</b>	Cumulative trapezoidal numerical integration

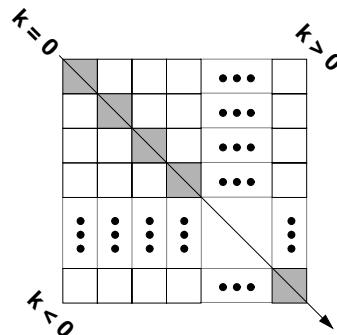
**Purpose** Lower triangular part of a matrix

**Syntax**

```
L = tril(X)
L = tril(X, k)
```

**Description**  $L = \text{tril}(X)$  returns the lower triangular part of  $X$ .

$L = \text{tril}(X, k)$  returns the elements on and below the  $k$ th 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**  $\text{tril}(\text{ones}(4, 4), -1)$  is

$$\begin{matrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{matrix}$$

**See Also** [di ag](#) Diagonal matrices and diagonals of a matrix  
[tri u](#) Upper triangular part of a matrix

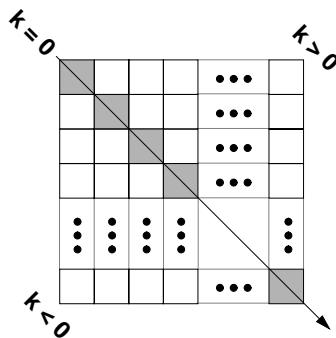
# triu

**Purpose**      Upper triangular part of a matrix

**Syntax**       $U = \text{triu}(X)$   
                   $U = \text{triu}(X, k)$

**Description**       $U = \text{triu}(X)$  returns the upper triangular part of  $X$ .

$U = \text{triu}(X, k)$  returns the element on and above the  $k$ th 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**       $\text{triu}(\text{ones}(4, 4), -1)$  is

$$\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{matrix}$$

**See Also**      [di ag](#)      Diagonal matrices and diagonals of a matrix  
[tril](#)      Lower triangular part of a matrix

---

<b>Purpose</b>	Begin try block
<b>Description</b>	<p>The general form of a try statement is:</p> <p>try statement, ..., statement, catch statement, ..., statement end</p> <p>Normally, only the statements between the try and catch are executed. However, if an error occurs while executing any of the statements, the error is captured into lasterr, and the statements between the catch and end are executed. If an error occurs within the catch statements, execution stops unless caught by another try...catch block. The error string produced by a failed try block can be obtained with lasterr.</p>
<b>See Also</b>	catch, end, eval , eval i n,

# tsearch

---

<b>Purpose</b>	Search for enclosing Delaunay triangle
<b>Syntax</b>	<code>T = tsearch(x, y, TRI, xi, yi)</code>
<b>Description</b>	<code>T = tsearch(x, y, TRI, xi, yi)</code> returns an index into the rows of <code>TRI</code> for each point in <code>xi, yi</code> . The <code>tsearch</code> command returns <code>NaN</code> for all points outside the convex hull. Requires a triangulation <code>TRI</code> of the points <code>x,y</code> obtained from <code>del aunay</code> .
<b>See Also</b>	<code>del aunay</code> Delaunay triangulation <code>dsearch</code> Search for nearest point

<b>Purpose</b>	List file																
<b>Syntax</b>	<code>type filename</code>																
<b>Description</b>	<code>type filename</code> 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 <code>type</code> command adds the <code>.m</code> extension by default. The <code>type</code> command checks the directories specified in MATLAB's search path, which makes it convenient for listing the contents of M-files on the screen.																
<b>Examples</b>	<code>type foo.bar</code> lists the file <code>foo.bar</code> .  <code>type foo</code> lists the file <code>foo.m</code> .																
<b>See Also</b>	<table><tr><td>!</td><td>Operating system command</td></tr><tr><td><code>cd</code></td><td>Change working directory</td></tr><tr><td><code>dbtype</code></td><td>List M-file with line numbers</td></tr><tr><td><code>delete</code></td><td>Delete files and graphics objects</td></tr><tr><td><code>dir</code></td><td>Directory listing</td></tr><tr><td><code>path</code></td><td>Control MATLAB's directory search path</td></tr><tr><td><code>what</code></td><td>Directory listing of M-files, MAT-files, and MEX-files</td></tr><tr><td><code>who</code></td><td>List directory of variables in memory</td></tr></table> <p>See also <code>partialpath</code>.</p>	!	Operating system command	<code>cd</code>	Change working directory	<code>dbtype</code>	List M-file with line numbers	<code>delete</code>	Delete files and graphics objects	<code>dir</code>	Directory listing	<code>path</code>	Control MATLAB's directory search path	<code>what</code>	Directory listing of M-files, MAT-files, and MEX-files	<code>who</code>	List directory of variables in memory
!	Operating system command																
<code>cd</code>	Change working directory																
<code>dbtype</code>	List M-file with line numbers																
<code>delete</code>	Delete files and graphics objects																
<code>dir</code>	Directory listing																
<code>path</code>	Control MATLAB's directory search path																
<code>what</code>	Directory listing of M-files, MAT-files, and MEX-files																
<code>who</code>	List directory of variables in memory																

# **uint8**

---

<b>Purpose</b>	Convert to unsigned 8-bit integer
<b>Syntax</b>	<code>i = uint8(x)</code>
<b>Description</b>	<code>i = uint8(x)</code> converts the vector <code>x</code> into an unsigned 8-bit integer. <code>x</code> can be any numeric object (such as a <code>double</code> ). The elements of an <code>uint8</code> range from 0 to 255. The result for any elements of <code>x</code> outside this range is not defined (and may vary from platform to platform). If <code>x</code> is already an unsigned 8-bit integer, <code>uint8</code> has no effect.
	The <code>uint8</code> class is primarily meant to store integer values. Most operations that manipulate arrays without changing their elements are defined (examples are <code>reshape</code> , <code>size</code> , subscripted assignment and subscripted reference). No math operations are defined for <code>uint8</code> 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 <code>uint8</code> (as you can for any object) by placing the appropriately named method in an <code>@uint8</code> directory within a directory on your path. The Image Processing Toolbox does just that to define additional methods for the <code>uint8</code> (such as the logical operators <code>&lt;,&gt;,&amp;</code> , etc.).
	Type <code>help oopfun</code> for the names of the methods you can overload.
<b>See Also</b>	<code>double</code> Convert to double precision

---

<b>Purpose</b>	Set union of two vectors														
<b>Syntax</b>	$c = \text{union}(a, b)$ $c = \text{union}(A, B, 'rows')$ $[c, ia, ib] = \text{union}(\dots)$														
<b>Description</b>	<p><math>c = \text{union}(a, b)</math> returns the combined values from <math>a</math> and <math>b</math> but with no repetitions. The resulting vector is sorted in ascending order. In set theoretic terms, <math>c = a \cup b</math>. <math>a</math> and <math>b</math> can be cell arrays of strings.</p> <p><math>c = \text{union}(A, B, 'rows')</math> when <math>A</math> and <math>B</math> are matrices with the same number of columns returns the combined rows from <math>A</math> and <math>B</math> with no repetitions.</p> <p><math>[c, ia, ib] = \text{union}(\dots)</math> also returns index vectors <math>ia</math> and <math>ib</math> such that <math>c = a(ia)</math> and <math>c = b(ib)</math> or, for row combinations, <math>c = a(ia, :)</math> and <math>c = b(ib, :)</math>.</p>														
<b>Examples</b>	<pre> a = [-1 0 2 4 6]; b = [-1 0 1 3]; [c, ia, ib] = union(a, b); c = </pre> <table style="margin-left: 200px; margin-top: -20px;"> <tr><td>-1</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>6</td></tr> </table> <pre> ia = </pre> <table style="margin-left: 200px; margin-top: -20px;"> <tr><td>3</td><td>4</td><td>5</td></tr> </table> <pre> ib = </pre> <table style="margin-left: 200px; margin-top: -20px;"> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> </table>	-1	0	1	2	3	4	6	3	4	5	1	2	3	4
-1	0	1	2	3	4	6									
3	4	5													
1	2	3	4												
<b>See Also</b>	<a href="#">intersect</a> Set intersection of two vectors <a href="#">setdiff</a> Return the set difference of two vectors <a href="#">setxor</a> Set exclusive-or of two vectors <a href="#">unique</a> Unique elements of a vector														

# unique

---

<b>Purpose</b>	Unique elements of a vector										
<b>Syntax</b>	<pre>b = unique(a) b = unique(A, 'rows') [b, i, j] = unique(...)</pre>										
<b>Description</b>	<p><code>b = unique(a)</code> returns the same values as in <code>a</code> but with no repetitions. The resulting vector is sorted in ascending order. <code>a</code> can be a cell array of strings.</p> <p><code>b = unique(A, 'rows')</code> returns the unique rows of <code>A</code>.</p> <p><code>[b, i, j] = unique(...)</code> also returns index vectors <code>i</code> and <code>j</code> such that <code>b = a(i)</code> and <code>a = b(j)</code> (or <code>b = a(:, i)</code> and <code>a = b(:, j)</code>).</p>										
<b>Examples</b>	<pre>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</pre>										
<b>See Also</b>	<table><tr><td><code>intersect</code></td><td>Set intersection of two vectors</td></tr><tr><td><code>ismember</code></td><td>True for a set member</td></tr><tr><td><code>setdiff</code></td><td>Return the set difference of two vectors</td></tr><tr><td><code>setxor</code></td><td>Set exclusive-or of two vectors</td></tr><tr><td><code>union</code></td><td>Set union of two vectors</td></tr></table>	<code>intersect</code>	Set intersection of two vectors	<code>ismember</code>	True for a set member	<code>setdiff</code>	Return the set difference of two vectors	<code>setxor</code>	Set exclusive-or of two vectors	<code>union</code>	Set union of two vectors
<code>intersect</code>	Set intersection of two vectors										
<code>ismember</code>	True for a set member										
<code>setdiff</code>	Return the set difference of two vectors										
<code>setxor</code>	Set exclusive-or of two vectors										
<code>union</code>	Set union of two vectors										

---

<b>Purpose</b>	Correct phase angles																
<b>Syntax</b>	$Q = \text{unwrap}(P)$ $Q = \text{unwrap}(P, tol)$ $Q = \text{unwrap}(P, [ ], dim)$ $Q = \text{unwrap}(P, tol, dim)$																
<b>Description</b>	<p><math>Q = \text{unwrap}(P)</math> corrects the radian phase angles in array <math>P</math> by adding multiples of <math>\pm 2\pi</math> when absolute jumps between consecutive array elements are greater than <math>\pi</math> radians. If <math>P</math> is a matrix, <math>\text{unwrap}</math> operates columnwise. If <math>P</math> is a multidimensional array, <math>\text{unwrap}</math> operates on the first nonsingleton dimension.</p> <p><math>Q = \text{unwrap}(P, tol)</math> uses a jump tolerance <math>tol</math> instead of the default value, <math>\pi</math>.</p> <p><math>Q = \text{unwrap}(P, [ ], dim)</math> unwraps along <math>dim</math> using the default tolerance.</p> <p><math>Q = \text{unwrap}(P, tol, dim)</math> uses a jump tolerance of <math>tol</math>.</p>																
<b>Examples</b>	Array $P$ features smoothly increasing phase angles except for discontinuities at elements (3, 1) and (1, 2).																
	$P =$ <table style="margin-left: 100px; margin-top: -10px;"> <tbody> <tr> <td>0</td> <td><u>7.0686</u></td> <td>1.5708</td> <td>2.3562</td> </tr> <tr> <td>0.1963</td> <td>0.9817</td> <td>1.7671</td> <td>2.5525</td> </tr> <tr> <td><u>6.6759</u></td> <td>1.1781</td> <td>1.9635</td> <td>2.7489</td> </tr> <tr> <td>0.5890</td> <td>1.3744</td> <td>2.1598</td> <td>2.9452</td> </tr> </tbody> </table>	0	<u>7.0686</u>	1.5708	2.3562	0.1963	0.9817	1.7671	2.5525	<u>6.6759</u>	1.1781	1.9635	2.7489	0.5890	1.3744	2.1598	2.9452
0	<u>7.0686</u>	1.5708	2.3562														
0.1963	0.9817	1.7671	2.5525														
<u>6.6759</u>	1.1781	1.9635	2.7489														
0.5890	1.3744	2.1598	2.9452														
	The function $Q = \text{unwrap}(P)$ eliminates these discontinuities.																
	$Q =$ <table style="margin-left: 100px; margin-top: -10px;"> <tbody> <tr> <td>0</td> <td>0.7854</td> <td>1.5708</td> <td>2.3562</td> </tr> <tr> <td>0.1963</td> <td>0.9817</td> <td>1.7671</td> <td>2.5525</td> </tr> <tr> <td>0.3927</td> <td>1.1781</td> <td>1.9635</td> <td>2.7489</td> </tr> <tr> <td>0.5890</td> <td>1.3744</td> <td>2.1598</td> <td>2.9452</td> </tr> </tbody> </table>	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
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														
<b>Limitations</b>	The $\text{unwrap}$ function detects branch cut crossings, but it can be fooled by sparse, rapidly changing phase values.																
<b>See Also</b>	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 30%;"><code>abs</code></td> <td style="width: 70%;">Absolute value and complex magnitude</td> </tr> <tr> <td><code>angle</code></td> <td>Phase angle</td> </tr> </table>	<code>abs</code>	Absolute value and complex magnitude	<code>angle</code>	Phase angle												
<code>abs</code>	Absolute value and complex magnitude																
<code>angle</code>	Phase angle																

# **upper**

---

<b>Purpose</b>	Convert string to upper case
<b>Syntax</b>	<code>t = upper('str')</code> <code>B = upper(A)</code>
<b>Description</b>	<code>t = upper('str')</code> converts any lower-case characters in the string <i>str</i> to the corresponding upper-case characters and leaves all other characters unchanged.  <code>B = upper(A)</code> when <i>A</i> is a cell array of strings, returns a cell array the same size as <i>A</i> containing the result of applying <i>upper</i> to each string within <i>A</i> .
<b>Examples</b>	<code>upper('attention!')</code> is ATTENTION!.
<b>Remarks</b>	Character sets supported: <ul style="list-style-type: none"><li>• Mac: Standard Roman</li><li>• PC: Windows Latin-1</li><li>• Other: ISO Latin-1 (ISO 8859-1)</li></ul>
<b>See Also</b>	<code>lower</code> Convert string to lower case

<b>Purpose</b>	Pass or return variable numbers of arguments
<b>Syntax</b>	<pre>function varargout = foo(n) y = function bar(varargin)</pre>
<b>Description</b>	<p><code>function varargout = foo(n)</code> returns a variable number of arguments from function <code>foo.m</code>.</p> <p><code>y = function bar(varargin)</code> accepts a variable number of arguments into function <code>bar.m</code>.</p> <p>The <code>varargin</code> and <code>varargout</code> 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, <code>varargin</code> and <code>varargout</code> must be lowercase.</p>
<b>Examples</b>	<p>The function</p> <pre>function myplot(x, varargin) plot(x, varargin{:})</pre> <p>collects all the inputs starting with the second input into the variable <code>varargin</code>. <code>myplot</code> uses the comma-separated list syntax <code>varargin{:}</code> to pass the optional parameters to <code>plot</code>. The call</p> <pre>myplot(sin(0:1:1), 'color', [.5 .7 .3], 'linestyle', ':')</pre> <p>results in <code>varargin</code> being a 1-by-4 cell array containing the values '<code>color</code>', <code>[.5 .7 .3]</code>, '<code>linestyle</code>', and <code>:</code>'.</p> <p>The function</p> <pre>function [s, varargout] = mysize(x) nout = max(nargout, 1) - 1; s = size(x); for i=1:nout, varargout(i) = {s(i)}; end</pre> <p>returns the size vector and, optionally, individual sizes. So</p> <pre>[s, rows, cols] = mysize(rand(4, 5));</pre> <p>returns <code>s = [4 5]</code>, <code>rows = 4</code>, <code>cols = 5</code>.</p>

# **varargin, varargout**

---

## **See Also**

[nargin](#)      Number of function arguments  
[nargout](#)     Number of function arguments  
[nargchk](#)    Check number of input arguments

---

<b>Purpose</b>	Vectorize expression
<b>Syntax</b>	<code>vectorize(string)</code> <code>vectorize(function)</code>
<b>Description</b>	<code>vectorize(string)</code> inserts a <code>.</code> before any <code>^</code> , <code>*</code> or <code>/</code> in <code>string</code> . The result is a character string.  <code>vectorize(function)</code> when <code>function</code> is an inline function object, vectorizes the formula for <code>function</code> . The result is the vectorized version of the inline function.
<b>See Also</b>	<a href="#">inline</a> Construct an inline object

## version

<b>Purpose</b>	MATLAB version number
<b>Syntax</b>	<code>v = versi on</code> <code>[v, d] = versi on</code>
<b>Description</b>	<code>v = versi on</code> returns a string <code>v</code> containing the MATLAB version number.  <code>[v, d] = versi on</code> also returns a string <code>d</code> containing the date of the version.
<b>See Also</b>	<code>hel p</code> Online help for MATLAB functions and M-files <code>what snew</code> Display README files for MATLAB and toolboxes

<b>Purpose</b>	Voronoi diagram
<b>Syntax</b>	<code>voronoi (x, y)</code> <code>voronoi (x, y, TRI)</code> <code>h = voronoi ( . . . , ' LineSpec' )</code> <code>[vx, vy] = voronoi ( . . . )</code>
<b>Definition</b>	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 <i>Voronoi polygon</i> , and the set of all Voronoi polygons for a given point set is called a <i>Voronoi diagram</i> .
<b>Description</b>	<code>voronoi (x, y)</code> plots the Voronoi diagram for the points $x, y$ .  <code>voronoi (x, y, TRI)</code> uses the triangulation $TRI$ instead of computing it via <code>delaunay</code> .  <code>h = voronoi ( . . . , ' LineSpec' )</code> plots the diagram with color and line style specified and returns handles to the line objects created in $h$ .  <code>[vx, vy] = voronoi ( . . . )</code> returns the vertices of the Voronoi edges in $vx$ and $vy$ so that <code>plot(vx, vy, ' - ', x, y, ' . ')</code> creates the Voronoi diagram.

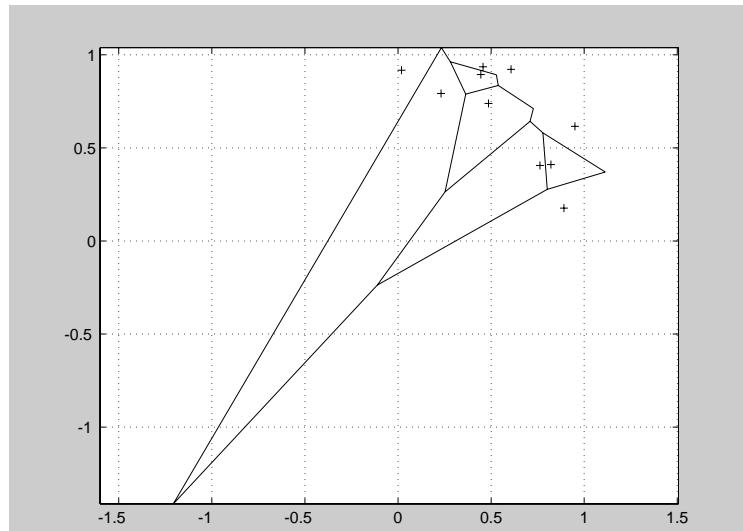
# voronoi

---

## 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`  
`delaunay`  
`dsearch`

Convex hull  
Delaunay triangulation  
Search for nearest point

<b>Purpose</b>	Display warning message						
<b>Syntax</b>	<pre>warning('message') warning on warning off warning backtrace warning debug warning once warning always [s, f] = warning</pre>						
<b>Description</b>	<p><code>warning('message')</code> displays the text '<code>message</code>' as does the <code>disp</code> function, except that with <code>warning</code>, message display can be suppressed.</p> <p><code>warning off</code> suppresses all subsequent warning messages.</p> <p><code>warning on</code> re-enables them.</p> <p><code>warning backtrace</code> is the same as <code>warning on</code> except that the file and line number that produced the warning are displayed.</p> <p><code>warning debug</code> is the same as <code>dbstop if warning</code> and triggers the debugger when a warning is encountered.</p> <p><code>warning once</code> displays Handle Graphics backwards compatibility warnings only once per session.</p> <p><code>warning always</code> displays Handle Graphics backwards compatibility warnings as they are encountered (subject to current warning state).</p> <p><code>[s, f] = warning</code> returns the current warning state <code>s</code> and the current warning frequency <code>f</code> as strings.</p>						
<b>Remarks</b>	Use <code>dbstop on warning</code> to trigger the debugger when a warning is encountered.						
<b>See Also</b>	<table border="0"> <tr> <td><code>dbstop</code></td> <td>Set breakpoints in an M-file function</td> </tr> <tr> <td><code>disp</code></td> <td>Display text or array</td> </tr> <tr> <td><code>error</code></td> <td>Display error messages</td> </tr> </table>	<code>dbstop</code>	Set breakpoints in an M-file function	<code>disp</code>	Display text or array	<code>error</code>	Display error messages
<code>dbstop</code>	Set breakpoints in an M-file function						
<code>disp</code>	Display text or array						
<code>error</code>	Display error messages						

# wavread

---

<b>Purpose</b>	Read Microsoft WAVE (. wav) sound file				
<b>Syntax</b>	<pre>y = wavread('filename') [y, Fs, bits] = wavread('filename') [...] = wavread('filename', N) [...] = wavread('filename', [N1 N2]) [...] = wavread('filename', 'size')</pre>				
<b>Description</b>	wavread supports multichannel data, with up to 16 bits per sample.				
	<p><code>y = wavread('filename')</code> loads a WAVE file specified by the string <code>filename</code>, returning the sampled data in <code>y</code>. The . wav extension is appended if no extension is given. Amplitude values are in the range [-1, +1].</p>				
	<p><code>[y, Fs, bits] = wavread('filename')</code> returns the sample rate (<code>Fs</code>) in Hertz and the number of bits per sample (<code>bits</code>) used to encode the data in the file.</p>				
	<p><code>[...] = wavread('filename', N)</code> returns only the first <code>N</code> samples from each channel in the file.</p>				
	<p><code>[...] = wavread('filename', [N1 N2])</code> returns only samples <code>N1</code> through <code>N2</code> from each channel in the file.</p>				
	<p><code>size = wavread('filename', 'size')</code> returns the size of the audio data contained in the file in place of the actual audio data, returning the vector <code>size = [samples channels]</code>.</p>				
<b>See Also</b>	<table><tr><td><code>auread</code></td><td>Read NeXT/SUN (. au) sound file</td></tr><tr><td><code>wavwrite</code></td><td>Write Microsoft WAVE (. wav) sound file</td></tr></table>	<code>auread</code>	Read NeXT/SUN (. au) sound file	<code>wavwrite</code>	Write Microsoft WAVE (. wav) sound file
<code>auread</code>	Read NeXT/SUN (. au) sound file				
<code>wavwrite</code>	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**

<code>afwrite</code>	Write NeXT/SUN (. au) sound file
<code>wavread</code>	Read Microsoft WAVE (. wav) sound file

# web

---

<b>Purpose</b>	Point Web browser at file or Web site				
<b>Syntax</b>	<code>web url</code>				
<b>Description</b>	<code>web url</code> 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.				
<b>Examples</b>	<code>web file:/disk/dir1/dir2/foo.html</code> points the browser to the file <code>foo.html</code> . If the file is on the MATLAB path, <code>web(['file:' whi ch(' foo.html ')])</code> also works. <code>web http://www.mathworks.com</code> loads The MathWorks Web page into your browser. Use <code>web mailto:email_address</code> to send e-mail to another site. The Web browser used is specified in the docopt M-file.				
<b>See Also</b>	<table><tr><td><code>doc</code></td><td>Display HTML documentation in Web browser</td></tr><tr><td><code>docopt</code></td><td>Configure local doc access defaults (in online help)</td></tr></table>	<code>doc</code>	Display HTML documentation in Web browser	<code>docopt</code>	Configure local doc access defaults (in online help)
<code>doc</code>	Display HTML documentation in Web browser				
<code>docopt</code>	Configure local doc access defaults (in online help)				

<b>Purpose</b>	Day of the week
<b>Syntax</b>	[N, S] = weekday(D)
<b>Description</b>	[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:
<b>N</b>	<b>S</b>
1	Sun
2	Mon
3	Tue
4	Wed
<b>N</b>	<b>S</b>
5	Thu
6	Fri
7	Sat
<b>Examples</b>	<p>Either</p> <p>[n, s] = weekday(728647)</p> <p>or</p> <p>[n, s] = weekday('19-Dec-1994')</p> <p>returns n = 2 and s = Mon.</p>
<b>See Also</b>	<p>datenum datevec eomday</p> <p>Serial date number Date components End of month</p>

# what

---

<b>Purpose</b>	Directory listing of M-files, MAT-files, and MEX-files														
<b>Syntax</b>	<code>what</code> <code>what <i>dirname</i></code> <code>what ('<i>dirname</i>')</code>														
<b>Description</b>	<p><code>what</code> by itself, lists the M-files, MAT-files, and MEX-files in the current directory.</p> <p><code>what <i>dirname</i></code> lists the files in directory <i>dirname</i> 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 <code>what <i>class</i></code> or <code>what <i>dirname</i>/private</code> to list the files in a method directory or a private directory (for the class named <i>class</i>).</p> <p><code>w = what ('<i>dirname</i>')</code> returns the results of <code>what</code> in a structure array with the fields:</p>														
	<table border="1"><thead><tr><th>path</th><th>path to directory</th></tr></thead><tbody><tr><td>M</td><td>cell array of M-file names</td></tr><tr><td>MAT</td><td>cell array of MAT-file names</td></tr><tr><td>MEX</td><td>cell array of MEX-file names</td></tr><tr><td>MDL</td><td>cell array of MDL-file names</td></tr><tr><td>P</td><td>cell array of P-file names</td></tr><tr><td>classes</td><td>cell array of class names</td></tr></tbody></table>	path	path to directory	M	cell array of M-file names	MAT	cell array of MAT-file names	MEX	cell array of MEX-file names	MDL	cell array of MDL-file names	P	cell array of P-file names	classes	cell array of class names
path	path to directory														
M	cell array of M-file names														
MAT	cell array of MAT-file names														
MEX	cell array of MEX-file names														
MDL	cell array of MDL-file names														
P	cell array of P-file names														
classes	cell array of class names														
<b>Examples</b>	<p>The statements</p> <p><code>what general</code></p> <p>and</p> <p><code>what matlab/general</code></p>														

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

# whatsnew

---

<b>Purpose</b>	Display README files for MATLAB and toolboxes
<b>Syntax</b>	<code>whatsnew</code> <code>whatsnew matlab</code> <code>whatsnew <i>toolboxpath</i></code>
<b>Description</b>	<code>whatsnew</code> , 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.  <code>whatsnew matlab</code> displays the README file for MATLAB.  <code>whatsnew <i>toolboxpath</i></code> displays the README file for the toolbox specified by the string <i>toolboxpath</i> .
<b>Examples</b>	<code>whatsnew matlab % MATLAB README file</code> <code>whatsnew signal % Signal Processing Tool box README file</code>
<b>See Also</b>	<code>help</code> Online help for MATLAB functions and M-files <code>lookfor</code> Keyword search through all help entries <code>path</code> Control MATLAB's directory search path <code>version</code> MATLAB version number <code>which</code> Locate functions and files

<b>Purpose</b>	Locate functions and files
<b>Syntax</b>	<pre>whi ch <i>fun</i> whi ch <i>fun</i> -al l whi ch <i>file. ext</i> whi ch <i>fun1</i> i n <i>fun2</i> whi ch <i>fun(a, b, c, . . .)</i> s = whi ch(. . .)</pre>
<b>Description</b>	<p>whi ch <i>fun</i> 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 whi ch pri vate/<i>fun</i> or whi ch <i>class/ fun</i> or whi ch <i>class/pri vate/ fun</i> to further qualify the function name for private functions, methods, and private methods (for the class named <i>class</i>).</p> <p>whi ch <i>fun</i> -al l displays the paths to all functions with the name <i>fun</i>. The first one in the list is the one normally returned by whi ch. The others are either shadowed or can be executed in special circumstances. The -al l flag can be used with all forms of whi ch.</p> <p>whi ch <i>file. ext</i> displays the full pathname of the specified file.</p> <p>whi ch <i>fun1</i> i n <i>fun2</i> displays the pathname to function <i>fun1</i> in the context of the M-file <i>fun2</i>. While debugging <i>fun2</i>, whi ch <i>fun1</i> 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.</p> <p>whi ch <i>fun(a, b, c, . . .)</i> displays the path to the specified function with the given input arguments. For example, whi ch feval (g), when g=i nl i ne(' si n(x)'), indicates that i nl i ne/feval . m is invoked.</p> <p>s = whi ch(. . .) returns the results of whi ch in the string s instead of printing it to the screen. s will be the string bui l t-i n or vari abl e for built-in functions or variables in the workspace. You must use the functional form of whi ch when there is an output argument.</p>

# which

---

## Examples

For example,

```
whi ch i nv
```

reveals that `i nv` is a built-in function, and

```
whi ch pi nv
```

indicates that `pi nv` is in the `matfun` directory of the MATLAB Toolbox.

The statement

```
whi ch j acobi an
```

probably says

```
j acobi an not found
```

because there is no file `j acobi an.m` on MATLAB's search path. Contrast this with `lookfor j acobi an`, which takes longer to run, but finds several matches to the keyword `j acobi an` in its search through all the help entries. (If `j acobi an.m` does exist in the current directory, or in some private directory that has been added to MATLAB's search path, `whi ch j acobi an` finds it.)

## See Also

`di r`, `exist`, `hel p`, `lookfor`, `path`, `what`, `who`

---

<b>Purpose</b>	Repeat statements an indefinite number of times																
<b>Syntax</b>	<pre>while <i>expression</i>       <i>statements</i> end</pre>																
<b>Description</b>	<p>while repeats statements an indefinite number of times. The statements are executed while the real part of <i>expression</i> has all nonzero elements. <i>expression</i> is usually of the form</p> $\text{expression} \text{ } rop \text{ expression}$ <p>where <i>rop</i> is ==, &lt;, &gt;, &lt;=, &gt;=, or ~=.</p> <p>The scope of a while statement is always terminated with a matching end.</p>																
<b>Examples</b>	<p>The variable <i>eps</i> is a tolerance used to determine such things as near singularity and rank. Its initial value is the <i>machine epsilon</i>, the distance from 1.0 to the next largest floating-point number on your machine. Its calculation demonstrates while loops:</p> <pre>eps = 1; while (1+eps) &gt; 1     eps = eps/2; end eps = eps*2</pre>																
<b>See Also</b>	<table> <tr> <td>all</td> <td>Test to determine if all elements are nonzero</td> </tr> <tr> <td>any</td> <td>Test for any nonzeros</td> </tr> <tr> <td>break</td> <td>Terminate execution of for or while loop</td> </tr> <tr> <td>end</td> <td>Terminate for, while, switch, try, and if statements or indicate last index</td> </tr> <tr> <td>for</td> <td>Repeat statements a specific number of times</td> </tr> <tr> <td>if</td> <td>Conditionally execute statements</td> </tr> <tr> <td>return</td> <td>Return to the invoking function</td> </tr> <tr> <td>switch</td> <td>Switch among several cases based on expression</td> </tr> </table>	all	Test to determine if all elements are nonzero	any	Test for any nonzeros	break	Terminate execution of for or while loop	end	Terminate for, while, switch, try, 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
all	Test to determine if all elements are nonzero																
any	Test for any nonzeros																
break	Terminate execution of for or while loop																
end	Terminate for, while, switch, try, 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																

## who, whos

---

<b>Purpose</b>	List directory of variables in memory
<b>Syntax</b>	<code>who</code> <code>whos</code> <code>who global</code> <code>whos global</code> <code>who -file <i>filename</i></code> <code>whos -file <i>filename</i></code> <code>who ... <i>var1 var2</i></code> <code>whos ... <i>var1 var2</i></code> <code>s = who(...)</code> <code>s = whos(...)</code>
<b>Description</b>	<p><code>who</code> by itself, lists the variables currently in memory.</p> <p><code>whos</code> by itself, lists the current variables, their sizes, and whether they have nonzero imaginary parts.</p> <p><code>who global</code> and <code>whos global</code> list the variables in the global workspace.</p> <p><code>who -file <i>filename</i></code> and <code>whos -file <i>filename</i></code> list the variables in the specified MAT-file.</p> <p><code>who ... <i>var1 var2</i></code> and <code>whos ... <i>var1 var2</i></code> restrict the display to the variables specified. The wildcard character * can be used to display variables that match a pattern. For instance, <code>who A*</code> finds all variables in the current workspace that start with A. Use the functional form, such as <code>whos(' -file', <i>filename</i>, <i>v1</i>, <i>v2</i>)</code>, when the filename or variable names are stored in strings.</p> <p><code>s = who(...)</code> returns a cell array containing the names of the variables in the workspace or file. Use the functional form of <code>who</code> when there is an output argument.</p>

`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`

# wilkinson

---

<b>Purpose</b>	Wilkinson's eigenvalue test matrix																																																	
<b>Syntax</b>	<code>W = wilkinson(n)</code>																																																	
<b>Description</b>	<code>W = wilkinson(n)</code> 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.																																																	
<b>Examples</b>	<code>wilkinson(7)</code> is																																																	
	<table><tr><td>3</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>2</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>2</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>3</td></tr></table>	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
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 <code>wilkinson(21)</code> . Its two largest eigenvalues are both about 10.746; they agree to 14, but not to 15, decimal places.																																																	
<b>See Also</b>	<table><tr><td><code>eig</code></td><td>Eigenvalues and eigenvectors</td></tr><tr><td><code>gallery</code></td><td>Test matrices</td></tr><tr><td><code>pascal</code></td><td>Pascal matrix</td></tr></table>	<code>eig</code>	Eigenvalues and eigenvectors	<code>gallery</code>	Test matrices	<code>pascal</code>	Pascal matrix																																											
<code>eig</code>	Eigenvalues and eigenvectors																																																	
<code>gallery</code>	Test matrices																																																	
<code>pascal</code>	Pascal matrix																																																	

**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)
```

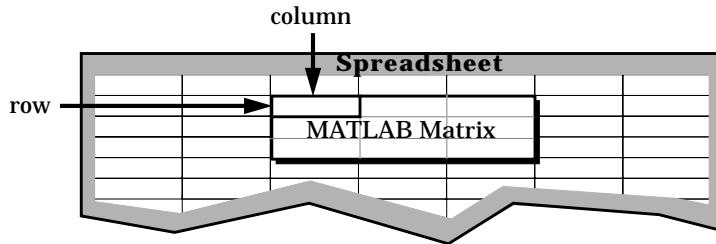
**Description**      M = wk1read(*filename*) reads a Lotus123 WK1 spreadsheet file into the matrix M.

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, 'Sal es' .

**See Also**

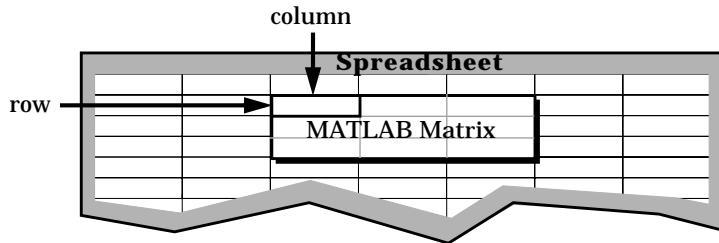
wk1write

Write a matrix to a Lotus123 WK1 spreadsheet file

# wk1write

---

<b>Purpose</b>	Write a matrix to a Lotus123 WK1 spreadsheet file
<b>Syntax</b>	<code>wk1write(filename, M)</code> <code>wk1write(filename, M, r, c)</code>
<b>Description</b>	<code>wk1write(filename, M)</code> writes the matrix <code>M</code> into a Lotus123 WK1 spreadsheet file named <code>filename</code> .  <code>wk1write(filename, M, r, c)</code> writes the matrix starting at the spreadsheet location <code>(r, c)</code> . <code>r</code> and <code>c</code> are zero based so that <code>r=0, c=0</code> specifies the first cell in the spreadsheet.
<b>See Also</b>	<a href="#">wk1read</a> Read a Lotus123 WK1 spreadsheet file into a matrix



---

<b>Purpose</b>	Write snd resources and files
<b>Syntax</b>	<code>writesnd(data, samplerate, bitspersample, filename)</code>
<b>Description</b>	<code>writesnd(data, samplerate, bitspersample, filename)</code> writes the sound information specified by <code>data</code> and <code>samplerate</code> into an snd resource in <code>filename</code> .
<b>Example</b>	<code>writesnd(y, Fs, 16, 'gong.snd')</code>

## **xlgetrange**

---

<b>Purpose</b>	Get range of cells from Microsoft Excel worksheet
<b>Syntax</b>	<code>xl getrange([ rmin, cmin, rmax, cmax], workbookname, worksheetnum)</code>
<b>Description</b>	<code>xl getrange([ rmin, cmin, rmax, cmax], workbookname, worksheetnum)</code> returns the data in the range $r < rmin > c < cmin > : r < rmax > c < cmax >$ of sheet <code>worksheetnum</code> of the Microsoft Excel workbook <code>workbookname</code> . <code>worksheetnum</code> defaults to 1 if not specified. Only numerical data is supported.
<b>See Also</b>	<a href="#">appl escript</a> Load a compiled AppleScript from a file and execute it <a href="#">xl setrange</a> Set range of cells in Microsoft Excel worksheet

<b>Purpose</b>	Set range of cells in Microsoft Excel worksheet
<b>Syntax</b>	<code>xl setrange(data, [rmin, cmin, rmax, cmax], workbookname, worksheetnum)</code>
<b>Description</b>	<code>xl setrange(data, [rmin, cmin, rmax, cmax], workbookname, worksheetnum)</code> sets the cells in the range $r < rmin > c < cmin : r < rmax > c < cmax$ of sheet worksheetnum of the Microsoft Excel workbook <code>workbookname</code> to <code>data</code> . <code>worksheetnum</code> defaults to 1 if not specified. Only numerical data is supported.
<b>See Also</b>	<code>appleScript</code> Load a compiled AppleScript from a file and execute it <code>xlgetrange</code> Get range of cells from Microsoft Excel worksheet

## xor

---

<b>Purpose</b>	Exclusive or															
<b>Syntax</b>	<code>C = xor(A, B)</code>															
<b>Description</b>	<code>C = xor(A, B)</code> performs an exclusive OR operation on the corresponding elements of arrays A and B. The resulting element $C(i, j, \dots)$ is logical true (1) if $A(i, j, \dots)$ or $B(i, j, \dots)$ , but not both, is nonzero.															
	<table><thead><tr><th><b>A</b></th><th><b>B</b></th><th><b>C</b></th></tr></thead><tbody><tr><td>zero</td><td>zero</td><td>0</td></tr><tr><td>zero</td><td>nonzero</td><td>1</td></tr><tr><td>nonzero</td><td>zero</td><td>1</td></tr><tr><td>nonzero</td><td>nonzero</td><td>0</td></tr></tbody></table>	<b>A</b>	<b>B</b>	<b>C</b>	zero	zero	0	zero	nonzero	1	nonzero	zero	1	nonzero	nonzero	0
<b>A</b>	<b>B</b>	<b>C</b>														
zero	zero	0														
zero	nonzero	1														
nonzero	zero	1														
nonzero	nonzero	0														
<b>Examples</b>	Given $A = [0 \ 0 \ \pi \ \text{eps}]$ and $B = [0 \ -2.4 \ 0 \ 1]$ , then  <code>C = xor(A, B)</code> <code>C =</code> <table><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr></table> To see where either A or B has a nonzero element and the other matrix does not,  <code>spy(xor(A, B))</code>	0	1	1	0											
0	1	1	0													
<b>See Also</b>	<table><tbody><tr><td><code>&amp;</code></td><td>Logical AND operator</td></tr><tr><td><code> </code></td><td>Logical OR operator</td></tr><tr><td><code>all</code></td><td>Test to determine if all elements are nonzero</td></tr><tr><td><code>any</code></td><td>Test for any nonzeros</td></tr><tr><td><code>find</code></td><td>Find indices and values of nonzero elements</td></tr></tbody></table>	<code>&amp;</code>	Logical AND operator	<code> </code>	Logical OR operator	<code>all</code>	Test to determine if all elements are nonzero	<code>any</code>	Test for any nonzeros	<code>find</code>	Find indices and values of nonzero elements					
<code>&amp;</code>	Logical AND operator															
<code> </code>	Logical OR operator															
<code>all</code>	Test to determine if all elements are nonzero															
<code>any</code>	Test for any nonzeros															
<code>find</code>	Find indices and values of nonzero elements															

---

<b>Purpose</b>	Create an array of all zeros								
<b>Syntax</b>	<pre>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))</pre>								
<b>Description</b>	<p><code>B = zeros(n)</code> returns an <math>n</math>-by-<math>n</math> matrix of zeros. An error message appears if <math>n</math> is not a scalar.</p> <p><code>B = zeros(m, n)</code> or <code>B = zeros([m n])</code> returns an <math>m</math>-by-<math>n</math> matrix of zeros.</p> <p><code>B = zeros(d1, d2, d3...)</code> or <code>B = zeros([d1 d2 d3...])</code> returns an array of zeros with dimensions <math>d_1</math>-by-<math>d_2</math>-by-<math>d_3</math>-by-<math>\dots</math>.</p> <p><code>B = zeros(size(A))</code> returns an array the same size as <math>A</math> consisting of all zeros.</p>								
<b>Remarks</b>	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.								
<b>Examples</b>	<p>With <math>n = 1000</math>, the for loop</p> <pre>for i = 1:n, x(i) = i; end</pre> <p>takes about 1.2 seconds to execute on a Sun SPARC-1. If the loop is preceded by the statement <code>x = zeros(1, n);</code> the computations require less than 0.2 seconds.</p>								
<b>See Also</b>	<table border="0"> <tr> <td><code>eye</code></td> <td>Identity matrix</td> </tr> <tr> <td><code>ones</code></td> <td>Create an array of all ones</td> </tr> <tr> <td><code>rand</code></td> <td>Uniformly distributed random numbers and arrays</td> </tr> <tr> <td><code>randn</code></td> <td>Normally distributed random numbers and arrays</td> </tr> </table>	<code>eye</code>	Identity matrix	<code>ones</code>	Create an array of all ones	<code>rand</code>	Uniformly distributed random numbers and arrays	<code>randn</code>	Normally distributed random numbers and arrays
<code>eye</code>	Identity matrix								
<code>ones</code>	Create an array of all ones								
<code>rand</code>	Uniformly distributed random numbers and arrays								
<code>randn</code>	Normally distributed random numbers and arrays								

**zeros**

---

**zeros**

---

**zeros**

---

**zeros**

---

**zeros**

---

# List of Commands

---

# Function Names

Arithmetic Operators + - * / \ ^	2-2
' .....	2-2
Relational Operators	
< > <= >= == ~= ~.....	2-9
Logical Operators &   ~	2-11
Special Characters [ ] () {} = ' . ... , ; % !	2-13
Colon :	2-16
abs .....	2-19
acopy .....	2-20
acos, acosh .....	2-21
acot, acoth .....	2-22
acsc,acsch .....	2-23
addpath .....	2-25
airy .....	2-26
all .....	2-28
amove .....	2-30
angle .....	2-31
ans .....	2-32
any .....	2-33
applescript .....	2-35
arename .....	2-36
areveal .....	2-37
asec, asech .....	2-38
asin, asinh .....	2-39
assignin .....	2-40
atan, atanh .....	2-41
atan2 .....	2-43
auread .....	2-44
auwrite .....	2-45
balance .....	2-47
base2dec .....	2-50
besselh .....	2-51
besseli, besselk .....	2-53
besselj, bessely .....	2-56
beta, betainc, betaln ..	2-59
bicg .....	2-61
bicgstab .....	2-68
bin2dec .....	2-72
bitand .....	2-73
bitcmp .....	2-74

bitget .....	2-75	corrcoef .....	2-140
bitmax .....	2-76	cos, cosh .....	2-141
bitor .....	2-77	cot, coth .....	2-142
bitset .....	2-78	cov .....	2-143
bitshift .....	2-79	cplxpairs .....	2-144
bitxor .....	2-80	cputime .....	2-145
blanks .....	2-81	cross .....	2-146
break .....	2-82	csc, csch .....	2-147
builtin .....	2-83	cumprod .....	2-148
calendar .....	2-85	cumsum .....	2-149
cart2pol .....	2-86	cumtrapz .....	2-150
cart2sph .....	2-88	date .....	2-155
case .....	2-89	datenum .....	2-156
cat .....	2-90	datestr .....	2-157
catch .....	2-91	datevec .....	2-159
cd .....	2-92	dbclear .....	2-160
cdf2rdf .....	2-93	dbcont .....	2-162
ceil .....	2-95	dbdown .....	2-163
cell .....	2-96	dblquad .....	2-164
cell2struct .....	2-97	dbmex .....	2-166
celldisp .....	2-98	dbquit .....	2-167
cellplot .....	2-99	dbstack .....	2-168
cellstr .....	2-100	dbstatus .....	2-169
cgs .....	2-101	dbstep .....	2-170
char .....	2-105	dbstop .....	2-171
chol .....	2-107	dbtype .....	2-174
cholinc .....	2-109	dbup .....	2-175
cholupdate .....	2-116	ddeadv .....	2-176
class .....	2-119	ddeexec .....	2-178
clear .....	2-120	ddeinit .....	2-179
clock .....	2-122	ddepoke .....	2-180
colmmd .....	2-123	ddereq .....	2-182
colperm .....	2-126	ddeterm .....	2-184
compan .....	2-127	ddeunadv .....	2-185
computer .....	2-128	deal .....	2-186
cond .....	2-130	deblank .....	2-189
condeig .....	2-131	dec2base .....	2-190
condest .....	2-132	dec2bin .....	2-191
conj .....	2-133	dec2hex .....	2-192
conv .....	2-134	deconv .....	2-193
conv2 .....	2-135	del2 .....	2-194
convhull .....	2-137	delaunay .....	2-197
convn .....	2-138	delete .....	2-200
copyfile .....	2-139	det .....	2-201

diag.....	2-202	fieldnames .....	2-266	help.....	2-365
diary .....	2-203	fileparts .....	2-267	hess.....	2-367
diff .....	2-204	filter .....	2-268	hex2dec .....	2-369
dir .....	2-206	filter2 .....	2-270	hex2num .....	2-370
disp .....	2-207	find .....	2-271	hilb .....	2-371
dlmread .....	2-208	findstr .....	2-273	i .....	2-372
dlmwrite .....	2-210	fix .....	2-274	if .....	2-373
dmperm .....	2-211	flipdim .....	2-275	ifft .....	2-375
doc .....	2-212	fliplr .....	2-276	ifft2 .....	2-376
double .....	2-213	flipud .....	2-277	ifftn .....	2-377
dsearch .....	2-214	floor .....	2-278	ifftshift .....	2-378
echo .....	2-215	flops .....	2-279	imag .....	2-379
edit .....	2-216	fmin .....	2-280	imfinfo .....	2-380
eig .....	2-217	fmins .....	2-282	imread .....	2-383
eigs .....	2-220	fopen.....	2-286	imwrite .....	2-386
ellipj .....	2-226	for .....	2-289	ind2sub .....	2-390
ellipke .....	2-228	format .....	2-291	Inf .....	2-391
else .....	2-230	fprintf .....	2-292	inferioro .....	2-392
elseif .....	2-231	fread .....	2-297	inline .....	2-393
end .....	2-233	freqspace .....	2-300	inmem .....	2-396
eomday .....	2-234	frewind .....	2-301	inpolygon .....	2-397
eps .....	2-235	fscanf .....	2-302	input .....	2-398
erf, erfc, erfcx, erfinv ...	2-236	fseek .....	2-305	inputname.....	2-399
error .....	2-238	fstell .....	2-306	int2str .....	2-400
errortrap .....	2-239	full .....	2-307	interp1 .....	2-401
etime .....	2-240	fullfile .....	2-308	interp2 .....	2-404
eval.....	2-241	function .....	2-309	interp3 .....	2-408
evalin .....	2-243	funm.....	2-311	interpft .....	2-410
exist .....	2-244	fwrite .....	2-313	interpnl .....	2-411
exp .....	2-246	fzero .....	2-316	intersect.....	2-413
expint .....	2-247	gallery .....	2-319	inv .....	2-414
expm .....	2-249	gamma, gammairc, gammaln		invhilb .....	2-417
eye .....	2-251	2-339		ipermute .....	2-418
factor .....	2-253	gcd .....	2-341	is* .....	2-419
fclose .....	2-254	gestalt .....	2-343	isa .....	2-423
feof .....	2-255	getfield.....	2-344	ismember .....	2-424
ferror .....	2-256	global .....	2-345	isstr.....	2-425
feval .....	2-257	gmres .....	2-347	j .....	2-430
fft .....	2-258	gradient .....	2-351	keyboard .....	2-431
fft2 .....	2-261	griddata .....	2-354	kron .....	2-432
fftn .....	2-262	gsvd .....	2-357	lasterr .....	2-433
fftshift .....	2-263	hadamard .....	2-362	lastwarn .....	2-435
fgetl .....	2-264	hankel .....	2-363	lcm .....	2-436
fgets .....	2-265	hdf.....	2-364	legendre .....	2-437

length .....	2-439	nonzeros .....	2-497	qrupdate .....	2-576
lin2mu .....	2-440	norm .....	2-498	qtwrite .....	2-579
linspace .....	2-441	normest.....	2-499	quad, quad8 .....	2-580
load .....	2-442	now .....	2-500	quit .....	2-582
log .....	2-444	null .....	2-501	qz .....	2-583
log2 .....	2-445	num2cell .....	2-502	rand .....	2-584
log10 .....	2-446	num2str .....	2-503	randn .....	2-586
logical .....	2-447	nzmax .....	2-504	randperm .....	2-588
logm.....	2-448	ode45, ode23, ode113, ode15s,		rank.....	2-589
logspace .....	2-450	ode23s, ode23t, ode23tb	2-505	rat, rats .....	2-590
lookfor.....	2-451	odefile .....	2-513	rcond .....	2-593
lower .....	2-452	odeget .....	2-519	readsnd .....	2-594
lscov.....	2-453	odeset .....	2-520	real .....	2-595
lu .....	2-454	ones .....	2-526	realmax .....	2-596
luinc .....	2-457	orth .....	2-527	realmin .....	2-597
magic .....	2-464	otherwise .....	2-528	recordsound .....	2-598
mat2str .....	2-466	pack .....	2-529	rem .....	2-599
matlabrc.....	2-467	partialpath .....	2-531	repmat .....	2-600
matlabroot .....	2-468	pascal .....	2-532	reshape .....	2-601
max .....	2-469	path .....	2-533	residue .....	2-602
mean .....	2-470	pause.....	2-535	return .....	2-604
median .....	2-471	pcg .....	2-536	rmfield .....	2-605
menu .....	2-472	pcode .....	2-540	rmpath .....	2-606
meshgrid .....	2-473	perms .....	2-541	roots.....	2-607
methods .....	2-475	permute .....	2-542	rot90 .....	2-609
mexext .....	2-476	persistent .....	2-543	round .....	2-610
mfilename .....	2-477	pi .....	2-544	rref, rrefmovie .....	2-611
min .....	2-478	pinv .....	2-545	rsf2csf .....	2-613
mislocked .....	2-479	pol2cart.....	2-548	save .....	2-616
mkdir .....	2-480	poly .....	2-550	schur .....	2-619
mlock .....	2-481	polyarea .....	2-553	script .....	2-621
mod .....	2-482	Polyder .....	2-554	sec, sech .....	2-622
more .....	2-483	polyeig .....	2-555	setdiff .....	2-624
mu2lin .....	2-484	polyfit .....	2-556	setfield .....	2-625
munlock .....	2-485	polyval .....	2-559	setstr .....	2-626
NaN.....	2-486	polyvalm .....	2-560	setxor .....	2-627
nargchk .....	2-487	pow2 .....	2-562	shiftdim .....	2-628
nargin, nargout .....	2-488	primes .....	2-563	sign .....	2-629
nchoosek.....	2-490	prod .....	2-564	sin, sinh .....	2-630
ndgrid .....	2-491	profile .....	2-565	size .....	2-632
ndims .....	2-492	qmr .....	2-567	sort .....	2-634
nextpow2 .....	2-493	qr .....	2-571	sortrows .....	2-635
nnls .....	2-494	qrdelete.....	2-574	sound .....	2-636
nnz .....	2-496	qrinsert.....	2-575	soundcap.....	2-637

soundsc .....	2-638	svd .....	2-700
spalloc .....	2-639	svds .....	2-702
sparse.....	2-640	switch .....	2-704
spconvert .....	2-642	symmmd .....	2-706
spdiags .....	2-644	symrcm .....	2-708
speak .....	2-647	tan, tanh .....	2-712
speye .....	2-648	tempdir .....	2-714
spfun .....	2-649	tempname .....	2-715
sph2cart .....	2-650	tic, toc .....	2-716
spline .....	2-651	toeplitz.....	2-717
spones .....	2-654	trace .....	2-718
spparms.....	2-655	trapz.....	2-719
sprand .....	2-658	tril.....	2-721
sprandn .....	2-659	triu .....	2-722
sprandsym .....	2-660	try .....	2-723
sprintf .....	2-661	tsearch .....	2-724
spy .....	2-665	type .....	2-725
sqrt.....	2-666	uint8.....	2-726
sqrtm .....	2-667	union .....	2-727
squeeze .....	2-670	unique .....	2-728
sscanf.....	2-671	unwrap .....	2-729
startup .....	2-674	upper .....	2-730
std .....	2-675	varargin, varargout .....	2-731
str2num.....	2-677	vectorize .....	2-733
strcat .....	2-678	version .....	2-734
strcmp .....	2-680	voronoi .....	2-735
strcmpi .....	2-682	warning .....	2-737
strings .....	2-683	wavread .....	2-738
strjust .....	2-684	wavwrite .....	2-739
strmatch .....	2-685	web .....	2-740
strncmp .....	2-686	weekday .....	2-741
strncmpi .....	2-687	what .....	2-742
strrep .....	2-688	whatsnew .....	2-744
strtok .....	2-689	which .....	2-745
struct .....	2-690	while.....	2-747
struct2cell .....	2-691	who, whos .....	2-748
strvcat .....	2-692	wilkinson.....	2-750
sub2ind .....	2-693	wk1read .....	2-751
subsasgn .....	2-694	wk1write .....	2-752
subsindex .....	2-695	writesnd .....	2-753
subsref .....	2-696	xlgetrange .....	2-754
subspace .....	2-697	xlsetrange .....	2-755
sum.....	2-698	xor .....	2-756
superiorto .....	2-699	zeros .....	2-757



## Symbols

! 2-13  
- 2-2  
% 2-13  
& 2-11  
' 2-2, 2-13  
( ) 2-13  
\* 2-2  
+ 2-2  
, 2-13  
. 2-13  
... 2-13  
/ 2-2  
: 2-16  
< 2-9  
= 2-13  
== 2-9  
> 2-9  
\ 2-2  
^ 2-2  
| 2-11  
~ 2-11  
~= 2-9  
2-9  
2-9

## Numerics

$\pi$  (pi) 2-544, 2-591, 2-630  
1-norm 2-498, 2-593  
2-norm (estimate of) 2-499

## A

abs **2-19**  
absolute value 2-19

accuracy  
of linear equation solution 2-130  
of matrix inversion 2-130  
relative floating-point 2-235  
acopy **2-20**  
acos **2-21**  
acosh **2-21**  
acot **2-22**  
acoth **2-22**  
acsca **2-23**  
acsch **2-23**  
Adams-Basforth-Moulton ODE solver 2-511  
addition (arithmetic operator) 2-2  
addpath **2-25**  
addressing selected array elements 2-16  
adjacency graph 2-211  
airy **2-26**  
aligning scattered data  
multi-dimensional 2-491  
two-dimensional 2-354  
all **2-28**  
allocation of storage (automatic) 2-757  
amove **2-30**  
and (M-file function equivalent for &) 2-11  
AND, logical  
bit-wise 2-73  
angle **2-31**  
ans **2-32**  
anti-diagonal 2-363  
any **2-33**  
applescript **2-35**  
arccosecant 2-23  
arccosine 2-21  
arccotangent 2-22  
arcsecant 2-38  
arcsine 2-39

- arctangent 2-41  
(four-quadrant) 2-43
- arename **2-36**
- arguments, M-file  
    checking number of input 2-487  
    number of input 2-488  
    number of output 2-488  
    passing variable numbers of 2-731
- arithmetic operations, matrix and array distinguished 2-2
- arithmetic operators 2-2
- array  
    addressing selected elements of 2-16  
    displaying 2-207  
    finding indices of 2-271  
    left division (arithmetic operator) 2-3  
    maximum elements of 2-469  
    maximum size of 2-128  
    mean elements of 2-470  
    median elements of 2-471  
    minimum elements of 2-478  
    multiplication (arithmetic operator) 2-2  
    of all ones 2-526  
    power (arithmetic operator) 2-3  
    product of elements 2-564  
    of random numbers 2-584, 2-586  
    removing first n singleton dimensions of 2-628  
    removing singleton dimensions of 2-670  
    reshaping 2-601  
    right division (arithmetic operator) 2-3  
    shifting dimensions of 2-628  
    size of 2-632  
    sorting elements of 2-634  
    structure 2-266, 2-344, 2-605, 2-625  
    sum of elements 2-698  
    swapping dimensions of 2-418, 2-542  
    transpose (arithmetic operator) 2-3
- of all zeros 2-757
- arrowhead matrix 2-126
- ASCII data  
    converting sparse matrix after loading from 2-642  
    printable characters (list of) 2-105  
    reading from disk 2-442  
    saving to disk 2-616
- ASCII delimited file  
    reading 2-208  
    writing matrix to 2-210
- asech **2-38**
- asin **2-39**
- asinh **2-39**
- atan **2-41**
- atan2 **2-43**
- atanh **2-41**
- auread **2-44**
- auwrite **2-45**
- average of array elements 2-470
- axis crossing *See* zero of a function
- azimuth (spherical coordinates) 2-650
- B**
- badly conditioned 2-593
- balance **2-47**
- bank format 2-291
- base to decimal conversion 2-50
- base two operations  
    conversion from decimal to binary 2-191  
    logarithm 2-445  
    next power of two 2-493
- base2dec **2-50**
- Bessel functions 2-51, 2-56  
    first kind 2-53  
    modified 2-53

second kind 2-54  
 third kind 2-57  
**Bessel's equation**  
 (defined) 2-51, 2-56  
 modified (defined) 2-53  
**bessel h 2-51**  
**bessel i 2-53**  
**bessel j 2-56**  
**bessel k 2-53**  
**bessel y 2-56**  
**beta 2-59**  
**beta function**  
 (defined) 2-59  
 incomplete (defined) 2-59  
 natural logarithm of 2-59  
**betainc 2-59**  
**betaln 2-59**  
**bicgstab 2-68**  
 Big Endian formats 2-287  
**bin2dec 2-72**  
 binary data  
     reading from disk 2-442  
     saving to disk 2-616  
     writing to file 2-313  
 binary to decimal conversion 2-72  
**bitand 2-73**  
**bitcmp 2-74**  
**bitget 2-75**  
**bitmax 2-76**  
**bitor 2-77**  
**bitset 2-78**  
**bitsshift 2-79**  
 bit-wise operations  
     AND 2-73  
     get 2-75  
     OR 2-77  
     set bit 2-78  
     shift 2-79  
     XOR 2-80  
**bitxor 2-80**  
 blanks  
     removing trailing 2-189  
**blanks 2-81**  
 braces, curly (special characters) 2-13  
 brackets (special characters) 2-13  
**break 2-82**  
 breakpoints  
     listing 2-169  
     setting 2-171  
 Buckminster Fuller 2-708  
**builtin 2-83**  
 built-in functions 2-745

**C**

calendar 2-85  
**cart2pol 2-86**  
**cart2sph 2-88**  
 Cartesian coordinates 2-86, 2-88, 2-548, 2-650  
**case**  
     in switch statement (defined) 2-704  
     lower to upper 2-730  
     upper to lower 2-452  
**case 2-89**  
**cat 2-90**  
**catch 2-91**  
 catching errors 2-241  
 Cayley-Hamilton theorem 2-561  
**cd 2-92**  
**cdf2rdf 2-93**  
**ceil 2-95**  
 cell array  
     conversion to from numeric array 2-502  
     creating 2-96

structure of, displaying 2-99  
**cell2struct 2-97**  
**cellplot 2-99**  
**cgs 2-101**  
changing working directory 2-92  
**char 2-105**  
characters (in format specification string)  
    conversion 2-294, 2-663  
    escape 2-293, 2-662  
checkerboard pattern (example) 2-600  
**chol 2-107**  
Cholesky factorization 2-107  
    (as algorithm for solving linear equations) 2-6  
    lower triangular factor 2-532  
    minimum degree ordering and (sparse) 2-706  
    preordering for 2-126  
**cholinc 2-109**  
**cholinc 2-109**  
**cholupdate 2-116**  
**class 2-119**  
class, object *See* object classes  
**clear 2-120**  
clearing  
    variables from workspace 2-120  
**clock 2-122**  
collapse dimensions, functions that 2-29  
**colmmd 2-123**  
**colperm 2-126**  
combinations of n elements 2-490  
**combs 2-490**  
comma (special characters) 2-15  
command window  
    controlling number of lines per page in 2-483  
common elements *See* set operations,  
    intersection  
**compan 2-127**  
companion matrix 2-127

complementary error function  
    (defined) 2-236  
    scaled (defined) 2-236  
complete elliptic integral  
    (defined) 2-228  
    modulus of 2-226, 2-228  
complex  
    exponential (defined) 2-246  
    logarithm 2-444, 2-446  
    modulus (magnitude) 2-19  
    numbers 2-372  
    numbers, sorting 2-634, 2-635  
    phase angle 2-31  
    unitary matrix 2-571  
    *See also* imaginary  
complex conjugate 2-133  
    sorting pairs of 2-144  
complex Schur form 2-619  
**computer 2-128**  
computers supported by MATLAB  
    numeric file formats of 2-287  
    numeric precision of 2-298, 2-313  
concatenating arrays 2-90  
**cond 2-130**  
**condeig 2-131**  
**condest 2-132**  
condition number of matrix 2-47, 2-130, 2-593  
    estimated 2-132  
conditional execution *See* flow control  
**conj 2-133**  
conjugate, complex 2-133  
    sorting pairs of 2-144  
contents file (contents.m) 2-365  
continuation (... , special characters) 2-14  
continued fraction expansion 2-590  
**conv 2-134**  
**conv2 2-135**

- conversion
- base to decimal 2-50
  - binary to decimal 2-72
  - Cartesian to cylindrical 2-86
  - Cartesian to polar 2-86
  - complex diagonal to real block diagonal 2-93
  - cylindrical to Cartesian 2-548
  - decimal number to base 2-186, 2-190
  - decimal to binary 2-191
  - decimal to hexadecimal 2-192
  - full to sparse 2-640
  - hexadecimal to decimal 2-369
  - hexadecimal to double precision 2-370
  - integer to string 2-400
  - lowercase to uppercase 2-730
  - matrix to string 2-466
  - numeric array to cell array 2-502
  - numeric array to logical array 2-447
  - numeric array to string 2-503
  - partial fraction expansion to pole-residue 2-602
  - polar to Cartesian 2-548
  - pole-residue to partial fraction expansion 2-602
  - real to complex Schur form 2-613
  - spherical to Cartesian 2-650
  - string matrix to cell array 2-100
  - string to matrix (formatted) 2-671
  - string to numeric array 2-677
  - uppercase to lowercase 2-452
  - vector to character string 2-105
- conversion characters (in format specification string) 2-294, 2-663
- convhul l 2-137**
- convn 2-138**
- convolution 2-134
- inverse *See deconvolution*
- two-dimensional 2-135
- coordinates
- Cartesian 2-86, 2-88, 2-548, 2-650
  - cylindrical 2-86, 2-88, 2-548
  - polar 2-86, 2-88, 2-548
  - spherical 2-650
- See also* conversion
- copyfile 2-139**
- corrcoef 2-140**
- cos 2-141**
- cosecant 2-147
- hyperbolic 2-147
  - inverse 2-23
  - inverse hyperbolic 2-23
- cosh 2-141**
- cosine 2-141
- hyperbolic 2-141
  - inverse 2-21
  - inverse hyperbolic 2-21
- cot 2-142**
- cotangent 2-142
- hyperbolic 2-142
  - inverse 2-22
  - inverse hyperbolic 2-22
- coth 2-142**
- cov 2-143**
- covariance
- least squares solution and 2-453
- cpl xpai r 2-144**
- cputime 2-145**
- creating online help for your own M-files 2-365
- creating your own MATLAB functions 2-309
- cross 2-146**
- cross product 2-146
- csc 2-147**
- csch 2-147**
- ctranspose (M-file function equivalent for ' ) 2-4**

cubic interpolation 2-401, 2-404  
cubic spline interpolation 2-401, 2-404, 2-408,  
    2-411  
**cumprod 2-148**  
**cumsum 2-149**  
**cumtrapz 2-150**  
cumulative  
    product 2-148  
    sum 2-149  
curly braces (special characters) 2-13  
curve fitting (polynomial) 2-556  
customizing your workspace 2-467, 2-674  
Cuthill-McKee ordering, reverse 2-706, 2-708  
cylindrical coordinates 2-86, 2-88, 2-548

**D**

data, aligning scattered  
    multi-dimensional 2-491  
    two-dimensional 2-354  
data, ASCII  
    converting sparse matrix after loading from  
        2-642  
    reading from disk 2-442  
    saving to disk 2-616  
data, binary  
    dependence upon array size and type 2-617  
    reading from disk 2-442  
    saving to disk 2-616  
    writing to file 2-313  
data, formatted  
    reading from file 2-302  
    writing to file 2-292  
**date 2-155**  
date and time functions 2-234  
date string  
    format of 2-157

date vector 2-159  
**datenum 2-156**  
**datestr 2-157**  
**datevec 2-159**  
**dbclear 2-160**  
**dbcont 2-162**  
**dbdown 2-163**  
**dbmex 2-166**  
**dbquit 2-167**  
**dbstack 2-168**  
**dbstatus 2-169**  
**dbstep 2-170**  
**dbstop 2-171**  
**dbtype 2-174**  
**dbup 2-175**  
**ddeadv 2-176**  
**ddeexec 2-178**  
**ddeinit 2-179**  
**ddepoke 2-180**  
**ddereq 2-182**  
**ddterm 2-184**  
**ddeunadv 2-185**  
**deal 2-186**  
**deblank 2-189**  
debugging  
    M-files 2-160-2-175, 2-431  
    quitting debug mode 2-167  
**dec2base 2-186, 2-190**  
**dec2bin 2-191**  
**dec2hex 2-192**  
decimal number to base conversion 2-186, 2-190  
decimal point (.)  
    (special characters) 2-14  
    to distinguish matrix and array operations 2-2  
decomposition  
    Dulmage-Mendelsohn 2-211  
    “economy-size” 2-571, 2-700

- orthogonal-triangular (QR) 2-453, 2-571
- Schur 2-619
- singular value 2-589, 2-700
- deconv 2-193**
- deconvolution 2-193
- default tolerance 2-235
- definite integral 2-580
- del operator** 2-194
- del 2-194**
- del aunay 2-197**
- del ete 2-200**
- deleting
  - files 2-200
  - workspace variables 2-120
- delimiter
  - in ASCII files 2-208, 2-210
- density
  - of sparse matrix 2-496
- dependence, linear 2-697
- derivative
  - approximate 2-204
  - polynomial 2-554
- det 2-201**
- Detect 2-419
- detecting
  - alphabetic characters 2-420
  - empty arrays 2-419
  - equal arrays 2-419
  - finite numbers 2-419
  - global variables 2-420
  - infinite elements 2-420
  - logical arrays 2-420
  - members of a set 2-424
  - NaNs 2-420
  - objects of a given class 2-423
  - positive, negative, and zero array elements 2-629
- prime numbers 2-421
- real numbers 2-421
- determinant of a matrix 2-201
- di ag 2-202**
- diagonal 2-202
  - anti- 2-363
  - k-th (illustration) 2-721
  - main 2-202
  - sparse 2-644
- di ary 2-203**
- di ff 2-204**
- differences
  - between adjacent array elements 2-204
  - between sets 2-624
- differential equation solvers 2-505
- adjusting parameters of 2-520
- extracting properties of 2-519
- digits
  - controlling number of displayed 2-291
- dimension statement (lack of in MATLAB) 2-757
- dimensions
  - functions that collapse 2-29
  - size of 2-632
- Diophantine equations 2-341
- di r 2-206**
- direct term of a partial fraction expansion 2-602
- directory
  - changing working 2-92
  - listing contents of 2-206, 2-742
  - root 2-468
  - temporary system 2-714
  - See also* search path
- discontinuities, eliminating (in arrays of phase angles) 2-729
- di sp 2-207**
- distribution
  - Gaussian 2-236

division  
 array, left (arithmetic operator) 2-3  
 array, right (arithmetic operator) 2-3  
 by zero 2-391  
 matrix, left (arithmetic operator) 2-3  
 matrix, right (arithmetic operator) 2-2  
 modulo 2-482  
 of polynomials 2-193  
 remainder after 2-599

divisor  
 greatest common 2-341

**dlmread 2-208**

**dlmwrit e 2-210**

**dmperm 2-211**

**doc 2-212**

documentation, HTML 2-212

dot product 2-146

**doubl e 2-213**

**dsearch 2-214**

dual vector 2-494

Dulmage-Mendelsohn decomposition 2-211

**E**

**echo 2-215**

edge finding, Sobel technique 2-135

**eig 2-217**

eigen system  
 transforming 2-93

eigen value  
 accuracy of 2-47, 2-217  
 complex 2-93  
 matrix logarithm and 2-448  
 modern approach to computation of 2-551  
 of companion matrix 2-127  
 poorly conditioned 2-47  
 problem 2-217, 2-555

problem, generalized 2-218, 2-555  
 problem, polynomial 2-555  
 repeated 2-218, 2-311  
 Wilkinson test matrix and 2-750

eigenvector  
 left 2-217  
 matrix, generalized 2-583  
 right 2-217

**ei gs 2-220**

elevation (spherical coordinates) 2-650

**elli pj 2-226**

**elli pke 2-228**

elliptic functions, Jacobian  
 (defined) 2-226

elliptic integral  
 complete (defined) 2-228  
 modulus of 2-226, 2-228

**el se 2-230**

**el sei f 2-231**

**end 2-233**

end of line, indicating 2-15

end-of-file indicator 2-255

**eomday 2-234**

**eps 2-235**

equal sign (special characters) 2-14

equations, linear  
 accuracy of solution 2-130

**erf 2-236**

**erfc 2-236**

**erfcx 2-236**

error  
 catching 2-241, 2-433  
 in file I/O 2-256  
 roundoff *See* roundoff error

**error 2-238**

error function  
 (defined) 2-236

- complementary 2-236  
 scaled complementary 2-236
- error message  
 displaying 2-238  
`Index into matrix is negative or zero`  
 2-447  
`Out of memory` 2-529  
 retrieving last generated 2-433
- escape characters (in format specification string)  
 2-293, 2-662
- etime** **2-240**
- eval** **2-241**
- evalin** **2-243**
- evaluating strings 2-241
- exclamation point (special characters) 2-15
- executing statements repeatedly 2-289, 2-747
- execution  
 conditional *See* flow control  
 improving speed of by setting aside storage  
 2-757  
 pausing M-file 2-535  
 resuming from breakpoint 2-162
- exist** **2-244**
- exp** **2-246**
- expint** **2-247**
- expm** **2-249**
- exponential 2-246  
 complex (defined) 2-246  
 integral 2-247  
 matrix 2-249
- exponentiation  
 array (arithmetic operator) 2-3  
 matrix (arithmetic operator) 2-3
- expression, MATLAB 2-373
- extension, filename  
`.m` 2-309, 2-725  
`.mat` 2-442, 2-616, 2-617
- eye** **2-251**
- F**
- factor** **2-253**
- factorization  
 LU 2-454  
 QZ 2-555, 2-583  
*See also* decomposition
- factorization, Cholesky 2-107  
 (as algorithm for solving linear equations) 2-6  
 minimum degree ordering and (sparse) 2-706  
 preordering for 2-126
- factors, prime 2-253
- fclose** **2-254**
- feof** **2-255**
- ferror** **2-256**
- feval** **2-257**
- fft** **2-258**
- FFT *See* Fourier transform
- fft2** **2-261**
- fftn** **2-262**
- fftshift** **2-263**
- fgetl** **2-264**
- fgets** **2-265**
- fid** *See* file identifier
- field names of a structure, obtaining 2-266
- fields, noncontiguous  
 inserting data into 2-313
- file  
 closing 2-254  
 deleting 2-200  
 finding position within 2-306  
 listing contents of 2-725  
 listing names of 2-206  
 MATLAB format 2-442  
 opening 2-286

reading ASCII delimited 2-208  
reading formatted data from 2-302  
returning next line of (with carriage returns) 2-264  
returning next line of (without carriage returns) 2-265  
rewinding to beginning of 2-301  
setting position within 2-305  
status of 2-244  
testing for end of 2-255  
text versus binary, opening 2-287  
writing binary data to 2-313  
writing formatted data to 2-292  
writing matrix as ASCII delimited 2-210

file identifier (`fild`) 2-255  
file position indicator  
  finding 2-306  
  setting 2-305  
  setting to start of file 2-301  
filename extension  
  `.m` 2-309, 2-725  
  `.mat` 2-442, 2-616, 2-617

`fileparts` **2-267**

filter 2-268  
  two-dimensional 2-135

`filter` **2-268**

`filter2` **2-270**

`find` **2-271**

finding  
  file position indicator 2-306  
  indices of arrays 2-271  
  sign of array elements 2-629  
  zero of a function 2-316  
  *See also* detecting

`findstr` **2-273**

finite numbers  
  detecting 2-419

FIR filter *See* filter

`fix` **2-274**

fixed-point  
  output format 2-291

`flint` *See* floating-point, integer

`flipdim` **2-275**

`flipplr` **2-276**

`flipud` **2-277**

floating-point  
  integer 2-74, 2-78  
  integer, maximum 2-76  
  numbers, interval between 2-235  
  operations, count of 2-279  
  output format 2-291

floating-point arithmetic, IEEE  
  largest positive number 2-596  
  relative accuracy of 2-235  
  smallest positive number 2-597

`floor` **2-278**

`flops` **2-279**

flow control  
  break 2-82  
  case 2-89  
  else 2-230  
  elseif 2-231  
  end 2-233  
  error 2-238  
  for 2-289  
  if 2-373  
  keyboard 2-431  
  otherwise 2-528  
  return 2-604  
  switch 2-704  
  while 2-747

`fmin` **2-280**

`fmins` **2-282**

F-norm 2-498

**fopen** **2-286**  
**for** **2-289**  
**format**  
  for saving ASCII data 2-616  
  output display 2-291  
**format** **2-291**  
**format specification string** 2-292, 2-661  
  matching file data to 2-302, 2-672  
**formatted data**  
  reading from file 2-302  
  writing to file 2-292  
**Fourier transform**  
  algorithm, optimal performance of 2-259, 2-375,  
  2-376, 2-493  
  convolution theorem and 2-134  
  discrete, one-dimensional 2-258  
  discrete, two-dimensional 2-261  
  fast 2-258  
  as method of interpolation 2-410  
  inverse, one-dimensional 2-375  
  inverse, two-dimensional 2-376  
  shifting the DC component of 2-263  
**fprintf** **2-292**  
**fraction**, continued 2-590  
**fragmented memory** 2-529  
**fread** **2-297**  
**freqspace** **2-300**  
**frequency response**  
  desired response matrix  
    frequency spacing 2-300  
**frequency vector** 2-450  
**frewind** **2-301**  
**fscanf** **2-302**  
**fseek** **2-305**  
**ftell** **2-306**  
**full** **2-307**  
**function**  
  locating 2-451, 2-745  
  minimizing (several variables) 2-282  
  minimizing (single variable) 2-280  
**function** **2-309**  
**functions**  
  that accept function name strings 2-257  
  that work down the first non-singleton dimension 2-628  
**funm** **2-311**  
**fwrite** **2-313**  
**fzero** **2-316**

**G**  
**gallery** **2-319**  
**gamma** **2-339**  
**gamma function**  
  (defined) 2-339  
  incomplete 2-339  
  logarithm of 2-339  
**gammainc** **2-339**  
**gammaln** **2-339**  
**Gaussian distribution function** 2-236  
**Gaussian elimination**  
  (as algorithm for solving linear equations) 2-7,  
  2-414  
  Gauss Jordan elimination with partial pivoting  
  2-611  
  LU factorization and 2-454  
**gcd** **2-341**  
**generalized eigenvalue problem** 2-218, 2-555  
**generating a sequence of matrix names (M1  
  through M12)** 2-241  
**geodesic dome** 2-708  
**gestalt** **2-343**  
**getfield** **2-344**  
**Givens rotations** 2-574, 2-575

global **2-345**  
global variable  
  clearing 2-120  
  defining 2-345  
**gmres 2-347**  
**gradient 2-351**  
gradient, numerical 2-351  
graph  
  adjacency 2-211  
graphics objects  
  deleting 2-200  
greatest common divisor 2-341  
grid  
  aligning data to a 2-354  
grid arrays  
  for volumetric plots 2-473  
  multi-dimensional 2-491  
**griddata 2-354**  
**gsvd 2-357**

**H**  
H1 line 2-366, 2-451  
**hadamard 2-362**  
Hadamard matrix 2-362  
  subspaces of 2-697  
Hager's method 2-132  
**hankel 2-363**  
Hankel functions, relationship to Bessel of 2-57  
Hankel matrix 2-363  
**hdf 2-364**  
help  
  keyword search 2-451  
  online 2-365  
**help 2-365**  
Help Desk 2-212  
Hermite transformations, elementary 2-341

**hess 2-367**  
Hessenberg form of a matrix 2-367  
**hex2dec 2-369**  
**hex2num 2-370**  
hexadecimal format 2-291  
**hilb 2-371**  
Hilbert matrix 2-371  
  inverse 2-417  
horzcat (M-file function equivalent for [ , ]) 2-15  
Householder reflections (as algorithm for solving  
  linear equations) 2-7  
HTML documentation 2-212  
hyperbolic  
  cosecant 2-147  
  cosecant, inverse 2-23  
  cosine 2-141  
  cosine, inverse 2-21  
  cotangent 2-142  
  cotangent, inverse 2-22  
  secant 2-38, 2-622  
  secant, inverse 2-38  
  sine 2-39, 2-630  
  sine, inverse 2-39  
  tangent 2-41, 2-712  
  tangent, inverse 2-41  
hyperplanes, angle between 2-697

**I**  
**i 2-372**  
identity matrix 2-251  
  sparse 2-648  
IEEE floating-point arithmetic  
  largest positive number 2-596  
  relative accuracy of 2-235  
  smallest positive number 2-597  
**if 2-373**

**i fft** **2-375**  
**i fft2** **2-376**  
**i fftn** **2-377**  
**i fftshift** **2-378**  
**IIR filter** *See filter*  
**i mag** **2-379**  
**imaginary**  
  part of complex number 2-379  
  parts of inverse FFT 2-375, 2-376  
  unit ( $\text{sqrt}(-1)$ ) 2-372, 2-430  
  *See also* complex  
**i mfi nfo** **2-380**  
**i mread** **2-383**  
**i mw rite** **2-386**  
**incomplete**  
  beta function (defined) 2-59  
  gamma function (defined) 2-339  
**i nd2sub** **2-390**  
**Index into matrix is negative or zero (error message)** 2-447  
**indexing**  
  logical 2-447  
**indicator**  
  end-of-file 2-255  
  file position 2-301, 2-305, 2-306  
**indices, array**  
  finding 2-271  
  of sorted elements 2-634  
**Inf** **2-391**  
**i nferior to** **2-392**  
**infinity** 2-391, 2-420  
  norm 2-498  
**inheritance, of objects** 2-119  
**i nl i ne** **2-393**  
**i npolygon** **2-397**  
**input**  
  checking number of M-file arguments 2-487  
**name of array passed as** 2-399  
**number of M-file arguments** 2-488  
**prompting users for** 2-398, 2-472  
**i nput** **2-398**  
**installation, root directory of** 2-468  
**i nt2str** **2-400**  
**integer**  
  floating-point 2-74, 2-78  
  floating-point, maximum 2-76  
**integrable singularities** 2-581  
**integration**  
  quadrature 2-580  
**i nterp1** **2-401**  
**i nterp2** **2-404**  
**i nterp3** **2-408**  
**i nterpf** **2-410**  
**i nterpn** **2-411**  
**interpolation**  
  one-dimensional 2-401  
  two-dimensional 2-404  
  three-dimensional 2-408  
  multidimensional 2-411  
  cubic method 2-354, 2-401, 2-404, 2-408, 2-411  
  cubic spline method 2-401  
  FFT method 2-410  
  linear method 2-401, 2-404  
  nearest neighbor method 2-354, 2-401, 2-404, 2-408, 2-411  
  trilinear method 2-354, 2-408, 2-411  
**interpreter, MATLAB**  
  search algorithm of 2-310  
**i ntersect** **2-413**  
**i nv** **2-414**  
**inverse**  
  cosecant 2-23  
  cosine 2-21  
  cotangent 2-22

Fourier transform 2-375, 2-376  
 four-quadrant tangent 2-43  
 Hilbert matrix 2-417  
 hyperbolic cosecant 2-23  
 hyperbolic cosine 2-21  
 hyperbolic cotangent 2-22  
 hyperbolic secant 2-38  
 hyperbolic sine 2-39  
 hyperbolic tangent 2-41  
 of a matrix 2-414  
 secant 2-38  
 sine 2-39  
 tangent 2-41  
 inversion, matrix  
     accuracy of 2-130  
**i nvhi lb 2-417**  
 involuntary matrix 2-532  
**i permute 2-418**  
**i s\* 2-419**  
**i sa 2-423**  
**i scell 2-419**  
**i scellstr 2-419**  
**i schar 2-419**  
**i seempty 2-419**  
**i sequal 2-419**  
**i sfi eld 2-419**  
**i sfini te 2-419**  
**i sgl obal 2-420**  
**i shand le 2-420**  
**i shold 2-420**  
**i si eee 2-420**  
**i si nf 2-420**  
**i sletter 2-420**  
**i slogi cal 2-420**  
**i smember 2-424**  
**i snan 2-420**  
**i snumeric 2-420**

**i sobj ect 2-420**  
**i sppc 2-421**  
**i spri me 2-421**  
**i sreal 2-421**  
**i sspace 2-421**  
**i ssparse 2-421**  
**i ssstr 2-425**  
**i ssstruct 2-421**  
**i ssstudent 2-421**  
**i suni x 2-421**  
**i svms 2-421**

**J**

**j 2-430**  
 Jacobi rotations 2-660  
 Jacobian elliptic functions  
     (defined) 2-226  
 joining arrays *See* concatenating arrays

**K**

K>> prompt 2-431  
 keyboard **2-431**  
 keyboard mode 2-431  
     terminating 2-604  
**kron 2-432**  
 Kronecker tensor product 2-432

**L**

labeling  
     matrix columns 2-207  
     plots (with numeric values) 2-503  
 Laplacian 2-194  
 largest array elements 2-469  
**lasterr 2-433**

**lastwarn** **2-435**  
**lcm** **2-436**  
**ldivide** (M-file function equivalent for .\ ) **2-4**  
**least common multiple** **2-436**  
**least squares**

- polynomial curve fitting **2-556**
- problem **2-453**
- problem, nonnegative **2-494**
- problem, overdetermined **2-545**

**legendre** **2-437**  
**Legendre functions**

- (defined) **2-437**
- Schmidt semi-normalized **2-437**

**length** **2-439**  
**lin2mu** **2-440**  
**line numbers**

- M-file, listing **2-174**
- linear dependence (of data) **2-697**
- linear equation systems
  - accuracy of solution **2-130**
  - solving overdetermined **2-572**-**2-573**
- linear equation systems, methods for solving
  - Cholesky factorization **2-6**
  - Gaussian elimination **2-7**
  - Householder reflections **2-7**
  - least squares **2-494**
    - matrix inversion (inaccuracy of) **2-414**
- linear interpolation **2-401**, **2-404**
- linearly spaced vectors, creating **2-441**
- lines per page, controlling in command window  
    **2-483**

- linspace** **2-441**  
**listing**
- breakpoints **2-169**
- directory contents **2-206**
- file contents **2-725**
- line numbers **2-174**

**M-files, MAT-files, and MEX-files** **2-742**  
**workspace variables** **2-748**  
**Little Endian formats** **2-287**  
**load** **2-442**  
**loading**

- WK1 spreadsheet files **2-751**
- local variables **2-309**, **2-345**
- locating MATLAB functions **2-451**, **2-745**

**log** **2-444**  
**log of MATLAB session, creating** **2-203**  
**log10** [**log010**] **2-446**  
**log2** **2-445**  
**logarithm**

- base ten **2-446**
- base two **2-445**
- complex **2-444**, **2-446**
- matrix (natural) **2-448**
- natural **2-444**
- of beta function (natural) **2-59**
- of gamma function (natural) **2-339**

**logarithmically spaced vectors, creating** **2-450**  
**logical** **2-447**  
**logical array**

- converting numeric array to **2-447**
- detecting **2-420**

**logical indexing** **2-447**  
**logical operations**

- AND, bit-wise **2-73**
- OR, bit-wise **2-77**
- XOR **2-756**
- XOR, bit-wise **2-80**

**logical operators** **2-11**  
**logical tests**

- all **2-28**
- any **2-33**
- See also* detecting

**logm** **2-448**

**l**ogspace **2-450**  
**l**ookfor **2-451**  
Lotus123 WK1 spreadsheet file  
  reading data from a **2-751**  
  writing a matrix to **2-752**  
**l**ower **2-452**  
lower triangular matrix **2-721**  
lowercase to uppercase **2-730**  
**l**scov **2-453**  
**l**u **2-454**  
LU factorization **2-454**  
  storage requirements of (sparse) **2-504**  
**l**uinc **2-457**

**M**

machine epsilon **2-747**  
**magi**c **2-464**  
magic squares **2-464**  
**mat2str** **2-466**  
MAT-file **2-442**, **2-617**  
  converting sparse matrix after loading from  
  **2-642**  
  listing **2-742**  
MATLAB format files **2-442**  
MATLAB interpreter  
  search algorithm of **2-310**  
MATLAB search path  
  adding directories to **2-25**  
  removing directories from **2-606**  
MATLAB startup file **2-467**, **2-674**  
MATLAB version number **2-734**  
**matlab.mat** **2-442**, **2-616**  
**matlabrc** **2-467**  
**matlabroot** **2-468**  
matrix  
  addressing selected rows and columns of **2-16**

arrowhead **2-126**  
companion **2-127**  
complex unitary **2-571**  
condition number of **2-47**, **2-130**, **2-593**  
converting to formatted data file **2-292**  
converting to vector **2-16**  
decomposition **2-571**  
defective (defined) **2-218**  
determinant of **2-201**  
diagonal of **2-202**  
Dulmage-Mendelsohn decomposition of **2-211**  
estimated condition number of **2-132**  
evaluating functions of **2-311**  
exponential **2-249**  
flipping left-right **2-276**  
flipping up-down **2-277**  
Hadamard **2-362**, **2-697**  
Hankel **2-363**  
Hermitian Toeplitz **2-717**  
Hessenberg form of **2-367**  
Hilbert **2-371**  
identity **2-251**  
inverse **2-414**  
inverse Hilbert **2-417**  
inversion, accuracy of **2-130**  
involuntary **2-532**  
left division (arithmetic operator) **2-3**  
lower triangular **2-721**  
magic squares **2-464**, **2-698**  
maximum size of **2-128**  
modal **2-217**  
multiplication (defined) **2-2**  
orthonormal **2-571**  
Pascal **2-532**, **2-560**  
permutation **2-454**, **2-571**  
poorly conditioned **2-371**  
power (arithmetic operator) **2-3**

- pseudoinverse 2-545
- reduced row echelon form of 2-611
- replicating 2-600
- right division (arithmetic operator) 2-2
- Rosser 2-334
- rotating  $90^\circ$  2-609
- Schur form of 2-613, 2-619
- singularity, test for 2-201
- sorting rows of 2-635
- sparse *See* sparse matrix
- specialized 2-319
- square root of 2-667
- storing as binary data 2-313
- subspaces of 2-697
- test 2-319
- Toeplitz 2-717
- trace of 2-202, 2-718
- transpose (arithmetic operator) 2-3
- transposing 2-14
- unimodular 2-341
- unitary 2-700
- upper triangular 2-722
- Vandermonde 2-558
- Wilkinson 2-645, 2-750
- writing formatted data to 2-302
- See also* array
- matrix functions
  - evaluating 2-311
- matrix names, (M1 through M12) generating a sequence of 2-241
- matrix power *See* matrix, exponential
- max **2-469**
- maximum array size 2-128
- mean **2-470**
- median **2-471**
- median value of array elements 2-471
- memory, consolidating information to minimize use of 2-529
- menu **2-472**
- menu (of user input choices) 2-472
- meshgrid **2-473**
- message
  - error *See* error message
  - warning *See* warning message
- methods
  - inheritance of 2-119
- MEX-file
  - clearing from memory 2-120
  - listing 2-742
- M-file
  - debugging 2-160-2-175, 2-431
  - displaying during execution 2-215
  - function 2-309
  - function file, echoing 2-215
  - listing 2-742
  - naming conventions 2-309
  - pausing execution of 2-535
  - programming 2-309
  - script 2-309
  - script file, echoing 2-215
- min **2-478**
- minimizing, function
  - of one variable 2-280
  - of several variables 2-282
- minimum degree ordering 2-706
- minus (M-file function equivalent for  $-$ ) 2-4
- misplaced **2-479**
- mkdir **2-480**
- mldivide (M-file function equivalent for  $\backslash$ ) 2-4
- mlock **2-481**
- mod **2-482**
- modal matrix 2-217
- modulo arithmetic 2-482

**modulus, complex** 2-19  
**Moore-Penrose pseudoinverse** 2-545  
**more** **2-483, 2-484**  
**mpower** (M-file function equivalent for  $\wedge$ ) 2-4  
**mrdivide** (M-file function equivalent for  $/$ ) 2-4  
**mtimes** (M-file function equivalent for  $*$ ) 2-4  
**mu2lin** **2-484**  
**multidimensional arrays**  
  concatenating 2-90  
  interpolation of 2-411  
  longest dimension of 2-439  
  number of dimensions of 2-492  
  rearranging dimensions of 2-418, 2-542  
  removing singleton dimensions of 2-670  
  reshaping 2-601  
  size of 2-632  
  sorting elements of 2-634  
  *See also* array  
**multiple**  
  least common 2-436  
**multiplication**  
  array (arithmetic operator) 2-2  
  matrix (defined) 2-2  
  of polynomials 2-134  
**multistep ODE solver** 2-511  
**munlock** **2-485**

**N**

**naming conventions**  
  M-file 2-309  
**NaN** **2-486**  
  NaN (Not-a-Number) 2-420, 2-486  
    returned by **rem** 2-599  
**nargchk** **2-487**  
**nargin** **2-488**  
**nargout** **2-488**

**ndgrid** **2-491**  
**ndims** **2-492**  
  nearest neighbor interpolation 2-354, 2-401, 2-404  
  Nelder-Mead simplex search 2-284  
**nextpow2** **2-493**  
**nnz** **2-494**  
**noncontiguous fields**  
  inserting data into 2-313  
**nonzero entries**  
  number of in sparse matrix 2-640  
**nonzero entries (in sparse matrix)**  
  allocated storage for 2-504  
  number of 2-496  
  replacing with ones 2-654  
  vector of 2-497  
**nonzeros** **2-497**  
**norm**  
  1-norm 2-498, 2-593  
  2-norm (estimate of) 2-499  
  F-norm 2-498  
  infinity 2-498  
  matrix 2-498  
  pseudoinverse and 2-545-2-547  
  vector 2-498  
**norm** **2-498**  
**normest** **2-499**  
**not** (M-file function equivalent for  $\sim$ ) 2-11  
**now** **2-500**  
**null** **2-501**  
  null space 2-501  
**num2cell** **2-502**  
**num2str** **2-503**  
**number**  
  of array dimensions 2-492  
  of digits displayed 2-291  
**numbers**

complex 2-31, 2-372  
 finite 2-419  
 imaginary 2-379  
 largest positive 2-596  
 minus infinity 2-420  
 NaN 2-420, 2-486  
 plus infinity 2-391, 2-420  
 prime 2-421, 2-563  
 random 2-584, 2-586  
 real 2-421, 2-595  
 smallest positive 2-597  
**numeric file formats** *See* computers supported by MATLAB  
 numeric precision (of hardware) 2-298, 2-313  
 numerical differentiation formula ODE solvers 2-511  
**nzmax 2-504**

## O

**object**  
 determining class of 2-423  
 inheritance 2-119

**object classes**, list of predefined 2-119, 2-423

**ODE** *See* differential equation solvers

**ode45** and other solvers **2-505**

**odefile 2-513**

**odeget 2-519**

**odeset 2-520**

**ones 2-526**

one-step ODE solver 2-511

**online**

  help 2-365

  keyword search 2-451

**operating system command**, issuing 2-15

**operators**

  arithmetic 2-2

logical 2-11  
 precedence of 2-12  
 relational 2-9, 2-447  
 special characters 2-13  
**Optimization Toolbox** 2-280, 2-283  
**logical OR**  
  bit-wise 2-77  
 or (M-file function equivalent for |) 2-11  
**ordering**  
  minimum degree 2-706  
 reverse Cuthill-McKee 2-706, 2-708  
**orth 2-527**  
 orthogonal-triangular decomposition 2-453, 2-571  
 orthonormal matrix 2-571  
**otherwise 2-528**  
 Out of memory (error message) 2-529  
**output**  
  controlling format of 2-291  
  controlling paging of 2-366, 2-483  
  number of M-file arguments 2-488  
**overdetermined equation systems, solving**  
  2-572-2-573  
**overflow 2-391**

## P

**pack 2-529**

Padé approximation (of matrix exponential) 2-249

**paging**

  controlling output in command window 2-366, 2-483

**parentheses (special characters)** 2-14

**Parlett's method** (of evaluating matrix functions) 2-311

**partial fraction expansion** 2-602

**partialpath 2-531**

**pascal 2-532**

Pascal matrix 2-532, 2-560  
**path** **2-533**  
pathname  
  of functions or files 2-745  
partial 2-531  
*See also* search path  
**pause** **2-535**  
pausing M-file execution 2-535  
**pcg** **2-536**  
**pcode** **2-540**  
percent sign (special characters) 2-15  
period (.), to distinguish matrix and array operations 2-2  
period (special characters) 2-14  
**perms** **2-541**  
permutation  
  of array dimensions 2-542  
  matrix 2-454, 2-571  
  random 2-588  
permutations of n elements 2-541  
**permute** **2-542**  
**persistent** **2-543**  
persistent variable 2-543  
phase, complex 2-31  
  correcting angles 2-729  
**pi** **2-544**  
 $\pi$  ( $\pi$ ) 2-544, 2-591, 2-630  
**pi nv** **2-545**  
platform *See* computers  
plot, volumetric  
  generating grid arrays for 2-473  
plotting *See* visualizing  
**plus** (M-file function equivalent for +) 2-4  
**pol2cart** **2-548**  
polar coordinates 2-86, 2-88, 2-548  
poles of transfer function 2-602  
**poly** **2-550**  
**polyarea** **2-553**  
**polyder** **2-554**  
**polyeig** **2-555**  
**polyfit** **2-556**  
polygon  
  area of 2-553  
  detecting points inside 2-397  
polynomial  
  characteristic 2-550-2-551, 2-607  
  coefficients (transfer function) 2-602  
  curve fitting with 2-556  
  derivative of 2-554  
  division 2-193  
  eigenvalue problem 2-555  
  evaluation 2-559  
  evaluation (matrix sense) 2-560  
  multiplication 2-134  
**polyval** **2-559**  
**polyvalm** **2-560**  
poorly conditioned  
  eigenvalues 2-47  
  matrix 2-371  
**pow2** **2-562**  
power  
  matrix *See* matrix exponential  
  of two, next 2-493  
power (M-file function equivalent for  $\cdot ^{\wedge}$ ) 2-4  
precedence of operators 2-12  
prime factors 2-253  
  dependence of Fourier transform on 2-261  
prime numbers 2-421, 2-563  
**primes** **2-563**  
printing, suppressing 2-15  
**prod** **2-564**  
product  
  cumulative 2-148  
  Kronecker tensor 2-432

- 
- of array elements 2-564  
 of vectors (cross) 2-146  
 scalar (dot) 2-146  
**profile 2-565**  
**K>> prompt** 2-431  
 prompting users for input 2-398, 2-472  
 pseudoinverse 2-545
- Q**
- qmr 2-567**  
**qr 2-571**  
 QR decomposition 2-453, 2-571  
     deleting a column from 2-574  
     inserting a column into 2-575  
**qrdelete 2-574**  
**qrinsert 2-575**  
**qtwrite 2-579**  
**quad 2-580**  
**quad8 2-580**  
 quadrature 2-580  
**quit 2-582**  
 quitting MATLAB 2-582  
 quotation mark, inserting in a string 2-296  
**qz 2-583**  
 QZ factorization 2-555, 2-583
- R**
- rand 2-584, 2-699**  
**randn 2-392, 2-586**  
 random  
     numbers 2-584, 2-586  
     permutation 2-588  
     sparse matrix 2-658, 2-659  
     symmetric sparse matrix 2-660  
**randperm 2-588**
- range space 2-527  
**rank 2-589**  
 rank of a matrix 2-589  
**rat 2-590**  
 rational fraction approximation 2-590  
**rats 2-590**  
**rcond 2-593**  
**rdivide** (M-file function equivalent for `. /`) 2-4  
 reading  
     ASCII delimited file 2-208  
     formatted data from file 2-302  
     WK1 spreadsheet files 2-751  
**README file** 2-744  
**readsnd 2-594**  
**real 2-595**  
 real numbers 2-421, 2-595  
 real Schur form 2-619  
**realmax 2-596**  
**realmint 2-597**  
 rearranging arrays  
     converting to vector 2-16  
     removing first  $n$  singleton dimensions 2-628  
     removing singleton dimensions 2-670  
     reshaping 2-601  
     shifting dimensions 2-628  
     swapping dimensions 2-418, 2-542  
 rearranging matrices  
     converting to vector 2-16  
     flipping left-right 2-276  
     flipping up-down 2-277  
     rotating  $90^\circ$  2-609  
     transposing 2-14  
**recordsound 2-598**  
 reduced row echelon form 2-611  
 regularly spaced vectors, creating 2-16, 2-441  
 relational operators 2-9, 2-447  
 relative accuracy

floating-point 2-235  
**rem** **2-599**  
remainder after division 2-599  
repeatedly executing statements 2-289, 2-747  
replicating a matrix 2-600  
**repmat** **2-600**  
**reshape** **2-601**  
**residue** **2-602**  
residues of transfer function 2-602  
resume execution (from breakpoint) 2-162  
**return** **2-604**  
reverse Cuthill-McKee ordering 2-706, 2-708  
**rmfile** **d** **2-605**  
**rmpath** **2-606**  
RMS *See* root-mean-square  
root directory 2-468  
root-mean-square  
  of vector 2-498  
**roots** **2-607**  
roots of a polynomial 2-550-2-551, 2-607  
Rosenbrock banana function 2-283  
Rosenbrock ODE solver 2-511  
Rosser matrix 2-334  
**rot90** **2-609**  
rotations  
  Givens 2-574, 2-575  
  Jacobi 2-660  
**round**  
  to nearest integer 2-610  
  towards infinity 2-95  
  towards minus infinity 2-278  
  towards zero 2-274  
**round** **2-610**  
roundoff error  
  characteristic polynomial and 2-551  
  convolution theorem and 2-134  
  effect on eigenvalues 2-47

evaluating matrix functions 2-311  
in inverse Hilbert matrix 2-417  
partial fraction expansion and 2-603  
polynomial roots and 2-607  
sparse matrix conversion and 2-643  
**rref** **2-611**  
**rrefmovie** **2-611**  
**rsf2csf** **2-613**  
Runge-Kutta ODE solvers 2-511

**S**  
**save** **2-616**  
saving  
  ASCII data 2-616  
  WK1 spreadsheet files 2-752  
  workspace variables 2-616  
scalar product (of vectors) 2-146  
scaled complementary error function (defined)  
  2-236  
scattered data, aligning  
  multi-dimensional 2-491  
  two-dimensional 2-354  
Schmidt semi-normalized Legendre functions  
  2-437  
**schur** **2-619**  
Schur decomposition 2-619  
  matrix functions and 2-311  
Schur form of matrix 2-613, 2-619  
**script** **2-621**  
scrolling, screen *See* paging  
search path  
  adding directories to 2-25  
  MATLAB's 2-533, 2-725  
  removing directories from 2-606  
**search**, **string** 2-273  
**sec** **2-622**

- secant 2-622  
 secant, inverse 2-38  
 secant, inverse hyperbolic 2-38  
**sech 2-622**  
 semicolon (special characters) 2-15  
 sequence of matrix names (M1 through M12)  
     generating 2-241  
 session  
     saving log of 2-203  
 set operations  
     difference 2-624  
     exclusive or 2-627  
     intersection 2-413  
     membership 2-424  
     union 2-727  
     unique 2-728  
**setdiff 2-624**  
**setfield 2-625**  
**setstr 2-626**  
**setxor 2-627**  
**shiftdim 2-628**  
**sign 2-629**  
 signum function 2-629  
 Simpson's rule, adaptive recursive 2-581  
**sin 2-630**  
 sine 2-630  
 sine, inverse 2-39  
 sine, inverse hyperbolic 2-39  
 single quote (special characters) 2-14  
 singular value  
     decomposition 2-589, 2-700  
     largest 2-498  
     rank and 2-589  
 singularities  
     integrable 2-581  
     soft 2-581  
**sinh 2-630**  
**size 2-632**  
 size of array dimensions 2-632  
 size vector 2-601, 2-632  
 skipping bytes (during file I/O) 2-313  
 smallest array elements 2-478  
 soccer ball (example) 2-708  
 soft singularities 2-581  
**sort 2-634**  
 sorting  
     array elements 2-634  
     complex conjugate pairs 2-144  
     matrix rows 2-635  
**sortrows 2-635**  
**sound**  
     converting vector into 2-636, 2-638  
**sound 2-636, 2-638**  
**soundcap 2-637**  
**spalloc 2-639**  
**sparse 2-640**  
 sparse matrix  
     allocating space for 2-639  
     applying function only to nonzero elements of 2-649  
     density of 2-496  
     diagonal 2-644  
     finding indices of nonzero elements of 2-271  
     identity 2-648  
     minimum degree ordering of 2-123  
     number of nonzero elements in 2-496, 2-640  
     permuting columns of 2-126  
     random 2-658, 2-659  
     random symmetric 2-660  
     replacing nonzero elements of with ones 2-654  
     results of mixed operations on 2-641  
     vector of nonzero elements 2-497  
     visualizing sparsity pattern of 2-665  
 sparse storage

criterion for using 2-307  
**spconvert** **2-642**  
**spdiags** **2-644**  
**speak** **2-647**  
**speye** **2-648**  
**spfun** **2-649**  
**sph2cart** **2-650**  
spherical coordinates 2-650  
**spline** **2-651**  
spline interpolation (cubic) 2-401, 2-404, 2-408, 2-411  
**Spline Toolbox** 2-403  
**spones** **2-654**  
**spparms** **2-655**  
**sprand** **2-658**  
**sprandn** **2-659**  
**sprandsym** **2-660**  
spreadsheet  
    loading WK1 files 2-751  
    reading ASCII delimited file into a matrix 2-208  
    writing matrix as ASCII delimited file 2-210  
    writing WK1 files 2-752  
**sprintf** **2-661**  
**spy** **2-665**  
**sqrt** **2-666**  
**sqrtm** **2-667**  
square root  
    of a matrix 2-667  
    of array elements 2-666  
**squeeze** **2-670**  
**sscanf** **2-671**  
standard deviation 2-675  
**startup** **2-674**  
startup file 2-467, 2-674  
status  
    of file or variable 2-244  
**std** **2-675**  
stopwatch timer 2-716  
storage  
    allocated for nonzero entries (sparse) 2-504  
    sparse 2-640  
**str2cell** **2-100**  
**str2num** **2-677**  
**strcat** **2-678**  
**strcmp** **2-680**  
**strcmpi** **2-682**  
string  
    comparing one to another 2-680  
    comparing the first n characters of two 2-686  
    converting from vector to 2-105  
    converting matrix into 2-466, 2-503  
    converting to lowercase 2-452  
    converting to matrix (formatted) 2-671  
    converting to numeric array 2-677  
    converting to uppercase 2-730  
    dictionary sort of 2-635  
    evaluating as expression 2-241  
    finding first token in 2-689  
    inserting a quotation mark in 2-296  
    searching and replacing 2-688  
    searching for 2-273  
string matrix to cell array conversion 2-100  
**strings** **2-683**  
**strjust** **2-684**  
**strmatch** **2-685**  
**strncmp** **2-686**  
**strncmpi** **2-687**  
**strrep** **2-688**  
**strtok** **2-689**  
**struct2cell** **2-691**  
structure array  
    field names of 2-266  
    getting contents of field of 2-344

- remove field from 2-605  
 setting contents of a field of 2-625
- strvcat 2-692**
- sub2ind 2-693**
- subfunction 2-309**
- subsasgn 2-694**
- subspace 2-697**
- subsref 2-696**
- subsref** (M-file function equivalent for  $A(i, j, k, \dots)$ ) 2-15
- subtraction** (arithmetic operator) 2-2
- sum**
- cumulative 2-149
  - of array elements 2-698
- sum 2-698**
- superiority 2-699**
- svd 2-700**
- svds 2-702**
- swatch 2-704**
- symmd 2-706**
- symrcm 2-708**
- syntaxes**
- of M-file functions, defining 2-309
- T**
- table lookup** *See* interpolation
- tab-separated ASCII format** 2-616
- tan 2-712**
- tangent** 2-712
- hyperbolic 2-712
  - (four-quadrant), inverse 2-43
  - inverse 2-41
  - hyperbolic 2-41
- tanh 2-712**
- Taylor series** (matrix exponential approximation) 2-249
- tempdir 2-714**
- tempname 2-715**
- temporary**
- file 2-715
  - system directory 2-714
- tensor, Kronecker product** 2-432
- test matrices** 2-319
- test, logical** *See logical tests and detecting*
- tic 2-716**
- tiling** (copies of a matrix) 2-600
- time**
- CPU 2-145
  - elapsed (stopwatch timer) 2-716
  - required to execute commands 2-240
- time and date functions** 2-234
- times** (M-file function equivalent for  $\cdot \ast$ ) 2-4
- toc 2-716**
- toeplitz 2-717**
- Toeplitz matrix** 2-717
- token** *See also* string 2-689
- tolerance, default** 2-235
- Toolbox**
- Optimization 2-280, 2-283
  - Spline 2-403
- toolbox**
- undocumented functionality in 2-744
- trace 2-718**
- trace of a matrix** 2-202, 2-718
- trailing blanks**
- removing 2-189
- transform, Fourier**
- discrete, one-dimensional 2-258
  - discrete, two-dimensional 2-261
  - inverse, one-dimensional 2-375
  - inverse, two-dimensional 2-376
  - shifting the DC component of 2-263
- transformation**

elementary Hermite 2-341  
 left and right (QZ) 2-583  
*See also* conversion  
**transpose**  
     array (arithmetic operator) 2-3  
     matrix (arithmetic operator) 2-3  
**transpose** (M-file function equivalent for `.'`) 2-4  
**trapz** **2-719**  
**tricubic interpolation** 2-354  
**tril** **2-721**  
**trilinear interpolation** 2-354, 2-408, 2-411  
**triu** **2-722**  
**truth tables** (for logical operations) 2-11  
**try** **2-723**  
**tsearch** **2-724**  
**type** **2-725**

**U**

**uint8** **2-726**  
**uminus** (M-file function equivalent for unary `-`)  
     2-4  
**undefined numerical results** 2-486  
**undocumented functionality** 2-744  
**unimodular matrix** 2-341  
**union** **2-727**  
**unique** **2-728**  
**unitary matrix** (complex) 2-571  
**unwrap** **2-729**  
**uplus** (M-file function equivalent for unary `+`)  
     2-4  
**upper** **2-730**  
**upper triangular matrix** 2-722  
**uppercase to lowercase** 2-452

**V**

**Vandermonde matrix** 2-558  
**varargout** **2-731**  
**variable numbers of M-file arguments** 2-731  
**variables**  
     clearing 2-120  
     global 2-345  
     (workspace) listing 2-748  
     local 2-309, 2-345  
     name of passed 2-399  
     persistent 2-543  
     retrieving from disk 2-442  
     saving to disk 2-616  
     sizes of 2-748  
     status of 2-244  
*See also* workspace  
**vector**  
     dual 2-494  
     frequency 2-450  
     length of 2-439  
     product (cross) 2-146  
**vectorize** **2-733**  
**vectors, creating**  
     logarithmically spaced 2-450  
     regularly spaced 2-16, 2-441  
**ver** **2-734**  
**version** **2-734**  
**vertcat** (M-file function equivalent for `[ ; ]`) 2-15  
**visualizing**  
     cell array structure 2-99  
     sparse matrices 2-665  
**voronoi** **2-735**

**W**

**warning** **2-737**

warning message (enabling, suppressing, and displaying) 2-737

**wavread 2-738**

**wavwrite 2-739**

**weekday 2-741**

well conditioned 2-593

**what 2-742**

**whatsnew 2-744**

**which 2-745**

**while 2-747**

white space characters, ASCII 2-421, 2-689

**who 2-748**

**whos 2-748**

wildcard (\*)

- using with clear command 2-120
- using with dir command 2-206

**wilkinson 2-750**

Wilkinson matrix 2-645, 2-750

**wk1read 2-751**

**wk1write 2-752**

workspace

- changing context while debugging 2-163, 2-175
- clearing variables from 2-120
- consolidating memory 2-529
- predefining variables 2-467, 2-674
- saving 2-616
- See also* variables

**writesnd 2-753**

writing

- binary data to file 2-313
- formatted data to file 2-292
- matrix as ASCII delimited file 2-210
- string to matrix (formatted) 2-671
- WK1 spreadsheet files 2-752

**X**

**xlgetrange 2-754**

**xlsetrange 2-755**

logical XOR 2-756

- bit-wise 2-80

**xor 2-756**

xyz coordinates *See* Cartesian coordinates

**Z**

zero of a function, finding 2-316

zero-padding

- while converting hexadecimal numbers 2-370
- while reading binary files 2-297

**zeros 2-757**